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

Teoría computacional (en ACL2) sobre cálculos proposicionales

Memoria presentada por
Francisco Jesús Martín Mateos
para optar al grado de
Doctor en Matemáticas
por la Universidad de Sevilla

Francisco Jesús Martín Mateos

V. B. Director

D. José Antonio Alonso Jiménez

Sevilla, Junio de 2002

A mis padres ...

... a mis hermanos ...

... y sobre todo, a Inma.

Agradecimientos

El trabajo desarrollado en esta memoria está parcialmente financiado por los proyectos TIC2000-1368-C03-02 del Ministerio de Ciencia y Tecnología y PB96-1345 del Ministerio de Educación y Ciencia.

En primer lugar, quiero agradecer a José Antonio Alonso Jiménez su dirección, apoyo y comprensión, sin los cuáles este trabajo no habría sido posible.

Al Grupo de Lógica Computacional de la Universidad de Sevilla, cuyas reuniones de trabajo han sido fuente de ideas. En especial me gustaría dar las gracias a José Luis Ruiz Reina por su ayuda incondicional y a María José Hidalgo Doblado por sus palabras de ánimo.

A los miembros del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla, por lo mucho que me aprecian y me aguantan.

A J Moore y Matt Kaufmann, creadores del sistema ACL2, por aceptar con agrado cualquier comentario o sugerencia acerca del sistema y por permitirme incluir una copia del mismo junto con la memoria.

A mi familia, en especial a mis padres, Asunción y Manuel, que hace mucho tiempo depositaron su confianza en mi.

Y finalmente a Inma, que siempre está dispuesta a escucharme y animarme. Sin su apoyo, esta memoria no se habría terminado.

Índice General

1 Introducción **1**

2 El sistema de razonamiento ACL2 **19**
2.1 El lenguaje 21
2.2 La lógica 22
2.2.1 Ordinales 24
2.2.2 Relaciones bien fundamentadas 25
2.2.3 Principio de definición 26
2.2.4 Principio de inducción 27
2.2.5 Encapsulados 29
2.2.6 Instanciación funcional 30
2.2.7 Equivalencias y Congruencias 31
2.3 El demostrador 32

3 Un modelo formal de la lógica proposicional **35**
3.1 Sintaxis 35
3.1.1 Símbolos y conectivas 36
3.1.2 Representación de las fórmulas 37
3.1.3 Fórmulas proposicionales 39
3.1.4 Clasificación de las fórmulas 42
3.1.5 Teorema de descomposición única 43
3.1.6 Símbolos proposicionales de una fórmula 44
3.2 Semántica clásica 45
3.2.1 Asignaciones 46
3.2.2 Funciones de verdad 48
3.2.3 Valor de una fórmula en una asignación 49
3.2.4 Satisfacibilidad, validez y consecuencia lógica 50
3.3 Semántica de Kleene 54
3.3.1 $\mathbb{K}$-asignaciones 54
3.3.2 Funciones de verdad en $\mathbb{K}$ 56
3.3.3 Valor de una fórmula en una $\mathbb{K}$-asignación 57
3.3.4 Satisfacibilidad, validez y consecuencia lógica en $\mathbb{K}$ 58
3.4 Tablas de verdad 60

3.4.1 Asignaciones relevantes de una fórmula 61
3.4.2 Decisión de validez basada en tablas de verdad 63
3.4.3 Decisión de satisfacibilidad basada en tablas de verdad . . 65
3.4.4 Tablas de verdad en $\mathbb{K}$ 66
3.5 Ejemplos 69

4 Cálculo de tableros semánticos 73
4.1 Notación uniforme 73
4.1.1 Clasificación basada en la notación uniforme 74
4.1.2 Medida asociada a la notación uniforme 78
4.1.3 Notación uniforme en $\mathbb{K}$ 79
4.2 Tableros semánticos 79
4.2.1 Tableros semánticos proposicionales 80
4.2.2 Modelo de una rama: $MOD_{\mathcal{R}}$ 87
4.2.3 Corrección y completitud de $MOD_{\mathcal{R}}$ 89
4.2.4 Satisfacibilidad por tableros: $SAT_{\mathcal{T}}$ 91
4.2.5 Demostrabilidad por tableros: $DAT_{\mathcal{T}}$ 92
4.2.6 Deducibilidad por tableros: $DEDUC_{\mathcal{T}}$ 93
4.2.7 Tableros semánticos en $\mathbb{K}$ 95
4.3 Ejemplos 99

5 Cálculo de secuentes 103
5.1 Secuentes 103
5.1.1 El cálculo G' 104
5.1.2 Contramodelo de un secuente: $CONTRAMOD_{\mathcal{S}}$ 110
5.1.3 Corrección y completitud de $CONTRAMOD_{\mathcal{S}}$ 115
5.1.4 Demostrabilidad por secuentes: $DAT_{\mathcal{S}}$ 116
5.1.5 Satisfacibilidad por secuentes: $SAT_{\mathcal{S}}$ 117
5.1.6 Deducibilidad por secuentes: $DEDUC_{\mathcal{S}}$ 119
5.1.7 Secuentes en $\mathbb{K}$ 120
5.2 Ejemplos 125

6 Extensiones de ACL2 129
6.1 Pruebas de terminación basadas en multiconjuntos 130
6.1.1 Relaciones bien fundamentadas entre multiconjuntos . . . 131
6.1.2 El comando `defmul` 139
6.1.3 Ejemplo 140
6.2 Teorías genéricas 144
6.2.1 Teorías genéricas en ACL2 144
6.2.2 Herramienta de instanciación genérica 147
6.2.3 Ejemplo 151

7 Sistemas de transformación proposicionales 155
7.1 Un marco genérico para la decisión de satisfacibilidad 156
7.1.1 Un algoritmo genérico de decisión de satisfacibilidad: $SAT_{\mathcal{G}}$ 156
7.1.2 Terminación de $SAT_{\mathcal{G}}$ 162
7.1.3 Corrección y completitud de $SAT_{\mathcal{G}}$ 163
7.1.4 Un algoritmo genérico de decisión de validez: $DAT_{\mathcal{G}}$ 165
7.1.5 La teoría genérica `*sat-generico*` 165
7.1.6 Sistemas de transformación proposicionales en $\mathbb{K}$ 166
7.2 Un STP exitoso basado en tableros semánticos 168
7.2.1 Descripción del STP exitoso $\mathcal{T}$ 168
7.2.2 Formalización de $\mathcal{T}$ 170
7.2.3 Un $\mathbb{K}$-STP exitoso basado en tableros semánticos 176
7.3 Un STP exitoso basado en secuentes 176
7.3.1 Descripción del STP exitoso $\mathcal{S}$ 177
7.3.2 Formalización de $\mathcal{S}$ 179
7.3.3 Un $\mathbb{K}$-STP exitoso basado en secuentes 184
7.4 Ejemplos 185

8 Procedimiento de Davis y Putnam 189
8.1 Cláusulas y formas clausales 190
8.1.1 Sintaxis y semántica 190
8.1.2 Transformación a forma clausal: $\mathcal{FC}$ 195
8.1.3 Cláusulas y formas clausales en $\mathbb{K}$ 203
8.2 Davis y Putnam 205
8.2.1 Transformaciones de formas clausales 206
8.2.2 Decisión de satisfacibilidad por bifurcación: $SAT_{\mathcal{B}}$ 215
8.2.3 Decisión de satisfacibilidad por Davis-Putnam: $SAT_{\mathcal{D}}$ . . . 220
8.2.4 Procedimiento de Davis y Putnam en $\mathbb{K}$ 225
8.3 Un STP exitoso basado en Davis-Putnam 227
8.3.1 Descripción del STP exitoso $\mathcal{D}'$ 228
8.3.2 Formalización de $\mathcal{D}'$ 230
8.4 Ejemplos 236

9 Resolución 241
9.1 Resolución condicionada 242
9.1.1 Conjuntos saturados por resolución condicionada 242
9.1.2 Decisión de insatisfacibilidad por saturación: $INSAT_{\mathcal{R}}$ . . 248
9.1.3 Terminación de $INSAT_{\mathcal{R}}$ 252
9.1.4 Corrección de $INSAT_{\mathcal{R}}$ y otras propiedades 255
9.2 Teorema de completitud de Bezem 258
9.2.1 Conjuntos representantes 259
9.2.2 Modelo de un conjunto saturado por resolución 263
9.2.3 Completitud de $INSAT_{\mathcal{R}}$ 271

9.3 Teoría genérica e instancias 272
9.3.1 La teoría genérica `*resolucion*` 272
9.3.2 Instancias 272
9.4 Ejemplos 282

10 Conclusiones **285**

Bibliografía **297**

Glosario **298**

Capítulo 1

Introducción

Uno de los primeros objetivos de la Inteligencia Artificial, es el de desarrollar un sistema informático capaz de demostrar teoremas correspondientes a problemas abiertos en matemáticas. Esta idea es el punto de partida para el desarrollo de uno de los campos de investigación más activos dentro de la Inteligencia Artificial: el Razonamiento Automático. La definición que da Larry Wos [85] sobre este campo es la siguiente:

> Automated reasoning is concerned with the study of using the computer to assist in that part of problem solving requiring reasoning. Some questions arisinq during the study concern the representation of knowledge, the rules for deriving new knowledge frow that which is given, and strategies for controling the rules. Other questions concern the implementation of the resulting theory and concern various applications for which the corresponding software can be used. Theory, implementation, and application play equally vital and interconnecting roles for automated reasoning in its attempt to reach one of its primary goals – the goal of providing an automated reasoning assistant.

Los primeros trabajos en este campo utilizaban las lógicas clásicas (proposicional y de primer orden) como lenguaje para formalizar los problemas. Así, en los años 60 aparecen el procedimiento de Davis y Putnam [19] y los sistemas basados en la regla de resolución de Robinson [65]. En ambas aproximaciones los problemas se formalizan utilizando cláusulas. El procedimiento de Davis y Putnam utiliza unas reglas que van construyendo un modelo del conjunto de cláusulas, reduciendo el problema hasta hacerlo trivial. Por otro lado, la regla de resolución de Robinson es una regla de inferencia que deriva nuevas cláusulas combinando un algoritmo de unificación y una generalización de la regla de *modus ponens*. De esta forma, si la cláusula vacía es generada, se puede afirmar que el conjunto de cláusulas inicial es inconsistente y, por tanto, no tiene modelos.

El trabajo en este tipo de sistemas se centra en tres aspectos, la búsqueda de nuevas reglas de inferencia, la mejora de las estrategias de aplicación de las reglas y el diseño de implementaciones eficientes. En lo que respecta a los sistemas basados en resolución, el más extendido es Otter [56]. Es necesario observar que con el sistema EQP, una variante de Otter, se ha demostrado con éxito la conjetura de Robbins, un problema abierto en matemáticas [55]. Los sistemas basados en el procedimiento de Davis y Putnam se encuentran entre los más eficientes para la lógica proposicional. En este caso, destacamos que con uno de ellos, SATO [88], se han conseguido demostrar problemas abiertos en el estudio de quasigrupos [87].

Este tipo de sistemas resultan insuficientes para abordar problemas de mayor complejidad, como la formalización de sistemas lógicos (por ejemplo hardware y software informático) y la verificación de sus propiedades. Para algunas de estas aplicaciones es necesario utilizar otras lógicas, distintas de la proposicional o de primer orden, que proporcionen mayor expresividad: lógicas de orden superior, modales, inductivas, constructivas, etc.

Aunque la expresividad de estas lógicas es superior a la de la lógica de primer orden, la dificultad de automatizar el proceso de deducción es mayor. Este hecho hace que los sistemas de razonamiento basados en estas lógicas sean más interactivos que automáticos. Si en un extremo se encuentran los sistemas totalmente automáticos, como Otter o SATO, en el otro se encuentran los denominados "comprobadores de pruebas", cuyo objetivo es comprobar la validez de una prueba proporcionada por el usuario.

En un punto intermedio entre automatismo e interactividad se encuentran los sistemas que siguen el "estilo LCF" [31], en el que las pruebas se construyen con un pequeño núcleo de reglas de inferencia y el usuario puede construir *tácticas* que automatizan la aplicación de secuencias de estas reglas. El uso de estas tácticas simplifica el proceso de deducción.

Cuando se utiliza un sistema de razonamiento automático para formalizar un sistema, es deseable que se pueda ejecutar el modelo formal del mismo. Esto incrementa la confianza en que el modelo construido se ajusta al sistema real. En algunos sistemas esto no es posible. En los basados en una lógica constructiva, se pueden extraer programas ejecutables a partir de las pruebas desarrolladas. En otros, como ACL2, el lenguaje de la lógica es un lenguaje de programación que permite ejecutar los modelos construidos aún sin haber demostrado sus propiedades.

En [81] se puede encontrar una amplia relación de sistemas de razonamiento automático en la que se indican sus principales características.

El sistema ACL2

ACL2 es, según la descripción dada en [81], un "demostrador automático de teoremas dirigido por un conjunto de reglas, construido de forma incremental

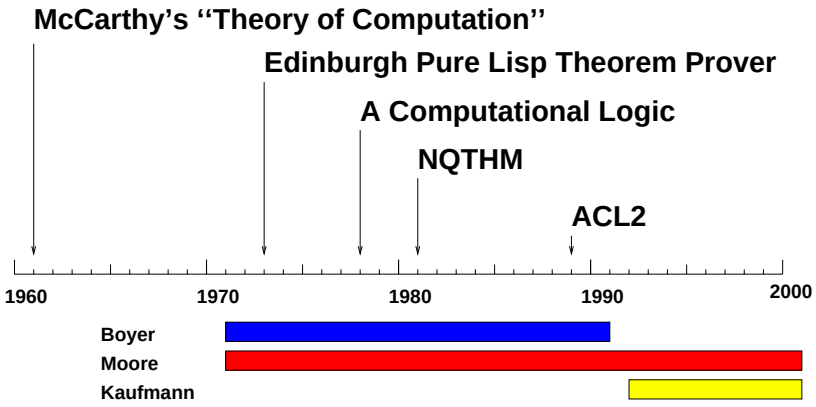

Figura 1.1: Evolución del demostrador ACL2

mediante la demostración de teoremas por parte del usuario. Incluye procedimientos específicos de decisión para lógica proposicional, igualdad y aritmética lineal, así como heurísticas de reescritura basada en congruencias, eliminación de destructores, generalización e inducción". El sistema ACL2 es un sucesor del demostrador de teoremas de Boyer y Moore, Nqthm [13] y su versión interactiva Pc-Nqthm [43]. La figura 1.1[1] muestra la evolución del sistema desde el punto de vista de uno de sus autores, J Moore.

El demostrador de teoremas de Boyer y Moore fue construido en 1971 en Edimburgo. Originalmente éste era un demostrador de teoremas completamente automático para una lógica basada en un dialecto de Lisp. Las ideas clave de este demostrador eran: el uso de Lisp como lógica base, un principio de definición para funciones recursivas totales, un uso extensivo de reescritura y "evaluación simbólica" y una regla de inducción heurística. El diseño de este sistema fue fuertemente influenciado por John McCarthy [54] y Woody Bledsoe.

A lo largo del tiempo el demostrador fue evolucionando e incorporando nuevas funcionalidades. Un cambio clave en la historia del mismo fué la incorporación en 1974 de un sistema de simplificación, que hacía uso de reglas de reescritura derivadas de teoremas demostrados por el usuario. Entonces el sistema dejó de ser completamente automático, ya que un usuario experimentado podía guiarlo hacia la demostración de un teorema complicado mediante la selección de un conjunto adecuado de lemas previos.

En 1979 los autores del demostrador, Robert Boyer y J Strother Moore, publican el libro "A Computational Logic" [11] en el que describen las técnicas que el demostrador emplea. En 1981 Boyer y Moore desarrollan Nqthm. Éste es un sucesor del demostrador ideado en Edimburgo, que dispone de nuevos tipos de

[1]Este figura esta tomada de las transparencias de la conferencia "Machines Reasoning about Machines", que se pueden encontrar en la página personal de J Moore, `http://www.cs.utexas.edu/users/moore/publications/index.html`

reglas mediante los que el usuario podía controlar el comportamiento del sistema, así como de un procedimiento de decisión para la aritmética lineal. En 1987 Matt Kaufmann implementa Pc-Nqthm proporcionando a Nqthm un sistema interactivo de comprobación de pruebas.

En 1989, Boyer y Moore escriben la primera versión de ACL2, con la idea de construir una lógica que permitiera razonar sobre un subconjunto aplicativo de Common Lisp. Una de sus principales novedades es que las funciones definidas en la lógica pueden ser ejecutadas eficientemente en cualquier implementación de Common Lisp. En 1990 se añade la instanciación funcional como una regla de inferencia derivada, proporcionando a la lógica del demostrador características de orden superior [10].

Con el tiempo, Matt Kaufmann se involucra en el proyecto ACL2 y la dedicación de Boyer decrece, hasta que éste decide dejar de ser considerado coautor del sistema. En la actualidad, la última versión de ACL2 es la 2.6 y puede ser considerado una obra de J Moore y Matt Kaufmann, nutriéndose de las ideas de un numeroso grupo de usuarios.

Cálculos proposicionales

La lógica proposicional y los cálculos proposicionales han sido objeto de estudio desde hace mucho tiempo. Por un lado, el problema de la satisfacibilidad proposicional fué el primer problema NP-completo conocido [17] de ahí el gran interés en resolverlo de manera eficiente. En la práctica este problema es fundamental en la resolución de muchos problemas que aparecen en razonamiento automático, diseño asistido por ordenador, bases de datos, robótica, diseño de circuitos integrados, etc. Un análisis detallado de las distintas aproximaciones para resolver el problema y sus aplicaciones se puede encontrar en [32]. En este sentido se han realizado algunos trabajos sobre formalización y verificación de propiedades de algoritmos concretos para resolver el problema de satisfacibilidad, [33].

Por otro lado, los trabajos en lógica proposicional son punto de referencia para resolver problemas similares en otras lógicas [25], [24]. En este sentido Caldwell [15] realiza una formalización de un cálculo de secuentes para la lógica proposicional intuicionista, partiendo de un desarrollo similar para la lógica proposicional clásica. Además, ciertos resultados en lógica de primer orden se obtienen gracias a un "lema de ascenso" que los relaciona con resultados similares en la lógica proposicional.

En esta memoria presentamos un modelo formal de la lógica proposicional, formalizando algunos de los cálculos proposicionales más conocidos, para los cuales se desarrollan y verifican procedimientos de decisión de satisfacibilidad. Este desarrollo es punto de partida para realizar modelos formales de otras lógicas. Por otro lado, como indica Shankar en [74], el uso combinado de pequeños demostradores especializados puede ser un pilar importante en el futuro del razonamiento automático. En este trabajo formalizamos y verificamos las propiedades

de corrección y completitud de algunos de estos demostradores para la lógica proposicional.

Un modelo formal de la lógica proposicional

El punto de partida para el desarrollo de esta memoria es la formalización en ACL2 de la lógica proposicional clásica. Las principales decisiones de diseño que tomamos son las relativas a la representación de las fórmulas. De hecho no es tan importante la representación de las mismas como el conjunto de propiedades que tiene que verificar dicha representación. Esta representación se trata como un tipo abstracto de dato definido mediante un conjunto de funciones de acceso y constructoras, caracterizadas por un conjunto de propiedades que las relacionan. De esta forma la representación se puede modificar para ajustarse a distintas situaciones siempre y cuando el conjunto de propiedades que la caracteriza se mantenga.

Para representar las fórmulas utilizamos listas en notación prefija. Es decir, el primer elemento de estas listas es la conectiva principal de la fórmula y los restantes son las subfórmulas componentes. Las conectivas consideradas son negación, conjunción, disyunción, implicación y equivalencia. Los símbolos proposicionales están representados mediante símbolos del sistema o números enteros positivos. Esto último fue añadido para poder generar fácilmente fórmulas dependientes de un parámetro, como las fórmulas de Urquhart [83] o las que representan el problema de las 8 reinas [62]. Al igual que ésta, se han hecho a posteriori otras modificaciones en la representación para "ajustarla" a conjuntos de ejemplos disponibles en internet (IFIP[2], DIMACS[3] [82], etc). En estas situaciones sólo ha sido necesario comprobar las propiedades que caracterizan la representación, siendo igual el desarrollo de los cálculos lógicos.

En lo que a la semántica se refiere, la principal dificultad aparece con la representación de las asignaciones. Una asignación es una función del conjunto de símbolos proposicionales en el conjunto de valores de verdad, por tanto, se trata de un objeto infinito. Una posibilidad de representar las asignaciones es mediante funciones del lenguaje. Esta opción dificulta el uso de las asignaciones como argumento en otras funciones, por ejemplo para calcular el valor de verdad de una fórmula, o para expresar propiedades, por ejemplo para afirmar que una fórmula es válida. No descartamos completamente esta posibilidad, aunque queda fuera de los objetivos de esta memoria ya que supondría profundizar en la capacidad de ACL2 para formalizar resultados de orden superior.

Una posibilidad más asequible de representar las asignaciones es la utilización de listas de asociación. Una lista de asociación es una lista de pares en los que se relacionan los símbolos de la lógica con el valor de verdad que se les asigna. Una restricción debida a este tipo de representación es que, al tratarse de un

[2]Disponibles en la página de Fabio Massacci: `http://www.ing.unitn.it/~massacci/`.
[3]Disponibles en `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`.

objeto finito, explícitamente sólo se puede hacer corresponder un valor de verdad a un conjunto finito de símbolos proposicionales. Sin embargo, esta restricción no limita el modelo formal construido, ya que el valor de verdad de una fórmula depende de los valores de verdad de los símbolos proposicionales que en ella aparecen, y éste es un conjunto finito. Por otro lado tendremos que decidir qué valor de verdad le corresponde por defecto a aquellos símbolos que no se encuentran en la lista de asociación que representa una asignación.

En esta memoria se han considerado dos opciones a la hora de hacer corresponder un valor de verdad por defecto a los símbolos que no aparecen en la representación de una asignación. En la primera opción, este valor es *falso*. De esta forma la semántica que se obtiene es la clásica. Una segunda opción es *indeterminado*, dando lugar a una formalización de la semántica trivalorada fuerte de Kleene. En esta última se hace explícito el hecho de que se puede determinar el valor de verdad de una fórmula, sin necesidad de conocer los valores de verdad de todos los símbolos proposicionales que en ella aparecen. Por ejemplo, si el valor de verdad del símbolo p es cierto entonces no necesitamos saber el valor de verdad del símbolo q para afirmar que la fórmula $p \vee q$ es cierta. De hecho, si en la semántica de Kleene se define el concepto de validez con respecto a los valores cierto e indeterminado, el resultado es homomórfico a la semántica clásica, es decir, tiene el mismo conjunto de fórmulas válidas.

Hemos verificado las propiedades de completitud y corrección de los cálculos lógicos desarrollados en esta memoria tanto en la semántica clásica como en la semántica de Kleene. Las formalizaciones obtenidas son bastante similares. De esta forma, presentamos en detalle las correspondientes a la semántica clásica e indicamos las principales diferencias que aparecen cuando se considera la semántica de Kleene.

Tableros semánticos y secuentes

Una vez realizado el modelo formal de la lógica proposicional, hemos formalizado algunos cálculos bien conocidos en dicho modelo. El objetivo es demostrar que los algoritmos basados en estos cálculos son correctos y completos.

El primer cálculo lógico que consideramos es el basado en tableros semánticos, para el que hemos seguido la descripción que se da en [26]. Un tablero semántico es un árbol cuyos nodos están etiquetados con fórmulas. Las ramas de un tablero se interpretan como la conjunción de todas las fórmulas que etiquetan sus nodos. De esta forma, cuando en una rama hay una fórmula que se comporta como la conjunción de otras dos, la rama se puede expandir añadiéndole estas componentes y preservando el conjunto de modelos de la misma. El propio tablero se interpreta como la disyunción de todas sus ramas. De esta forma, cuando en una rama hay una fórmula que se comporta como la disyunción de dos componentes, podemos añadir una bifurcación en dicha rama, añadiendo cada una de las dos componentes a una de las ramas de la bifurcación. Si en una rama apare-

cen fórmulas complementarias entonces ni ella ni las que se obtengan a partir de ella tendrán modelos. Este proceso de expansión de las ramas se realiza hasta que en todas aparezcan fórmulas complementarias o hasta obtener una rama sin fórmulas complementarias en la que el procedimiento de expansión descrito no genere nuevos nodos. La terminación de este proceso queda asegurada pues en cada paso quedan menos fórmulas por expandir en las distintas ramas del tablero.

El resultado de corrección de un procedimiento de decisión de satisfacibilidad consiste en demostrar que si el procedimiento tiene éxito al ser evaluado sobre una fórmula, entonces esta fórmula tiene al menos un modelo. La manera de expresar este tipo de resultados en ACL2 es proporcionando explícitamente dicho modelo. En el cálculo de tableros semánticos este modelo se obtiene a partir de una rama sin fórmulas complementarias en la que el procedimiento de expansión no genere nuevos nodos. En este caso cualquier asignación en la que los literales de dicha rama sean ciertos es modelo de dicha rama y, por las propiedades de las reglas de expansión, modelo de la fórmula original.

Este proceso se ha formalizado dando lugar a procedimientos de decisión de validez, satisfacibilidad y consecuencia lógica. Para estos procedimientos se han demostrado resultados de corrección y completitud tanto en la semántica clásica como en la semántica de Kleene. La formalización realizada es genérica en cierto sentido, ya que se ha considerado la existencia de una función, sin proporcionarla de forma explícita, que selecciona la fórmula a expandir en cada paso. De esta forma, hemos construido toda una familia de procedimientos de decisión basados en tableros semánticos y hemos demostrado la corrección y completitud de todos los elementos de esta familia.

A continuación presentamos el cálculo de secuentes G' de G. Gentzen [29]. Los secuentes son pares de listas de fórmulas entre las que hay una relación de consecuencia: la conjunción de todas las fórmulas del primer elemento del par (o *parte izquierda*) implica la disyunción de todas las fórmulas del segundo elemento del par (o *parte derecha*). Este cálculo consta de un conjunto de axiomas y un conjunto de reglas de inferencia. En el proceso para determinar la validez de una fórmula, las reglas de inferencia del cálculo G' se utilizan hacia atrás, es decir, se obtienen las premisas a partir de la conclusión de las reglas. De esta forma, a partir del secuente que representa a una fórmula se obtienen las premisas de una regla de inferencia que tiene a dicho secuente en la conclusión. Las características de las reglas de inferencia nos aseguran que si todas las premisas son válidas entonces la conclusión también lo es. Para los secuentes obtenidos se vuelve a aplicar el mismo proceso, de forma que en todo momento se considera un conjunto de premisas tal que la validez de todas ellas implica la validez de la fórmula original. El proceso continua hasta que todos los secuentes que estemos considerando sean axiomas del cálculo G', que son secuentes válidos o se obtenga uno que no sea un axioma ni se pueda obtener como consecuencia de aplicar una regla de inferencia. La terminación de este proceso queda asegurada pues las premisas de una regla son menos complejas que la conclusión, es decir, en cada

paso estamos considerando un conjunto de premisas más simples (aunque quizá haya mayor cantidad de ellas).

El cálculo de secuentes utiliza un conjunto de reglas de expansión que preservan la validez. De esta forma, si el proceso tiene éxito asegura la validez de la fórmula original y si falla, indica que la fórmula original es insatisfacible. En este caso se puede construir un contramodelo de dicha fórmula a partir del secuente que provoca el fallo en el proceso. Cualquier asignación que sea modelo de todos los literales de su parte izquierda y no lo sea de los literales de la parte derecha es contramodelo de dicho secuente y, por las propiedades de las reglas de inferencia, también es contramodelo de la fórmula original.

Al igual que para el cálculo basado en tableros semánticos, este proceso se ha formalizado dando lugar a procedimientos de decisión de validez, satisfacibilidad y consecuencia lógica, para los que se han demostrado resultados de corrección y completitud tanto en la semántica clásica como en la de Kleene. La formalización realizada también es genérica en el mismo sentido que la realizada para el cálculo de tableros semánticos. Se ha considerado la existencia de una función, sin proporcionarla de forma explícita, que selecciona la regla a aplicar en cada paso. Por tanto, en este caso también hemos construido y verificado formalmente una familia de procedimientos de decisión basados en el cálculo de secuentes.

Sistemas de transformación proposicionales

Los dos cálculos presentados tienen un patrón de comportamiento común. Estos métodos no trabajan directamente con fórmulas sino con unos objetos construidos a partir de ellas (ramas de un tablero semántico o secuentes). Los objetos se modifican repetidamente usando unas reglas de expansión que reducen su complejidad. Estas reglas preservan el significado de los objetos. Eventualmente, a partir de cierto tipo de objetos simples se pueden obtener asignaciones que prueban la satisfacibilidad de la fórmula original. A estas asignaciones las llamaremos *asignaciones distinguidas*. Si no se encuentra ninguno de estos objetos simples entonces la fórmula original es insatisfacible.

Este patrón común se formaliza en lo que llamamos *sistemas de transformación proposicionales* o *STP*. Estos son ternas formadas por un conjunto de los *objetos proposicionales* construidos a partir de las fórmulas, un conjunto de *reglas de expansión*, que indican cómo modificar un objeto para obtener una lista de componentes, y una relación entre los objetos proposicionales y las asignaciones, que establece qué asignaciones son distinguidas para cada uno de los objetos. En el conjunto de reglas de expansión pueden existir reglas en las que la lista de componentes sea vacía y reglas que modifiquen un objeto para obtener el símbolo **t**. Las primeras son reglas que sirven para eliminar objetos a partir de los cuales es imposible obtener una asignación distinguida. Las segundas sirven para indicar que a partir de determinados objetos se puede obtener una asignación distinguida.

Para asegurar que, a partir de un STP, se puede construir un procedimiento de decisión de satisfacibilidad, tenemos que exigirle ciertas propiedades. Por un lado, tenemos que disponer de una *función de representacion* mediante la cual se pueda construir un objeto proposicional asociado a una fórmula, tal que las asignaciones distinguidas de dicho objeto sean los modelos de la fórmula. Este objeto será transformado usando las reglas de expansión, así que tenemos que asegurarnos de que hay una forma de utilizar dichas reglas de expansión (a lo que llamaremos una *regla de computación*), tal que las asignaciones distinguidas del objeto sean también asignaciones distinguidas de alguna de sus componentes. De esta forma, las reglas de expansión en las que la lista de componentes es vacía sirven para identificar objetos que no tienen asignaciones distinguidas. También es necesario asegurar que aquellos objetos que la regla de computación transforma en el símbolo **t**, tienen una asignación distinguida y, para poder utilizar dicha asignación en los resultados de corrección del procedimiento, esta asignación viene proporcionada por una *función modelo*.

En esta situación podemos definir un procedimiento genérico para decidir la satisfacibilidad de una fórmula. El primer paso consiste en considerar la lista formada por el resultado de aplicar la función de representación a dicha fórmula. A continuación, y de forma repetitiva, se considera un objeto de dicha lista y se le aplica la regla de computación obteniendo las componentes resultado de utilizar cierta regla de expansión. Si se obtiene el símbolo **t**, el proceso termina con éxito, la fórmula original es satisfacible y un modelo se obtiene aplicando la función modelo sobre el último objeto considerado. Si se obtiene una lista de componentes, el proceso continua con el resultado de reemplazar el objeto considerado por estas componentes. Si, tras iterar este procedimiento, se obtiene la lista de objetos vacía, entonces la fórmula original es insatisfacible.

Para formalizar este procedimiento en ACL2 se ha de asumir la existencia de funciones que describen el STP, una función de representación, una regla de computación y una función modelo, con las propiedades que hemos descrito. Sin embargo, existe una dificultad adicional relacionada con la terminación del procedimiento: se trabaja con una lista de objetos que, como resultado de aplicar ciertas reglas de expansión, aumenta su tamaño en algunas iteraciones.

Para demostrar la terminación de un proceso recursivo en ACL2, tenemos que proporcionar una medida de los argumentos y una relación bien fundamentada, definida sobre los valores obtenidos por dicha medida, de forma que, la medida de los argumentos en cada una de las llamadas recursivas sea menor, con respecto a la relación bien fundamentada, que la medida de los argumentos originales. En nuestro caso el único argumento del proceso recursivo es una lista de objetos proposicionales, para los que sabemos que las reglas de expansión "reducen su complejidad". De esta forma, en cada paso se toma un objeto de dicha lista y se sustituye por varios de menor complejidad. Por tanto, se puede afirmar que la lista de objetos "disminuye" en cierto modo.

Hemos construido una herramienta ACL2 para extender una relación bien

fundamentada, definida sobre los valores obtenidos por cierta medida, a multiconjuntos (es decir conjuntos con repeticiones) de estas medidas, de forma que la relación resultante también es bien fundamentada. Esta herramienta se puede utilizar para demostrar la terminación del proceso de decisión de satisfacibilidad asociado a un STP. Para ello basta con proporcionar una *función de medida* de los objetos proposicionales, tal que, para toda regla de expansión, la medida de las componentes obtenidas sea más pequeña que la medida del objeto al que se aplica dicha regla.

Para poder reutilizar el trabajo realizado en la formalización de un STP genérico y obtener procedimientos de decisión de satisfacibilidad basados en el cálculo de tableros semánticos o en el de secuentes, se puede utilizar el proceso de *instanciación funcional* del que dispone ACL2. Con este proceso, si se ha demostrado un resultado para determinadas funciones con ciertas propiedades, se puede obtener de forma inmediata el mismo resultado para otras funciones siempre y cuando cumplan las mismas propiedades. Hemos construido otra herramienta ACL2 para automatizar este proceso. Así, conseguimos versiones "concretas" del desarrollo realizado para un STP genérico, proporcionando las funciones que describen el caso concreto y demostrando las propiedades mínimas que hay que exigir a un STP para poder realizar dicho desarrollo.

Todo el desarrollo de los STP, su formalización, la construcción de procedimientos de decisión de satisfacibilidad y validez asociados y la demostración de los resultados de corrección y completitud de los mismos, se ha realizado con éxito tanto en la semántica clásica como en la de Kleene. También se han construido STP basados en el cálculo de tableros semánticos y en el de secuentes obteniendo, a partir del desarrollo realizado para un STP genérico, procedimientos verificados de decisión de satisfacibilidad y validez basados en estos dos cálculos.

El procedimiento de Davis y Putnam

Presentamos a continuación una formalización del procedimiento de Davis y Putnam para comprobar la satisfacibilidad de un conjunto de cláusulas o *forma clausal*. Previamente formalizamos la sintaxis y semántica de las cláusulas y las formas clausales y proporcionamos un procedimiento que transforma una fórmula en una forma clausal lógicamente equivalente. Este procedimiento sirve para caracterizar las fórmulas válidas, es por esto que no desarrollamos procesos de decisión de validez basados en el procedimiento de Davis y Putnam (ni tampoco lo hacemos para resolución).

El procedimiento de Davis y Putnam va construyendo un modelo de una forma clausal, considerando en cada paso que un determinado literal es cierto. La forma clausal se reduce en cada paso eliminando aquellas cláusulas en las que aparece el literal considerado cierto (pues ya son ciertas) y eliminando las ocurrencias del complementario de dicho literal (pues son falsas). Si en algún momento se obtiene la cláusula vacía entonces la forma clausal original es insatisfacible. Si el

proceso termina con la lista vacía de cláusulas entonces la forma clausal original es satisfacible y se puede obtener un modelo a partir de los literales que se han considerado ciertos en el proceso de reducción de la misma. Obviamente, para cada literal considerado en cada paso existen dos posibilidades, que dicho literal sea cierto o que lo sea su complementario. La clave del procedimiento de Davis y Putnam consiste en identificar aquellos literales para los que sólo es necesario analizar una de las dos posibilidades. Sin embargo, en algunas ocasiones no existirá ninguno de estos literales y será necesario tomar uno cualquiera para considerar las dos opciones.

Hemos formalizado este procedimiento de forma genérica. Se ha considerado una función que selecciona un literal en cada paso, dando prioridad a aquellos que no bifurcan el proceso frente a los que sí lo hacen. De esta forma se pueden utilizar distintas heurísticas de selección de un literal. Hemos demostrado que el procedimiento obtenido tiene las propiedades de corrección y completitud tanto en la semántica clásica como en la de Kleene. También hemos construido un STP basado en el procedimiento de Davis y Putnam, obteniendo procedimientos verificados de decisión de satisfacibilidad y validez como versiones concretas del desarrollo realizado para un STP genérico.

Procedimientos basados en resolución

El último de los cálculos proposicionales formalizados es el que proporciona la regla de resolución de Robinson. Esta regla actúa sobre dos cláusulas y obtiene una tercera, llamada *resolvente*, que es consecuencia lógica de las iniciales. Los procedimientos basados en resolución generan todas las resolventes posibles a partir de un conjunto de cláusulas, hasta obtener un conjunto saturado por resolución, es decir, tal que cualquier resolvente entre dos elementos de dicho conjunto ya se encuentra en el mismo. Si en este proceso aparece la cláusula vacía, entonces el conjunto de cláusulas original es insatisfacible. Si el conjunto saturado por resolución no contiene la cláusula vacía, entonces el conjunto de cláusulas original es satisfacible. Hemos utilizado este proceso para construir un procedimiento de decisión de insatisfacibilidad de un conjunto de cláusulas.

Existe distintas variantes de la regla de resolución de Robinson que dan lugar a "refinamientos" del proceso de resolución. Estas variantes consisten en comprobar determinada propiedad sobre dos cláusulas antes de generar su resolvente. A este proceso lo llamamos *resolución condicionada*. Nuestra formalización es general en el sentido de que se asume la existencia de tal propiedad y se desarrolla un procedimiento basado en resolución condicionada del que se pueden obtener como versiones concretas los distintos refinamientos del proceso de resolución.

Es relativamente fácil demostrar que el procedimiento de decisión de insatisfacibilidad basado en resolución condicionada es correcto. Sin embargo, la demostración de la propiedad de completitud se complica debido a la generalidad que se ha incorporado al proceso. La formalización de la misma se basa en la de-

mostración de Bezem de la completitud de la resolución condicionada [8]: a partir de una asignación, relacionada en cierta forma con la condición de resolución, se construye otra que es modelo de un conjunto de cláusulas saturado por resolución condicionada. La formalización que presentamos es la única automatización que conocemos de dicha prueba.

Puesto que el desarrollo del procedimiento de decisión de insatisfacibilidad basado en resolución condicionada se ha realizado de forma genérica, utilizando las herramientas desarrolladas para tal fin, se pueden obtener versiones concretas y verificadas de dicho procedimiento para distintos refinamientos de resolución, siempre que se proporcionen la condición que describe el refinamiento y la asignación asociada necesaria para la prueba del resultado de completitud. De esta forma hemos desarrollado procedimientos de decisión de insatisfacibilidad basados en resolución binaria, positiva, negativa, semántica y con conjunto soporte.

Observaciones sobre el contenido de la memoria

Los capítulos de esta memoria están escritos para poder ser leidos sin necesidad de tener un amplio conocimiento de Lisp o del sistema ACL2. Tras la definición de cada concepto se incluye su formalización en el sistema. De la misma forma, tras el enunciado de todo teorema se describe de forma matemática su prueba y después se presenta su formalización. Todas las formalizaciones están incluidas en cajas enumeradas a las que se hace referencia de la forma $\boxed{143}$. Al final de la memoria hay un glosario de las funciones que forman parte de la formalización desarrollada, en el que se indica la página donde se utilizan por primera vez.

En la memoria sólo aparecen las definiciones y teoremas más importantes, el resto, para guiar a ACL2 hacia las demostraciones realizadas, se pueden consultar en los libros ACL2 que se encuentran en el CD que acompaña a esta memoria. Para mayor claridad, no hemos incluido las anotaciones específicas de ACL2, como sugerencias de prueba o tipos de regla asociados, salvo en aquellos casos en los que la explicación lo requiere.

El contenido del CD

El CD que acompaña a la memoria contiene los ficheros correspondientes a la formalización desarrollada, una copia del sistema ACL2 e instrucciones para instalarlo y evaluar en él la formalización. Veamos una descripción más detallada de los libros:

Descripción de los libros ACL2

El conjunto de definiciones y teoremas (denominados *eventos* en la terminología de ACL2) que formalizan la teoría presentada en esta memoria se encuentran organizados en una serie de ficheros, llamados *libros*. Estos libros están *certificados* en ACL2, es decir, todas las definiciones son admisibles y el sistema es

capaz de demostrar de forma automática todos los teoremas (utilizando quizá un conjunto de eventos previo). Estos libros, junto con la salida que proporciona el sistema al certificarlos, se encuentran en el CD adjunto en el directorio `calculos-proposicionales`. Todos los libros tienen extensión `.lisp`. La salida que proporciona el sistema al certificar un libro está almacenada en un fichero con el mismo nombre que el del libro, pero con extensión `.log`. Para cada capítulo de la memoria hay un subdirectorio cuyo nombre comienza con el número asociado a dicho capítulo. A continuación describimos brevemente el contenido de cada libro, indicando el capítulo al que corresponde y el nombre del directorio que lo contiene.

La figura 1.2 muestra la dependencia entre los libros correspondientes al desarrollo realizado en la semántica clásica. Esta grafo da una idea de como se interrelacionan dichos libros. Existe una interrelación similar para los libros correspondientes al desarrollo realizado en la semántica de Kleene.

- Capítulo 3: Directorio `3-logica`.
  - `sintaxis`: formalización de la sintaxis de la lógica proposicional.
  - `semantica`: formalización de la semántica clásica de la lógica proposicional.
  - `semantica-K`: formalización de la semántica de Kleene de la lógica proposicional.
  - `tablas-de-verdad`: desarrollo de procedimientos de decisión de satisfacibilidad y validez basados en tablas de verdad. Los resultados de corrección y completitud se demuestran con respecto a la semántica clásica.
  - `tablas-de-verdad-K`: desarrollo de procedimientos de decisión de satisfacibilidad y validez basados en tablas de verdad. Los resultados de corrección y completitud se demuestran con respecto a la semántica de Kleene.
- Capítulo 4: Directorio `4-tableros`.
  - `uniforme`: formalización de la notación uniforme con la semántica clásica.
  - `uniforme-K`: formalización de la notación uniforme con la semántica de Kleene.
  - `tableros`: desarrollo de procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica basados en el cálculo de tableros semánticos. Los resultados de corrección y completitud se demuestran con respecto a la semántica clásica.

  - `tableros-K`: desarrollo de procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica basados en el cálculo de tableros semánticos. Los resultados de corrección y completitud se demuestran con respecto a la semántica de Kleene.

- Capítulo 5: Directorio `5-secuentes`.

  - `secuentes`: desarrollo de procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica basados en el cálculo de secuentes. Los resultados de corrección y completitud se demuestran con respecto a la semántica clásica.
  - `secuentes-K`: desarrollo de procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica basados en el cálculo de secuentes. Los resultados de corrección y completitud se demuestran con respecto a la semántica de Kleene.

- Capítulo 6: Directorio `6-extensiones`.

  - `multiconjuntos`: 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.
  - `arboles-binarios`: prueba de la terminación (usando multiconjuntos) de una versión iterativa de un esquema recursivo sobre árboles binarios.
  - `teorias-genericas`: definición de una herramienta para la instanciación de teorías genéricas.
  - `insercion-ordenada`: ejemplo de uso de la herramienta de instanciación genérica.
  - `listas`: estrategia de prueba para extender un resultado de manera conjuntiva a los elementos de una lista.

- Capítulo 7: Directorio `7-generico`.

  - `SAT-generico`: definición de procedimientos de decisión de satisfacibilidad y validez para un sistema de transformación proposicional cualquiera. Las propiedades de corrección y completitud se demuestran con respecto a la semántica clásica. También se define una teoría genérica para las funciones que describen dichos procedimientos.
  - `SAT-generico-K`: definición de procedimientos de decisión de satisfacibilidad y validez para un $\mathbb{K}$-sistema de transformación proposicional cualquiera. Las propiedades de corrección y completitud se demuestran con respecto a la semántica de Kleene. También se define una

teoría genérica para las funciones que describen dichos procedimientos.

- `SAT-tableros`: instancia basada en el cálculo de tableros semánticos, de la teoría genérica definida para un sistema de transformación proposicional.
- `SAT-tableros-K`: instancia basada en el cálculo de tableros semánticos, de la teoría genérica definida para un $\mathbb{K}$-sistema de transformación proposicional.
- `SAT-secuentes`: instancia basada en el cálculo de secuentes, de la teoría genérica definida para un sistema de transformación proposicional.
- `SAT-secuentes-K`: instancia basada en el cálculo de secuentes, de la teoría genérica definida para un $\mathbb{K}$-sistema de transformación proposicional.

- Capítulo 8: Directorio `8-davisputnam`.
  - `clausulas`: formalización de los conceptos de cláusula y forma clausal con la semántica clásica.
  - `clausulas-K`: formalización de los conceptos de cláusula y forma clausal con la semántica de Kleene.
  - `forma-clausal`: definición de un procedimiento de transformación a forma clausal. Las propiedades de este procedimiento se demuestran con respecto a la semántica clásica.
  - `forma-clausal-K`: definición de un procedimiento de transformación a forma clausal. Las propiedades de este procedimiento se demuestran con respecto a la semántica de Kleene.
  - `davis-putnam`: desarrollo de procedimientos de decisión de satisfacibilidad basado en el procedimiento de Davis y Putnam. Los resultados de corrección y completitud se demuestran con respecto a la semántica clásica.
  - `davis-putnam-K`: desarrollo de procedimientos de decisión de satisfacibilidad basados en el procedimiento de Davis y Putnam. Los resultados de corrección y completitud se demuestran con respecto a la semántica de Kleene.
  - `SAT-davisputnam`: instancia basada en el procedimiento de Davis y Putnam, de la teoría genérica definida para un sistema de transformación proposicional.

- Capítulo 9: Directorio `9-resolucion`.
  - `conjuntos`: definición y propiedades básicas acerca de conjuntos.
  - `representantes`: definición de conjunto de representantes de una colección de conjuntos. Teorema de existencia del conjunto de representantes minimal.
  - `resolucion-sat`: definición de resolución condicionada y conjunto saturado por resolución condicionada. Definición y prueba de terminación de un procedimiento de decisión de insatisfacibilidad basado en resolución condicionada.
  - `resolucion-thm`: formalización de la prueba de Bezem del teorema de completitud de la resolución condicionada.
  - `resolucion-aux`: demostración de las propiedades de corrección y completitud del procedimiento de decisión de insatisfacibilidad basado en resolución condicionada.
  - `resolucion-gen`: definición de una teoría generica para las funciones a partir de las que se construye el procedimiento de decisión de insatisfacibilidad basado en resolución condicionada. Se incluyen instancias para de esta teoría genérica para la resolución binaria, positiva, negativa, semántica y con conjunto soporte.
- Directorio `0-ejemplos`.
  - `urquhart`: generador de fórmulas de Urquhart.
  - `pelletier`: 17 primeras fórmulas de Pelletier.
  - `plaisted`: fórmulas de Plaisted.
  - `reinas`: generador de fórmulas codificando el problema de las N reinas.

Estos libros también pueden ser obtenidos a través de la siguiente página web:

`http://www.cs.us.es/~fmartin/tesis/`

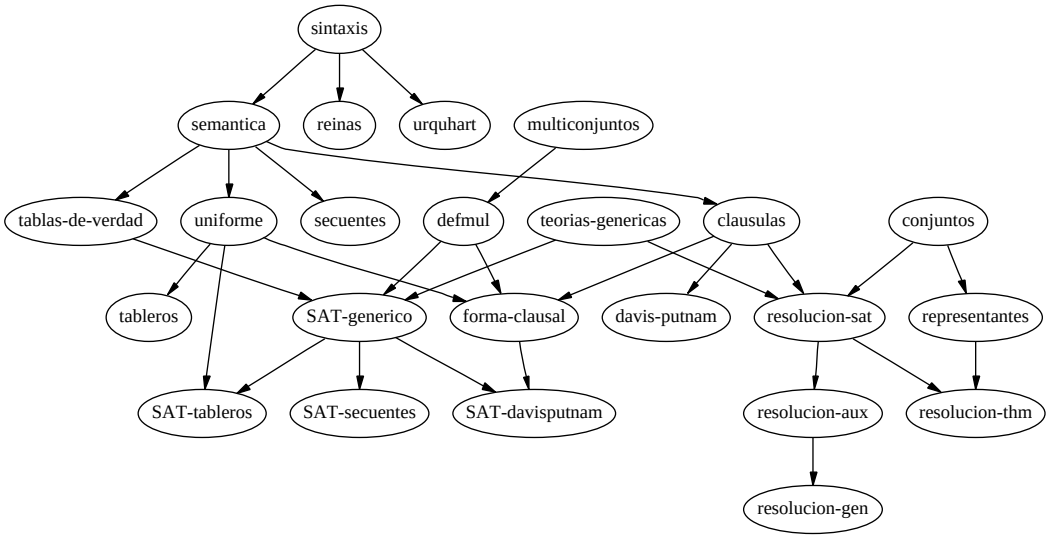

Figura 1.2: Dependencias entre los libros desarrollados en la semántica clásica

Capítulo 2

El sistema de razonamiento ACL2

Presentamos en este capítulo una introducción al sistema ACL2. El principal objetivo del mismo es facilitar al lector la comprensión del código que formaliza los distintos conceptos y propiedades desarrollados en esta memoria. Una introducción más completa acerca del funcionamiento del sistema se puede encontrar en [39].

ACL2 son las siglas de *A Computational Logic for Applicative Common Lisp*. El sistema ACL2 es, al mismo tiempo, un lenguaje de programación, una lógica para razonar sobre los programas construidos en ese lenguaje y un demostrador que automatiza, en cierta medida, el razonamiento en la lógica. El sistema ACL2 es un descendiente directo del demostrador automático de Boyer y Moore, Nqthm [13], del que hereda parte de su filosofía y su lógica, pero se modifica para poder razonar sobre un subconjunto aplicativo de Common Lisp.

Ambos sistemas han sido utilizados para formalizar gran variedad de resultados:

- El Teorema Fundamental del Cálculo [37].
- La ley de cuadrados recíprocos de Gauss [68].
- La indecidibilidad del problema de la parada [12].
- El teorema de Church-Rosser para el lambda-cálculo [71].
- El teorema de Ramsey [47].
- El teorema de incompletitud de Gödel [72].
- Corrección de un algoritmo de la transformada rápida de Fourier [28].
- El teorema de pares críticos de Knuth y Bendix [66].

También 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 [58].
- 21 de las 22 rutinas de la biblioteca de cadenas de Berkeley C (compiladas para el procesador Motorola 6820) [86].
- Programas de microcódigo de la ROM del procesador de señales digitales CAP, de Motorola [14].
- El microcódigo del algoritmo de división de coma flotante y de raíz cuadrada del procesador AMD K5 [59, 70].
- El código RTL que implementa las operaciones elementales sobre números de coma flotante del procesador AMD Athlon [69].

Se puede encontrar más referencias sobre éstos y otros proyectos de verificación desarrollados con Nqthm y ACL2 en las páginas web de los sistemas ([42] y [13]).

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

- La mejor introducción a ACL2 es el libro [39]. Viene acompañado de un segundo volumen [38] en el que se describen una serie de proyectos de verificación que se han llevado a cabo usando el sistema. En [44] también se proporciona una introducción rápida al sistema, explicando su evolución a partir de Nqthm.
- La página web de ACL2 [42] 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.
- En el aspecto técnico, la información más completa sobre el sistema es el manual de referencia, cuya versión más actualizada se encuentra en [42].

La teoría formal presentada en esta memoria ha sido desarrollada usando la versión 2.6 del sistema. En lo que sigue describimos algunas características del lenguaje de programación, la lógica y el demostrador automático de ACL2, que facilitan la comprensión de esta memoria.

2.1 El lenguaje

El lenguaje de programación ACL2 es una *extensión* de un *subconjunto aplicativo* de Common Lisp. Este subconjunto contiene los tipos de datos básicos de Common Lisp y la mayoría de los elementos del mismo que no producen efectos colaterales. De esta forma, elementos como las variables globales y las operaciones destructivas no son parte del lenguaje de programación ACL2. De hecho, 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. Es por ello que decimos que ACL2 es un subconjunto aplicativo (o *funcional*) de Common Lisp. Esta restricción permite simplificar considerablemente el razonamiento formal sobre los programas desarrollados con el lenguaje. Finalmente, ACL2 incorpora algunas funciones predefinidas que no se encuentran en el estándar Common Lisp, extendiéndolo.

No comentamos aquí las características que ACL2 comparte con Common Lisp. Sin duda alguna, la mejor descripción del estándar Common Lisp se encuentra en [79], aunque cualquier manual de programación Lisp proporciona información suficiente para comprender la mayor parte del código desarrollado en esta memoria. Por tanto, supondremos que el lector está familiarizado con las nociones básicas del lenguaje: notación prefija, constantes y variables, valores lógicos (`t` y `nil`), definición de funciones y macros (`defun` y `defmacro`), estructuras condicionales (`cond` e `if`), entornos locales (`let` y `let*`), predicados lógicos (`and`, `or` y `not`), funciones de manipulación de listas (`endp`, `consp`, `equal`, `member`, `cons`, `car`, `cdr`, `append` ...), listas de asociación (`assoc` y `acons`), etc.

Una de las características de Common Lisp que no comparte ACL2, es la posibilidad de utilizar funciones como argumento en la evaluación de una expresión. Así, la expresión Common Lisp `(member e l :test #'equal)`, comprueba si el elemento `e` pertenece a la lista `l`, utilizando la función `equal` para comparar `e` con los elementos de `l`. La función ACL2 que realiza esta comprobación es `member-equal`. De igual forma, en ACL2 están predefinidas las funciones `subsetp-equal`, `assoc-equal`, ... , que actúan como las correspondientes funciones Common Lisp sin el sufijo "`-equal`", pero utilizando la función `equal` para comparar los elementos.

A lo largo de esta memoria se utilizan otras características ACL2 que no son propias de Common Lisp, cuyo comportamiento se explica la primera vez que se usan. Destacamos aquí dos funciones propias de ACL2 que son utilizadas con mucha frecuencia:

- `iff`: Es una función de comparación lógica. La expresión `(iff x y)` es cierta si y sólo si tanto `x` como `y` tienen el mismo valor lógico, es decir, o ambos son falsos (iguales a `nil`), o ambos son ciertos (distintos de `nil`).

- `implies`: Esta función es la implicación lógica. La expresión `(implies x y)` es falsa si y sólo si `x` es cierto (tiene un valor distinto de `nil`) e `y` es falso (igual a `nil`).

2.2 La lógica

Presentamos en esta sección una introducción informal de la lógica del sistema. Una descripción precisa, detallada y formal de la misma se puede encontrar en [40]. Aquí sólo comentamos algunos aspectos que permiten una mejor comprensión de la teoría presentada en la memoria.

Consideremos el siguiente ejemplo, tomado de [39]: Supongamos definida, en el lenguaje de ACL2, una función de un argumento llamada `ordena`, que ordena listas de números. Supongamos también que queremos comprobar que dicha función es correcta. Para ello debemos verificar que devuelve una lista ordenada con los mismos elementos que la de entrada. Centrémonos en la segunda de estas propiedades. Una primera aproximación sería comprobarlo manualmente evaluando algunos ejemplos. Así, el resultado de evaluar `(ordena '(4 1 7 3))` es `(1 3 4 7)`, que efectivamente es una permutación de `(4 1 7 3)`. Una forma de automatizar este proceso es definiendo en ACL2 una función `permutacion`, que implemente el concepto de permutación mediante un predicado binario. De esta forma, podríamos comprobar que el valor devuelto por la función `ordena` cumple la propiedad descrita por la función `permutacion` para distintas situaciones[1]:

```
ACL2 !>(permutacion '(4 1 7 3) (ordena '(4 1 7 3)))
T
ACL2 !>(permutacion '(2 3 1 5) (ordena '(2 3 1 5)))
T
```

Suponiendo que la definición de la función `permutacion` es correcta, las evaluaciones anteriores aumentan nuestra confianza en que la función `ordena` devuelve una permutación de su argumento, pero no se puede afirmar que, para cualquier objeto `x`, el valor de `(permutacion x (ordena x))` es siempre `t`. Para demostrar esta afirmación utilizamos la lógica de ACL2.

ACL2 es una lógica de primer orden con igualdad sin cuantificadores. Los *términos* de la lógica son las expresiones, tal y como se entiende este concepto en Common Lisp, construidas con el lenguaje descrito en la sección anterior. Los *operadores lógicos* son la igualdad, representada con el predicado `equal`, y las conectivas proposicionales habituales: $\neg$, $\vee$, $\wedge$, $\rightarrow$ y $\leftrightarrow$, representadas con las funciones `not`, `or`, `and`, `implies` y `iff`, respectivamente. Las *fórmulas atómicas* son igualdades de la forma `(equal e1 e2)`, donde `e1` y `e2` son términos. Las *fórmulas* se definen recursivamente de la manera habitual. Una característica importante de las fórmulas es que sus variables se consideran universalmente cuantificadas, así la fórmula `(equal (permutacion x (ordena x)) t)` expresa el hecho de que para cualquier valor de `x`, el valor de `(permutacion x (ordena x))` es `t`.

[1] En estas evaluaciones la cadena `ACL2 !>` es el "prompt" del sistema y, al igual que en los interpretes de Common Lisp, indica que éste está a la espera de que el usuario introduzca una expresión.

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. En la sección 2.2.3 se explica en qué condiciones se admite la definición de una nueva función y cómo se incorpora ésta a la lógica.

Los axiomas y reglas de inferencia describen la lógica de ACL2 como una lógica proposicional con igualdad correspondiente a la axiomatización de Shoenfield [75] (ver [40] para una descripción más precisa de la lógica de ACL2). Además, como las variables de una fórmula se consideran universalmente cuantificadas, se dispone de la siguiente regla de *instanciación*: A partir de una fórmula ϕ se deriva $\sigma(\phi)$. Donde σ representa una asignación de términos a variables.

Las funciones predefinidas en ACL2 están caracterizadas mediante axiomas que especifican su comportamiento. Por ejemplo, los axiomas que caracterizan el comportamiento de `cons`, `car`, `cdr` y `consp` son los siguientes:

- `(equal (consp (cons x y)) t).`
- ```
  (implies (equal (consp x) t)
           (equal (cons (car x) (cdr x)) x)).
  ```
- `(equal (car (cons x y)) x).`
- `(equal (cdr (cons x y)) y).`

La última regla de inferencia de la lógica es la de *inducción*, que será comentada en la sección 2.2.4.

La manera de indicar al sistema que se ha de intentar la prueba de una fórmula en la lógica, es mediante el comando `defthm`. Así, la siguiente expresión indica al sistema que debe intentar demostrar en la lógica, la fórmula que representa la propiedad acerca de las funciones `permutacion` y `ordena`:

```
(defthm permutacion-ordena
  (equal (permutacion x (ordena x)) t))
```

Cuando la fórmula que se quiere demostrar es del tipo `(equal <exp> t)`, la expresión anterior puede reducirse a:

```
(defthm permutacion-ordena
  (permutacion x (ordena x)))
```

Si el intento de prueba tiene éxito, la fórmula es almacenada en la lógica para su posible uso en otros intentos de prueba. El símbolo `permutacion-ordena` es el nombre con el que se asocia la fórmula demostrada.
```

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

Figura 2.1: Ordinales en ACL2

2.2.1 Ordinales

No incluimos aquí un desarrollo completo de la teoría de ordinales, una buena referencia para su estudio es [36]. Goodstein establece en [30] una forma de representar de manera constructiva los ordinales menores que ε_0, mediante números naturales y listas. En la figura 2.1 se muestran algunos ejemplos de la correspondencia entre ordinales (expresados en forma normal de Cantor) y los objetos ACL2 asociados según dicha representación.

Como se observa, los números naturales son ordinales ACL2. Un objeto de la forma `(o1 o2 ... on . n)` es un ordinal ACL2 si a su vez los objetos `o1`, ... , `on`, son ordinales ACL2 distintos de `0`, en orden decreciente y `n` es un número natural. Intuitivamente, los `o1`, ... , `on` 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 corresponden con repeticiones de las potencias. El número `n` es el "término independiente" de la forma normal. En ACL2 se define la función `e0-ordinalp` para reconocer aquellos objetos que representan ordinales y la función `e0-ord-<` que formaliza el orden entre objetos ACL2 que representan ordinales, implementando el orden usual entre éstos.

2.2.2 Relaciones bien fundamentadas

Un concepto importante dentro de la lógica de ACL2 es el de relación noetheriana. Informalmente, una relación $\lhd$ se dice *noetheriana* (o que *termina*) si no existen sucesiones decrecientes infinitas en el conjunto en el que está definida: $\ldots \lhd x_3 \lhd x_2 \lhd x_1$. Este concepto no se puede formalizar directamente en ACL2, sin embargo sí es posible hacerlo con la caracterización que proporciona el siguiente resultado[2]:

> Una relación $\lhd$ en un conjunto A es noetheriana si y sólo si existe una función $F : A \longrightarrow \mathcal{O}rd$ tal que si $x \lhd y$ entonces $F(x) < F(y)$, donde $<$ es el orden usual entre ordinales.

Usando esta caracterización se pueden formalizar relaciones noetherianas cuyo tipo ordinal no sea mayor que ε_0, ya que existe una representación de los ordinales hasta este valor. De esta forma, la siguiente fórmula expresa que la relación binaria representada con la función ***rel*** es noetheriana en el conjunto caracterizado por la función ***mp***, donde la función ***fn*** es una inmersión en los ordinales que garantiza esta propiedad:

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

Un concepto equivalente al de relación noetheriana es el de relación *bien fundamentada*, que son aquellas en las que todo subconjunto no vacío tiene un elemento minimal con respecto a la relación[3]. Es por esto que a las relaciones que verifican la propiedad anterior se las llama *relaciones bien fundamentadas* y a la propiedad *teorema de buena fundamentación*.

Es posible demostrar que `e0-ord-<` es una relación bien fundamentada sobre el conjunto de objetos ACL2 que representan ordinales. Es decir, que no es posible obtener una cadena infinita descendente (respecto de `e0-ord-<`) de objetos ACL2 que verifiquen `e0-ordinalp`. Esta es la única relación bien fundamentada primitiva en la lógica de ACL2. El usuario puede definir nuevas relaciones bien fundamentadas demostrando que verifican la propiedad anterior.

[2]En el apéndice A de [23] se encuentra una prueba del mismo.
[3]En el apéndice A.2 de [66] se encuentra una prueba de este hecho.

2.2.3 Principio de definición

Al definir nuevas funciones en el lenguaje de ACL2, no sólo describimos una forma de realizar determinado cálculo, además ampliamos la lógica del sistema con axiomas que caracterizan las funciones definidas. Por ello, el sistema restringe el tipo de funciones que se pueden definir, para evitar inconsistencias en la teoría desarrollada. Las condiciones bajo las cuales se admite una nueva definición, `(defun f (x1 ... xn) <cuerpo>)`, son las siguientes:

- `f` es un símbolo de función nuevo.
- `x1`, ... , `xn`, son símbolos de variables distintos.
- `<cuerpo>` es una expresión en la que aparecen a lo sumo las funciones definidas previamente a la definición de `f` y la propia `f`. Además las únicas variables libres que pueden aparecer en `<cuerpo>` son `x1`, ... , `xn`.
- Existe una función `m` de n argumentos (a la que llamaremos *medida de terminación*) y una relación bien fundamentada `rel` definida sobre un conjunto caracterizado por la función `mp`, tales que:
  - Se verifica `(mp (m (x1 ... xn)))`
  - Para toda llamada recursiva de la función `f`, `(f u1 ... un)`, condicionada por las expresiones `t1`, ... , `tm`, se verifica que la medida de `u1`, ... , `un` es menor que la medida de `x1`, ... , `xn`, con respecto a la relación bien fundamentada `rel`, cuando se verifican las condiciones `t1`, ... , `tm`:

    ```
    (implies (and t1 ... tm)
             (rel (m u1 ... un) (m x1 ... xn)))
    ```

Donde decimos que una expresión `t` condiciona una llamada recursiva de la función `f` si, para que el cálculo que describe la función `f` se realice mediante dicha llamada recursiva, se tiene que verificar `t` o `(not t)`. Por ejemplo, en la siguiente definición, los términos `(not (endp l1))` y `(member-equal (car l1) l2)` condicionan la única llamada recursiva que existe:

```
(defun permutacion (l1 l2)
  (cond ((endp l1) (endp l2))
        ((member-equal (car l1) l2)
         (permutacion (cdr l1) (elimina-una (car l1) l2)))
        (t nil)))
```

Si la definición de `f` es admisible, se introduce en el lenguaje de ACL2 un nuevo símbolo de función de aridad n que realiza el cálculo descrito en dicha

definición y se añade a la lógica el axioma `(equal (f x1 ... xn) <cuerpo>)`, de forma que la teoría ampliada con dicho axioma es una extensión conservativa de la inicial.

Al probar la admisibilidad de una función, el sistema considera por defecto la relación bien fundamentada `e0-ord-<` y usa una serie de heurísticas para determinar la función de medida `m`. En los casos en los que el sitema no es capaz de demostrar automáticamente la terminación de una función, el usuario puede indicar, mediante la palabra clave `:measure`, una función de medida y/o, mediante la palabra clave `:well-founded-relation`, una relación bien fundamentada con respecto a la que la medida decrece en cada llamada recursiva. Con estas indicaciones, la definición de una función queda como sigue:

```
(defun f (x1 ... xn)
   (declare (xargs :measure m
                   :well-founded-relation rel))
   <cuerpo>)
```

2.2.4 Principio de inducción

Veamos ahora la última regla de inferencia de la lógica de ACL2: el principio de inducción. Supongamos que `(f x1 ... xn)` es una fórmula del lenguaje de ACL2, donde `x1`, ... , `xn` son sus variables libres, entonces dicha fórmula se deriva a partir de las siguientes fórmulas:

- *Caso base*: `(implies (and (not q1) ... (not qm))`
  `(f x1 ... xn))`
- *Casos de inducción*: Para cada `q1`, ... , `qm`:

  ```
  (implies (and qi
                (f ti-1-1 ... ti-1-n)
                ...
                (f ti-ki-1 ... ti-ki-n))
           (f x1 ... xn)))
  ```

donde `q1`, ... , `qm`, `ti-1-1`, ... , `ti-1-n`, ... , `ti-ki-1`, ... , `ti-ki-n` son términos y, para una función `m` de n argumentos (a la que llamaremos *medida de inducción*) y una relación bien fundamentada `rel` definida sobre un conjunto caracterizado por la función `mp`, se pueden demostrar las siguientes fórmulas en la lógica de ACL2:

- `(mp (m x1 ... xm))`
- Para cada `ti-j-1`, ... , `ti-j-n` ($1 \leq$`i`$\leq m$ y $1 \leq$`j`$\leq$`ki`):

  ```
  (implies qi
           (rel (m ti-j-1 ... ti-j-n) (m x1 ... xn)))
  ```

La idea es que para probar una fórmula `(f x1 ... xn)` 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 `qi` (casos inductivos), o bien por la negación de todos los `qi` (caso base). Para la demostración de cada uno de los casos inductivos se permite asumir que determinadas instancias de la fórmula `(f x1 ... xn)` son ciertas. En concreto, para probar el caso correspondiente a `qi` se suponen ciertas un número `ki` de instancias `(f ti-j-1 ... ti-j-n)`, $1 \leq$ `j` $\leq$ `ki`. Cada una de estas instancias se denomina una *hipótesis de inducción*. Para que este esquema de demostración sea correcto, se debe demostrar previamente que para cada instancia de la fórmula `(f x1 ... xn)` asumida en cada hipótesis de inducción, la medida de sus argumentos (dada por la expresión `(m x1 ... xn)`, que toma valores que verifican `mp`) es menor, con respecto a la relación bien fundamentada `rel`, que la medida de los argumentos de la fórmula que se quiere demostrar.

Los esquemas de inducción están ligados a definiciones recursivas. Así, la definición de la función `permutacion` proporciona un esquema de inducción en el que, para demostrar una fórmula `(:P X Y)`, el caso base es:

```
(IMPLIES (AND (NOT (ENDP X))
              (NOT (MEMBER-EQUAL (CAR X) Y)))
         (:P X Y))
```

y los casos inductivos son:

```
(IMPLIES (AND (NOT (ENDP X))
              (MEMBER-EQUAL (CAR X) Y)
              (:P (CDR X) (ELIMINA-UNA (CAR X) Y)))
         (:P X Y))

(IMPLIES (ENDP X)
         (:P X Y)))
```

Este esquema de inducción se justifica por la admisibilidad de la función `permutacion`.

Así, para probar la fórmula `(equal (permutacion x (ordena x)) t)`, lo normal será utilizar el esquema de inducción asociado a la función `permutacion` o la función `ordena`. Si usamos el primero, se tendrán que demostrar los siguientes casos:

```
(IMPLIES (AND (NOT (ENDP X))
              (NOT (MEMBER-EQUAL (CAR X) (ORDENA X))))
         (PERMUTACION X (ORDENA X)))

(IMPLIES (AND (NOT (ENDP X))
              (MEMBER-EQUAL (CAR X) (ORDENA X))
              (PERMUTACION (CDR X)
                           (ELIMINA-UNA (CAR X) (ORDENA X))))
         (PERMUTACION X (ORDENA X)))

(IMPLIES (ENDP X)
         (PERMUTACION X (ORDENA X))))
```

El sistema utiliza un conjunto de heurísticas con el objeto de escoger un esquema de inducción para demostrar una fórmula. Si el esquema de inducción que elige no es el adecuado, la prueba puede fallar aún siendo cierta la fórmula. El usuario puede indicar al sistema el esquema de inducción que quiere utilizar mediante un *consejo*, de esta forma:

```
(defthm permutacion-ordena
  (permutacion x (ordena x))
  :hints (("Goal" :induct (permutacion x (ordena x)))))
```

La expresión que aparece a partir de la segunda línea del enunciado del resultado `permutacion-ordena` se denomina *consejo*. En este caso es un consejo de inducción (`:induct`), e indica al sistema que para demostrar la fórmula se ha de utilizar el esquema de inducción asociado a la función `permutacion`, cuando sus argumentos son `x` y `(ordena x)`

2.2.5 Encapsulados

El principio de encapsulado permite introducir determinados símbolos de función, sin especificar completamente la función que representan, asumiendo solamente una serie de propiedades que los definen de forma parcial. Para que una aplicación del principio de encapsulado sea admisible y se preserve la consistencia, ha de demostrarse previamente que existen funciones, llamadas *testigos locales*, que verifican las propiedades que se quieren asumir. Una vez probado esto, se prescinde de las definiciones correspondientes a los testigos locales y los símbolos de función introducidos por el principio de encapsulado quedan parcialmente especificados mediante las propiedades asumidas. Así, es admisible asumir una propiedad sobre las funciones `f1`, ... , `fn`, descrita por la fórmula `F` si se verifican:

- `f1`, ... , `fn` son símbolos de función nuevos.

- Existe definiciones admisibles de `n` funciones cuyos nombres son `f1`, ... , `fn`, tales que se puede demostrar la fórmula `F` para dichas funciones.

Si el encapsulado es admisible, se introduce en el lenguaje de ACL2 los símbolos de función `f1`, ... , `fn` y se añade a la lógica la fórmula `F` como axioma.

Nótese que los testigos locales sólo se usan para asegurar la admisibilidad del encapsulado, ya que los axiomas correspondientes a sus definiciones no se conservan. El axioma introducido por un encapsulado preserva la consistencia de la teoría extiendida. Véase [41] para una demostración rigurosa de esta afirmación.

Veamos la sintaxis del encapsulado en ACL2 con un ejemplo: supongamos que queremos asumir la existencia de una función `ordena` que devuelve una permutación del argumento que recibe como entrada. El comando ACL2 que realiza esto es el siguiente:

```
(encapsulate
 (((ordena *) => *))

 (local
   (defun ordena (x)
     x))

 (defthm permutacion-ordena
   (permutacion x (ordena x)))

 )
```

La primera expresión es una lista con las descripciones de las aridades de las funciones que se introducen con el principio del encapsulado (denominada *signatura*). En este caso se indica que `ordena` es una función de un argumento que devuelve un único valor. Los testigos locales se definen usando `defun`, pero se declaran como locales mediante `local`. La propiedad asumida sobre las funciones del encapsulado es expresada usando `defthm`. El encapsulado puede contener definiciones y propiedades declaradas como locales. Una vez demostrada la admisibilidad del encapsulado, sólo las definiciones y propiedades no declaradas como locales son incorporadas a la lógica de ACL2.

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.

2.2.6 Instanciación funcional

La regla de instanciación funcional es la forma en que la lógica de ACL2 utiliza resultados de segundo orden. Un típico resultado de segundo orden tiene la

siguiente forma: "Para toda función f que verifique la propiedad P, entonces f también verifica la propiedad Q". La manera de formalizar este resultado en ACL2 es mediante un encapsulado, en el que se asume la existencia de una función `f` que cumple la propiedad P descrita por la fórmula `P` y, una vez admitido dicho encapsulado, la demostración en la lógica de la fórmula `Q` que describe la propiedad Q. El resultado de segundo orden nos asegura que cualquier otra función `g`, que cumpla la propiedad descrita por `P`, también cumple la propiedad descrita por `Q`.

El sistema no utiliza de forma automática la regla de instanciación funcional, sólo lo hace si el usuario se lo indica con un *consejo* adecuado. Por ejemplo, supongamos que después de admitir el encapsulado de la sección anterior acerca de la función `ordena`, que devuelve una permutación de su argumento, hemos demostrado la siguiente propiedad:

```
(defthm permutacion-ordena-ordena
  (permutacion x (ordena (ordena x))))
```

Entonces se puede probar la misma propiedad para cualquier otra función `mezcla` para la que se pueda demostrar `(permutacion x (mezcla x))`:

```
(defthm permutacion-mezcla
  (permutacion x (mezcla x)))

(defthm permutacion-mezcla-mezcla
  (permutacion x (mezcla (mezcla x)))
  :hints (("Goal"
           :by (:functional-instance
                  permutacion-ordena-ordena
                  ((ordena mezcla))))))
```

La expresión que aparece a partir de la segunda línea del enunciado del resultado `permutacion-mezcla-mezcla` se denomina *consejo*. En este caso se trata de un consejo de instanciación funcional (`:functional-instance`), en el que hay que reutilizar la propiedad descrita por el resultado `permutacion-ordena-ordena` reemplazando el símbolo de función `ordena` por `mezcla`. A la lista que asocia los símbolos de función del resultado de segundo orden con los símbolos de función del caso concreto, la denominamos *sustitución funcional*.

2.2.7 Equivalencias y Congruencias

Una relación de equivalencia en ACL2 es una función binaria para la que se han demostrado las propiedades de reflexividad, simetría y transitividad. La manera habitual de hacer esto es evaluando la macro `defequiv` sobre el símbolo de función

que se quiere incorporar como relación de equivalencia en ACL2. Un ejemplo de su uso y la fórmula en la que se expande la evaluación de esta macro, se muestran a continuación:

```
(defequiv equiv)

(defthm equiv-is-an-equivalence
  (and (booleanp (equiv x y))
       (equiv x x)
       (implies (equiv x y)
                (equiv y x))
       (implies (and (equiv x y)
                     (equiv y z))
                (equiv x z)))
```

Una de las reglas de la lógica de ACL2 es la de igualdad con respecto a funciones: si los términos `xi` e `yi` son iguales, entonces cualquier ocurrencia de `xi` en un término `(f x1 ... xi ... xn)` se puede sustituir por `yi`, dando lugar a `(f x1 ... yi ... xn)`. De una forma similar, el usuario puede indicar al sistema que, para cierto símbolo de función `f`, si se sustituye su `i`-ésimo argumento por uno equivalente con respecto a una relación de equivalencia `equiv1`, el resultado es equivalente al original con respecto a otra relación de equivalencia `equiv2`. La manera habitual de hacer esto es mediante una evaluación de la macro `defcong`. Un ejemplo de su uso y la fórmula en la que se expande la evaluación de esta macro, se muestran a continuación:

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

(defthm equiv1-implies-equiv2-f-i
  (implies (equiv1 xi xi-equiv)
           (equiv2 (f x1 ... xi ... xn)
                   (f x1 ... xi-equiv ... xn))))
```

2.3 El demostrador

La tercera componente del sistema ACL2 es un demostrador automático para la lógica descrita en la sección anterior. El demostrador es automático en el sentido de que una vez que comienza un intento de prueba, 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 cierta. 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.

La estrategia de prueba se comunica al demostrador mediante la construcción de lo que llamamos un *mundo lógico*. Éste se construye mediante la definición de funciones que realizan cálculos o formalizan propiedades y la demostración de una serie de propiedades acerca de estas funciones, que el demostrador interpreta en forma de *reglas* y que dirigen su comportamiento al intentar abordar la prueba automática de una fórmula. Si la prueba 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.

La forma usual de construir un mundo lógico consiste en proporcionar al demostrador funciones (definidas con `defun`), propiedades de estas funciones (establecidas con `defthm`) o encapsulados en los que se asumen propiedades sobre símbolos de función (expresados con `encapsulate`). A estas expresiones que sirven para construir el mundo lógico las llamaremos *eventos*. Los eventos se suelen almacenar en ficheros, a los que llamaremos *libros*, en el mismo orden en que el sistema es capaz de admitirlos. El contenido de un libro puede ser incluido en el sistema con la orden `(include-book <fichero>)`. Si el sistema es capaz de admitir todos los eventos incluidos en un libro, éste puede ser *certificado*, con lo que se evita que, cuando el libro sea incluido en el sistema, se vuelvan a realizar todas las pruebas necesarias para admitir los eventos que contiene.

Sumario

En este capítulo:

- Se ha introducido el lenguaje de programación de ACL2, relacionándolo con el de Common Lisp.
- Se ha comentado en líneas generales la lógica de ACL2 y se han descrito algunos elementos importantes de la misma.
- Se ha establecido la manera usual de interaccionar con el demostrador de ACL2.

Capítulo 3

Un modelo formal de la lógica proposicional

En este capítulo presentamos una formalización en ACL2 de la lógica proposicional. El desarrollo se basa fundamentalmente en el capítulo 2 de [26], siendo similar a otras presentaciones que se pueden encontrar en la literatura. Paralelamente a la definición de los conceptos de la lógica se presenta la formalización de los mismos en ACL2. Esta formalización constituye la base sobre la que se desarrollan los capítulos posteriores.

En la primera sección se presenta la sintaxis de la lógica proposicional y se discute la representación de las fórmulas. Se establecen una serie de propiedades que caracterizan esta representación, de forma que el desarrollo posterior se basa fundamentalmente en dichas propiedades. De esta forma, los cambios en la representación no afectarán en la prueba de los resultados demostrados, siempre y cuando las propiedades sigan siendo ciertas en la nueva representación.

A continuación, se presenta la semántica clásica de la lógica proposicional y se discute la representación de las asignaciones. En esta sección se indica cómo se incorporan los conceptos de satisfacibilidad, validez, consecuencia lógica y equivalencia lógica en la formalización de los teoremas.

En la tercera sección se realiza un desarrollo similar al de la sección previa para una semántica de la lógica proposicional basada en la lógica trivalorada fuerte de Kleene [45].

Finalmente se desarrollan algoritmos para decidir la validez y la satisfacibilidad de una fórmula proposicional, basados en tablas de verdad. Se demuestran resultados de corrección y completitud de estos algoritmos en las dos semánticas presentadas.

3.1 Sintaxis

En esta sección presentamos la formalización de la sintaxis de la lógica proposicional. Se centra principalmente en la representación de las fórmulas proposicionales,

para la que se han utilizado listas en notación prefija. Esta representación está formalizada mediante un conjunto de funciones de acceso y constructoras. Estas funciones se caracterizan por un conjunto de propiedades que no hacen referencia directa a la representación escogida. Las definiciones de las funciones de acceso y constructoras se han ocultado al sistema, quedando disponibles únicamente sus propiedades. Esta forma de actuar permite que los cambios en la representación no afecten a los resultados desarrollados posteriormente, bastará con redefinir las funciones de acceso y constructoras y demostrar las propiedades que las caracterizan.

En esta sección también se formalizan conceptos como el rango o el conjunto de símbolos de una fórmula, así como el teorema de descomposición única para la lógica proposicional. Los eventos que se presentan en esta sección se encuentran en el libro `sintaxis.lisp`.

3.1.1 Símbolos y conectivas

Los elementos básicos de la lógica proposicional son un conjunto infinito de símbolos, Σ, que servirán para expresar proposiciones básicas, es decir, enunciados susceptibles de ser verdaderos o falsos. En el modelo formal de la lógica proposicional el conjunto Σ será el formado por todos los símbolos del sistema, reconocidos con el predicado `symbolp`, excepto el símbolo `nil`, junto con todos los números enteros positivos.

1
```
(defun es-simbolo-proposicional (x)
  (or (and (symbolp x) (not (equal x nil)))
      (and (integerp x) (> x 0))))
```

Algunos símbolos proposicionales en nuestra formalización son `p1`, `p2`, `q1`, `q2`, `1`, `2`, ... Así, la expresión `(es-simbolo-proposicional x)` será cierta si y sólo si se verifica la propiedad `x` $\in \Sigma$.

Una fórmula de la lógica proposicional será un objeto construido a partir de los elementos de Σ, utilizando conectivas proposicionales. Las conectivas pueden ser de aridad 0 (constantes), aridad 1 (monarias), aridad 2 (binarias), ... De todas las posibles conectivas que se pueden definir para la lógica proposicional, las de aridad superior a 2 raramente aparecen en la literatura, así que sólo consideraremos conectivas monarias y binarias. La única conectiva monaria de interés es la negación ($\neg$), las conectivas binarias que consideramos son: conjunción ($\wedge$), disyunción ($\vee$), implicación ($\rightarrow$) y equivalencia ($\leftrightarrow$). Estas conectivas son representadas en la formalización de la lógica de la siguiente forma, el símbolo `-` representa la negación, `&` la conjunción, `\/` la disyunción, `->` la implicación y `<->` la equivalencia. En los sucesivo utilizaremos el término *conectiva* para hacer referencia tanto a las conectivas proposicionales como a sus representaciones en la formalización.

Para definir algunos conceptos como el de fórmula proposicional, nos bastará con poder distinguir entre fórmulas monarias y binarias. Esta distinción se establece en función del tipo de la conectiva. Para ello introducimos dos funciones que sirven para reconocer los dos tipos de conectivas, `es-conectiva-monaria` y `es-conectiva-binaria`

2

```
(defun es-conectiva-monaria (c)
  (member-equal c '(-)))

(defun es-conectiva-binaria (c)
  (member-equal c '(& \/ -> <->)))
```

3.1.2 Representación de las fórmulas

Las fórmulas son representadas mediante listas utilizando una notación prefija. Así, el primer elemento será una conectiva y el resto los argumentos, cuyo número dependerá de la conectiva. De esta forma, la fórmula $p_1 \wedge (p_2 \rightarrow p_3)$ se representa por la lista `(& p1 (-> p2 p3))`. De forma general si `F` y `G` son, respectivamente, las representaciones de las fórmulas F y G, entonces la fórmula $\neg F$ se representa por la lista `(- F)` y la fórmula $F \circ G$ se representa por la lista `(· F G)`, donde $\circ$ es una conectiva binaria y $\cdot$ su representación. En lo sucesivo utilizaremos el término *fórmula* para hacer referencia tanto a las fórmulas proposicionales como a sus representaciones en la formalización.

Se han considerado funciones de acceso a los elementos de una fórmula y funciones constructoras de fórmulas. Estas funciones son las únicas que hacen referencia a la representación. Una vez establecidas las propiedades que las relacionan, que a su vez no hacen referencia a la representación, las definiciones de estas funciones se han ocultado. De esta forma, el sistema sólo puede utilizar las propiedades y por tanto el desarrollo posterior no dependerá de la representación escogida. Esta forma de actuar permite que los cambios en la representación no afecten a los resultados desarrollados posteriormente, siempre y cuando se hayan demostrado en la nueva representación, las propiedades que caracterizan a las funciones de acceso y constructoras.

3.1.2.1 Funciones de acceso

Muchos conceptos definidos sobre fórmulas proposicionales hacen referencia a la conectiva principal y las subfórmulas componentes. De manera implícita se están considerando funciones de acceso a estos elementos. Estas funciones sirven para obtener las componentes de una fórmula a partir de su representación. Su definición no depende del concepto de fórmula proposicional, sólo de la representación escogida. El valor devuelto por estas funciones es un elemento concreto de una lista que se les pasa como argumento, de forma que si dicha lista es la representación

de un fórmula, dicho valor coincide con la conectiva principal o las subfórmulas componentes. Estas funciones son, `op`, que devuelve el primer elemento de una lista (si la lista representa una fórmula será la conectiva principal de la misma) y `arg1` y `arg2`, que devuelven, respectivamente, el segundo y tercer elementos de una lista (si la lista representa una fórmula, son sus subfórmulas componentes).

A la hora de implementar estas funciones se ha de comprobar que la expresión que se pasa como argumento tiene la longitud suficiente para poder acceder a su primer, segundo y tercer elemento. Estas funciones devuelven `nil` en el caso de que la expresión argumento no sea una lista o no tenga la longitud adecuada. Esta es la razón por la que el símbolo `nil` no es considerado un símbolo proposicional, puesto que es comúnmente utilizado para indicar un error en la evaluación de una función.

Desde el punto de vista de la representación, será necesario distinguir las expresiones que representan fórmulas de las que no. Llamaremos *expresiones monarias* o *expresiones binarias* a aquellas que representan fórmulas construidas con una conectiva monaria o binaria, respectivamente. Estos conceptos son implementados por las funciones `es-monaria` y `es-binaria`, que comprueban si la expresión que se pasa como argumento es una lista de la longitud adecuada (2 para las expresiones monarias y 3 para las binarias) y si su conectiva principal es, respectivamente, monaria o binaria. Estas funciones sirven para comprobar que la estructura de una expresión es la adecuada, no comprueban si dicha expresión representa una fórmula proposicional correctamente construida.

3.1.2.2 Funciones constructoras

Otro aspecto importante de la representación de las fórmulas es la construcción de expresiones con la estructura correcta. Para cada conectiva definimos una función que construye una expresión para representar una fórmula construida con dicha conectiva y el número de argumentos correspondiente. Estas funciones son `negacion`, `conjuncion`, `disyuncion`, `implicacion` y `equivalencia`. La expresión construida será una lista en la que el primer elemento es una conectiva y los restantes elementos son las expresiones que representan la subfórmulas. Así, si `F` y `G` son expresiones que representan las fórmulas F y G respectivamente, entonces:

- Al evaluar `(negacion F)` se obtiene la expresión `(- F)`.
- Al evaluar `(conjuncion F G)` se obtiene la expresión `(& F G)`.
- Al evaluar `(disyuncion F G)` se obtiene la expresión `(\/ F G)`.
- Al evaluar `(implicacion F G)` se obtiene la expresión `(-> F G)`.
- Al evaluar `(equivalencia F G)` se obtiene la expresión `(<-> F G)`.

Obviamente, el resultado devuelto por la función `negacion` es una expresión monaria y los resultados de las funciones `conjuncion`, `disyuncion`, `implicacion` y `equivalencia` son expresiones binarias.

3.1.2.3 Relación entre funciones de acceso y constructoras

Las funciones constructoras presentadas en el apartado anterior tienen la relación adecuada con las funciones de acceso. La conectiva (`op`) de una expresión obtenida con alguna de las funciones constructoras coincide con la conectiva que dicha función utiliza. También se verifica que las componentes (`arg1` o `arg2`) de una expresión obtenida con alguna de las funciones constructoras coincide con el correspondiente argumento en su llamada. Así `(op (conjuncion F G))` es igual a `&`, `(arg1 (negacion F))` es igual a `F` y `(arg2 (implicacion F G))` es igual a `G`. Estos resultados se establecen en ACL2 de la siguiente forma:

3

```
(defthm acceso-negacion
  (and (equal (op (negacion F)) '-)
       (equal (arg1 (negacion F)) F)))

(defthm acceso-conjuncion
  (and (equal (op (conjuncion F G)) '&)
       (equal (arg1 (conjuncion F G)) F)
       (equal (arg2 (conjuncion F G)) G)))

(defthm acceso-disyuncion
  (and (equal (op (disyuncion F G)) '\/)
       (equal (arg1 (disyuncion F G)) F)
       (equal (arg2 (disyuncion F G)) G)))

(defthm acceso-implicacion
  (and (equal (op (implicacion F G)) '->)
       (equal (arg1 (implicacion F G)) F)
       (equal (arg2 (implicacion F G)) G)))

(defthm acceso-equivalencia
  (and (equal (op (equivalencia F G)) '<->)
       (equal (arg1 (equivalencia F G)) F)
       (equal (arg2 (equivalencia F G)) G)))
```

3.1.3 Fórmulas proposicionales

Definición 3.1 *Una fórmula atómica es un símbolo proposicional. Notaremos por $At(\Sigma)$ al conjunto de las fórmulas atómicas.*

Este concepto está implementado por la función `es-atomica`, de forma que la expresión `(es-atomica F)` es cierta si y sólo si `F` $\in At(\Sigma)$:

4
```
(defun es-atomica (F)
  (es-simbolo-proposicional F))
```

Definición 3.2 *El conjunto de las fórmulas proposicionales es el menor conjunto* $\mathbb{P}(\Sigma)$ *tal que:*

1. $At(\Sigma) \subseteq \mathbb{P}(\Sigma)$.
2. *Si* $F \in \mathbb{P}(\Sigma)$, *entonces* $\neg F \in \mathbb{P}(\Sigma)$.
3. *Si* $\circ$ *es una conectiva binaria y* $F, G \in \mathbb{P}(\Sigma)$, *entonces* $(F \circ G) \in \mathbb{P}(\Sigma)$.

Este concepto está implementado por la función `es-proposicional`. Así, la expresión `(es-proposicional F)` será cierta si y sólo si `F` $\in \mathbb{P}(\Sigma)$. Para realizar tal definición hay que analizar recursivamente el objeto `F`:

1. Si `F` es una expresión monaria, su argumento ha de ser una fórmula proposicional.
2. Si `F` es una expresión binaria, sus dos argumentos han de ser fórmulas proposicionales.
3. En otro caso `F` ha de ser una fórmula atómica.

5
```
(defun es-proposicional (F)
  (declare (xargs :measure (numero-conectivas F)))
  (cond ((es-monaria F)
         (es-proposicional (arg1 F)))
        ((es-binaria F)
         (and (es-proposicional (arg1 F))
              (es-proposicional (arg2 F))))
        (t (es-atomica F))))
```

Para que ACL2 admita esta función, se necesita una medida de los argumentos que decrezca en las llamadas recursivas. Una medida habitual sobre fórmulas es el número de conectivas que en ellas aparece. Esta medida decrece en las llamadas recursivas y, por tanto, puede ser utilizada para probar la terminación de funciones que hagan recursión sobre la estructura de una fórmula.

Definición 3.3 *El número de conectivas de una fórmula proposicional F, también llamado rango de F, viene dado por la función r:*

1. *Si F es atómica, $r(F) = 0$.*
2. *Si F es la fórmula monaria $\neg G$, $r(F) = 1 + r(G)$.*
3. *Si F es la fórmula binaria $(G_1 \circ G_2)$, $r(F) = 1 + r(G_1) + r(G_2)$, (donde $\circ$ es cualquier conectiva binaria).*

La función `numero-conectivas` presentada en 6 formaliza este concepto. Es importante notar que la definición se hace independientemente de que el argumento sea una fórmula proposicional. Así, esta definición puede realizarse antes de la definición de `es-proposicional` y, por tanto, puede ser utilizada en la prueba de terminación de esta última.

6

```
(defun numero-conectivas (F)
  (cond ((es-monaria F)
         (+ (numero-conectivas (arg1 F)) 1))
        ((es-binaria F)
         (+ (numero-conectivas (arg1 F))
            (numero-conectivas (arg2 F)) 1))
        (t 0)))
```

El siguiente resultado no es necesario para la prueba de terminación de la función `es-proposicional`, sin embargo simplifica mucho las pruebas cuando se usa `numero-conectivas` como medida de terminación:

Teorema 3.4 *El número de conectivas de una fórmula proposicional compuesta es mayor que el número de conectivas de sus subfórmulas componentes.*

La formulación en ACL2 de este teorema es más general puesto que hace referencia a expresiones monarias y binarias: el número de conectivas de una expresión monaria o binaria es mayor que el número de conectivas de sus argumentos. De esta forma, el resultado se puede establecer en ACL2 antes de la definición de `es-proposicional` y por tanto, también puede ser utilizado en la prueba de terminación de esta última.

7

```
(defthm numero-conectivas-monaria-decrece
  (implies (es-monaria F)
           (< (numero-conectivas (arg1 F))
              (numero-conectivas F))))

(defthm numero-conectivas-binaria-decrece
  (implies (es-binaria F)
           (and (< (numero-conectivas (arg1 F))
                   (numero-conectivas F))
                (< (numero-conectivas (arg2 F))
                   (numero-conectivas F)))))
```

Con estos resultados se demuestra fácilmente la terminación de la función `es-proposicional`.

3.1.4 Clasificación de las fórmulas

Una vez que está definida la función `es-proposicional`, podemos definir las funciones que sirven para clasificar las fórmulas proposicionales en función de su conectiva principal. Estas funciones comprueban que la expresión que se les pasa como argumento es una fórmula proposicional correctamente construida y que su conectiva principal es del tipo adecuado:

8

```
(defun es-negacion (F)
  (and (equal (op F) '-)
       (es-proposicional F)))

(defun es-conjuncion (F)
  (and (equal (op F) '&)
       (es-proposicional F)))

(defun es-disyuncion (F)
  (and (equal (op F) '\/)
       (es-proposicional F)))

(defun es-implicacion (F)
  (and (equal (op F) '->)
       (es-proposicional F)))

(defun es-equivalencia (F)
  (and (equal (op F) '<->)
       (es-proposicional F)))
```

Obviamente, estas funciones se comportan de la forma esperada cuando se utilizan sobre fórmulas construidas con las funciones `negacion`, `conjuncion`, `disyuncion`, `implicacion` y `equivalencia`, a partir de componentes proposicionales. Así, si `F` y `G` son fórmulas proposicionales entonces se verifican las propiedades `(es-negacion (negacion F))`, `(es-conjuncion (conjuncion F G))`, etc.

Utilizando estas funciones se puede proporcionar una nueva caracterización de las fórmulas proposicionales: `F` es una fórmula proposicional si y sólo si `F` es atómica, una negación, una conjunción, una disyunción, una implicación o una equivalencia:

9

```
(defthm otra-definicion-de-es-proposicional
  (iff (es-proposicional F)
       (or (es-atomica F)
           (es-negacion F)
           (es-conjuncion F)
           (es-disyuncion F)
           (es-implicacion F)
           (es-equivalencia F))))
```

3.1.5 Teorema de descomposición única

En la formalización de la lógica proposicional, un mismo objeto ACL2 no puede representar dos fórmulas distintas. Este resultado es conocido como el teorema de descomposición única:

Teorema 3.5 *Toda fórmula proposicional tiene una y sólo una de las siguientes formas:*

1. *Fórmula atómica.*

2. *$\neg F$, para una única fórmula proposicional F.*

3. *$(F \circ G)$, para una única conectiva binaria $\circ$, y unas únicas fórmulas proposicionales F y G.*

En nuestra formalización este teorema se expresa de la siguiente forma: Si una fórmula proposicional es de un tipo concreto (`es-atomica`, `es-negacion`, `es-conjuncion`, `es-disyuncion`, `es-implicacion` o `es-equivalencia`), entonces no puede ser de ningún otro tipo. Se trata de 6 resultados, uno para cada tipo de fórmula, de los cuales mostramos en *10* los relativos a fórmulas atómicas y conjunciones. Los restantes se establecen de forma similar.

10

```
(defthm unicidad-es-atomica
  (implies (es-atomica F)
           (and (not (es-negacion F))
                (not (es-conjuncion F))
                (not (es-disyuncion F))
                (not (es-implicacion F))
                (not (es-equivalencia F)))))

(defthm unicidad-es-conjuncion
  (implies (es-conjuncion F)
           (and (not (es-atomica F))
                (not (es-negacion F))
                (not (es-disyuncion F))
                (not (es-implicacion F))
                (not (es-equivalencia F)))))
```

Estos resultados junto con *9* nos permiten demostrar propiedades por distinción de casos: si una propiedad `q0` es cierta para toda fórmula atómica, negación, conjunción, disyunción, implicación o equivalencia entonces es cierta para toda fórmula proposicional.

3.1.6 Símbolos proposicionales de una fórmula

Terminamos esta sección definiendo el conjunto de símbolos proposicionales de una fórmula.

Definición 3.6 *Dada una fórmula proposicional $F \in \mathbb{P}(\Sigma)$, el conjunto de símbolos proposicionales de F, $Var(F)$, se define como sigue:*

1. *Si $F \in At(\Sigma)$, $Var(F) = \{F\}$.*
2. *Si $F = \neg G$, $Var(F) = Var(G)$.*
3. *Si $F = (G_1 \circ G_2)$, $Var(F) = Var(G_1) \cup Var(G_2)$ (donde $\circ$ es cualquier conectiva binaria).*

La construcción en ACL2 es recursiva sobre la estructura de la fórmula. Si la fórmula es monaria, el conjunto de sus símbolos coincidirá con el conjunto de símbolos de su argumento. Si la fórmula es binaria, el conjunto de sus símbolos será la unión de los símbolos de sus dos argumentos. Finalmente, si la fórmula es un símbolo proposicional, el resultado es la lista formada por dicho símbolo. En cualquier otro caso el resultado es vacío. El número de conectivas de la fórmula de partida, es la medida que garantiza que esta función recursiva siempre termina.

11

```
(defun simbolos-proposicionales (F)
  (declare (xargs :measure (numero-conectivas F)))
  (cond ((es-monaria F)
         (simbolos-proposicionales (arg1 F)))
        ((es-binaria F)
         (union-equal (simbolos-proposicionales (arg1 F))
                      (simbolos-proposicionales (arg2 F))))
        ((es-simbolo-proposicional F) (list F))
        (t nil)))
```

La función `union-equal` realiza la unión de los elementos de dos listas eliminando las repeticiones. De esta forma se asegura que el resultado es un conjunto.

Teorema 3.7 *Dada una fórmula cualquiera $F \in \mathbb{P}(\Sigma)$, se verifica:*

1. *Si F es $\neg G$, entonces $Var(G) \subseteq Var(F)$.*
2. *Si F es $(G_1 \circ G_2)$ entonces $Var(G_1) \subseteq Var(F)$ y $Var(G_2) \subseteq Var(F)$ (donde $\circ$ es cualquier conectiva binaria).*

La demostración de este resultado en ACL2 se basa en dos lemas previos, en los que se prueba que cualquier lista que contenga todos los símbolos proposicionales de una fórmula compuesta, también contiene los símbolos proposicionales de sus argumentos.

12

```
(defthm subsetp-equal-simbolos-proposicionales-monaria
  (implies
    (es-monaria F)
    (subsetp-equal (simbolos-proposicionales (arg1 F))
                   (simbolos-proposicionales F))))

(defthm subsetp-equal-simbolos-proposicionales-binaria
  (implies
    (es-binaria F)
    (and (subsetp-equal (simbolos-proposicionales (arg1 F))
                        (simbolos-proposicionales F))
         (subsetp-equal (simbolos-proposicionales (arg2 F))
                        (simbolos-proposicionales F)))))
```

3.2 Semántica clásica

Presentamos en esta sección la formalización de la semántica clásica de la lógica proposicional. Se discute la representación de las asignaciones y su utilización

para determinar el valor de verdad de una fórmula proposicional. También se definen los conceptos de satisfacibilidad y validez, y se indica cómo son utilizados en la formalización de teoremas en el sistema. Los eventos ACL2 que se presentan en esta sección se encuentran en el libro `semantica.lisp`.

3.2.1 Asignaciones

La lógica proposicional clásica es bivaluada. Notaremos como $\mathbb{B}$ al conjunto de los valores de verdad, $\mathbb{B} = \{\top, \bot\}$. Donde $\top$ y $\bot$ representan los valores cierto y falso. En la formalización utilizaremos las constantes `*V*` y `*F*` para representar los valores de verdad. El valor de `*V*` es `'(1)` y el valor de `*F*` es `'(0)`. Se han escogido estos valores para asegurar que el conjunto de símbolos proposicionales y el conjunto de valores de verdad son disjuntos, aunque esta propiedad no es necesaria para el desarrollo posterior. Definimos el predicado `es-valor-de-verdad` para comprobar si un objeto `v` pertenece al conjunto $\mathbb{B}$. Así, la expresión `(es-valor-de-verdad v)` será cierta si y sólo si se verifica la propiedad `v` $\in \mathbb{B}$:

13
```
(defun es-valor-de-verdad (v)
  (or (equal v *V*)
      (equal v *F*)))
```

Se podrían utilizar los valores de verdad del sistema, `nil` para falso y cualquier cosa distinta de `nil` para verdadero. Esta opción no ha sido considerada ya que podría dar lugar a confusiones entre los valores de verdad de la lógica y los del sistema. Con nuestra elección los valores de verdad de la lógica son independientes de los del sistema y además, es más fácil ampliar la representación cuando se aumenta el número de valores de verdad. También permite considerar otros valores de verdad, sin más que cambiar el valor de las constantes `*V*` y `*F*`.

Definición 3.8 *Una asignación sobre* Σ*, es una función* $\sigma : \Sigma \longrightarrow \mathbb{B}$*. Notaremos* $\mathbb{V}_\Sigma$ *al conjunto de todas las asignaciones sobre* Σ*.*

El valor de verdad de una fórmula proposicional se determina a partir de los valores de verdad de los símbolos proposicionales que en ella aparecen, utilizando un conjunto de funciones asociadas a las conectivas lógicas. Las asignaciones establecen un valor de verdad concreto para cada símbolo proposicional. Para representar las asignaciones utilizamos listas de asociación, en las que se relacionan los símbolos proposicionales con el valor de verdad asignado. Una restricción debida a esta representación es que únicamente podemos expresar asignaciones en las que sólo un número finito de símbolos proposicionales tienen un valor explícitamente asignado. Llamamos *asignaciones parciales* a estas asignaciones.

Definición 3.9 *Una asignación parcial sobre* Σ*, es una función parcial* $\sigma : \Sigma \longmapsto \mathbb{B}$*, definida sobre un conjunto finito de elementos.*

Así, la lista de asociación `( (p . *F*) (q . *F*) (r . *V*) (s . *V*) )` representa la asignación parcial σ, en la que $\sigma(p) = \sigma(q) = \bot$ y $\sigma(r) = \sigma(s) = \top$.

A la hora de utilizar una asignación parcial para determinar el valor de verdad de una fórmula, no siempre tendremos la seguridad de que todos los símbolos de la fórmula tengan un valor de verdad asociado en dicha asignación. Por ello, tendremos que extender la definición de las asignaciones parciales, considerando un valor de verdad por defecto para los símbolos sobre los que no están definidas:

Definición 3.10 *Dada una asignación parcial sobre Σ, $\sigma : \Sigma \longmapsto \mathbb{B}$, consideraremos la extensión a una función total definida de la siguiente manera: $\hat{\sigma} : \Sigma \longrightarrow \mathbb{B}$, tal que, dado $x \in \Sigma$*

$$\hat{\sigma}(x) = \begin{cases} \sigma(x) & \textit{si } \sigma \textit{ está definida sobre } x \\ \bot & \textit{en otro caso} \end{cases}$$

Por tanto, para calcular el valor de verdad de un símbolo en la extensión de una asignación parcial, basta con comprobar si dicho símbolo tiene asociado el valor `*V*` en la asignación parcial y, en caso contrario, su valor será `*F*`. Este valor es devuelto por la función `valor-de-simbolo` definida en *14*. En lo sucesivo, cuando hablemos de valor de verdad de un símbolo en una asignación parcial estaremos haciendo referencia al valor en la extensión de dicha asignación.

14
```
(defun valor-de-simbolo (x sigma)
  (cond ((es-simbolo-proposicional x)
         (let ((valor (cdr (assoc-equal x sigma))))
           (if (es-valor-de-verdad valor)
               valor
               *F*)))
        (t *F*)))
```

Hay que tener en cuenta que si un símbolo aparece en varias ocasiones en una asignación, el valor de verdad que se considera es el que proporciona la primera ocurrencia del mismo. Por ejemplo, el valor de `b` en la asignación parcial `( (a . *V*) (b . *F*) (b . *V*) )` es `*F*`. Además, si un símbolo no está asociado a un valor de verdad correcto, la asignación no se considera definida para dicho símbolo y por tanto su valor de verdad será `*F*`. Por ejemplo, el valor de `c` en la asignación parcial `( (a . *V*) (c . t) )` es `*F*`.

Definición 3.11 *Sea σ una asignación, x un símbolo proposicional y v un valor de verdad, $\sigma[x/v]$ es la asignación definida por:*

$$\sigma[x/v](y) = \begin{cases} v & \textit{si } y \textit{ es } x \\ \sigma(x) & \textit{en otro caso} \end{cases}$$

Este concepto se formaliza mediante la función `asume-valor`:

15
```
(defun asume-valor (x valor sigma)
  (acons x valor sigma))
```

3.2.2 Funciones de verdad

Las funciones de verdad definen el comportamiento semántico de las conectivas lógicas. Estas funciones determinan el valor de verdad de una fórmula compuesta a partir de los valores de verdad de sus componentes.

Definición 3.12 *Para cada conectiva proposicional,* $\neg$, $\wedge$, $\vee$, $\rightarrow$ *y* $\leftrightarrow$, *consideramos las funciones de verdad* $\mathcal{F}_\neg : \mathbb{B} \longrightarrow \mathbb{B}$, *y* $\mathcal{F}_\wedge, \mathcal{F}_\vee, \mathcal{F}_\rightarrow, \mathcal{F}_\leftrightarrow : \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$, *cuya definición viene dada en las siguientes tablas:*

X	$\mathcal{F}_\neg$
$\top$	$\bot$
$\bot$	$\top$

X	Y	$\mathcal{F}_\wedge$	$\mathcal{F}_\vee$	$\mathcal{F}_\rightarrow$	$\mathcal{F}_\leftrightarrow$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

La formalización de las funciones de verdad asociadas a las conectivas es la siguiente:

16
```
(defun funcion-de-verdad-de-negacion (i)
  (if (equal i *V*) *F* *V*))

(defun funcion-de-verdad-de-conjuncion (i j)
  (if (equal i *V*)
      (if (equal j *V*) *V* *F*)
      *F*))

(defun funcion-de-verdad-de-disyuncion (i j)
  (if (equal i *V*)
      *V*
      (if (equal j *V*) *V* *F*)))

(defun funcion-de-verdad-de-implicacion (i j)
  (if (equal i *V*)
      (if (equal j *V*) j *F*)
      *V*))

(defun funcion-de-verdad-de-equivalencia (i j)
  (if (equal i *V*)
      (if (equal j *V*) *V* *F*)
      (if (equal j *V*) *F* *V*)))
```

3.2.3 Valor de una fórmula en una asignación

Definición 3.13 *La valoración asociada a una asignación σ es la función $\tilde{\sigma} : \mathbb{P}(\Sigma) \longrightarrow \mathbb{B}$, tal que, dado $F \in \mathbb{P}(\Sigma)$:*

1. *Si F es atómica, $\tilde{\sigma}(F) = \sigma(F)$.*
2. *Si F es la fórmula monaria $\neg G$, $\tilde{\sigma}(F) = \mathcal{F}_{\neg}(\tilde{\sigma}(G))$.*
3. *Si F es la fórmula binaria $G_1 \circ G_2$, $\tilde{\sigma}(F) = \mathcal{F}_{\circ}(\tilde{\sigma}(G_1), \tilde{\sigma}(G_2))$, donde $\circ$ es cualquier conectiva binaria.*

En nuestra formalización definimos la función `valor`, mostrada en 17, que dados una fórmula proposicional y una asignación parcial, calcula el valor de verdad de la fórmula en la valoración asociada a la extensión de la asignación parcial. El número de conectivas de la fórmula de partida, es la medida que garantiza que esta función recursiva siempre termina. En lo sucesivo, cuando hablemos del valor de verdad de una fórmula en una asignación, nos estaremos refiriendo al valor de verdad de dicha fórmula en la valoración asociada.

17
```
(defun valor (F sigma)
  (declare (xargs :measure (numero-conectivas F)))
  (cond ((es-negacion F)
         (funcion-de-verdad-de-negacion
          (valor (arg1 F) sigma)))
        ((es-conjuncion F)
         (funcion-de-verdad-de-conjuncion
          (valor (arg1 F) sigma)
          (valor (arg2 F) sigma)))
        ((es-disyuncion F)
         (funcion-de-verdad-de-disyuncion
          (valor (arg1 F) sigma)
          (valor (arg2 F) sigma)))
        ((es-implicacion F)
         (funcion-de-verdad-de-implicacion
          (valor (arg1 F) sigma)
          (valor (arg2 F) sigma)))
        ((es-equivalencia F)
         (funcion-de-verdad-de-equivalencia
          (valor (arg1 F) sigma)
          (valor (arg2 F) sigma)))
        (t (valor-de-simbolo F sigma))))
```

Definición 3.14 *Una asignación σ es modelo de una fórmula F, $\sigma \models F$, si el valor de verdad de F en dicha asignación es verdadero, $\tilde{\sigma}(F) = \top$.*

Este concepto es incorporado a nuestra formalización de la siguiente forma:

18
```
(defun modelo (sigma F)
  (equal (valor F sigma) *V*))
```

Finalmente definimos el concepto de complementario, que relaciona una fórmula con otra cuyo valor de verdad es el contrario que el de la primera.

Definición 3.15 *El complementario de una fórmula F, que notaremos $\overline{F}$, se define de la siguiente forma: si $F = \neg G$ entonces $\overline{F} = G$, en otro caso $\overline{F} = \neg F$.*

La función que implementa este concepto es la siguiente:

19
```
(defun complementario (F)
  (if (es-negacion F)
      (arg1 F)
      (negacion F)))
```

La principal propiedad del complementario de una fórmula es la que relaciona su valor de verdad con el de la original:

Teorema 3.16 *Dada una fórmula F y una asignación σ, se tiene $\sigma \models \overline{F}$ si y sólo si $\sigma \models F$.*

El siguiente evento formaliza esta propiedad:

20
```
(defthm modelo-complementario
  (implies (es-proposicional F)
           (iff (modelo sigma (complementario F))
                (not (modelo sigma F)))))
```

3.2.4 Satisfacibilidad, validez y consecuencia lógica

Definición 3.17 *Una fórmula F es satisfacible si existe una asignación σ tal que $\sigma \models F$.*

La formalización de este concepto supone la búsqueda de una asignación modelo de la fórmula de partida. Dado que el conjunto de asignaciones es infinito (aún considerando únicamente las asignaciones parciales), no podemos definir una función que compruebe si existe una que sea modelo de una fórmula. En su lugar, cuando queramos utilizar como condición de un teorema el hecho de que

una fórmula es satisfacible, utilizaremos la expresión `(modelo SIGMA F)`, donde `SIGMA` es una variable libre nueva. Así, si `SAT` es una función que implementa un algoritmo para decidir la satisfacibilidad de una fórmula, el resultado de completitud para dicha función será:

```
(defthm completitud-SAT
  (implies (and (es-proposicional F)
                (modelo SIGMA F))
           (SAT F)))
```

Este evento se interpreta de la siguiente forma: "Si `F` es una fórmula proposicional para la que existe `SIGMA` modelo de `F`, entonces la función `SAT` tiene éxito al ser evaluada sobre `F`".

Cuando la afirmación sobre la satisfacibilidad de una fórmula aparece como conclusión de un resultado, tendremos que construir explícitamente una asignación modelo de dicha fórmula. Esto ocurre con el resultado de corrección asociado a la función `SAT`, donde se considera una función `MOD` que construye un modelo de `F`.

```
(defthm correccion-SAT
  (implies (and (es-proposicional F)
                (SAT F))
           (modelo (MOD F) F)))
```

Definición 3.18 *Una fórmula F es válida, y lo notaremos $\models F$, si para cualquier asignación σ se tiene que $\tilde{\sigma}(F) = \top$.*

En nuestra formalización tampoco podemos definir una función que implemente este concepto, ya que eso supondría trabajar con todas las asignaciones y éste es un conjunto infinito. En su lugar, cuando queramos utilizar como conclusión de un teorema el hecho de que una fórmula es válida, usaremos de nuevo la expresión `(modelo SIGMA F)`, donde `SIGMA` es una variable libre nueva. Así, si `DAT` es una función que implementa un algoritmo para decidir la validez de una fórmula, el teorema de corrección de dicha función es el siguiente:

```
(defthm correccion-DAT
  (implies (and (es-proposicional F)
                (DAT F))
           (modelo SIGMA F)))
```

Puesto que el sistema interpreta los teoremas cuantificando universalmente las variables libres, el evento anterior se interpreta de la siguiente forma: "Si la

función `DAT` tiene éxito al ser evaluada sobre una fórmula `F`, entonces para toda asignación `SIGMA`, `SIGMA` es modelo de `F`".

La formalización no es tan simple cuando la afirmación sobre la validez de una fórmula aparece entre las hipótesis del teorema. Este es el caso del teorema de completitud asociado a la función `DAT` que afirma lo siguiente:

$$(\forall \sigma : (\texttt{modelo}\ \sigma\ F)) \Rightarrow (\texttt{DAT}\ F)$$

Este resultado no se puede expresar tal cual en el sistema, debido a la cuantificación universal que hay en el antecedente. Normalmente, modificamos esta formulación hasta conseguir otra equivalente en la que no aparezcan cuantificaciones en el antecedente:

$$\begin{array}{rl} & (\forall \sigma : (\texttt{modelo}\ \sigma\ F)) \Rightarrow (\texttt{DAT}\ F) \quad \leftrightarrow \\ \leftrightarrow & \neg(\forall \sigma : (\texttt{modelo}\ \sigma\ F)) \vee (\texttt{DAT}\ F) \quad \leftrightarrow \\ \leftrightarrow & \exists \sigma, \neg(\texttt{modelo}\ \sigma\ F) \vee (\texttt{DAT}\ F) \quad \leftrightarrow \\ \leftrightarrow & \neg\neg(\texttt{DAT}\ F) \vee \exists \sigma, \neg(\texttt{modelo}\ \sigma\ F) \quad \leftrightarrow \\ \leftrightarrow & \neg(\texttt{DAT}\ F) \Rightarrow \exists \sigma, \neg(\texttt{modelo}\ \sigma\ F) \end{array}$$

Esta última expresión se puede formalizar como sigue:

```
(defthm completitud-DAT
  (implies (and (es-proposicional F)
                (not (DAT F)))
           (not (modelo (contramodelo-DAT F)))))
```

donde la función `contramodelo-DAT` proporciona una asignación que, en las condiciones del resultado, no es modelo de la fórmula `F`.

Definición 3.19 *Una fórmula F es consecuencia lógica de las fórmulas $H_1, \ldots, H_n$, y lo notaremos $H_1, \ldots, H_n \models F$, si para toda asignación σ que sea modelo de todas las fórmulas H_i, σ es modelo de F.*

De nuevo se trata de un concepto que no podemos definir en nuestra formalización, en su lugar utilizaremos expresiones relativas a la función `modelo`. Así, si `CONSEC` es una función que implementa un procedimiento para decidir si una fórmula `F` es consecuencia lógica de una lista de fórmulas `lista-H`, el teorema de corrección de dicha función es el siguiente:

```
(defthm correccion-CONSEC
  (implies (and (lista-formulas lista-H)
                (es-proposicional F)
                (CONSEC lista-H F)
                (modelo-lista SIGMA lista-H))
           (modelo SIGMA F)))
```

donde `lista-formulas` es una función que comprueba si su argumento es una lista de fórmulas proposicionales y `modelo-lista` es una función que comprueba si una asignación es modelo de todos los elementos de una lista.

Este evento se interpreta de la siguiente forma: "Si la función `CONSEC` tiene éxito al ser evaluada sobre la lista de fórmulas `lista-H` y la fórmula `F`, entonces para toda asignación `SIGMA` que sea modelo de todos los elementos de `lista-H`, `SIGMA` es modelo de `F`".

Para expresar el resultado de completitud asociado a la función `CONSEC`, se utiliza una formulación similar al del resultado de completitud de una función para decidir la validez: "Si la función `CONSEC` no tiene éxito al ser evaluada sobre una lista de fórmulas `lista-H` y una fórmula `F`, entonces existe una asignación que siendo modelo de todos los elementos de `lista-H`, no es modelo de `F`". La función `contramodelo-CONSEC` proporciona dicha asignación:

```
(defthm completitud-CONSEC
  (implies
    (and (lista-formulas lista-H)
         (es-proposicional F)
         (not (CONSEC lista-H F)))
    (and (modelo-lista (contramodelo-CONSEC lista-H F) lista-H)
         (not (modelo (contramodelo-CONSEC lista-H F) F)))))
```

Definición 3.20 *Dos fórmulas F_1 y F_2 son lógicamente equivalentes, y lo notaremos $F_1 \equiv F_2$, si tienen el mismo valor de verdad en cualquier asignación.*

Este concepto tampoco se puede definir en nuestra formalización, en su lugar usaremos la expresión: `(equal (valor F1 SIGMA) (valor F2 SIGMA))`. Así, si `FN` es un procedimiento para calcular cierto tipo de formas normales, el evento que afirma que dicho procedimiento mantiene la equivalencia lógica será el siguiente:

```
(defthm equivalencia-logica-FN
  (implies (es-proposicional F)
           (equal (valor (FN F) SIGMA)
                  (valor F SIGMA))))
```

3.3 Semántica de Kleene

En determinadas ocasiones se puede determinar si una fórmula es cierta o falsa, aún cuando se desconoce el valor de verdad de sus componentes. Por ejemplo, para afirmar que la fórmula $p \vee q$ es cierta en una asignación, basta con saber que una de sus componentes es cierta. Este tipo de situaciones es el que se formaliza con la semántica de Kleene, en la que se consideran los valores de verdad de la semántica clásica junto con un tercer valor que se interpreta como "indeterminado" o "desconocido".

En esta sección presentamos la formalización de una semántica de la lógica proposicional basada en la lógica trivalorada fuerte de Kleene [45]. Siguiendo un desarrollo similar al de la sección anterior se presentan los conceptos de $\mathbb{K}$-asignación, funciones de verdad de las conectivas, $\mathbb{K}$-satisfacibilidad, $\mathbb{K}$-validez, consecuencia $\mathbb{K}$-lógica y equivalencia $\mathbb{K}$-lógica. Los eventos ACL2 que se presentan en esta sección se encuentran en el libro `semantica-K.lisp`.

3.3.1 $\mathbb{K}$-asignaciones

La lógica trivalorada fuerte de Kleene utiliza los valores de verdad clásicos, $\top$ y $\bot$, interpretados como cierto y falso y un tercer valor, $\mathtt{I}$, que se interpreta como "indeterminado" o "desconocido". Así, si el valor de una fórmula en una asignación es $\mathtt{I}$, entonces no se tiene suficiente información para decidir si es verdadera o falsa. Teniendo en cuenta este hecho, llamaremos valores de verdad a los valores $\top$ y $\bot$ y valor indeterminado a $\mathtt{I}$. Notaremos como $\mathbb{K}$ al conjunto formado por los tres valores: $\mathbb{K} = \{\top, \bot, \mathtt{I}\} = \mathbb{B} \cup \{\mathtt{I}\}$.

Al igual que en la semántica clásica utilizamos las constantes `*V*` y `*F*` para representar los valores de verdad $\top$ y $\bot$. El valor indeterminado está representado por `nil`. De esta forma sólo habrá que modificar ligeramente los eventos que describen la semántica clásica para obtener los correspondientes de la semántica de Kleene.

Definición 3.21 *Una $\mathbb{K}$-asignación sobre Σ es una función $\sigma : \Sigma \longrightarrow \mathbb{K}$.*

Al igual que para la semántica clásica, utilizaremos listas de asociación para representar las $\mathbb{K}$-asignaciones. Con esta representación sólo se pueden expresar asignaciones en las que un número finito de símbolos tienen un valor explícitamente asignado, así que tendremos que extenderlas definiendo su comportamiento cuando el valor no esté proporcionado de forma explícita. El valor por defecto será $\mathtt{I}$. De esta forma, para representar una $\mathbb{K}$-asignación basta con indicar los símbolos que tienen asignado un valor distinto de $\mathtt{I}$, con la limitación de que ha de ser un número finito. Es decir, la representación de una $\mathbb{K}$-asignación será una asignación parcial (ver 3.9) extendida para asignar el valor $\mathtt{I}$ a todos los símbolos que no estén en su dominio.

Definición 3.22 *Dada una asignación parcial sobre* Σ, $\sigma : \Sigma \longmapsto \mathbb{B}$, *consideraremos la* $\mathbb{K}$*-asignación definida de la siguiente manera:* $\hat{\sigma}_{\mathbb{K}} : \Sigma \longrightarrow \mathbb{K}$, *tal que, dado* $x \in \Sigma$

$$\hat{\sigma}_{\mathbb{K}}(x) = \begin{cases} \sigma(x) & \textit{si } \sigma \textit{ está definida sobre } x \\ \bot & \textit{en otro caso} \end{cases}$$

Por tanto, para calcular el valor de un símbolo en la $\mathbb{K}$-asignación obtenida según la definición anterior, comprobamos si dicho símbolo tiene asociado un valor de verdad en la asignación parcial y, en caso contrario, su valor será `nil`. Una versión ligeramente modificada de la función `valor-de-simbolo` nos proporciona este valor:

21
```
(defun valor-de-simbolo-K (x sigma)
  (cond ((es-simbolo-proposicional x)
         (let ((valor (cdr (assoc-equal x sigma))))
           (if (es-valor-de-verdad valor)
               valor
               nil)))
        (t nil)))
```

Con respecto a esta función y la representación de las asignaciones se tienen las mismas características que en la semántica clásica. El valor de un símbolo `b` será el que proporciona el primer par `(b . v)` que aparezca en la asignación y si un símbolo no está asociado a un valor de verdad (`*V*` o `*F*`) correcto entonces el valor de dicho símbolo en la asignación es `nil`.

La forma de construir una nueva $\mathbb{K}$-asignación a partir de otra, cambiando el valor asignado para un símbolo, es igual que en la semántica clásica:

Definición 3.23 *Sea* σ *una* $\mathbb{K}$*-asignación,* x *un símbolo proposicional y* v *un valor de* $\mathbb{K}$, $\sigma[x/v]$ *es la asignación definida por:*

$$\sigma[x/v](y) = \begin{cases} v & \textit{si } y \textit{ es } x \\ \sigma(x) & \textit{en otro caso} \end{cases}$$

Este concepto se formaliza mediante la función `asume-valor` presentada en *15*.

3.3.2 Funciones de verdad en $\mathbb{K}$

La definición de las funciones de verdad de las conectivas lógicas en la semántica de Kleene, es una extensión de las funciones de verdad para la semántica clásica. En esta extensión, en determinadas ocasiones, es posible determinar un valor de verdad para una fórmula aún cuando los valores de verdad de alguna de sus componentes es indeterminado.

Definición 3.24 *Para cada conectiva proposicional,* $\neg$, $\wedge$, $\vee$, $\rightarrow$ *y* $\leftrightarrow$, *consideramos las funciones de verdad en* $\mathbb{K}$*:* $\mathcal{K}_\neg : \mathbb{K} \longrightarrow \mathbb{K}$, *y* $\mathcal{K}_\wedge, \mathcal{K}_\vee, \mathcal{K}_\rightarrow, \mathcal{K}_\leftrightarrow : \mathbb{K} \times \mathbb{K} \longrightarrow \mathbb{K}$, *cuya definición viene dada en las siguientes tablas:*

X	$\mathcal{K}_\neg$
$\top$	$\bot$
$\bot$	$\top$
I	I

X	Y	$\mathcal{K}_\wedge$	$\mathcal{K}_\vee$	$\mathcal{K}_\rightarrow$	$\mathcal{K}_\leftrightarrow$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	I	I	$\top$	I	I
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	I	$\bot$	I	$\top$	I
I	$\top$	I	$\top$	$\top$	I
I	$\bot$	$\bot$	I	I	I
I	I	I	I	I	I

La formalización de las funciones de verdad en la semántica de Kleene es la siguiente:

22

```
(defun funcion-de-verdad-de-negacion-K (i)
  (cond ((equal i *F*) *V*)
        ((equal i *V*) *F*)
        (t nil)))

(defun funcion-de-verdad-de-conjuncion-K (i j)
  (cond ((equal i *F*) *F*)
        ((equal i *V*) j)
        (t (cond ((equal j *F*) *F*)
                 (t nil)))))

(defun funcion-de-verdad-de-disyuncion-K (i j)
  (cond ((equal i *F*) j)
        ((equal i *V*) *V*)
        (t (cond ((equal j *V*) *V*)
                 (t nil)))))
```

22
```
(defun funcion-de-verdad-de-implicacion-K (i j)
  (cond ((equal i *F*) *V*)
        ((equal i *V*) j)
        (t (cond ((equal j *V*) *V*)
                 (t nil)))))

(defun funcion-de-verdad-de-equivalencia-K (i j)
  (cond ((equal i *F*) (cond ((equal j *F*) *V*)
                             ((equal j *V*) *F*)
                             (t nil)))
        ((equal i *V*) (cond ((equal j *F*) *F*)
                             ((equal j *V*) *V*)
                             (t nil)))
        (t nil)))
```

3.3.3 Valor de una fórmula en una $\mathbb{K}$-asignación

El valor de una fórmula en una $\mathbb{K}$-asignación se determina a partir de los valores de los símbolos proposicionales que en ella aparecen, utilizando las funciones de verdad en $\mathbb{K}$:

Definición 3.25 *La $\mathbb{K}$-valoración asociada a una $\mathbb{K}$-asignación σ es la función $\tilde{\sigma}_{\mathbb{K}} : \mathbb{P}(\Sigma) \longrightarrow \mathbb{K}$, tal que, dado $F \in \mathbb{P}(\Sigma)$:*

1. *Si F es atómica, $\tilde{\sigma}_{\mathbb{K}}(F) = \sigma(F)$.*
2. *Si F es la fórmula monaria $\neg G$, $\tilde{\sigma}_{\mathbb{K}}(F) = \mathcal{K}_{\neg}(\tilde{\sigma}_{\mathbb{K}}(G))$.*
3. *Si F es la fórmula binaria $G_1 \circ G_2$, $\tilde{\sigma}_{\mathbb{K}}(F) = \mathcal{K}_{\circ}(\tilde{\sigma}_{\mathbb{K}}(G_1), \tilde{\sigma}_{\mathbb{K}}(G_2))$, donde $\circ$ es cualquier conectiva binaria.*

Para calcular el valor de una fórmula en una $\mathbb{K}$-asignación utilizamos la función `valor-K`, cuya definición es igual a la correspondiente para la lógica proposicional, definida en 17, salvo que se utilizan las funciones de verdad en $\mathbb{K}$ presentadas en 22, y la función que determina el valor de un símbolo en una $\mathbb{K}$-asignación, definida en 21. En lo sucesivo, cuando hablemos del valor de una fórmula en una $\mathbb{K}$-asignación estaremos haciendo referencia al valor de dicha fórmula en la valoración asociada a la $\mathbb{K}$-asignación.

Definición 3.26 *Dada una $\mathbb{K}$-asignación σ y una fórmula F, diremos que*

1. *σ es modelo de F, $\sigma \models_{\mathbb{K}} F$, si el valor de F en σ es $\top$.*
2. *σ es no–modelo de F, $\sigma \not\models_{\mathbb{K}} F$, si el valor de F en σ es $\bot$.*

La formalización de estos conceptos es la siguiente:

23

```
(defun modelo-K (sigma F)
  (equal (valor-K F sigma) *V*))

(defun no-modelo-K (sigma F)
  (equal (valor-K F sigma) *F*))
```

En la semántica clásica es suficiente con definir el concepto de modelo, ya que el concepto de no–modelo es la negación del primero. No ocurre esto en la semántica de Kleene puesto que hay tres valores de verdad. Así, el concepto de contramodelo en la semántica clásica se corresponde con el de no–modelo en la semántica de Kleene.

3.3.4 Satisfacibilidad, validez y consecuencia lógica en $\mathbb{K}$

Definición 3.27 *Una fórmula F es satisfacible en $\mathbb{K}$ o $\mathbb{K}$-satisfacible, si existe una $\mathbb{K}$-asignación σ tal que $\sigma \models_{\mathbb{K}} F$.*

Al igual que ocurre en la semántica clásica, no podemos definir una función que implemente este concepto, ni ninguno de los de esta sección, pues eso supondría trabajar con todas las $\mathbb{K}$-asignaciones, y éste es un conjunto infinito. Analicemos pues cómo son incorporados estos conceptos a los teoremas en la formalización. Con respecto a la satisfacibilidad en $\mathbb{K}$, las observaciones son las mimas que en el caso de la semántica clásica. Así, los resultados de corrección y completitud para una función `SAT` que implementa un algoritmo para decidir la $\mathbb{K}$-satisfacibilidad de una fórmula son:

```
(defthm correccion-SAT
  (implies (and (es-proposicional F)
                (SAT F))
           (modelo-K (MOD F) F)))

(defthm completitud-SAT
  (implies (and (es-proposicional F)
                (modelo-K sigma F))
           (SAT F)))
```

donde `MOD` es una función que proporciona un modelo en $\mathbb{K}$ de la fórmula `F`.

Definición 3.28 *Una fórmula F es válida en $\mathbb{K}$ o $\mathbb{K}$-válida, si para toda $\mathbb{K}$-asignación σ, el valor de F en σ es $\top$ o $\bot$.*

Obsérvese que el concepto de fórmula $\mathbb{K}$-válida no se define de igual forma que en la semántica clásica. Esto se debe a que para cualquier fórmula F siempre existen $\mathbb{K}$-asignaciones que no son modelo en $\mathbb{K}$ de F. De hecho la $\mathbb{K}$-asignación definida de forma que el valor de cualquier símbolo es indeterminado, no es modelo en $\mathbb{K}$ de ninguna fórmula. Una formulación equivalente del concepto de $\mathbb{K}$-validez es la siguiente: una fórmula F es $\mathbb{K}$-válida si no existen $\mathbb{K}$-asignaciones que sean no–modelo de F. De hecho, la semántica de Kleene con este concepto de validez es homomórfica a la semántica clásica, es decir, tiene el mismo conjunto de fórmulas válidas ([9] pg 74).

Si `DAT` es una función que implementa un algoritmo para decidir la $\mathbb{K}$-validez de una fórmula, entonces los resultados de corrección y completitud para dicha función son:

```
(defthm correccion-DAT
  (implies (and (es-proposicional F)
                (DAT F)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-DAT
  (implies (and (es-proposicional F)
                (not (DAT F)))
           (no-modelo-K (contramodelo-DAT F))))
```

En el resultado de corrección se ha añadido la hipótesis `(valor-K F sigma)` para asegurar que la $\mathbb{K}$-asignación `sigma` no deja indeterminada a la fórmula `F`. De esta forma, si se verifica `(DAT F)` entonces, para toda $\mathbb{K}$-asignación `sigma`, el valor de `F` en `sigma` es `nil` o `*V*`. La función `contramodelo-DAT` del teorema de completitud proporciona una $\mathbb{K}$-asignación en la que F es falsa, de ahí que se utilice la función `no-modelo-K` en la conclusión.

Definición 3.29 *Una fórmula F es consecuencia lógica en $\mathbb{K}$ o consecuencia $\mathbb{K}$-lógica de las fórmulas $H_1, \ldots, H_n$, y lo notaremos $H_1, \ldots, H_n \models_{\mathbb{K}} F$, si para toda $\mathbb{K}$-asignación σ que sea modelo en $\mathbb{K}$ de todas las fórmulas H_i, el valor de F en σ es $\top$ o $\bot$.*

Al igual que con el concepto de $\mathbb{K}$-validez, el de consecuencia $\mathbb{K}$-lógica no se define como el correspondiente para la semántica clásica. La razón es la misma, en determinados casos no se puede afirmar que toda $\mathbb{K}$-asignación que sea modelo en $\mathbb{K}$ de las hipótesis, hace cierta la conclusión. Por ejemplo, esto ocurre cuando la conclusión es una fórmula $\mathbb{K}$-válida y no hay hipótesis. Si `CONSEC` es una función que implementa un procedimiento para decidir si una fórmula `F` es consecuencia lógica en $\mathbb{K}$ de una lista de fórmulas `lista-H`, los resultados de corrección y completitud para `CONSEC` son:

```
(defthm correccion-CONSEC
  (implies (and (lista-formulas lista-H)
                (es-proposicional F)
                (CONSEC lista-H F)
                (modelo-lista-K sigma lista-H)
                (valor-K F sigma))
           (modelo-K SIGMA F)))

(defthm completitud-CONSEC
  (implies
    (and (lista-formulas lista-H)
         (es-proposicional F)
         (not (CONSEC lista-H F)))
    (and (modelo-lista-K (contramodelo-CONSEC lista-H F) lista-H)
         (no-modelo-K (contramodelo-CONSEC lista-H F) F))))
```

En el resultado de corrección se ha añadido la hipótesis `(valor-K F sigma)` para asegurar que la $\mathbb{K}$-asignación `sigma` no deja la fórmula `F` indeterminada. La función `contramodelo-CONSEC` del teorema de completitud, proporciona una $\mathbb{K}$-asignación que es modelo en $\mathbb{K}$ de todas las fórmulas de `lista-H`, pero hace falsa a `F`. En ambos eventos se utiliza la función `lista-formulas`, que comprueba si su argumento es una lista fórmulas proposicionales, y `modelo-lista-K`, que comprueba si una $\mathbb{K}$-asignación es modelo en $\mathbb{K}$ de todas las fórmulas de una lista.

Definición 3.30 *Dos fórmulas F_1 y F_2 son lógicamente equivalentes en $\mathbb{K}$, y lo notaremos $F_1 \equiv_{\mathbb{K}} F_2$, si tienen el mismo valor en cualquier $\mathbb{K}$-asignación.*

Si `FN` es un procedimiento para calcular cierto tipo de formas normales, el evento que afirma que dicho procedimiento mantiene la equivalencia lógica en $\mathbb{K}$ es el siguiente:

```
(defthm equivalencia-logica-FN
  (implies (es-proposicional F)
           (equal (valor-K (FN F) SIGMA)
                  (valor-K F SIGMA))))
```

3.4 Tablas de verdad

En esta sección se definen funciones que implementan algoritmos para decidir la satisfacibilidad y la validez de una fórmula proposicional, basados en tablas de verdad. Con las tablas de verdad se analiza el valor de verdad de una fórmula, con respecto a las asignaciones parciales que están definidas únicamente para los

símbolos que en ella aparecen. A estas asignaciones las llamaremos asignaciones relevantes. Una fórmula será satisfacible si y sólo si una de sus asignaciones relevantes es modelo de ella, y será válida si y sólo si todas sus asignaciones relevantes son modelo de ella. Estos resultados son formalizados y demostrados en el sistema. Las definiciones y teoremas que se presentan en las tres primeras subsecciones se encuentran en el libro `tablas-de-verdad.lisp`, el desarrollo del método de las tablas de verdad en la semántica de Kleene se encuentra en el libro `tablas-de-verdad-K.lisp`.

3.4.1 Asignaciones relevantes de una fórmula

Definición 3.31 *Dado un conjunto finito de símbolos, $V \subseteq \Sigma$, la familia de sus asignaciones parciales es* $\mathcal{A}(V) = \{\sigma : V \longrightarrow \mathbb{B}\}$

Para construir la familia de las asignaciones parciales de una lista de símbolos, procedemos recursivamente sobre dicha lista. En cada paso, añadimos las ligaduras de un nuevo símbolo con un valor de verdad, a la familia de asignaciones conseguida en el paso anterior.

24
```
(defun asignaciones (Xs)
  (if (endp Xs)
      (list nil)
      (let ((sigma-lst (asignaciones (cdr Xs))))
        (append (asume-valor-lista (car Xs) *F* sigma-lst)
                (asume-valor-lista (car Xs) *V* sigma-lst)))))

(defun asume-valor-lista (x v sigma-lst)
  (if (endp sigma-lst)
      nil
      (cons (asume-valor x v (car sigma-lst))
            (asume-valor-lista x v (cdr sigma-lst)))))
```

Definición 3.32 *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, la familia de sus asignaciones relevantes es* $\mathcal{A}_R(F) = \mathcal{A}(Var(F))$

La familia de las asignaciones relevantes de una fórmula se construye a partir de la lista de símbolos de dicha fórmula. Para definir este concepto utilizamos la función `asignaciones` sobre la lista de símbolos de la fórmula obtenida con la función `simbolos-proposicionales`, definida en *11*.

25
```
(defun asignaciones-formula (F)
  (asignaciones (simbolos-proposicionales F)))
```

Definición 3.33 *Dado un conjunto de símbolos, $V \subseteq \Sigma$, y una asignación σ, la asignación relevante de V inducida por σ es la función $\sigma_V : V \longrightarrow \mathbb{B}$ tal que $\sigma_V(x) = \sigma(x)$ para todo $x \in V$.*

La función `asignacion-inducida` construye la asignación relevante de un conjunto de símbolos inducida por una asignación:

26
```
(defun asignacion-inducida (Xs sigma)
  (if (endp Xs)
      nil
      (asume-valor (car Xs)
                   (valor-de-simbolo (car Xs) sigma)
                   (asignacion-inducida (cdr Xs) sigma))))
```

El siguiente teorema establece un par de propiedades importantes de la asignación inducida:

Teorema 3.34 *Dado un conjunto de símbolos, $V \subseteq \Sigma$, y una asignación σ, se verifican:*

1. *$\sigma_V \in \mathcal{A}(V)$.*

2. *Para toda fórmula F tal que $Var(F) \subseteq V$ se verifica $\tilde{\sigma}(F) = \tilde{\sigma}_V(F)$.*

La primera parte de este resultado se obtiene fácilmente a partir de la definición de σ_V. La segunda parte se debe a que el valor de verdad de una fórmula se determina a partir de los valores de verdad de los símbolos que en ella aparecen, utilizando las funciones de verdad asociadas a las conectivas. Como $Var(F) \subseteq V$ entonces σ y σ_V asignan el mismo valor de verdad a todos los símbolos de F y por tanto, el valor de verdad de F en ambas asignaciones es el mismo.

Los siguientes eventos formalizan el teorema anterior:

27
```
(defthm asignacion-inducida-es-asignacion
  (member-equal (asignacion-inducida Xs sigma)
                (asignaciones Xs)))

(defthm asignacion-inducida-equivale-a-la-original
  (implies (subsetp-equal (simbolos-proposicionales F) Xs)
           (equal (valor F (asignacion-inducida Xs sigma))
                  (valor F sigma))))
```

3.4.2 Decisión de validez basada en tablas de verdad

La validez de una fórmula se puede decidir comprobando que todas sus asignaciones relevantes son modelo de ella. Si se encuentra una en la que la fórmula sea falsa, entonces se puede afirmar que la fórmula no es válida.

Definición 3.35 *Una fórmula F es válida en una familia de asignaciones $\mathcal{A}$ si para toda $\sigma \in \mathcal{A}$ se tiene $\sigma \models F$.*

Definimos este concepto para familias finitas de asignaciones:

28
```
(defun valida-en-asignaciones (F sigma-lst)
  (if (endp sigma-lst)
      t
      (and (modelo (car sigma-lst) F)
           (valida-en-asignaciones F (cdr sigma-lst)))))
```

Algoritmo 3.36 (*$TAUT_{tv}$*) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $TAUT_{tv}(F)$ es* **t** *si y sólo si F es válida en la familia de asignaciones $\mathcal{A}_R(F)$. En caso contrario el valor de $TAUT_{tv}(F)$ es* **f**.

Este algoritmo está implementado por la siguiente función:

29
```
(defun TAUT-tablas-de-verdad (F)
  (valida-en-asignaciones F (asignaciones-formula F)))
```

Obsérvese que, tanto en la descripción del algoritmo como en los enunciados de las propiedades relacionadas con él, utilizamos los símbolos **t** y **f** para representar, respectivamente, los valores cierto y falso del sistema en el que se implementa dicho algoritmo. En nuestra formalización **t** representa el valor `t` y **f** el valor `nil`. Esta notación es utilizada en los restantes algoritmos que se presentan en esta memoria.

3.4.2.1 Corrección del algoritmo

Teorema 3.37 *Dada una fórmula F, si $TAUT_{tv}(F) = \mathbf{t}$ entonces F es válida.*

Demostración:
Sean una fórmula F tal que $TAUT_{tv}(F) = \mathbf{t}$ y una asignación σ. Por el teorema 3.34 se tiene $\sigma_{Var(F)} \in \mathcal{A}(Var(F)) = \mathcal{A}_R(F)$, luego $\tilde{\sigma}_{Var(F)}(F) = \top$ pues $TAUT_{tv}(F) = \mathbf{t}$. Por otro lado el teorema 3.34 también afirma que $\tilde{\sigma}(F) = \tilde{\sigma}_{Var(F)}(F)$, luego $\sigma \models F$.

Por tanto, para cualquier asignación σ se tiene $\sigma \models F$ y de aquí que F sea válida.

□

Este resultado se formaliza como se indicó en la sección 3.2.4:

30
```
(defthm correccion-TAUT-tablas-de-verdad
  (implies (and (es-proposicional F)
                (TAUT-tablas-de-verdad F))
           (modelo sigma F)))
```

La prueba de este resultado se obtiene a partir de los eventos establecidos en *27*.

3.4.2.2 Completitud del algoritmo

Teorema 3.38 *Dada una fórmula F, si $TAUT_{tv}(F) = \mathbf{f}$ entonces F no es válida.*

Demostración:
Sea F una fórmula tal que $TAUT_{tv}(F) = \mathbf{f}$ entonces existe una asignación $\sigma \in \mathcal{A}_R(F)$ tal que $\sigma \not\models F$. Luego F no es válida.

□

La demostración del teorema consiste en encontrar una asignación relevante de la fórmula `F` que no sea modelo de ella. Esto es lo que hace la función `NO-MOD-tablas-de-verdad`. Para obtener dicha asignación se utiliza la función `primer-contramodelo` que devuelve la primera asignación de una lista que no es modelo de una fórmula:

31
```
(defun NO-MOD-tablas-de-verdad (F)
  (primer-contramodelo F (asignaciones-formula F)))

(defun primer-contramodelo (F sigma-lst)
  (if (endp sigma-lst)
      nil
      (if (not (modelo (car sigma-lst) F))
          (car sigma-lst)
          (primer-contramodelo F (cdr sigma-lst)))))
```

El teorema de completitud se formaliza siguiendo las indicaciones de la sección 3.2.4:

32
```
(defthm completitud-TAUT-tablas-de-verdad
  (implies (and (es-proposicional F)
                (not (TAUT-tablas-de-verdad F)))
           (not (modelo (NO-MOD-tablas-de-verdad F) F))))
```

3.4.3 Decisión de satisfacibilidad basada en tablas de verdad

El algoritmo que decide la satisfacibilidad de una fórmula basado en las tablas de verdad, calcula el valor de verdad de la fórmula en todas y cada una de sus asignaciones relevantes. Si encuentra una en la que la fórmula sea cierta, entonces se puede afirmar que la fórmula es satisfacible.

Definición 3.39 *Una fórmula F es satisfacible en una familia de asignaciones $\mathcal{A}$ si $\exists \sigma \in \mathcal{A}, \sigma \models F$.*

Este concepto se define fácilmente para familias finitas de asignaciones:

33
```
(defun satisfacible-en-asignaciones (F sigma-lst)
  (if (endp sigma-lst)
      nil
      (or (modelo (car sigma-lst) F)
          (satisfacible-en-asignaciones F (cdr sigma-lst)))))
```

Algoritmo 3.40 (SAT_{tv}) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $SAT_{tv}(F)$ es* **t** *si y sólo si F es satisfacible en la familia de asignaciones $\mathcal{A}_R(F)$. En caso contrario el valor de $SAT_{tv}(F)$ es* **f**.

Este algoritmo está implementado por la siguiente función:

34
```
(defun SAT-tablas-de-verdad (F)
  (satisfacible-en-asignaciones F (asignaciones-formula F)))
```

3.4.3.1 Corrección del algoritmo

Teorema 3.41 *Dada una fórmula F, si $SAT_{tv}(F) =$* **t** *entonces F es satisfacible.*

El resultado de corrección de la función que implementa el algoritmo tiene la siguiente forma (ver 3.2.4):

35
```
(defthm correccion-SAT-tablas-de-verdad
  (implies (and (es-satisfacible F)
                (SAT-tablas-de-verdad F))
           (modelo (MOD-tablas-de-verdad F) F)))
```

Donde `MOD-tablas-de-verdad` proporciona la primera asignación del conjunto $\mathcal{A}_R(F)$ que es modelo de F. La función `MOD-tablas-de-verdad` se ha definido a partir de la función `primer-modelo`, que devuelve la primera asignación de una lista que es modelo de una fórmula:

36
```
(defun MOD-tablas-de-verdad (F)
  (primer-modelo F (asignaciones-formula F)))

(defun primer-modelo (F sigma-lst)
  (if (endp sigma-lst)
      nil
      (if (modelo (car sigma-lst) F)
          (car sigma-lst)
          (primer-modelo F (cdr sigma-lst)))))
```

3.4.3.2 Completitud del algoritmo

Teorema 3.42 *Dada una fórmula F, si F es satisfacible entonces $SAT_{tv}(F) = \mathbf{t}$.*

Demostración:

Puesto que F es satisfacible, existe una asignación σ tal que $\sigma \models F$. Por el teorema 3.34 se tienen $\sigma_{Var(F)} \in \mathcal{A}(Var(F)) = \mathcal{A}_R(F)$ y $\tilde{\sigma}_{Var(F)}(F) = \tilde{\sigma}(F)$. Luego F es satisfacible en la familia de asignaciones $\mathcal{A}_R(F)$, y de aquí $SAT_{tv}(F) = \mathbf{t}$. □

El resultado de completitud de la función que implementa el algoritmo tiene la siguiente forma (ver 3.2.4):

37
```
(defthm completitud-SAT-tablas-de-verdad
  (implies (and (es-proposicional F)
                (modelo sigma F))
           (SAT-tablas-de-verdad F)))
```

La prueba de este resultado se obtiene a partir de los eventos establecidos en *27*.

3.4.4 Tablas de verdad en $\mathbb{K}$

El método de las tablas de verdad en la semántica de Kleene se basa en la misma idea que en la semántica clásica: basta con analizar el valor de una fórmula en sus asignaciones relevantes para determinar su validez o su satisfacibilidad en $\mathbb{K}$.

Para la formalización de este método también se considera la familia de las asignaciones relevantes de una fórmula tal y como se definió en 3.32. Estas asignaciones no dejan indeterminada a la fórmula a partir de la que se construyen.

Teorema 3.43 *Dada una fórmula F y una asignación parcial $\sigma \in \mathcal{A}_R(F)$ entonces $\tilde{\sigma}_{\mathbb{K}}(F) \neq \mathbb{I}$.*

La prueba de este teorema se debe a que las asignaciones relevantes asociadas a una fórmula, asignan valores de verdad a todos los símbolos que en ella aparecen y las funciones de verdad de las conectivas, no generan indeterminaciones a partir de valores de verdad. La formalización es la siguiente:

38
```
(defthm asignaciones-asigna-valor-K
  (implies (and (es-proposicional F)
                (member-equal sigma (asignaciones-formula F)))
           (valor-K F sigma)))
```

Definición 3.44 *Dado un conjunto de símbolos, $V \subseteq \Sigma$ y una $\mathbb{K}$-asignación σ, la asignación relevante de V inducida por σ es la función $\sigma_V : V \longrightarrow \mathbb{B}$ tal que:*

$$\sigma_V(x) = \begin{cases} \bot & si\ \sigma(x) = \perp\!\!\!\perp \\ \sigma(x) & en\ otro\ caso \end{cases}$$

La principal diferencia con el concepto de asignación inducida en la semántica clásica, es que en este caso la $\mathbb{K}$-asignación puede no asignar un valor de verdad a todos los elementos de V. Para construir una asignación parcial definida en V se asigna el valor $\bot$ a todos los símbolos que la $\mathbb{K}$-asignación deja indeterminados. La función `asigna-inducida-K` construye esta asignación relevante:

39
```
(defun asignacion-inducida-K (Xs sigma)
  (if (endp Xs)
      nil
      (if (valor-de-simbolo (car Xs) sigma)
          (asume-valor (car Xs)
                       (valor-de-simbolo-K (car Xs) sigma)
                       (asignacion-inducida-K (cdr Xs) sigma))
          (asume-valor (car Xs) *F*
                       (asignacion-inducida-K (cdr Xs) sigma)))))
```

El siguiente teorema establece las principales propiedades de la asignación inducida:

Teorema 3.45 *Dado un conjunto de símbolos, $V \subseteq \Sigma$, y una $\mathbb{K}$-asignación σ, se verifican:*

1. *$\sigma_V \in \mathcal{A}(V)$.*
2. *Dada una fórmula F tal que $Var(F) \subseteq V$ y $\tilde{\sigma}_{\mathbb{K}}(F) \neq \perp\!\!\!\perp$, entonces F tiene el mismo valor en σ y en σ_V.*

La demostración de este teorema se obtiene de forma similar a la del teorema 3.34 para la semántica clásica. Su formalización es la siguiente:

40
```
(defthm asignacion-inducida-K-es-asignacion
  (member-equal (asignacion-inducida-K Xs sigma)
                (asignaciones Xs)))

(defthm asignacion-inducida-K-equivale-a-la-original
  (implies (and (subsetp-equal (simbolos-proposicionales F) Xs)
                (valor-K F sigma))
           (equal (valor-K F (asignacion-inducida-K Xs sigma))
                  (valor-K F sigma))))
```

3.4.4.1 Decisión de $\mathbb{K}$-validez basada en tablas de verdad

Al igual que en el caso de la semántica clásica, la $\mathbb{K}$-validez de una fórmula se determina comprobando que dicha fórmula es cierta en todas sus asignaciones relevantes. La función que implementa el algoritmo de decisión de $\mathbb{K}$-validez es la siguiente:

41
```
(defun TAUT-tablas-de-verdad-K (F)
  (valida-en-asignaciones-K F (asignaciones-formula F)))
```

donde `valida-en-asignaciones-K` es una función que comprueba si todas las $\mathbb{K}$-asignaciones de una lista son modelo de una fórmula. Su definición es similar a la de `valida-en-asignaciones` presentada en *28* salvo que se utiliza la función `modelo-K` para comprobar que las $\mathbb{K}$-asignaciones son modelo de la fórmula.

Los resultados de corrección y completitud para esta función tienen la forma indicada en la sección 3.3.4:

42
```
(defthm correccion-TAUT-tablas-de-verdad-K
  (implies (and (es-proposicional F)
                (TAUT-tablas-de-verdad-K F)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-TAUT-tablas-de-verdad-K
  (implies (and (es-proposicional F)
                (not (TAUT-tablas-de-verdad-K F)))
           (no-modelo-K (NO-MOD-tablas-de-verdad-K F) F)))
```

donde la función `NO-MOD-tablas-de-verdad-K` devuelve la primera asignación relevante de una fórmula que es no–modelo de ella.

3.4.4.2 Decisión de $\mathbb{K}$-satisfacibilidad basada en tablas de verdad

La $\mathbb{K}$-satisfacibilidad de una fórmula se determina comprobando que dicha fórmula es cierta en alguna de sus asignaciones relevantes. La función que implementa el algoritmo de decisión de $\mathbb{K}$-satisfacibilidad es la siguiente:

43
```
(defun SAT-tablas-de-verdad-K (F)
  (satisfacible-en-asignaciones-K F (asignaciones-formula F)))
```

donde la función `satisfacible-en-asignaciones-K` comprueba si una fórmula es cierta en alguna de las $\mathbb{K}$-asignaciones de una lista. La definición de esta función es similar a la de `satisfacible-en-asignaciones` presentada en *33* salvo que se utiliza la función `modelo-K` en lugar de `modelo`.

Los resultados de corrección y completitud para esta función son los siguientes:

44
```
(defthm correccion-SAT-tablas-de-verdad-K
  (implies (and (es-proposicional F)
                (SAT-tablas-de-verdad-K F))
           (modelo-K (MOD-tablas-de-verdad-K F) F)))

(defthm completitud-SAT-tablas-de-verdad-K
  (implies (and (es-proposicional F)
                (modelo-K sigma F))
           (SAT-tablas-de-verdad-K F)))
```

donde la función `MOD-tablas-de-verdad-K` devuelve la primera asignación relevante de una fórmula que es modelo de ella.

3.5 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, evaluándolos sobre las 17 primeras fórmulas de Pelletier [63]. Antes de realizar cualquier evaluación, conviene compilar los ficheros que contienen el código. De esta forma los tiempos de respuesta obtenidos son menores. Una forma de hacer esto es certificando los libros ACL2. Una vez hecho esto, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `3-logica`:

```
../calculos-proposicionales/3-logica> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/3-logica".

ACL2 !>
```

A continación incluimos el libro que contiene los procedimientos de decisión basados en tablas de verdad. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "tablas-de-verdad")
```

La familia de ejemplos que presentamos está formada por las 17 primeras fórmulas de Pelletier [63]. Esta familia está definida en el libro `pelletier.lisp` del directorio `0-ejemplos`. Cada una de las fórmulas está almacenada en una constante `*pelletier-`n`*`, para los valores de n de `1` a `17`:

```
ACL2 !>(include-book "../0-ejemplos/pelletier")
...
ACL2 !>*pelletier-9*
(|->| (& (& (/ P Q) (/ (- P) Q)) (/ P (- Q)))
      (- (/ (- P) (- Q))))
```

Se puede comprobar que cada una de estas fórmulas es válida evaluando sobre ellas la función `TAUT-tablas-de-verdad`:

```
ACL2 !>(TAUT-tablas-de-verdad *pelletier-9*)
T
```

Para evaluar todas las fórmulas de la familia, obteniendo información sobre el tiempo de evaluación, hemos definido la función `evalua-ejemplos`. Esta función no puede ser utilizada en ACL2 puesto que no es completamente aplicativa. Para utilizarla, usamos el comando `(value :q)` para salir del entorno ACL2 y pasar al Common Lisp en el que está desarrollado[1], y cargamos el archivo `ejemplos.lisp` que se encuentra en el directorio `0-ejemplos`. La función `evalua-ejemplos` recibe como primer argumento el nombre del procedimiento de decisición que se quiere utilizar, y como segundo argumento una lista en la que están almacenados los ejemplos que se quieren evaluar. En este caso hemos definido la constante `*pelletier*` que contiene dicha lista:

[1]Se puede volver al entorno ACL2 evaluando `(lp)`.

```
ACL2 !>(value :q)
Exiting the ACL2 read-eval-print loop. To re-enter, execute (LP).
ACL2>(load "../0-ejemplos/ejemplos.lisp")

Loading ../0-ejemplos/ejemplos.lisp
Finished loading ../0-ejemplos/ejemplos.lisp
T

ACL2>(evalua-ejemplos 'TAUT-tablas-de-verdad *pelletier*)

 +=============================================+============+
 | Fórmula                                     | Tiempo     |
 +---------------------------------------------+------------+
 |  1) (p -> q) <-> (-q -> -p)                 |      0.000 |
 |  2) (-(-p) <-> p)                           |      0.000 |
 |  3) -(p -> q) -> (q -> p)                   |      0.000 |
 |  4) (-p -> q) <-> (-q -> p)                 |      0.000 |
 |  5) ((p | q) -> (p | r)) -> (p | (q -> r))  |      0.000 |
 |  6) (p | -p)                                 |      0.000 |
 |  7) (p | -(-(-p)))                          |      0.000 |
 |  8) ((p -> q) -> p) -> p                    |      0.000 |
 | ....                                        |      ..... |
 +=============================================+============+
T
```

Estas fórmulas son muy simples y el tiempo de evaluación es prácticamente nulo para todas ellas.

Sumario

En este capítulo:

- Se han visto cuestiones relativas a la representación de las fórmulas. Esta representación se ha caracterizado mediante un conjunto de propiedades, en los que se basa el desarrollo posterior. La representación puede cambiarse siempre que se mantengan dichas propiedades
- Se ha discutido la representación de las asignaciones y cómo se utilizan para determinar el valor de verdad de una fórmula. Se ha indicado como se incorporan los conceptos de satisfacibilidad, validez, consecuencia lógica y equivalencia lógica en la formalización de los teoremas. Esto se ha hecho tanto para la semántica clásica como para una semántica basada en la lógica trivalorada fuerte de Kleene.

- Se han desarrollado algoritmos para decidir la validez y la satisfacibilidad de una fórmula proposicional basados en tablas de verdad. Se han demostrado la corrección y completitud de estos algoritmos. Se ha analizado como estos algoritmos también sirven para decidir la validez y la satisfacibilidad en la semántica de Kleene.

Capítulo 4

Cálculo de tableros semánticos

Los sistemas de razonamiento basados en tableros semánticos han ganado atención en la última década. Se han utilizado para construir demostradores tales como SETHEO [60] o leanTAP [6], que pueden competir con sistemas actuales basados en resolución, [80]. También son adecuados para cooperar con demostradores interactivos usados en verificación de software, [1]. Otra razón de su auge es la necesidad de desarrollar métodos de deducción en lógicas no clásicas para las que los cálculos basados en tableros son particularmente adecuados, [34].

En este capítulo presentamos el método de los tableros semánticos para la lógica proposicional. Una descripción más detallada de este método, analizando sus variantes y la relación con otros métodos como el procedimiento de Davis y Putnam, el cálculo KE o el algoritmo de Stålmarck pueden encontrarse en [35] o [18].

El capítulo está dividido en dos secciones. En la primera se presenta la notación uniforme para la lógica proposicional. Usando esta notación se expresan las reglas del cálculo de tableros de una forma más concisa. En esta sección también se establecen los eventos relacionados con su formalización. La notación uniforme se puede utilizar tanto en la semántica clásica como en la semántica de Kleene. Al final de esta sección se comentan las diferencias entre las dos versiones.

En la segunda sección se presenta el método de los tableros semánticos. Basándonos en esta presentación, se formaliza un algoritmo para demostrar la satisfacibilidad de una fórmula, tanto en la semántica clásica como en la semántica de Kleene. Se presentan los resultados de corrección y completitud de este algoritmo y los eventos que los establecen. Un desarrollo similar se realiza con un algoritmo para decidir la validez de una fórmula y otro para la consecuencia lógica.

4.1 Notación uniforme

La notación uniforme fue desarrollada independientemente por Smullyan [78] y Lis [49]. Esta notación sirve para clasificar las fórmulas proposicionales en cua-

tro categorías distintas: las que se comportan de forma conjuntiva, las que se comportan de forma disyuntiva, las dobles negaciones y los literales. Al tener sólo cuatro situaciones distintas, las definiciones de determinados conceptos se simplifican al considerar la notación uniforme. Este es el caso del verificador de satisfacibilidad basado en tableros semánticos presentado en la sección 4.2. También se presenta una medida asociada a la notación uniforme, que será útil para demostrar la terminación de funciones recursivas basadas en esta notación. Los eventos que se presentan en las dos primeras subsecciones se encuentran en el libro `uniforme.lisp`. La versión para la semántica de Kleene comentada en la última subsección se encuentra en el libro `uniforme-K.lisp`.

4.1.1 Clasificación basada en la notación uniforme

Agruparemos todas las fórmulas proposicionales de la forma $(G_1 \circ G_2)$ y $\neg(G_1 \circ G_2)$, en dos categorías, las que actúan de manera conjuntiva, a las que llamaremos *α-fórmulas*, y las que actúan de manera disyuntiva, a las que llamaremos *β-fórmulas*. La siguiente definición establece estas dos categorías:

Definición 4.1 *Una α-fórmula es una fórmula del tipo $G_1 \wedge G_2$, $\neg(G_1 \vee G_2)$ o $\neg(G_1 \rightarrow G_2)$. Una β-fórmula es una fórmula del tipo $G_1 \vee G_2$, $\neg(G_1 \wedge G_2)$, $G_1 \rightarrow G_2$, $G_1 \leftrightarrow G_2$ o $\neg(G_1 \leftrightarrow G_2)$. Toda α-fórmula (β-fórmula) tiene dos componentes α_1 y α_2 (β_1 y β_2) que se indican en las siguientes tablas:*

α	α_1	α_2
$G_1 \wedge G_2$	G_1	G_2
$\neg(G_1 \vee G_2)$	$\neg G_1$	$\neg G_2$
$\neg(G_1 \rightarrow G_2)$	G_1	$\neg G_2$

β	β_1	β_2
$G_1 \vee G_2$	G_1	G_2
$\neg(G_1 \wedge G_2)$	$\neg G_1$	$\neg G_2$
$G_1 \rightarrow G_2$	$\neg G_1$	G_2
$G_1 \leftrightarrow G_2$	$G_1 \wedge G_2$	$\neg G_1 \wedge \neg G_2$
$\neg(G_1 \leftrightarrow G_2)$	$G_1 \wedge \neg G_2$	$\neg G_1 \wedge G_2$

Donde G_1 y G_2 son fórmulas proposicionales cualesquiera.

Las fórmulas de la forma $(G_1 \leftrightarrow G_2)$ y $\neg(G_1 \leftrightarrow G_2)$, se pueden considerar tanto α-fórmulas como β-fórmulas. Si se tiene en cuenta que $G_1 \leftrightarrow G_2 \equiv (G_1 \wedge G_2) \vee (\neg G_1 \wedge \neg G_2)$ y $\neg(G_1 \leftrightarrow G_2) \equiv (G_1 \wedge \neg G_2) \vee (\neg G_1 \wedge G_2)$, entonces se trata de β-fórmulas. Si, por el contrario, se consideran las equivalencias $G_1 \leftrightarrow G_2 \equiv (G_1 \vee \neg G_2) \wedge (\neg G_1 \vee G_2)$ y $\neg(G_1 \leftrightarrow G_2) \equiv (G_1 \vee G_2) \wedge (\neg G_1 \vee \neg G_2)$, entonces se trata de α-fórmulas. En esta presentación las hemos considerado β-fórmulas.

Definimos las funciones `alfa-formula` y `beta-formula`, presentadas en [45], para distinguir los dos tipos de fórmulas. A su vez, para construir las componentes, consideramos las funciones `componente-1`, que calcula α_1 y β_1, y `componente-2`, que calcula α_2 y β_2. Estas funciones se encuentran definidas en [46].

45

```
(defun alfa-formula (F)
  (or (es-conjuncion F)
      (and (es-negacion F) (or (es-disyuncion (arg1 F))
                               (es-implicacion (arg1 F))))))

(defun beta-formula (F)
  (or (es-disyuncion F)
      (es-implicacion F)
      (es-equivalencia F)
      (and (es-negacion F) (or (es-conjuncion (arg1 F))
                               (es-equivalencia (arg1 F))))))
```

46

```
(defun componente-1 (F)
  (cond ((es-negacion F)
         (let ((F-1 (arg1 F)))
           (cond ((es-conjuncion F-1)    (negacion (arg1 F-1)))
                 ((es-disyuncion F-1)    (negacion (arg1 F-1)))
                 ((es-implicacion F-1)   (arg1 F-1))
                 ((es-equivalencia F-1)
                  (conjuncion (arg1 F-1) (negacion (arg2 F-1))))
                 (t F))))
        ((es-conjuncion F)    (arg1 F))
        ((es-disyuncion F)    (arg1 F))
        ((es-implicacion F)   (negacion (arg1 F)))
        ((es-equivalencia F)  (conjuncion (arg1 F) (arg2 F)))
        (t F)))

(defun componente-2 (F)
  (cond ((es-negacion F)
         (let ((F-1 (arg1 F)))
           (cond ((es-conjuncion F-1)    (negacion (arg2 F-1)))
                 ((es-disyuncion F-1)    (negacion (arg2 F-1)))
                 ((es-implicacion F-1)   (negacion (arg2 F-1)))
                 ((es-equivalencia F-1)
                  (conjuncion (negacion (arg1 F-1)) (arg2 F-1)))
                 (t F))))
        ((es-conjuncion F)    (arg2 F))
        ((es-disyuncion F)    (arg2 F))
        ((es-implicacion F)   (arg2 F))
        ((es-equivalencia F)  (conjuncion (negacion (arg1 F))
                                          (negacion (arg2 F))))
        (t F)))
```

Definición 4.2 *Una doble negación es una fórmula de la forma* $\neg(\neg G)$*. La única componente de una doble negación* $\neg(\neg G)$ *es la fórmula* G*.*

El predicado `doble-negacion` sirve para reconocer este tipo de fórmulas y la función `componente-neg-neg` proporciona la componente.

47
```
(defun doble-negacion (F)
  (and (es-negacion F)
       (es-negacion (arg1 F))))

(defun componente-neg-neg (F)
  (if (doble-negacion F)
      (arg1 (arg1 F))
      F))
```

Definición 4.3 *Un literal es una fórmula atómica o su negación. Llamaremos literales positivos a las fórmulas atómicas y literales negativos a las negaciones de fórmulas atómicas.*

Consideramos las funciones `literal-positivo` y `literal-negativo` para reconocer los dos tipos de literales. Los literales son los elementos más básicos en la clasificación de las fórmulas basada en la notación uniforme y no tienen componentes.

48
```
(defun es-literal-positivo (F)
  (es-atomica F))

(defun es-literal-negativo (F)
  (and (es-negacion F)
       (es-atomica (arg1 F))))

(defun es-literal (F)
  (or (es-literal-positivo F)
      (es-literal-negativo F)))
```

El siguiente teorema establece la idea intuitiva en la que se basa la clasificación anterior: las α-fórmulas tienen un comportamiento conjuntivo y las β-fórmulas tienen un comportamiento disyuntivo.

Teorema 4.4 *Toda* α*-fórmula (*β*-fórmula) es lógicamente equivalente a la conjunción (disyunción) de sus componentes.*

49
```
(defthm valor-componentes-alfa-formula
  (implies (alfa-formula F)
           (equal (valor F sigma)
                  (funcion-de-verdad-de-conjuncion
                    (valor (componente-1 F) sigma)
                    (valor (componente-2 F) sigma)))))

(defthm valor-componentes-beta-formula
  (implies (beta-formula F)
           (equal (valor F sigma)
                  (funcion-de-verdad-de-disyuncion
                    (valor (componente-1 F) sigma)
                    (valor (componente-2 F) sigma)))))
```

Teorema 4.5 *La componente de una doble negación es lógicamente equivalente a la fórmula original.*

50
```
(defthm valor-componentes-doble-negacion
  (implies (doble-negacion F)
           (equal (valor F sigma)
                  (valor (componente-neg-neg F) sigma))))
```

En los eventos que formalizan los dos teoremas anteriores la variable `sigma` representa una asignación cualquiera. De esta forma, la interpretación del evento `valor-componente-alfa-formula` es la siguiente: "Para toda asignación `sigma`, el valor de la fórmula `F` en `sigma` es igual a la conjunción de los valores en `sigma` de sus componentes".

El siguiente teorema establece la clasificación basada en la notación uniforme:

Teorema 4.6 *Toda fórmula proposicional es una α-fórmula, una β-fórmula, una doble negación o un literal.*

Este resultado proporciona una nueva caracterización de `es-proposicional` y, por tanto, una nueva manera de definir funciones por recursión en la estructura de las fórmulas.

51
```
(defthm definicion-uniforme-de-es-proposicional
   (iff (es-proposicional F)
        (or (doble-negacion F)
            (alfa-formula F)
            (beta-formula F)
            (es-literal F))))
```

4.1.2 Medida asociada a la notación uniforme

En la sección 3.1.3 hemos visto una medida con respecto a la que las subfórmulas de toda fórmula proposicional compuesta son menores que la original. Esta medida es útil para demostrar la terminación de funciones definidas recursivamente sobre fórmulas. La notación uniforme proporciona una nueva manera de clasificar las fórmulas, de aquí que tengamos que definir una nueva medida, con respecto a la cual las componentes de una fórmula compuesta en la notación uniforme sean menores que la fórmula original. Utilizamos una extensión de la medida proporcionada en [7] para tener en cuenta las equivalencias:

Definición 4.7 *La medida uniforme de una fórmula proposicional $u(F)$ es:*

1. *Si F es atómica, $u(F) = 0$.*
2. *Si $F = \neg G$, $u(F) = 1 + u(G)$.*
3. *Si $F = G_1 \circ G_2$, donde $\circ$ es una conectiva binaria distinta de la equivalencia, $u(F) = 2 + u(G_1) + u(G_2)$.*
4. *Si $F = G_1 \leftrightarrow G_2$, $u(F) = 5 + u(G_1) + u(G_2)$.*

La función `medida-uniforme` implementa esta medida:

52
```
(defun medida-uniforme (F)
  (cond ((es-equivalencia F)
         (+ 5 (medida-uniforme (arg1 F))
              (medida-uniforme (arg2 F))))
        ((or (es-conjuncion F)
             (es-disyuncion F)
             (es-implicacion F))
         (+ 2 (medida-uniforme (arg1 F))
              (medida-uniforme (arg2 F))))
        ((es-negacion F)
         (+ 1 (medida-uniforme (arg1 F))))
        (t 0)))
```

El siguiente teorema demuestra que las medidas uniformes de las componentes de una doble negación, α-fórmula o β-fórmula son menores que las medidas uniformes de la fórmula original.

Teorema 4.8 *Sea F una fórmula proposicional:*

1. *Si $F = \neg(\neg G)$, entonces $u(G) < u(F)$.*

2. *Si F es una α-fórmula de componentes α_1 y α_2, entonces $u(\alpha_1) < u(F)$ y $u(\alpha_2) < u(F)$.*

3. *Si F es una β-fórmula de componentes β_1 y β_2, entonces $u(\beta_1) < u(F)$ y $u(\beta_2) < u(F)$.*

La formalización de estas propiedades es la siguiente:

53
```
(defthm medida-uniforme-componente-neg-neg-disminuye
  (implies (doble-negacion F)
           (< (medida-uniforme (componente-neg-neg F))
              (medida-uniforme F))))

(defthm medida-uniforme-componentes-alfa-disminuye
  (implies (alfa-formula F)
           (and (< (medida-uniforme (componente-1 F))
                   (medida-uniforme F))
                (< (medida-uniforme (componente-2 F))
                   (medida-uniforme F)))))

(defthm medida-uniforme-componentes-beta-disminuye
  (implies (beta-formula F)
           (and (< (medida-uniforme (componente-1 F))
                   (medida-uniforme F))
                (< (medida-uniforme (componente-2 F))
                   (medida-uniforme F)))))
```

La prueba de estos resultados se obtiene gracias al procedimiento de decisión para la aritmética lineal que tiene implementado ACL2.

4.1.3 Notación uniforme en $\mathbb{K}$

El único aspecto relativo a la notación uniforme que está relacionado con la semántica, es el que establece el comportamiento de las α-fórmulas, β-formulas y dobles negaciones con respecto a sus componentes, teoremas 4.4 y 4.5. Estos resultados siguen siendo ciertos en la semántica de Kleene y su prueba se obtiene de forma similar. La formalización de los mismos es igual a la presentada en *49* y *50* pero utilizando las funciones `valor-K`, `funcion-de-verdad-de-conjuncion-K` y `funcion-de-verdad-de-disyuncion-K`.

4.2 Tableros semánticos

En esta sección se presenta el método de los tableros semánticos proposicionales, siguiendo la descripción dada en [26]. Éste es un sistema de refutación, es decir,

$$\frac{\neg\neg G}{G} \qquad \frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta}{\beta_1 \mid \beta_2}$$

Figura 4.1: Reglas de expansión de tableros

para probar que una fórmula F es válida, se comienza a trabajar con $\neg F$ hasta producir una contradicción. Desde un punto de vista más constructivo, el método comienza a trabajar con un conjunto de fórmulas, y devuelve un modelo de dicho conjunto. Así, si no es posible construir un modelo para la fórmula $\neg F$, entonces F es válida.

Simultáneamente a la presentación del método, se describen los elementos necesarios para su formalización hasta desarrollar un algoritmo para comprobar la satisfacibilidad de una fórmula en la semántica clásica. A continuación se presentan los principales eventos relacionados con la prueba automática de la corrección y completitud de dicho algoritmo. El método de los tableros semánticos también es utilizado para construir algoritmos para decidir la validez de una fórmula y la consecuencia lógica. En la última sección se comenta cómo estos algoritmos se pueden utilizar en la semántica de Kleene y se indican las principales diferencias con los de la semántica clásica. Los eventos relativos a la semántica clásica presentados en esta sección se encuentran en el libro `tableros.lisp`, los relativos a la semántica de Kleene en el libro `tableros-K.lisp`.

4.2.1 Tableros semánticos proposicionales

El método de los tableros semánticos trabaja con árboles finitos con sus nodos etiquetados con fórmulas. Estos árboles son modificados de acuerdo al conjunto de reglas de expansión mostrado en la figura 4.1. Para simplificar el número de reglas, estas se expresan utilizando la notación uniforme.

Las reglas de expansión se aplican como se describe a continuación: dado un árbol finito T con sus nodos etiquetados con fórmulas proposicionales, se selecciona una rama θ en la que haya una ocurrencia de una fórmula no literal F. Si F es $\neg\neg G$, entonces se aumenta la rama θ, añadiendo al final un nodo etiquetado con G. Si F es una α-fórmula, entonces se aumenta la rama θ, añadiendo al final dos nodos etiquetados con las componentes α_1 y α_2 de la fórmula original. Finalmente, si F es una β-fórmula, entonces añadimos dos hijos al nodo final de θ, cada uno de ellos etiquetado con una de las componentes β_1 y β_2 de la fórmula original. Si llamamos T^* al árbol resultante, diremos que T^* se obtiene a partir de T mediante la aplicación de una regla de expansión de tableros.

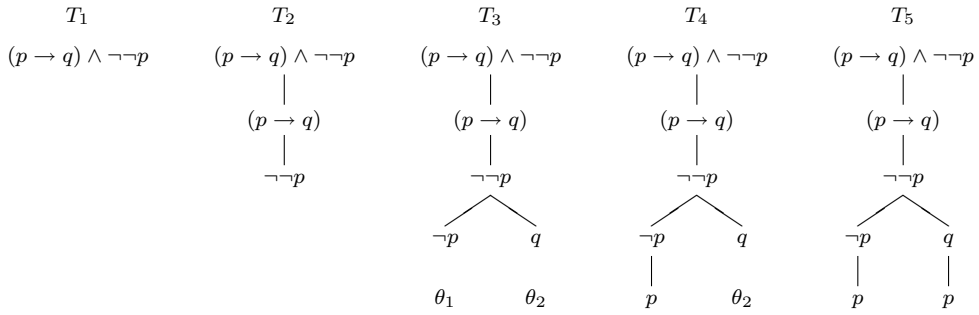

Figura 4.2: Un ejemplo del método de los tableros semánticos

Definición 4.9 *Sea $\{A_1, \ldots, A_n\}$ un conjunto finito de fórmulas proposicionales.*

1. *El siguiente árbol de un rama es un tablero para $\{A_1, \ldots, A_n\}$:*

$$\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_n \end{array}$$

2. *Si T es un tablero para $\{A_1, \ldots, A_n\}$ y T^* se obtiene a partir de T mediante una regla de expansión de tableros, entonces T^* es un tablero para $\{A_1, \ldots, A_n\}$.*

La figura 4.2 muestra una secuencia de transformaciones mediante el uso de reglas de expansión de tableros. El tablero inicial T_1 consta de una única rama con la fórmula $(p \to q) \wedge \neg\neg p$, ésta es una α-fórmula cuyas componentes son $(p \to q)$ y $\neg\neg p$. Al aplicar la regla de expansión para las α-fórmulas se obtiene el tablero T_2. En T_2 hay una ocurrencia de la fórmula $(p \to q)$, ésta es una β-fórmula cuyas componentes son $\neg p$ y q. Al aplicar la regla de expansión para las β-fórmulas se obtiene el tablero T_3. En T_3 hay una ocurrencia de la fórmula $\neg\neg p$ en la rama θ_1, ésta es una doble negación cuya componente es p. Al aplicar la regla de expansión para las dobles negaciones en la rama θ_1, se obtiene el tablero T_4. Se puede aplicar otra vez la regla de expansión para las dobles negaciones a la ocurrencia de la fórmula $\neg\neg p$ en la rama θ_2 del tablero T_4, dando lugar al tablero T_5.

Obsérvese que una misma ocurrencia de una fórmula puede pertenecer a distintas ramas y por tanto, se le puede aplicar una regla de expansión en cada una de ellas. Esto ocurre en el tablero T_3 de la figura 4.2, la misma ocurrencia de la fórmula $\neg\neg p$ aparece en las ramas θ_1 y θ_2, por tanto la regla de expansión para las dobles negaciones se puede aplicar para esta fórmula tanto en θ_1 como en θ_2.

En la formalización representamos las ramas de un tablero utilizando listas de fórmulas. La función `es-rama-tablero`, mostrada en 54, sirve para comprobar

esta propiedad. De esta forma, el resultado de aplicar una regla de expansión a una rama consistirá en añadir las correspondientes componentes a la lista que la representa. Cuando se aplica la regla de expansión para las β-fórmulas, se crean dos copias de la lista que representa la rama original y se añade una componente a cada una de ellas. Así, si la misma ocurrencia de una fórmula aparece en dos ramas distintas, dicha fórmula aparecerá en las dos listas que las representan.

54
```
(defun es-rama-tablero (rama)
  (or (endp rama)
      (and (es-proposicional (car rama))
           (es-rama-tablero (cdr rama)))))
```

Definición 4.10 *Dado un tablero T y una asignación σ. Diremos que σ es modelo de una rama θ de T y lo notaremos $\sigma \models \theta$, si y sólo si σ es modelo de todas las fórmulas que aparecen en θ. Diremos que σ es modelo de T y lo notaremos $\sigma \models T$, si y sólo si σ es modelo de alguna de sus ramas. Diremos que dos tableros T_1 y T_2 son lógicamente equivalentes y lo notaremos $T_1 \equiv T_2$ si para toda asignación σ, $\sigma \models T_1$ si y sólo si $\sigma \models T_2$.*

La función `modelo-rama` formaliza el concepto de modelo de una rama.

55
```
(defun modelo-rama (sigma rama)
  (cond ((endp rama) t)
        (t (and (modelo sigma (car rama))
                (modelo-rama sigma (cdr rama))))))
```

De esta forma, un tablero se interpreta como la disyunción de sus ramas, de aquí que la regla para las β-fórmulas genere bifurcaciones. Las ramas de un tablero se interpretan como la conjunción de las fórmulas que en ellas aparecen, es por esto que la regla para las α-fórmulas añade las dos componentes a la misma rama.

Teorema 4.11 *Dada una rama θ y una asignación σ:*

1. *Si θ' es la rama obtenida aplicando una regla de expansión a una α-fórmula de θ entonces $\sigma \models \theta'$ si y sólo si $\sigma \models \theta$.*
2. *Si θ'_1 y θ'_2 son las ramas obtenidas aplicando una regla de expansión a una β-fórmula de θ entonces $\sigma \models \theta$ si y sólo si $\sigma \models \theta'_1$ o $\sigma \models \theta'_2$.*
3. *Si θ' es la rama obtenida aplicando una regla de expansión a una doble negación de θ entonces $\sigma \models \theta'$ si y sólo si $\sigma \models \theta$.*

La prueba de este teorema se obtiene fácilmente a partir de la semántica de los tableros y las propiedades 4.4 y 4.5. Su formalización aparece en *59*.

Corolario 4.12 *Si T^* se obtiene a partir de T mediante una regla de expansión de tableros entonces $T^* \equiv T$.*

Teniendo en cuenta esta interpretación de los tableros, las repeticiones de una fórmula en una misma rama pueden ser eliminadas dando lugar a un tablero lógicamente equivalente al original. Por tanto, desde el punto de vista de la interpretación, no es necesario aplicar varias veces una misma regla de expansión a la ocurrencia de una fórmula dentro de una rama, ya que esto genera repeticiones innecesarias. En el tablero T_3 de la figura 4.2, aplicar la regla de expansión para las α-fórmulas a $(p \rightarrow q) \wedge \neg\neg p$ en cualquiera de las dos ramas genera repeticiones de las fórmulas $(p \rightarrow q)$ y $\neg\neg p$. Diremos que la ocurrencia de una fórmula F en una rama θ está expandida si se le ha aplicado una regla de expansión de tableros en dicha rama.

Definición 4.13 *Una rama θ de un tablero se dice que está cerrada si contiene fórmulas complementarias.*

La función `rama-cerrada` formaliza este concepto:

56
```
(defun rama-cerrada (rama)
  (cond ((endp rama) nil)
        ((member-equal (complementario (car rama)) (cdr rama)) t)
        (t (rama-cerrada (cdr rama)))))
```

Teorema 4.14 *Si θ es una rama cerrada de un tablero, entonces θ no posee modelos, es decir, para toda asignación σ, $\sigma \not\models \theta$.*

La demostración de este teorema es trivial debido a la definición de modelo de una rama. Su formalización es la siguiente:

57
```
(defthm rama-cerrada-no-tiene-modelos
  (implies (and (rama-cerrada rama)
                (es-rama-tablero rama))
           (not (modelo-rama sigma rama))))
```

Teorema 4.15 *Si θ es una rama no cerrada de un tablero, en la que todas las ocurrencias de fórmulas no literales están expandidas, entonces la asignación σ definida de forma que $\sigma(p) = \top$ si y sólo si p es un literal positivo que aparece en θ, es un modelo de θ.*

Demostración:

La prueba de este resultado se basa en los siguientes hechos:

1. Si F es una α-fórmula que aparece en θ, entonces sus componentes α_1 y α_2 también están en θ. Por tanto, si θ' es la rama que se obtiene eliminando de θ las ocurrencias de F, se verifica $\sigma \models \theta$ si y sólo si $\sigma \models \theta'$, puesto que α_1 y α_2 aparecen en θ'.

2. Si F es una β-fórmula que aparece en θ, entonces alguna de sus componentes β_1 o β_2 también está en θ. Por tanto, si θ' es la rama que se obtiene eliminando de θ las ocurrencias de F, se verifica $\sigma \models \theta$ si y sólo si $\sigma \models \theta'$, puesto que una de las componentes β_1 o β_2 aparece en θ'.

3. Si $F = \neg\neg G$ aparece en θ, entonces su componente G también está en θ. Por tanto, si θ' es la rama que se obtiene eliminando de θ las ocurrencias de F, se verifica $\sigma \models \theta$ si y sólo si $\sigma \models \theta'$, puesto que G aparece en θ'.

De esta forma, sólo tenemos que demostrar que σ es modelo de todos los literales que aparecen en θ. Por definición de σ esto es cierto para los literales positivos. Si $\neg q$ es un literal negativo que aparece en θ, entonces q no aparece en θ puesto que no es una rama cerrada y por tanto $\sigma(q) = \bot$, luego σ es modelo de $\neg q$.

□

La prueba de este teorema sugiere que las ocurrencias de fórmulas expandidas dentro de una rama son innecesarias desde el punto de vista de la interpretación de la misma. Teniendo en cuenta esta idea y la observación acerca de las repeticiones de elementos dentro de una rama, las reglas de expansión quedan como sigue:

1. Para aplicar una regla de expansión a una α-fórmula de la rama θ, se elimina la fórmula a expandir y se añaden sus componentes, si es que no estaban ya en la rama.

2. Para aplicar una regla de expansión a una β-fórmula de la rama θ, se crean dos nuevas ramas iguales a la original en las que se ha eliminado la fórmula expandida y a cada una de ellas se añade una componente de la β-fórmula expandida, si es que dicha componente no estaba ya en la rama.

3. Para aplicar una regla de expansión a una doble negación de la rama θ, se elimina la fórmula a expandir y se añade su componente, si es que no estaba ya en la rama.

De esta forma, los tableros de la figura 4.2 se pueden representar de la siguiente forma:

$$
\begin{aligned}
T_1 &= \langle\langle (p \to q) \wedge \neg\neg p\rangle\rangle \\
T_2 &= \langle\langle (p \to q), \neg\neg p\rangle\rangle \\
T_3 &= \langle\langle \neg p, \neg\neg p\rangle, \langle q, \neg\neg p\rangle\rangle \\
T_4 &= \langle\langle \neg p, p\rangle, \langle q, \neg\neg p\rangle\rangle \\
T_5 &= \langle\langle \neg p, p\rangle, \langle q, p\rangle\rangle
\end{aligned}
$$

El árbol T_1 está formado por una única rama con la fórmula $(p \to q) \wedge \neg\neg p$. Al aplicarle la regla de expansión para las α-fórmulas, se sustituye dicha fórmula por sus componentes, dando lugar a la lista $\langle (p \to q), \neg\neg p \rangle$, que es la única rama del árbol T_2. A este árbol se le puede aplicar la regla de expansión para las β-fórmulas, eliminando la fórmula $(p \to q)$ y generando dos ramas, cada una de ellas con una componente de $(p \to q)$. El resultado es el árbol T_3, que tiene dos ramas: $\langle \neg p, \neg\neg p \rangle$ y $\langle q, \neg\neg p \rangle$. En la primera rama de este árbol se puede aplicar la regla de expansión para las dobles negaciones. Esta regla elimina la ocurrencia de la fórmula $\neg\neg p$ y añade su componente p a la primera rama. El resultado es el árbol T_4 cuyas ramas son $\langle \neg p, p \rangle$ y $\langle q, \neg\neg p \rangle$. En este punto, sólo queda una fórmula a la que se pueda aplicar una regla de expansión, se trata de $\neg\neg p$ en la segunda rama de T_4. Al aplicar dicha regla la fórmula $\neg\neg p$ es reemplazada por p, dando lugar al árbol T_5.

En la formalización, utilizamos la función `elimina-una` para eliminar una fórmula de una lista y la función `añade-elemento` para añadir una fórmula a una lista, si es que no aparece ya en ella:

58

```
(defun elimina-una (F lista)
  (cond ((endp lista) lista)
        ((equal (car lista) F) (cdr lista))
        (t (cons (car lista) (elimina-una F (cdr lista))))))

(defun añade-elemento (F lista)
  (if (member-equal F lista)
      lista
      (cons F lista)))
```

Así, las reglas de expansión actúan de la siguiente forma: sea `rama` una rama de un tablero y `F` una fórmula no literal que aparece en `rama`:

1. Si `F` es una α-fórmula, el resultado de aplicar la correspondiente regla de expansión es:

   ```
   (añade-elemento (componente-1 F)
                   (añade-elemento (componente-2 F)
                                   (elimina-una F rama)))
   ```

2. Si `F` es una β-fórmula, el resultado de aplicar la correspondiente regla de expansión es:

   ```
   (añade-elemento (componente-1 F) (elimina-una F rama))

   (añade-elemento (componente-2 F) (elimina-una F rama))
   ```

3. Si `F` es una doble negación, el resultado de aplicar la correspondiente regla de expansión es:

```
(añade-elemento (componente-neg-neg F) (elimina-una F rama))
```

Una vez establecido el comportamiento de las reglas de expansión, presentamos la formalización del teorema 4.11:

59

```
(defthm modelo-rama-doble-negacion
  (implies
    (and (member-equal F rama)
         (doble-negacion F))
    (iff (modelo-rama
          sigma (añade-elemento (componente-neg-neg F)
                                (elimina-una F rama)))
         (modelo-rama sigma rama))))

(defthm modelo-rama-alfa-formula
  (implies
    (and (member-equal F rama)
         (alfa-formula F))
    (iff (modelo-rama
          sigma
          (añade-elemento (componente-1 F)
                          (añade-elemento (componente-2 F)
                                          (elimina-una F rama))))
         (modelo-rama sigma rama))))

(defthm modelo-rama-beta-formula
  (implies
    (and (beta-formula F)
         (member-equal F rama))
    (iff (modelo-rama sigma rama)
         (or (modelo-rama
              sigma (añade-elemento (componente-1 F)
                                    (elimina-una F rama)))
             (modelo-rama
              sigma (añade-elemento (componente-2 F)
                                    (elimina-una F rama)))))))
```

4.2.2 Modelo de una rama: MOD_{π}

Puesto que las reglas de expansión de tableros preservan la validez, se pueden utilizar para obtener un modelo de un conjunto de fórmulas. Para ello basta con aplicar las reglas de expansión a la rama formada por dichas fórmulas, hasta obtener una rama no cerrada en la que todas las ocurrencias de fórmulas no literales estén expandidas. A partir de esta rama se puede construir un modelo del conjunto de fórmulas inicial. El siguiente algoritmo realiza dicho proceso:

Algoritmo 4.16 (MOD_{π}) *El dato de entrada de este algoritmo es una rama de un tablero θ y actúa como se describe a continuación:*

1. *Si θ es una rama cerrada, el algoritmo termina y devuelve* **f**.

2. *Se selecciona una fórmula F no literal y no expandida de la rama θ.*

   (a) *Si F es una α-fórmula o una doble negación, se aplica a θ la correspondiente regla expansión de tableros y se evalúa el algoritmo con la nueva rama obtenida.*

   (b) *Si F es una β-fórmula, se le aplica a θ la regla de expansión para las β-fórmulas obteniendo dos nuevas ramas, θ_1 y θ_2. Se evalúa el algoritmo sobre una de ellas, θ_1, y si devuelve* **f** *entonces se evalúa sobre la otra, θ_2.*

3. *Si θ no tiene ocurrencias de fórmulas no literales no expandidas, el algoritmo termina y devuelve θ.*

El algoritmo anterior no está completamente determinado, puesto que no se indica cómo se escoge la fórmula F. Para resolver esta indeterminación hemos considerado una función `seleccion`, que escoge una fórmula no literal de una rama, si es que existe. Si la rama no contiene fórmulas no literales, la función `seleccion` devuelve `nil`. Se ha asumido la existencia de esta función en un encapsulado ACL2, caracterizándola por las siguientes propiedades:

- Si la función `seleccion` devuelve un valor, entonces dicho valor pertenece a la rama.

- Si la función `seleccion` devuelve un valor, entonces dicho valor es un literal.

- Si la función `seleccion` no devuelve ningún valor, entonces todos los elementos de la rama son literales.

La formalización de estas propiedades es la siguiente:

60

```
(defthm si-existe-no-literal-seleccion-pertenece
  (implies (seleccion rama)
           (member-equal (seleccion rama) rama)))

(defthm si-existe-no-literal-seleccion-no-es-literal
  (implies (seleccion rama)
           (not (es-literal (seleccion rama)))))

(defthm si-no-seleccion-miembros-literales
  (implies (and (not (seleccion rama))
                (es-rama-tablero rama)
                (member-equal F rama))
           (es-literal F)))
```

De esta forma, el resto del desarrollo presentado en esta sección es genérico, en el sentido de que sirve para cualquier función `seleccion` que verifique estas propiedades.

La función que implementa el algoritmo $MOD_{\mathcal{R}}$ es la siguiente:

61

```
(defun MOD-rama (rama)
  (declare (xargs :measure (medida-uniforme-rama rama)))
  (cond
    ((endp rama) nil)
    ((rama-cerrada rama) nil)
    (t (let ((F (seleccion rama)))
         (cond ((doble-negacion F)
                (MOD-rama (añade-elemento (componente-neg-neg F)
                                          (elimina-una F rama))))
               ((alfa-formula F)
                (MOD-rama
                  (añade-elemento
                    (componente-1 F)
                    (añade-elemento (componente-2 F)
                                    (elimina-una F rama)))))
               ((beta-formula F)
                (or (MOD-rama
                      (añade-elemento (componente-1 F)
                                      (elimina-una F rama)))
                    (MOD-rama
                      (añade-elemento (componente-2 F)
                                      (elimina-una F rama)))))
               (t rama))))))
```

Utilizando una función `seleccion` concreta, por ejemplo la que devuelve la primera fórmula no literal de una lista, podemos utilizar la función anterior para comprobar la satisfacibilidad de una rama de un tablero. En caso de que la rama sea satisfacible, la función devuelve una lista de literales a partir de la que se puede construir un modelo de la rama original. Si la rama no es satisfacible, la función devuelve `nil`.

```
ACL2 !>(MOD-rama '((-> (- p) q) (-> q r) (\/ r s)))
(R (- Q) P)
ACL2 !>(MOD-rama '((-> p q) (-> q (- r)) (& p r)))
NIL
```

En el primer ejemplo la rama $\langle (p \rightarrow q), (q \rightarrow r), (r \vee s) \rangle$ es satisfacible. Cualquier asignación en la que todos los literales de la lista que se obtiene como resultado sean ciertos, es modelo de dicha rama. En el segundo ejemplo, la rama $\langle (p \rightarrow q), (q \rightarrow \neg r), (p \wedge r) \rangle$ es instatisfacible.

Para que la función `MOD-rama` sea admitida por el sistema es necesario probar su terminación. Para ello hemos proporcionado una medida de los argumentos, `medida-uniforme-rama`, que decrece en las llamadas recursivas.

Definición 4.17 *La medida uniforme de una rama $u^+(\theta)$ es la suma de las medidas uniformes de todas las fórmulas que aparecen en dicha rama. Así, $u^+(\langle F_1, \ldots, F_n \rangle) = u(F_1) + \cdots + u(F_n)$.*

La función `medida-uniforme-rama` formaliza este concepto:

62

```
(defun medida-uniforme-rama (rama)
  (cond ((endp rama) 0)
        (t (+ (medida-uniforme (car rama))
              (medida-uniforme-rama (cdr rama))))))
```

Teorema 4.18 *Si θ' es una rama obtenida al aplicar una regla de expansión a otra rama θ entonces $u^+(\theta') < u^+(\theta)$.*

El evento asociado a este teorema se genera de forma automática al evaluar el evento *61*. Su demostración se obtiene gracias a los teoremas *53* y al hecho de que para toda α-fórmula F, $u(\alpha_1) + u(\alpha_2) < u(\alpha)$.

4.2.3 Corrección y completitud de $MOD_{\mathcal{R}}$

Teorema 4.19 (Corrección de $MOD_{\mathcal{R}}$) *Dada una rama θ, si $MOD_{\mathcal{R}}(\theta) = \theta' \neq \mathbf{f}$ entonces θ tiene modelos. Además si σ es la asignación definida de forma que $\sigma(p) = \top$ si y sólo si p es un literal positivo que aparece en θ', entonces $\sigma \models \theta$.*

Demostración:

Si $MOD_{\mathcal{R}}(\theta) = \theta' \neq \mathbf{f}$, entonces θ' es una rama no cerrada en la que todas las fórmulas no literales están expandidas. Por el teorema 4.15, la asignación σ definida en el enunciado es modelo de la rama θ'. Por el teorema 4.11, esta asignación es modelo de cualquier rama a partir de la cual se haya obtenido θ' mediante reglas de expansión de tableros. Por tanto σ es modelo de la rama inicial.

□

La función `genera-modelo-literales` construye la asignación σ mencionada en el teorema de corrección:

63
```
(defun genera-modelo-literales (lista-L)
  (cond ((endp lista-L) nil)
        ((es-literal-positivo (car lista-L))
         (asume-valor (car lista-L) *V*
                      (genera-modelo-literales (cdr lista-L))))
        (t (genera-modelo-literales (cdr lista-L)))))
```

La formalización del teorema de corrección es la siguiente:

64
```
(defthm correccion-MOD-rama
  (implies (and (es-rama-tablero rama)
                (MOD-rama rama))
           (modelo-rama (genera-modelo-literales (MOD-rama rama))
                        rama)))
```

Teorema 4.20 (Completitud de $MOD_{\mathcal{R}}$) *Dada una rama θ, si existe una asignación σ modelo de θ, entonces $MOD_{\mathcal{R}}(\theta) \neq \mathbf{f}$*

La prueba de este resultado se obtiene fácilmente a partir del teorema 4.11. Su formalización es la siguiente:

65
```
(defthm completitud-MOD-rama
  (implies (and (consp rama)
                (es-rama-tablero rama)
                (modelo-rama sigma rama))
           (MOD-rama rama)))
```

4.2.4 Satisfacibilidad por tableros: $SAT_{\mathcal{T}}$

Para determinar la satisfacibilidad de una fórmula F y, si es satisfacible, obtener la información necesaria para construir un modelo de F, basta con utilizar el algoritmo $MOD_{\mathcal{R}}$ con la rama formada únicamente por la fórmula F.

Algoritmo 4.21 ($SAT_{\mathcal{T}}$) *Dada $F \in \mathbb{P}(\Sigma)$, $SAT_{\mathcal{T}}(F)$ es $MOD_{\mathcal{R}}(\langle F \rangle)$.*

La función que implementa este algoritmo es la siguiente:

66
```
(defun SAT-tableros-semanticos (F)
  (MOD-rama (list F)))
```

Si una fórmula es satisfacible, esta función devuelve una lista de literales a partir de la que se puede obtener un modelo de dicha fórmula. Si la fórmula es insatisfacible, la función devuelve `nil`:

```
ACL2 !>(SAT-tableros-semanticos '(& (-> p q) (- (- p))))
(P Q)
ACL2 !>(SAT-tableros-semanticos '(& (/ p q) (& (- p) (- q))))
NIL
```

En el primer ejemplo, la fórmula $(p \rightarrow q) \wedge \neg\neg p$ es satisfacible y una asignación que la hace cierta es cualquiera en la que p y q sean ciertos. En el segundo ejemplo, la fórmula $(p \vee q) \wedge (\neg p \wedge \neg q)$ es insatisfacible.

Teorema 4.22 *Dada $F \in \mathbb{P}(\Sigma)$, F es satisfacible si y sólo si $SAT_{\mathcal{T}}(F) \neq \mathbf{f}$. Además, si $SAT_{\mathcal{T}}(F) = \theta \neq \mathbf{f}$, y σ es la asignación definida de forma que $\sigma(p) = \top$ si y sólo si p es un literal positivo que aparece en θ, entonces $\sigma \models F$.*

Demostración:

Por definición $SAT_{\mathcal{T}}(F) \neq \mathbf{f}$ si y sólo si $MOD_{\mathcal{R}}(\langle F \rangle) \neq \mathbf{f}$ y, por los teoremas de corrección y completitud de $MOD_{\mathcal{R}}$, esto ocurre si y sólo si $\langle F \rangle$ tiene un modelo; es decir, F es satisfacible.

Si $SAT_{\mathcal{T}}(F) = \theta \neq \mathbf{f}$ y σ es la asignación definida en el enunciado, entonces por el teorema de corrección de $MOD_{\mathcal{R}}$, σ es modelo de $\langle F \rangle$ y por tanto, modelo de F.

□

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $SAT_{\mathcal{T}}$:

67
```
(defthm correccion-SAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (SAT-tableros-semanticos F))
           (modelo (genera-modelo-literales
                     (SAT-tableros-semanticos F)) F)))

(defthm completitud-SAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (modelo sigma F))
           (SAT-tableros-semanticos F)))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en *64* y *65*, cuando el valor de la variable `rama` es `(list F)`.

4.2.5 Demostrabilidad por tableros: $DAT_{\mathcal{T}}$

El método de los tableros semánticos también es utilizado para verificar la validez de una fórmula F, para ello basta con comprobar que la rama formada por $\neg F$ no tiene modelos.

Definición 4.23 *Una fórmula F es demostrable por tableros, y lo notaremos $\vdash_{\mathcal{T}} F$, si existe una sucesión de tableros $T_1, \ldots, T_m$, tal que:*

- *T_1 es el tablero de una única rama con la fórmula $\neg F$, $T_1 = \langle\langle \neg F \rangle\rangle$.*
- *Para todo $i > 1$, T_i se obtiene a partir de T_{i-1} mediante una regla de expansión de tableros.*
- *Todas las ramas de T_m están cerradas.*

Algoritmo 4.24 ($DAT_{\mathcal{T}}$) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $DAT_{\mathcal{T}}(F)$ es* **t** *si y sólo si $MOD_{\mathcal{R}}(\langle \neg F \rangle)$ es* **f**.

La función que implementa este algoritmo es la siguiente:

68
```
(defun DAT-tableros-semanticos (F)
  (not (MOD-rama (list (negacion F)))))
```

Esta función devuelve `T` si la fórmula que se le pasa como argumento es demostrable por tableros. En caso contrario devuelve `nil`.

```
ACL2 !>(DAT-tableros-semanticos '(<-> (<-> p q) (<-> q p)))
T
ACL2 !>(DAT-tableros-semanticos '(-> (-> p q) (& r s)))
NIL
```

En el primer ejemplo la fórmula $(p \leftrightarrow q) \leftrightarrow (q \leftrightarrow p)$ es demostrable por tableros, es decir, es válida. En el segundo, la fórmula $(p \rightarrow q) \rightarrow (r \wedge s)$ no es demostrable por tableros.

Teorema 4.25 *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, F es válida si y sólo si $DAT_{\mathcal{T}}(F)$ es* $\mathbf{t}$.

Demostración:
Por definición $DAT_{\mathcal{T}}(F) \neq \mathbf{t}$ si y sólo si $MOD_{\mathcal{R}}(\langle \neg F \rangle) \neq \mathbf{f}$ y, por los teoremas de corrección y completitud de $MOD_{\mathcal{R}}$, esto ocurre si y sólo si la rama $\langle \neg F \rangle$ es satisfacible, es decir, F no es válida. Luego $DAT_{\mathcal{T}}(F) = \mathbf{t}$ si y sólo si F es válida. □

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $DAT_{\mathcal{T}}$:

69
```
(defthm correccion-DAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (DAT-tableros-semanticos F))
           (modelo sigma F)))

(defthm completitud-DAT-tableros-semanticos
  (implies
    (and (es-proposicional F)
         (not (DAT-tableros-semanticos F)))
    (not (modelo (genera-modelo-literales
                  (SAT-tableros-semanticos (negacion F))) F))))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en *65* y *64* cuando el valor de la variable `rama` es `(list (negacion F))`.

4.2.6 Deducibilidad por tableros: $DEDUC_{\mathcal{T}}$

Finalmente, también se puede utilizar el método de los tableros semánticos para comprobar si una fórmula F, es consecuencia lógica de un conjunto de hipótesis $H_1, \ldots, H_n$. Para ello basta con comprobar que la rama $\langle \neg F, H_1, \ldots, H_n \rangle$ no tiene modelos.

Definición 4.26 *Dadas las fórmulas $F, H_1, \ldots, H_n$, decimos que F es deducible por tableros a partir de las hipótesis $H_1 \ldots, H_n$, y lo notaremos $H_1, \ldots, H_n \vdash_{\mathcal{T}} F$, si existe una sucesión de tableros $T_1, \ldots, T_m$, tal que:*

- *T_1 es el tablero de una única rama con las fórmulas $\neg F, H_1, \ldots, H_n$, $T_1 = \langle\langle \neg F, H_1, \ldots, H_n\rangle\rangle$.*
- *Para todo $i > 1$, T_i se obtiene a partir de T_{i-1} mediante una regla de expansión de tableros.*
- *Todas las ramas de T_n están cerradas.*

Algoritmo 4.27 ($DEDUC_{\mathcal{T}}$) *Dadas las fórmulas $F, H_1, \ldots, H_n$, el valor de $DEDUC_{\mathcal{T}}(\langle H_1, \ldots, H_n\rangle, F)$ es* **t** *si y sólo si $MOD_{\mathcal{R}}(\langle \neg F, H_1, \ldots, H_n\rangle)$ es* **f**.

La función que implementa este algoritmo es la siguiente:

70
```
(defun DEDUC-tableros-semanticos (lista-H F)
  (not (MOD-rama (cons (negacion F) lista-H))))
```

Esta función devuelve `T` si la fórmula `F` es deducible por tableros a partir de las hipótesis `lista-H`. En caso contrario devuelve `nil`.

```
ACL2 !>(DEDUC-tableros-semanticos '((-> p q) p) 'q)
T
ACL2 !>(DEDUC-tableros-semanticos '((-> p q) p) '(- q))
NIL
```

En el primer ejemplo la fórmula q es deducible por tableros a partir de las fórmulas $(p \to q)$ y p. En el segundo ejemplo, la fórmula $\neg q$ no es deducible por tableros a partir de las fórmulas $(p \to q)$ y p.

Teorema 4.28 *Dadas las fórmulas $F, H_1, \ldots, H_n \in \mathbb{P}(\Sigma)$, F es consecuencia lógica de $\{H_1, \ldots, H_n\}$ si y sólo si $DEDUC_{\mathcal{T}}(\langle H_1, \ldots, H_n\rangle, F)$ es* **t**.

Demostración:

Por definición $DEDUC_{\mathcal{T}}(\langle H_1, \ldots, H_n\rangle, F) \neq \mathbf{t}$ si y sólo si se verifica $MOD_{\mathcal{R}}(\langle \neg F, H_1, \ldots, H_n\rangle) \neq \mathbf{f}$ y, por los teoremas de corrección y completitud de $MOD_{\mathcal{R}}$, esto ocurre si y sólo si la rama $\langle \neg F, H_1, \ldots, H_n\rangle$ es satisfacible, es decir, si y sólo si existe una asignación σ modelo de los H_i y modelo de $\neg F$. Finalmente esto es equivalente a que F no sea consecuencia lógica de $\{H_1, \ldots, H_n\}$.

Por tanto F es consecuencia lógica de $\{H_1, \ldots, H_n\}$ si y sólo si el valor de $DEDUC_{\mathcal{T}}(\langle H_1, \ldots, H_n\rangle, F)$ es **t**.

□

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $DEDUC_{\mathcal{T}}$:

71
```
(defthm correccion-DEDUC-tableros-semanticos
  (implies (and (es-rama-tablero lista-H)
                (es-proposicional F)
                (DEDUC-tableros-semanticos lista-H F)
                (modelo-rama sigma lista-H))
           (modelo sigma F)))

(defthm completitud-consecuencia-por-tableros
  (let ((sigma (genera-modelo-literales
                (MOD-rama (cons (negacion F) lista-H)))))
    (implies (and (es-rama-tablero lista-H)
                  (es-proposicional F)
                  (not (DEDUC-tableros-semanticos lista-H F)))
             (and (modelo-rama sigma lista-H)
                  (not (modelo sigma F))))))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en 65 y 64 cuando el valor de la variable `rama` es `(cons (negacion F) lista-H)`.

4.2.7 Tableros semánticos en $\mathbb{K}$

Los algoritmos desarrollados en las subsecciones previas también pueden ser utilizados para decidir la satisfacibilidad, validez y consecuencia lógica en $\mathbb{K}$. Los algoritmos son los mismos, puesto que no se basan en aspectos semánticos, pero no ocurre lo mismo con los conceptos y propiedades que aseguran su corrección y completitud.

La semántica en $\mathbb{K}$ de las ramas de un tablero sigue siendo la misma: una $\mathbb{K}$-asignación es modelo de una rama si y sólo si lo es de todas las fórmulas que en ella aparecen. La función que implementa este concepto es ligeramente diferente puesto que utiliza la función `modelo-K` en lugar de `modelo`:

72
```
(defun modelo-rama-K (sigma rama)
  (cond ((endp rama) t)
        (t (and (modelo-K sigma (car rama))
                (modelo-rama-K sigma (cdr rama))))))
```

Una rama cerrada es una lista de fórmulas en la que aparecen fórmulas complementarias. De esta forma una $\mathbb{K}$-asignación no puede ser modelo de todas las fórmulas de una rama cerrada. La formalización de esta propiedad es similar a la presentada en 57 pero utilizando la función `modelo-rama-K` en lugar de `modelo-rama`.

Las propiedades 4.4 y 4.5 de la notación uniforme se mantienen en la semántica de Kleene. Es por esto que las reglas de expansión también preserven los modelos en $\mathbb{K}$ tal y como se establece en el teorema 4.11. La formalización de estas propiedades es similar a la presentada en *59*, utilizando `modelo-rama-K` en lugar de `modelo-rama`

4.2.7.1 Modelo en $\mathbb{K}$ de una rama

Los algoritmos para decidir la satisfacibilidad, validez y consecuencia lógica presentados en las subsecciones anteriores se basan en un procedimiento que busca una asignación modelo de una rama. Este procedimiento es igualmente válido en la semántica de Kleene, ya que no hace referencia a aspectos semánticos de las fórmulas. Utilizamos la función `MOD-rama` presentada en *61* para formalizar este procedimiento.

Los resultados de corrección y completitud de este procedimiento en la semántica de Kleene se establecen de igual forma que en la semántica clásica. Su formalización es la siguiente:

73

```
(defthm correccion-MOD-rama
  (implies (and (es-rama-tablero rama)
                (MOD-rama rama))
           (modelo-rama-K (genera-modelo-literales-K
                           (MOD-rama rama)) rama)))

(defthm completitud-MOD-rama
  (implies (and (consp rama)
                (es-rama-tablero rama)
                (modelo-rama-K sigma rama))
           (MOD-rama rama)))
```

La única diferencia entre estos eventos y los presentados en *64* y *65* es el uso de la función `modelo-rama-K`, para comprobar que una $\mathbb{K}$-asignación es modelo de una rama, y la función `genera-modelo-literales-K`, para construir una $\mathbb{K}$-asignación a partir de una rama no cerrada con todas sus fórmulas no literales expandidas.

Para que la $\mathbb{K}$-asignación construida por `genera-modelo-literales-K` sea modelo de la rama a partir de la que se construye, tenemos que asegurarnos de que asigna un valor de verdad a todos los literales que aparecen en dicha rama. La definición presentada en *63* sólo nos asegura que los literales positivos tendrán asignado `*V*` como valor de verdad, el valor de los literales negativos se obtiene por el comportamiento por defecto de las extensiones de las asignaciones parciales en la semántica clásica. En la semántica de Kleene el valor por defecto es $\perp$ y

por tanto dicha construcción no es la adecuada para generar un modelo de una rama.

La $\mathbb{K}$-asignación que se construye a partir de una rama θ no cerrada en la que todas las fórmulas no literales están expandidas es la siguiente:

$$\sigma(x) = \begin{cases} \top & \text{si } x \text{ es un literal positivo que aparece en } \theta \\ \bot & \text{si } \neg x \text{ es un literal negativo que aparece en } \theta \\ \text{I} & \text{en otro caso} \end{cases}$$

La función `genera-modelo-literales-K` formaliza esta construcción:

74
```
(defun genera-modelo-literales-K (lista-L)
  (cond ((endp lista-L) nil)
        ((es-literal-positivo (car lista-L))
         (asume-valor (car lista-L) *V*
                      (genera-modelo-literales-K (cdr lista-L))))
        ((es-literal-negativo (car lista-L))
         (asume-valor (arg1 (car lista-L)) *F*
                      (genera-modelo-literales-K (cdr lista-L))))
        (t (genera-modelo-literales-K (cdr lista-L)))))
```

4.2.7.2 Decisión de $\mathbb{K}$-satisfacibilidad basada en tableros semánticos

El algoritmo para decidir la $\mathbb{K}$-satisfacibilidad es el mismo que el de la semántica clásica. La función que lo implementa es `SAT-tableros-semanticos`, presentada en *66*. La formalización y prueba de sus propiedades de corrección y completitud es similar a las de la semántica clásica. Los eventos son los siguientes:

75
```
(defthm correccion-SAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (SAT-tableros-semanticos F))
           (modelo-K (genera-modelo-literales-K
                      (SAT-tableros-semanticos F)) F)))

(defthm completitud-SAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (modelo-K sigma F))
           (SAT-tableros-semanticos F)))
```

4.2.7.3 Decisión de $\mathbb{K}$-validez basada en tableros semánticos

El procedimiento para determinar si una fórmula es $\mathbb{K}$-válida es el mismo que el de la semántica clásica. La función `DAT-tableros-semanticos`, presentada en *68*, implementa este procedimiento. Los resultados de corrección y completitud son los siguientes:

76
```
(defthm correccion-DAT-tableros-semanticos
  (implies (and (es-proposicional F)
                (DAT-tableros-semanticos F)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-DAT-tableros-semanticos
  (implies
    (and (es-proposicional F)
         (not (DAT-tableros-semanticos F)))
    (no-modelo-K (genera-modelo-literales-K
                   (SAT-tableros-semanticos (negacion F))) F)))
```

Como se indicó en la sección 3.3.4 el resultado de corrección tiene una hipótesis adicional en la que se afirma que la asignación `sigma` asigna un valor de verdad a la fórmula `F`, es decir un valor distinto de `nil`. En el resultado de completitud la conclusión afirma la existencia de una $\mathbb{K}$-asignación que es no–modelo de la fórmula `F`.

4.2.7.4 Decisión de consecuencia $\mathbb{K}$-lógica basada en tableros semánticos

Al igual que en los dos casos anteriores, el algoritmo para decidir la consecuencia lógica en $\mathbb{K}$ es el mismo que para la semántica clásica. La función que lo implementa, `DEDUC-tableros-semanticos`, está definida en *70*. Los resultados de corrección y completitud se han construido siguiendo las indicaciones de la sección 3.3.4.

77
```
(defthm correccion-DEDUC-tableros-semanticos
  (implies (and (es-rama-tablero lista-H)
                (es-proposicional F)
                (DEDUC-tableros-semanticos lista-H F)
                (modelo-rama-K sigma lista-H)
                (valor-K F sigma))
           (modelo-K sigma F)))
```

78
```
(defthm completitud-consecuencia-por-tableros
  (let ((sigma (genera-modelo-literales-K
                (MOD-rama (cons (negacion F) lista-H)))))
    (implies (and (es-rama-tablero lista-H)
                  (es-proposicional F)
                  (not (DEDUC-tableros-semanticos lista-H F)))
             (and (modelo-rama-K sigma lista-H)
                  (no-modelo-K sigma F)))))
```

4.3 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, evaluándolos sobre algunos ejemplos: algunas fórmulas de Urquhart [83] y fórmulas representando el problema de las N reinas [62].

Antes de realizar cualquier evaluación, conviene compilar los ficheros que contienen el código. De esta forma los tiempos de respuesta obtenidos son menores. Una forma de hacer esto es certificando los libros ACL2. Una vez hecho esto, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `4-tableros`:

```
../calculos-proposicionales/4-tableros> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/4-tableros".

ACL2 !>
```

A continación incluimos el libro que contiene la teoría desarrollada en este capítulo. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "tableros")
```

La primera familia de ejemplos que presentamos está formada por las siguientes fórmulas de Urquhart:

$$
\begin{array}{l}
U_1 : v_1 \leftrightarrow v_1 \\
U_2 : v_2 \leftrightarrow (v_1 \leftrightarrow (v_2 \leftrightarrow v_1)) \\
U_3 : v_3 \leftrightarrow (v_2 \leftrightarrow (v_1 \leftrightarrow (v_3 \leftrightarrow (v_2 \leftrightarrow v_1)))) \\
U_4 : v_4 \leftrightarrow (v_3 \leftrightarrow (v_2 \leftrightarrow (v_1 \leftrightarrow (v_4 \leftrightarrow (v_3 \leftrightarrow (v_2 \leftrightarrow v_1))))))
\end{array}
$$

...

Esta familia está definida en el libro `urquhart.lisp` del directorio `0-ejemplos`. Cada una de las fórmulas se obtiene evaluando `(urquhart` n`)`, para distintos valores de n. En este caso hemos utilizado números enteros positivos como símbolos proposicionales:

```
ACL2 !>(include-book "../0-ejemplos/urquhart")
...
ACL2 !>(urquhart 1)
(<-> 1 1)
ACL2 !>(urquhart 2)
(<-> 2 (<-> 1 (<-> 2 1)))
ACL2 !>(urquhart 3)
(<-> 3 (<-> 2 (<-> 1 (<-> 3 (<-> 2 1)))))
ACL2 !>(urquhart 4)
(<-> 4 (<-> 3 (<-> 2 (<-> 1 (<-> 4 (<-> 3 (<-> 2 1)))))))
```

Para poder utilizar los procedimientos de decisión desarrollados en este capítulo, tenemos que proporcionar una definición concreta de la función que selecciona una fórmula de una rama. El símbolo asociado a esta función ya existe en el sistema por lo que tendremos que redefinirla. Las propiedades de los procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica se mantienen, siempre que la nueva definición de la función `seleccion` tenga las propiedades presentadas en *60*.

```
ACL2 !>(set-ld-redefinition-action '(:warn . :overwrite) state)

ACL2 !>(defun seleccion (lista)
         (cond ((endp lista) nil)
               ((es-literal (car lista)) (seleccion (cdr lista)))
               (t (car lista))))
```

Una vez hecho esto, se puede utilizar la función `DAT-tableros-semanticos` para comprobar la validez de las fórmulas de Urquhart:

```
ACL2 !>(DAT-tableros-semanticos (urquhart 1))
T
ACL2 !>(DAT-tableros-semanticos (urquhart 2))
T
```

En la tabla 4.1 se muestran los tiempos de evaluación de la función `DAT-tableros-semanticos` para las 15 primeras fórmulas de Urquhart.

La segunda familia de ejemplos es una formulación del problema de las N reinas en lógica proposicional. Fijado un valor de N, cada casilla de un tablero

Fórmula	tiempo	Fórmula	tiempo	Fórmula	tiempo
U_1	0.000	U_6	0.160	U_{11}	15.810
U_2	0.010	U_7	0.420	U_{12}	38.470
U_3	0.000	U_8	1.090	U_{13}	86.460
U_4	0.020	U_9	2.850	U_{14}	197.000
U_5	0.060	U_{10}	6.600	U_{15}	473.540

Tabla 4.1: Tiempos de evaluación de las fórmulas de Urquhart

Tamaño del tablero	2	3	4	5	6
Tiempo	0.000	0.140	1.240	5.740	527.110

Tabla 4.2: Tiempos de evaluación para el problema de las N reinas

de tamaño $N \times N$ se representa por una variable proposicional distinta. Dada una asignación cualquiera, si una variable es cierta en dicha asignación, interpretamos que hay una reina situada en la casilla correspondiente, si la variable es falsa, interpretamos que la casilla está libre. El problema consiste en obtener una asignación en la que sean ciertas las fórmulas que describen las restricciones del problema:

- En cada fila hay al menos una reina.
- Si en una casilla hay una reina entonces no puede haber reinas situadas en la misma fila, columna o diagonales que dicha casilla.

Esta familia está definida en el libro `reinas.lisp` del directorio `0-ejemplos`. Cada una de las fórmulas se obtiene evaluando `(n-reinas` n`)`, para distintos valores de n. Hemos utilizado números enteros positivos como símbolos proposicionales: la casilla de la fila i, columna j, se representa con el número $(i-1)*n+j$, con $1 \leq i, j \leq n$.

```
ACL2 !>(include-book "../0-ejemplos/reinas")
...
ACL2 !>(n-reinas 2)
(& (/ 1 2)
   (& (/ 3 4)
      (& (|->| 1 (& (- 2) (& (- 3) (- 4))))
         (& (|->| 2 (& (- 1) (& (- 4) (- 3))))
            (& (|->| 3 (& (- 4) (& (- 1) (- 2))))
               (|->| 4 (& (- 3) (& (- 2) (- 1)))))))))
```

En la tabla 4.2 se muestran los tiempos de evaluación de la función `SAT-tableros-semanticos` para algunas de estas fórmulas.

Sumario

En este capítulo:

- Se ha formalizado la notación uniforme para la lógica proposicional clásica. Se han presentado los eventos que establecen sus principales propiedades. Se ha definido una medida asociada a la notación uniforme que resulta útil para demostrar la terminación de funciones recursivas basadas en la misma. La notación uniforme es utilizada tanto en la semántica clásica como en la semántica de Kleene.

- Se ha formalizado el método de los tableros semánticos. La notación uniforme es utilizada para expresar las reglas de expansión en las que se basa este método.

- Se ha desarrollado una función que implementa un algoritmo para comprobar la satisfacibilidad de una fórmula en el sentido clásico, basado en el método de los tableros semánticos. Se han establecido los eventos que demuestran la corrección y completitud de este algoritmo.

- También se han desarrollado una función que implementa un algoritmo para comprobar la validez de una fórmula y otro que comprueba si una fórmula es consecuencia lógica de un conjunto de hipótesis. Se han presentado los eventos que establecen sus propiedades de corrección y completitud en la semántica clásica.

- Se ha demostrado que los algoritmos desarrollados también pueden ser utilizados en la semántica de Kleene, presentado los eventos que establecen sus propiedades de corrección y completitud en $\mathbb{K}$.

Capítulo 5

Cálculo de secuentes

El cálculo de secuentes fue introducido por G. Gentzen en 1935 [29]. Desde entonces ha sido ampliamente estudiado y utilizado siendo la base de algunos sistemas de razonamiento automático como Nuprl [16]. Como sus duales, los basados en tableros, los cálculos basados en secuentes son muy útiles en el desarrollo de métodos de deducción en lógicas multivaluadas [3]. Para más información sobre los cálculos de secuentes se puede consultar [2].

En este capítulo presentamos el cálculo de secuentes proposicionales. Basándonos en esta presentación, se formaliza un algoritmo para decidir la satisfacibilidad de una fórmula en la semántica clásica. Se presentan los resultados de corrección y completitud de este algoritmo y los eventos que los establecen. Un desarrollo similar es realizado con un algoritmo para decidir la validez de una fórmula y otro para la consecuencia lógica.

Los algoritmos desarrollados en el caso clásico son igualmente válidos en la semántica de Kleene, sólo hay que caracterizar adecuadamente la semántica de los secuentes para demostrar sus propiedades de corrección y completitud. En el caso clásico se caracterizan aquellas situaciones en las que un secuente es cierto, aunque se utiliza la propiedad contraria. En la semántica de Kleene se caracterizan las situaciones en las que un secuente es falso. Un desarrollo similar del cálculo de secuentes proposicionales en la semántica de Kleene, ha sido realizado en Nuprl en [15].

5.1 Secuentes

Los secuentes son pares de listas de fórmulas entre las que existe una relación de consecuencia. El cálculo de secuentes consta de un conjunto de axiomas y un conjunto de reglas de inferencia. Para probar la validez de una fórmula, se construye un secuente a partir de dicha fórmula y se comprueba si este secuente se puede obtener como resultado de aplicar sucesivamente las reglas de inferencia a los axiomas. Cuando el intento de prueba falla, proporciona un contramodelo de la fórmula inicial. De forma simultánea a la presentación del método, se describen

los elementos necesarios para su formalización y se desarrolla un algoritmo para decidir la validez de una fórmula. A continuación se describen los principales eventos relacionados con la prueba automática de la corrección y completitud de dicho algoritmo. También se describe un algoritmo para comprobar la satisfacibilidad de una fórmula. Los eventos relativos a la semántica clásica presentados en esta sección se encuentran en el libro `secuentes.lisp`, los relativos a la semántica de Kleene en el libro `secuentes-K.lisp`.

5.1.1 El cálculo G'

Definición 5.1 *Un secuente es un par $\langle \Gamma, \Delta \rangle$ de secuencias finitas de fórmulas.*

En lo sucesivo notaremos los secuentes de la forma $\Gamma \Rightarrow \Delta$, llamaremos a Γ parte izquierda del secuente y a Δ parte derecha. El símbolo $\Rightarrow$ sugiere que existe cierta relación de consecuencia entre las dos partes y, de hecho, esa es la intención. Usaremos la expresión $\langle \Gamma_1, F, \Gamma_2 \rangle$ para distinguir una ocurrencia de la fórmula F entre las secuencias de fórmulas Γ_1 y Γ_2. Los secuentes en los que una de las dos secuencias de fórmulas es vacía se notarán como $\Rightarrow \Delta$ y $\Gamma \Rightarrow$.

En la formalización usamos listas para representar los secuentes. En estas listas el primer elemento es la parte izquierda del secuente y el resto de los elementos forman la parte derecha. Así, si `S` es la representación del secuente $\Gamma \Rightarrow \Delta$, entonces `(car S)` es la secuencia de fórmulas Γ y `(cdr S)` es la secuencia de fórmulas Δ. A continuación se muestran algunos ejemplos de secuentes y sus representaciones:

$\langle \neg r, p \rangle \Rightarrow \langle q, \neg s \rangle$	$\longrightarrow$	`( ( (- r) p ) q (- s) )`
$\langle p, \neg q \rangle \Rightarrow$	$\longrightarrow$	`( ( p (- q) ) )`
$\Rightarrow \langle (p \wedge q), r \rangle$	$\longrightarrow$	`( ( ) (& p q) r )`

La función `es-secuente` sirve para comprobar si una expresión es un secuente representado de la forma descrita anteriormente. Para ello se define previamente la función `lista-formulas` que comprueba si su argumento es una lista de fórmulas.

79

```
(defun es-secuente (S)
  (and (consp S)
       (lista-formulas (car S))
       (lista-formulas (cdr S))))

(defun lista-formulas (lista-F)
  (or (endp lista-F)
      (and (es-proposicional (car lista-F))
           (lista-formulas (cdr lista-F)))))
```

Como se ha comentado antes, el símbolo $\Rightarrow$ sirve para indicar una relación de consecuencia entre la partes izquierda y derecha de un secuente: si todas las fórmulas de la parte izquierda son ciertas entonces alguna de la parte derecha también lo es. Se trata de una extensión a secuentes del concepto de modelo de una fórmula.

Definición 5.2 *Una asignación σ es modelo de un secuente $\Gamma \Rightarrow \Delta$, y lo notaremos $\sigma \models \Gamma \Rightarrow \Delta$, si existe una fórmula F en Γ tal que $\sigma \not\models F$ o existe una fórmula G en Δ tal que $\sigma \models G$. Una asignación σ es contramodelo de un secuente $\Gamma \Rightarrow \Delta$ si no es modelo suyo. Diremos que un secuente $\Gamma \Rightarrow \Delta$ es válido, si para toda asignación σ se verifica $\sigma \models \Gamma \Rightarrow \Delta$.*

De esta forma, una asignación es modelo de un secuente si no es modelo de todas las fórmulas de su parte izquierda o bien si, siendo modelo de todas las fórmulas de su parte izquierda, es modelo de alguna de las fórmulas de la parte derecha. Por tanto, desde el punto de vista de su interpretación, un secuente $\langle F_1, \ldots, F_n \rangle \Rightarrow \langle G_1, \ldots, G_m \rangle$ es lógicamente equivalente a la fórmula $(F_1 \wedge \ldots \wedge F_n) \rightarrow (G_1 \vee \ldots \vee G_m)$.

La función `modelo-secuente` implementa el concepto de modelo de un secuente. Esta función utiliza las funciones `modelo-conjuncion` y `modelo-disyuncion`. La primera comprueba si una asignación es modelo de todas las fórmulas de una lista y la segunda, si una asignación es modelo de alguna fórmula de una lista. Obviamente, si una asignación no es modelo de todas las fórmulas de la parte izquierda de un secuente entonces es modelo del secuente.

80

```
(defun modelo-secuente (sigma S)
  (if (modelo-conjuncion sigma (car S))
      (modelo-disyuncion sigma (cdr S))
      t))

(defun modelo-conjuncion (sigma Gamma)
  (cond ((endp Gamma) t)
        (t (and (modelo sigma (car Gamma))
                (modelo-conjuncion sigma (cdr Gamma))))))

(defun modelo-disyuncion (sigma Delta)
  (cond ((endp Delta) nil)
        (t (or (modelo sigma (car Delta))
               (modelo-disyuncion sigma (cdr Delta))))))
```

El concepto de secuente y el cálculo G' fueron introducidos originariamente por G. Gentzen [29]. Este cálculo consta de un conjunto de axiomas y un conjunto de reglas de inferencia. Los axiomas son un tipo simple de secuentes válidos y las reglas de inferencia preservan la validez. De esta forma cualquier secuente obtenido como resultado de aplicar sucesivamente las reglas de inferencia a los axiomas es válido.

Definición 5.3 *Un secuente $\Gamma \Rightarrow \Delta$ es un axioma si existe alguna fórmula en común en sus partes izquierda y derecha: $\Gamma \cap \Delta \neq \emptyset$.*

La función `secuente-axioma` formaliza este concepto, comprobando que existe algún elemento en la intersección entre las dos partes de un secuente. La función `intersectp-equal` comprueba que la intersección de dos listas es no vacía:

81
```
(defun secuente-axioma (S)
  (intersectp-equal (car S) (cdr S)))
```

Teorema 5.4 *Si el secuente $\Gamma \Rightarrow \Delta$ es un axioma entonces es válido.*

Demostración:
Si el secuente $\Gamma \Rightarrow \Delta$ es un axioma entonces existe una fórmula F que aparece en Γ y en Δ. Dada una asignación cualquiera σ, si σ es modelo de F entonces existe una fórmula en Δ de la que σ es modelo. Si σ no es modelo de F, entonces existe una fórmula en Γ de la que σ no es modelo. Luego σ es modelo de $\Gamma \Rightarrow \Delta$. □

La prueba de este resultado en la formalización se obtiene fácilmente a partir de las definiciones de `modelo-conjuncion` y `modelo-disyuncion`. El evento asociado es el siguiente:

82
```
(defthm secuente-axioma-es-valido
  (implies (secuente-axioma S)
           (modelo-secuente sigma S)))
```

Nótese que para afirmar que el secuente `S` es válido, se utiliza una nueva variable `sigma` que representa una asignación cualquiera, tal y cómo se indicó en la sección 3.2.4.

$$\dfrac{\dfrac{\dfrac{\langle p, \neg q\rangle \Rightarrow \langle p\rangle}{\langle \neg q\rangle \Rightarrow \langle \neg p, p\rangle}\,\neg\text{-der}}{\Rightarrow \langle p, (\neg q \to \neg p)\rangle}\,\to\text{-der} \qquad \dfrac{\dfrac{\langle q\rangle \Rightarrow \langle q, \neg p\rangle}{\langle \neg q, q\rangle \Rightarrow \langle \neg p\rangle}\,\neg\text{-izq}}{\langle q\rangle \Rightarrow \langle \neg q \to \neg p\rangle}\,\to\text{-der}}{\langle p \to q\rangle \Rightarrow \langle \neg q \to \neg p\rangle}\,\to\text{-izq}$$

Figura 5.1: Un ejemplo de inferencia en el cálculo G'

Definición 5.5 *El conjunto de reglas de inferencia del cálculo G' es el siguiente:*

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow F, \Delta}{\Gamma_1, \neg F, \Gamma_2 \Rightarrow \Delta}\ (\neg\text{-}izq) \qquad \frac{F, \Gamma \Rightarrow \Delta_1, \Delta_2}{\Gamma \Rightarrow \Delta_1, \neg F, \Delta_2}\ (\neg\text{-}der)$$

$$\frac{\Gamma_1, F, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \wedge G, \Gamma_2 \Rightarrow \Delta}\ (\wedge\text{-}izq) \qquad \frac{\Gamma \Rightarrow \Delta_1, F, \Delta_2 \quad \Gamma \Rightarrow \Delta_1, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \wedge G, \Delta_2}\ (\wedge\text{-}der)$$

$$\frac{\Gamma_1, F, \Gamma_2 \Rightarrow \Delta \quad \Gamma_1, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \vee G, \Gamma_2 \Rightarrow \Delta}\ (\vee\text{-}izq) \qquad \frac{\Gamma \Rightarrow \Delta_1, F, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \vee G, \Delta_2}\ (\vee\text{-}der)$$

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow F, \Delta \quad \Gamma_1, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \to G, \Gamma_2 \Rightarrow \Delta}\ (\to\text{-}izq) \qquad \frac{F, \Gamma \Rightarrow \Delta_1, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \to G, \Delta_2}\ (\to\text{-}der)$$

$$\frac{\Gamma_1, F, G, \Gamma_2 \Rightarrow \Delta \quad \Gamma_1, \Gamma_2 \Rightarrow F, G, \Delta}{\Gamma_1, F \leftrightarrow G, \Gamma_2 \Rightarrow \Delta}\ (\leftrightarrow\text{-}izq)$$

$$\frac{F, \Gamma \Rightarrow \Delta_1, G, \Delta_2 \quad G, \Gamma \Rightarrow \Delta_1, F, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \leftrightarrow G, \Delta_2}\ (\leftrightarrow\text{-}der)$$

Donde Γ, Γ_1, Γ_2, Δ, Δ_1 *y* Δ_2 *son secuencias finitas de fórmulas.*

A la derecha de cada regla hemos indicado el nombre por el que se le hará referencia en el resto del capítulo. Cada regla consta de uno o dos secuentes (encima de la línea) llamados premisas, a partir de los cuales se infiere un único secuente (debajo de la línea) llamado conclusión. Gentzen también consideró reglas estructurales para eliminar fórmulas repetidas en la parte izquierda (derecha) de un secuente. Estas repeticiones no afectan a la satisfacibilidad de un secuente, sin embargo conviene eliminarlas a la hora de implementar un algoritmo basado en el cálculo G', pues llevan a repetir en varias ocasiones una misma tarea.

La figura 5.1 muestra un ejemplo de inferencia en el cálculo de secuentes. Los secuentes $\langle p, \neg q\rangle \Rightarrow \langle p\rangle$ y $\langle q\rangle \Rightarrow \langle q, \neg p\rangle$ son axiomas a partir de los que se obtienen $\langle \neg q\rangle \Rightarrow \langle \neg p, p\rangle$ y $\langle \neg q, q\rangle \Rightarrow \langle \neg p\rangle$ utilizando respectivamente las reglas de inferencia $\neg$-der y $\neg$-izq. A partir de estos dos secuentes y mediante la aplicación de la regla $\to$-der se obtienen, respectivamente, $\Rightarrow \langle p, (\neg q \to \neg p)\rangle$ y $\langle q\rangle \Rightarrow \langle \neg q \to \neg p\rangle$.

Estos dos secuentes son las premisas necesarias para la obtención del secuente $\langle p \to q \rangle \Rightarrow \langle \neg q \to \neg p \rangle$ mediante la regla de inferencia $\to$-izq.

El cálculo G' se utiliza para determinar la validez de un secuente considerando las reglas de inferencia hacia atrás. Esto es posible puesto que todos los elementos de las premisas de las reglas se pueden obtener a partir de la conclusión. Llamaremos reglas de expansión a las reglas del cálculo G' cuando son utilizadas hacia atrás, es decir, para determinar un conjunto de premisas a partir de las que se puede inferir un secuente. Las funciones `secuentes-expansion-izq` y `secuentes-expansion-der` utilizan las reglas de esta forma.

La función `secuentes-expansion-izq` recibe como argumentos una fórmula `F` y dos listas de fórmulas `Gamma` y `Delta`. Las listas de fórmulas `Gamma` y `Delta` representan respectivamente las partes izquierda y derecha de un secuente y `F` es una fórmula no atómica que aparece en `Gamma`. La función devuelve las premisas de las reglas de inferencia $\neg$-izq, $\wedge$-izq, $\vee$-izq, $\to$-izq y $\leftrightarrow$-izq, en función del tipo de fórmula que sea `F`. El resultado es devuelto como una lista de secuentes dado que algunas de las reglas tienen una única premisa y otras tienen dos. Para la implementación de esta función se han utilizado las funciones `elimina-una` y `añade-elemento`, 58. El uso de esta última evita incluir elementos repetidos en las partes izquierda y derecha de las premisas.

83

```
(defun secuentes-expansion-izq (F Gamma Delta)
  (let ((Gamma- (elimina-una F Gamma)))
    (cond ((es-negacion F)
           (list (cons Gamma- (añade-elemento (arg1 F) Delta))))
          ((es-conjuncion F)
           (list (cons (añade-elemento
                        (arg1 F) (añade-elemento
                                  (arg2 F) Gamma-)) Delta)))
          ((es-disyuncion F)
           (list (cons (añade-elemento (arg1 F) Gamma-) Delta)
                 (cons (añade-elemento (arg2 F) Gamma-) Delta)))
          ((es-implicacion F)
           (list (cons Gamma- (añade-elemento (arg1 F) Delta))
                 (cons (añade-elemento (arg2 F) Gamma-) Delta)))
          ((es-equivalencia F)
           (list (cons (añade-elemento
                        (arg1 F) (añade-elemento
                                  (arg2 F) Gamma-)) Delta)
                 (cons Gamma- (añade-elemento
                               (arg1 F) (añade-elemento
                                         (arg2 F) Delta)))))
          (t nil))))
```

La función `secuentes-expansion-der` se define de manera similar para las reglas de inferencia ¬-der, ∧-der, ∨-der, →-der y ↔-der:

84

```
(defun secuentes-expansion-der (F Gamma Delta)
  (let ((Delta- (elimina-una F Delta)))
    (cond ((es-negacion F)
           (list (cons (añade-elemento (arg1 F) Gamma) Delta-)))
          ((es-conjuncion F)
           (list (cons Gamma (añade-elemento (arg1 F) Delta-))
                 (cons Gamma (añade-elemento (arg2 F) Delta-))))
          ((es-disyuncion F)
           (list (cons Gamma (añade-elemento
                              (arg1 F) (añade-elemento
                                        (arg2 F) Delta-)))))
          ((es-implicacion F)
           (list (cons (añade-elemento (arg1 F) Gamma)
                       (añade-elemento (arg2 F) Delta-))))
          ((es-equivalencia F)
           (list (cons (añade-elemento (arg1 F) Gamma)
                       (añade-elemento (arg2 F) Delta-))
                 (cons (añade-elemento (arg2 F) Gamma)
                       (añade-elemento (arg1 F) Delta-))))
          (t nil))))
```

En el ejemplo de la figura 5.1, el secuente cuya validez se establece es $\langle p \rightarrow q\rangle \Rightarrow \langle \neg q \rightarrow \neg p\rangle$. A este secuente se le puede aplicar la regla de expansión →-izq para obtener los secuentes $\Rightarrow \langle p, (\neg q \rightarrow \neg p)\rangle$ y $\langle q\rangle \Rightarrow \langle \neg q \rightarrow \neg p\rangle$. Aplicando la regla de expansión →-der a cada uno de estos dos secuentes se obtienen respectivamente $\langle \neg q\rangle \Rightarrow \langle \neg p, p\rangle$ y $\langle \neg q, q\rangle \Rightarrow \langle \neg p\rangle$. Finalmente al aplicar la regla de expansión ¬-der al primero de ellos se obtiene el axioma $\langle p, \neg q\rangle \Rightarrow \langle p\rangle$ y al aplicar la regla de expansión ¬-izq al segundo, el axioma $\langle q\rangle \Rightarrow \langle q, \neg p\rangle$.

Teorema 5.6 *Si Pre es el conjunto de premisas de una de las reglas de inferencia del cálculo G' y $\Gamma \Rightarrow \Delta$ es la conclusión, entonces para toda asignación σ se verifica: $\sigma \models \Gamma \Rightarrow \Delta$ si y sólo si para todo secuente $S \in Pre$, $\sigma \models S$.*

Este teorema se demuestra fácilmente a partir del concepto de modelo de un secuente y las funciones de verdad de las conectivas proposicionales. La prueba es tediosa dado el número de casos distintos que tiene. Para formalizar este teorema se han utilizado cuatro eventos:

- `modelo-expansion-secuentes-izq-1` prueba el resultado para las reglas ¬-izq y ∧-izq.

- `modelo-expansion-secuentes-izq-2` prueba el resultado para las reglas $\vee$-izq, $\rightarrow$-izq y $\leftrightarrow$-izq.
- `modelo-expansion-secuentes-der-1` prueba el resultado para las reglas $\neg$-der, $\vee$-der y $\rightarrow$-der.
- `modelo-expansion-secuentes-der-2` prueba el resultado para las reglas $\wedge$-der y $\leftrightarrow$-der.

En 85 se muestran los dos primeros eventos, los otros dos han sido construidos de forma similar. Los dos casos se diferencian en el número de premisas de la regla de inferencia utilizada. Para distinguirlos se comprueba la longitud de la lista de premisas con la función `len`:

85

```
(defthm modelo-expansion-secuentes-izq-1
  (implies
   (and (equal (len (secuentes-expansion-izq F Gamma Delta)) 1)
        (es-proposicional F)
        (not (es-atomica F))
        (member-equal F Gamma))
   (let ((Premisas (secuentes-expansion-izq F Gamma Delta)))
     (iff (modelo-secuente sigma (car Premisas))
          (modelo-secuente sigma (cons Gamma Delta))))))

(defthm modelo-expansion-secuentes-izq-2
  (implies
   (and (equal (len (secuentes-expansion-izq F Gamma Delta)) 2)
        (es-proposicional F)
        (not (es-atomica F))
        (member-equal F Gamma))
   (let ((Premisas (secuentes-expansion-izq F Gamma Delta)))
     (iff (modelo-secuente sigma (cons Gamma Delta))
          (and (modelo-secuente sigma (car Premisas))
               (modelo-secuente sigma (cadr Premisas)))))))
```

5.1.2 Contramodelo de un secuente: $CONTRAMOD_s$

Puesto que las reglas de inferencia preservan la validez, se pueden utilizar para comprobar si un secuente es válido. Para ello basta con encontrar una forma de obtener dicho secuente aplicando reglas de inferencia a los axiomas. Es necesario observar que la aplicación de las reglas de inferencia se puede conmutar, es decir, si un secuente se puede obtener como consecuencia de aplicar dos reglas de inferencia distintas a un conjunto de premisas, entonces da igual el orden en que se utilicen dichas reglas. Esto se debe a que no hay dos reglas de inferencia que sirvan para

$$\dfrac{\dfrac{\dfrac{\langle p, \neg q\rangle \Rightarrow \langle p\rangle}{\langle \neg q\rangle \Rightarrow \langle p, \neg p\rangle}\ \neg\text{-der} \qquad \dfrac{\langle q\rangle \Rightarrow \langle q, \neg p\rangle}{\langle \neg q, q\rangle \Rightarrow \langle \neg p\rangle}\ \neg\text{-izq}}{\langle \neg q, (p \to q)\rangle \Rightarrow \langle \neg p\rangle}\ \to\text{-izq}}{\langle p \to q\rangle \Rightarrow \langle \neg q \to \neg p\rangle}\ \to\text{-der}$$

Figura 5.2: Conmutación de reglas en G'

obtener la misma fórmula dentro de un secuente y a que las reglas de inferencia sólo afectan a la fórmula que se infiere, el resto no son alteradas. En el ejemplo de la figura 5.1 se pueden conmutar las aplicaciones de las reglas $\to$-izq y $\to$-der y se obtiene la derivación mostrada en la figura 5.2, en la que el conjunto de premisas a partir de las que se aplican dichas reglas no cambia.

Si el proceso para determinar la validez de un secuente falla, obtendremos un secuente en el que todas las fórmulas son atómicas y que no es un axioma. Estos secuentes no se pueden obtener como resultado de aplicar una regla de inferencia. A partir de los elementos de este tipo de secuentes se puede construir un contramodelo de los mismos:

Definición 5.7 *Un secuente $\Gamma \Rightarrow \Delta$ es atómico si Γ y Δ son secuencias finitas de fórmulas atómicas.*

Teorema 5.8 *Sea $\Gamma \Rightarrow \Delta$ un secuente atómico que no es un axioma. La asignación σ definida de forma que $\sigma(p) = \top$ si y sólo si p aparece en Γ, no es modelo de $\Gamma \Rightarrow \Delta$.*

Demostración:

Puesto que σ es modelo de todos los elementos de Γ, para que sea modelo de $\Gamma \Rightarrow \Delta$ tendría que ser modelo de algún elemento de Δ, pero esto no ocurre puesto que Δ y Γ no tienen elementos en común. Luego $\sigma \not\models \Gamma \Rightarrow \Delta$.

□

Dado que las reglas de inferencia del cálculo G' preservan la validez, la asignación así construida a partir de un secuente atómico S que no sea un axioma, será contramodelo de cualquier secuente obtenido mediante una aplicación de las reglas de inferencia, en las que se utilice S entre las premisas.

De esta forma se dispone de un procedimiento que comprueba si un secuente es válido y, en caso de que esto no sea cierto, proporciona información suficiente para construir un contramodelo del mismo. El siguiente algoritmo aplica las reglas de G' como reglas de expansión, hasta encontrar un secuente a partir del cual se puede construir un contramodelo.

Algoritmo 5.9 *($CONTRAMOD_s$) El dato de entrada de este algoritmo es un secuente $\Gamma \Rightarrow \Delta$ y actúa como se describe a continuación:*

1. *Si $\Gamma \Rightarrow \Delta$ es una axioma, el algoritmo termina y devuelve* **f**.
2. *Se selecciona una fórmula F no atómica de la lista Γ y se utiliza la regla de expansión para la ocurrencia de F en el lado izquierdo del secuente.*
   (a) *Si la regla de expansión utilizada tiene una única premisa S, se evalúa el algoritmo sobre dicha premisa.*
   (b) *Si la regla de expansión utilizada tiene dos premisas S_1 y S_2, se evalúa el algoritmo sobre una de ellas, S_1, y si devuelve* **f** *entonces se evalúa sobre la otra, S_2.*
3. *Si en Γ no hay fórmulas no atómicas, se selecciona una fórmula F no atómica de la lista Δ y se utiliza la regla de expansión para la ocurrencia de F en el lado derecho del secuente.*
   (a) *Si la regla de expansión utilizada tiene una única premisa S, se evalúa el algoritmo sobre dicha premisa.*
   (b) *Si la regla de expansión utilizada tiene dos premisas S_1 y S_2, se evalúa el algoritmo sobre una de ellas, S_1, y si devuelve* **f** *entonces se evalúa sobre la otra, S_2.*
4. *Si Γ y Δ no tienen fórmulas no atómicas, se devuelve el secuente $\Gamma \Rightarrow \Delta$.*

Este algoritmo no está completamente determinado, puesto que no se indica cómo se escoge la fórmula F, ya sea de Γ o de Δ. Para resolver esta indeterminación hemos considerado una función `seleccion`, que escoge una fórmula no atómica de una lista de fórmulas, si es que existe. Si la lista no contiene fórmulas no atómicas devuelve `nil`. Se ha asumido la existencia de esta función en un encapsulado ACL2, caracterizándola por las siguientes propiedades:

- Si `(seleccion Gamma)` devuelve un valor, entonces dicho valor pertenece a `Gamma`.
- Si la función `seleccion` devuelve un valor, entonces dicho valor no es una formula atómica.
- Si `(seleccion Gamma)` no devuelve ningún valor, entonces todos los elementos de `Gamma` son fórmulas atómicas.

De esta forma, el resto del desarrollo presentado en esta sección es genérico, en el sentido de que es válido para cualquier función `seleccion` verificando estas propiedades. La formalización de estas propiedades es la siguiente:

86

```
(defthm si-existe-no-atomica-seleccion-pertenece
  (implies (seleccion Gamma)
           (member-equal (seleccion Gamma) Gamma)))

(defthm si-existe-no-atomica-seleccion-no-es-atomica
  (implies (seleccion Gamma)
           (not (es-atomica (seleccion Gamma)))))

(defthm si-no-seleccion-miembros-atomicos
  (implies (and (not (seleccion Gamma))
                (lista-formulas Gamma)
                (member-equal F Gamma))
           (es-atomica F)))
```

La función que implementa el algoritmo $CONTRAMOD_s$ es la siguiente:

87

```
(defun CONTRAMOD-secuentes (S)
  (declare (xargs :measure (medida-secuente S)))
  (cond ((endp S) nil)
    ((secuente-axioma S) nil)
    (t (let* ((Gamma (car S))
              (Delta (cdr S))
              (F-1 (seleccion Gamma))
              (F-2 (seleccion Delta)))
         (cond
          (F-1 (let ((Premisas
                       (secuentes-expansion-izq F-1 Gamma Delta)))
                 (cond ((equal (len Premisas) 1)
                        (CONTRAMOD-secuentes (car Premisas)))
                       ((equal (len Premisas) 2)
                        (or (CONTRAMOD-secuentes (car Premisas))
                          (CONTRAMOD-secuentes (cadr Premisas))))
                       (t S))))
          (F-2 (let ((Premisas
                       (secuentes-expansion-der F-2 Gamma Delta)))
                 (cond ((equal (len Premisas) 1)
                        (CONTRAMOD-secuentes (car Premisas)))
                       ((equal (len Premisas) 2)
                        (or (CONTRAMOD-secuentes (car Premisas))
                          (CONTRAMOD-secuentes (cadr Premisas))))
                       (t S))))
          (t S))))))
```

Utilizando una función `seleccion` concreta, por ejemplo la que devuelve la primera fórmula no atómica de una lista, podemos utilizar la función anterior para comprobar la existencia de un contramodelo de un secuente. Si el secuente tiene contramodelos, la función devuelve un secuente atómico a partir del que se puede construir uno. Si el secuente no tiene contramodelos la función devuelve `nil`.

```
ACL2 !>(NO-MOD-secuentes '( ( (-> p q) ) (-> (- p) (- q)) ))
((Q) P)
ACL2 !>(NO-MOD-secuentes '( ( (-> p q) ) (-> (- q) (- p)) ))
NIL
```

En el primer ejemplo el secuente $\langle p \to q\rangle \Rightarrow \langle \neg p \to \neg q\rangle$ tiene un contramodelo. Cualquier asignación en la que q sea cierta y p falsa es contramodelo de dicho secuente. En el segundo ejemplo, el secuente $\langle p \to q\rangle \Rightarrow \langle \neg p \to \neg q\rangle$ no tiene contramodelos.

Para que la función `CONTRAMOD-secuentes` sea admitida por ACL2, se ha de demostrar que termina para cualquier dato de entrada. Para ello hemos proporcionado una medida del argumento, `medida-secuente`, que decrece en las llamadas recursivas.

Definición 5.10 *El número de conectivas de un secuente $r_s(\Gamma \Rightarrow \Delta)$ es la suma del número de conectivas que aparecen en todas las fórmulas que lo componen. Así, $r_s(\langle F_1, \ldots, F_n\rangle \Rightarrow \langle G_1, \ldots, G_m\rangle) = r(F_1)+\cdots+r(F_n)+r(G_1)+\cdots+r(G_m)$.*

La función `medida-secuente` formaliza este concepto. Para ello se utiliza la función `medida-lista-formulas` que suma el número de conectivas de todas las fórmulas que aparecen en una lista:

88

```
(defun medida-secuente (S)
  (+ (medida-lista-formulas (car S))
     (medida-lista-formulas (cdr S))))

(defun medida-lista-formulas (Gamma)
  (cond ((endp Gamma) 0)
        (t (+ (numero-conectivas (car Gamma))
              (medida-lista-formulas (cdr Gamma))))))
```

Teorema 5.11 *Si el secuente S' es obtenido como premisa al utilizar una regla de inferencia para concluir otro secuente S, entonces $r_s(S') < r_s(S)$.*

El evento asociado a este teorema se genera de forma automática al evaluar el evento *87*. Su demostración se obtiene fácilmente dado que el número de conectivas de las subfórmulas de una fórmula proposicional compuesta es menor que el de la fórmula original, resultado establecido en *7*.

5.1.3 Corrección y completitud de $CONTRAMOD_s$

Teorema 5.12 (Corrección de $CONTRAMOD_s$) *Dado un secuente $\Gamma \Rightarrow \Delta$, si $CONTRAMOD_s(\Gamma \Rightarrow \Delta) = S \neq \mathbf{f}$ entonces $\Gamma \Rightarrow \Delta$ no es un secuente válido. Además, si σ es la asignación definida de forma que $\sigma(p) = \top$ si y sólo si p aparece en la parte izquierda de S, entonces $\sigma \not\models \Gamma \Rightarrow \Delta$.*

Demostración:

Si $CONTRAMOD_s(\Gamma \Rightarrow \Delta) = S \neq \mathbf{f}$ entonces S es un secuente atómico y por el teorema 5.8, se tiene que la asignación σ definida en el enunciado no es modelo de S. Por el teorema 5.6, esta asignación no es modelo de ningún secuente obtenido al aplicar una secuencia de reglas de inferencia, en las que se utilice S entre las premisas. Al obtener S como resultado, el algoritmo nos asegura que dicho secuente aparece como premisa en una secuencia de reglas de inferencia que concluye con $\Gamma \Rightarrow \Delta$. Por tanto, σ no es modelo de $\Gamma \Rightarrow \Delta$.

□

La función `genera-contramodelo-secuente` construye la asignación mencionada en el teorema anterior. Para definir esta función consideramos la función `genera-asignacion-positiva` que construye una asignación en la que todos los símbolos proposicionales de una lista son verdaderos.

89

```
(defun genera-contramodelo-secuente (S)
  (genera-asignacion-positiva (car S)))

(defun genera-asignacion-positiva (atomos)
  (cond ((endp atomos) nil)
        ((es-simbolo-proposicional (car atomos))
         (asume-valor (car atomos) *V*
                      (genera-asignacion-positiva (cdr atomos))))
        (t (genera-asignacion-positiva (cdr atomos)))))
```

La formalización del teorema de corrección es la siguiente:

90

```
(defthm correccion-CONTRAMOD-secuentes
  (implies (and (es-secuente S)
                (CONTRAMOD-secuentes S))
           (not (modelo-secuente (genera-contramodelo-secuente
                                  (CONTRAMOD-secuentes S))
                                 S))))
```

Teorema 5.13 (Completitud de $CONTRAMOD_s$) *Dado un secuente $\Gamma \Rightarrow \Delta$, si existe una asignación σ que no es modelo de $\Gamma \Rightarrow \Delta$, entonces se tiene $CONTRAMOD_s(\Gamma \Rightarrow \Delta) \neq \mathbf{f}$.*

La prueba de este resultado se obtiene fácilmente a partir del teorema 5.6. Su formalización es la siguiente:

91
```
(defthm completitud-CONTRAMOD-secuentes
  (implies (and (es-secuente S)
                (not (modelo-secuente sigma S)))
           (CONTRAMOD-secuentes S))
```

5.1.4 Demostrabilidad por secuentes: DAT_s

Teorema 5.14 *Dado una fórmula $F \in \mathbb{P}(\Sigma)$, una asignación σ es modelo de F si y sólo si es modelo del secuente $\Rightarrow F$.*

La prueba de este teorema es trivial a partir del concepto de modelo de un secuente. Este teorema permite utilizar el cálculo de secuentes para decidir la validez de una fórmula F, para ello basta con comprobar que el secuente $\Rightarrow F$ es válido, es decir, no tiene contramodelos.

Definición 5.15 *Una fórmula F es demostrable por secuentes, y lo notaremos $\vdash_s F$, si, aplicando las reglas de expansión de G', no se puede obtener un contramodelo del secuente $\Rightarrow F$.*

Algoritmo 5.16 (DAT_s) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $DAT_s(F)$ es* **t** *si y sólo si $CONTRAMOD_s(\Rightarrow F)$ es* **f**.

La función que implementa este algoritmo es la siguiente:

92
```
(defun DAT-secuentes (F)
  (not (CONTRAMOD-secuentes (cons nil (list F)))))
```

Esta función devuelve `T` si la fórmula que se le pasa como argumento es demostrable por secuentes. En caso contrario devuelve `nil`.

```
ACL2 !>(DAT-secuentes '(<-> (<-> p q) (<-> q p)))
T
ACL2 !>(DAT-secuentes '(-> (-> p q) (& r s)))
NIL
```

En el primer ejemplo la fórmula $(p \leftrightarrow q) \leftrightarrow (q \leftrightarrow p)$ es demostrable por secuentes, es decir, es válida. En el segundo, la fórmula $(p \rightarrow q) \rightarrow (r \wedge s)$ no es demostrable por secuentes.

Teorema 5.17 *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, F es válida si y sólo si $DAT_s(F)$ es* **t**.

Demostración:

Por definición $DAT_s(F) \neq \mathbf{t}$ si y sólo si $CONTRAMOD_s(\Rightarrow F) \neq \mathbf{f}$ y, por los teoremas de corrección y completitud de $CONTRAMOD_s$, esto ocurre si y sólo si $\Rightarrow F$ tiene un contramodelo. Luego $DAT_s(F) = \mathbf{t}$ si y sólo si $\Rightarrow F$ no tiene contramodelos, es decir es válido, lo que, por el teorema 5.14, es equivalente a que F sea válida

□

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo DAT_s. La condición acerca de la validez de la fórmula `F` se establece como se indicó en la sección 3.2.4.

93
```
(defthm correccion-DAT-secuentes
  (implies (and (es-proposicional F)
                (DAT-secuentes F))
           (modelo sigma F)))

(defthm completitud-DAT-secuentes
  (implies
    (and (es-proposicional F)
         (not (DAT-secuentes F)))
    (not (modelo (genera-contramodelo-secuente
                   (CONTRAMOD-secuentes (cons nil (list F))))
                 F))))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en *91* y *90* cuando el valor de la variable `S` es `(cons nil (list F))`.

5.1.5 Satisfacibilidad por secuentes: SAT_s

Teorema 5.18 *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, F es satisfacible si y sólo si el secuente $F \Rightarrow$ tiene un contramodelo.*

Este teorema es una consecuencia del teorema 5.14 y permite utilizar el cálculo de secuentes para obtener un modelo de una fórmula F, para ello basta con generar un contramodelo del secuente $F \Rightarrow$.

Algoritmo 5.19 (SAT_s) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $SAT_s(F)$ es igual a $CONTRAMOD_s(F \Rightarrow)$.*

La función que implementa este algoritmo es la siguiente:

94
```
(defun SAT-secuentes (F)
  (CONTRAMOD-secuentes (cons (list F) nil)))
```

Si una fórmula es satisfacible, esta función devuelve un secuente atómico a partir del que se puede obtener un modelo de dicha fórmula. Si la fórmula es insatisfacible, la función devuelve `nil`:

```
ACL2 !>(SAT-secuentes '(& (-> p (- q)) (- (- p))))
((P) Q)
ACL2 !>(SAT-secuentes '(& (/ p q) (& (- p) (- q))))
NIL
```

En el primer ejemplo, la fórmula $(p \rightarrow \neg q) \wedge \neg\neg p$ es satisfacible y una asignación que la hace cierta es cualquiera en la que p sea cierto y q falso. En el segundo ejemplo, la fórmula $(p \vee q) \wedge (\neg p \wedge \neg q)$ es insatisfacible.

Teorema 5.20 *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, F es satisfacible si y sólo si $SAT_{\mathcal{S}}(F) = S \neq \mathbf{f}$. Además, si $SAT_{\mathcal{S}}(F) = S \neq \mathbf{f}$ y σ es la asignación definida de forma que $\sigma(p) = \top$ si y sólo si p está en la parte izquierda del secuente S, entonces $\sigma \models F$.*

Demostración:

Por definición $SAT_{\mathcal{S}}(F) = S \neq \mathbf{f}$ si y sólo si $CONTRAMOD_{\mathcal{S}}(F \Rightarrow) = S \neq \mathbf{f}$. Por los resultados de corrección y completitud de $CONTRAMOD_{\mathcal{S}}$ esto ocurre si y sólo si el secuente $F \Rightarrow$ tiene un contramodelo y esto, por el teorema 5.18, es equivalente a que F sea satisfacible. Si $SAT_{\mathcal{S}}(F) = S \neq \mathbf{f}$ y σ es la asignación definida en el enunciado, entonces por el teorema de corrección de $CONTRAMOD_{\mathcal{S}}$, σ es contramodelo de $F \Rightarrow$, luego, por la definición de modelo de un secuente, σ es modelo de F.

□

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $SAT_{\mathcal{S}}$. La condición acerca de la satisfacibilidad de la fórmula `F` se establece como se indicó en la sección 3.2.4.

95

```
(defthm correccion-SAT-secuentes
  (implies (and (es-proposicional F)
                (SAT-secuentes F))
           (modelo (genera-contramodelo-secuente
                     (SAT-secuentes F)) F)))

(defthm completitud-SAT-secuentes
  (implies (and (es-proposicional F)
                (modelo sigma F))
           (SAT-secuentes F)))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en *90* y *91* cuando el valor de la variable `S` es `(cons (list F) nil)`.

5.1.6 Deducibilidad por secuentes: $DEDUC_s$

Finalmente, también se puede utilizar el cálculo de secuentes para comprobar si una fórmula F es consecuencia lógica de un conjunto de hipótesis $H_1, \ldots, H_n$. Para ello basta con comprobar que el secuente $\langle H_1, \ldots, H_n \rangle \Rightarrow F$ es válido, es decir, que no tiene contramodelos.

Definición 5.21 *Dadas las fórmulas $F, H_1, \ldots, H_n$, decimos que F es deducible por secuentes a partir de las hipótesis $H_1 \ldots, H_n$, y lo notaremos $H_1, \ldots, H_n \vdash_s F$, si, aplicando las reglas de expansión de G', no se puede obtener un contramodelo del secuente $H_1, \ldots, H_n \Rightarrow F$.*

Algoritmo 5.22 ($DEDUC_s$) *Dadas las fórmulas F, H_1, $\ldots$, $H_n \in \mathbb{P}(\Sigma)$, $DEDUC_s(\langle H_1, \ldots, H_n \rangle, F)$ es* $\mathbf{t}$ *si y sólo si $CONTRAMOD_s(\langle H_1, \ldots, H_n \rangle \Rightarrow F)$ es* $\mathbf{f}$.

La función que implementa este algoritmo es la siguiente:

96
```
(defun DEDUC-secuentes (lista-H F)
  (not (CONTRAMOD-secuentes (cons lista-H (list F)))))
```

Esta función devuelve `T` si la fórmula `F` es deducible por secuentes a partir de las hipótesis `lista-H`. En caso contrario devuelve `nil`.

```
ACL2 !>(DEDUC-secuentes '((-> p q) p) 'q)
T
ACL2 !>(DEDUC-secuentes '((-> p q) p) '(- q))
NIL
```

En el primer ejemplo la fórmula q es deducible por secuentes a partir de las fórmulas $(p \rightarrow q)$ y p. En el segundo ejemplo, la fórmula $\neg q$ no es deducible por secuentes a partir de las fórmulas $(p \rightarrow q)$ y p.

Teorema 5.23 *Dadas las fórmulas $F, H_1 \ldots, H_n \in \mathbb{P}(\Sigma)$, F es consecuencia lógica de $\{H_1, \ldots, H_n\}$ si y sólo si $DEDUC_s(\langle H_1, \ldots, H_n \rangle, F)$ es* $\mathbf{t}$.

Demostración:

Por la definición de $DEDUC_s$ se tiene $DEDUC_s(\langle H_1, \ldots, H_n \rangle, F) \neq \mathbf{t}$ si y sólo si $CONTRAMOD_s(\langle H_1, \ldots, H_n \rangle \Rightarrow F) \neq \mathbf{f}$ y, por los teoremas de corrección y completitud de $CONTRAMOD_s$, esto ocurre si y sólo si el secuente

$\langle H_1, \ldots, H_n \rangle \Rightarrow F$ tiene un contramodelo, es decir, si y sólo si existe una asignación σ modelo de los H_i que no es modelo de F. Finalmente esto es equivalente a que F no sea consecuencia lógica de $\{H_1, \ldots, H_n\}$.

Por tanto F es consecuencia lógica del conjunto de hipótesis $\{H_1, \ldots, H_n\}$ si y sólo si $DEDUC_{\mathcal{S}}(\langle H_1, \ldots, H_n \rangle, F)$ es $\mathbf{t}$.

□

Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $DEDUC_{\mathcal{S}}$:

97
```
(defthm correccion-DEDUC-secuentes
  (implies (and (lista-formulas lista-H)
                (es-proposicional F)
                (DEDUC-secuentes lista-H F)
                (modelo-conjuncion sigma lista-H))
           (modelo sigma F)))

(defthm completitud-DEDUC-secuentes
  (let ((sigma (genera-contramodelo-secuente
                (CONTRAMOD-secuentes (cons lista-H (list F))))))
    (implies (and (lista-formulas lista-H)
                  (es-proposicional F)
                  (not (DEDUC-secuentes lista-H F)))
             (and (modelo-conjuncion sigma lista-H)
                  (not (modelo sigma F))))))
```

Las pruebas de estos resultados se obtienen respectivamente como instancias de los eventos presentados en *91* y *90* cuando el valor de la variable `S` es `(cons lista-H (list F))`.

5.1.7 Secuentes en $\mathbb{K}$

Los algoritmos desarrollados en las subsecciones previas también pueden ser utilizados para decidir la satisfacibilidad, validez y consecuencia lógica en $\mathbb{K}$. Los algoritmos son los mismos, puesto que no se basan en aspectos semánticos, pero no ocurre lo mismo con los conceptos y propiedades que aseguran su corrección y completitud.

La semántica en $\mathbb{K}$ de un secuente es similar a la semántica clásica: un secuente $\Gamma \Rightarrow \Delta$ es cierto en una $\mathbb{K}$-asignación si alguna de las fórmulas de Γ es falsa o alguna de las fórmulas de Δ es cierta. Sin embargo, el método basado en secuentes busca un contramodelo, de ahí que sea necesario caracterizar las situaciones en las que una asignación hace falso un secuente. En la semántica clásica basta con

utilizar la negación de la función `modelo-secuente`, esto no sirve en la semántica de Kleene pues no deja fuera el valor indeterminado.

Definición 5.24 *Una $\mathbb{K}$-asignación σ es no–modelo de un secuente $\Gamma \Rightarrow \Delta$ si σ es modelo de todos los elementos de Γ y de las negaciones de todos los elementos de Δ.*

La función `no-modelo-secuente-K` formaliza este concepto, para ello utiliza las funciones `modelo-conjuncion-K` y `no-modelo-conjuncion-K` que comprueban si una $\mathbb{K}$-asignación es modelo, respectivamente no–modelo, de todos los elementos de una lista:

98
```
(defun no-modelo-secuente-K (sigma S)
  (and (modelo-conjuncion-K sigma (car S))
       (no-modelo-conjuncion-K sigma (cdr S))))

(defun modelo-conjuncion-K (sigma Gamma)
  (cond ((endp Gamma) t)
        (t (and (modelo-K sigma (car Gamma))
                (modelo-conjuncion-K sigma (cdr Gamma))))))

(defun no-modelo-conjuncion-K (sigma Delta)
  (cond ((endp Delta) t)
        (t (and (no-modelo-K sigma (car Delta))
                (no-modelo-conjuncion-K sigma (cdr Delta))))))
```

Un secuente axioma es un secuente en el que alguna fórmula aparece simultáneamente en la parte izquierda y en la parte derecha. De esta forma no existe ninguna $\mathbb{K}$-asignación que sea no–modelo de un secuente axioma. La formalización de esta propiedad es la siguiente:

99
```
(defthm secuente-axioma-no-tiene-no-modelos
   (implies (secuente-axioma S)
            (not (no-modelo-secuente-K sigma S))))
```

Las reglas del cálculo G' no hacen referencia a la semántica de los secuentes, por tanto se aplican de igual forma tanto en la semántica clásica como en la semántica de Kleene. Es por esto por lo que utilizamos las funciones `secuentes-expansion-izq` y `secuentes-expansion-der`, presentadas en *83* y *84*, para formalizar el cálculo de secuentes en $\mathbb{K}$. Sin embargo, las propiedades semánticas de estas funciones han de ser reformuladas con respecto al concepto de no–modelo de un secuente: una $\mathbb{K}$-asignación es no–modelo de un secuente conclusión de una regla de expansión si y sólo si es no–modelo

de alguna de las premisas. La formalización de esta propiedad para la función `modelo-expansion-secuentes-izq` es la siguiente:

100
```
(defthm modelo-expansion-secuentes-izq-1
  (implies
   (and (equal (len (secuentes-expansion-izq F Gamma Delta)) 1)
        (es-proposicional F)
        (not (es-atomica F))
        (member-equal F Gamma))
   (let ((Premisas (secuentes-expansion-izq F Gamma Delta)))
     (iff (no-modelo-secuente-K sigma (car Premisas))
          (no-modelo-secuente-K sigma (cons Gamma Delta))))))

(defthm modelo-expansion-secuentes-izq-2
  (implies
   (and (equal (len (secuentes-expansion-izq F Gamma Delta)) 2)
        (es-proposicional F)
        (not (es-atomica F))
        (member-equal F Gamma))
   (let ((Premisas (secuentes-expansion-izq F Gamma Delta)))
     (iff (no-modelo-secuente-K sigma (cons Gamma Delta))
          (or (no-modelo-secuente-K sigma (car Premisas))
              (no-modelo-secuente-K sigma (cadr Premisas)))))))
```

Los eventos para la función `modelo-expansion-secuentes-der` se construyen de forma similar.

5.1.7.1 No–modelo en $\mathbb{K}$ de un secuente

Los algoritmos para decidir la validez, satisfacibilidad y consecuencia lógica presentados a lo largo de este capítulo, se basan en un procedimiento que busca un contramodelo de un secuente, es decir, una asignación que lo hace falso. Este procedimiento, al que hemos llamado `CONTRAMOD-secuentes` se define en la semántica de Kleene de igual forma a como se ha hecho en la semántica clásica. Su formalización se encuentra en *87*.

Los resultados de corrección y completitud de este procedimiento en la semántica de Kleene se establecen de forma similar al caso de la semántica clásica. Su formalización es la siguiente:

101
```
(defthm correccion-CONTRAMOD-secuentes
  (implies
    (and (es-secuente S)
         (CONTRAMOD-secuentes S))
    (no-modelo-secuente-K
      (genera-no-modelo-secuente-K (CONTRAMOD-secuentes S)) S)))

(defthm completitud-CONTRAMOD-secuentes
  (implies (and (es-secuente S)
                (no-modelo-secuente-K sigma S))
           (CONTRAMOD-secuentes S)))
```

En este caso hemos utilizado la función `no-modelo-secuente-K` para comprobar que una $\mathbb{K}$-asignación es no–modelo de un secuente. Obsérvese que en los correspondientes teoremas en la semántica clásica se utiliza la negación del predicado `modelo-secuente`.

102
```
(defun genera-no-modelo-secuente-K (S)
  (cond ((es-secuente S)
         (append (genera-asignacion-positiva-K (car S))
                 (genera-asignacion-negativa-K (cdr S))))
        (t nil)))

(defun genera-asignacion-positiva-K (Xs)
  (cond ((endp Xs) nil)
        ((es-simbolo-proposicional (car Xs))
         (asume-valor (car Xs) *V*
                      (genera-asignacion-positiva-K (cdr Xs))))
        (t (genera-asignacion-positiva-K (cdr Xs)))))

(defun genera-asignacion-negativa-K (Xs)
  (cond ((endp Xs) nil)
        ((es-simbolo-proposicional (car Xs))
         (asume-valor (car Xs) *F*
                      (genera-asignacion-negativa-K (cdr Xs))))
        (t (genera-asignacion-negativa-K (cdr Xs)))))
```

La función `genera-no-modelo-secuente-K` proporciona un no–modelo de un secuente atómico que no es un axioma. En el caso de la semántica clásica es suficiente con asegurar que todos los átomos del lado izquierdo del secuente son verdaderos. En la semántica de Kleene tenemos que asignar explícitamente un valor de verdad a todos los átomos del secuente para evitar que quede indefinido.

La función `genera-no-modelo-secuente-K` realiza esta tarea, para ello utiliza las funciones `genera-asignacion-positiva-K` y `genera-asignacion-negativa-K` que construyen el trozo de $\mathbb{K}$-asignación que hace verdaderos, respectivamente falsos, todas las fórmulas del lado izquierdo del secuente, respectivamente el lado derecho.

5.1.7.2 Decisión de $\mathbb{K}$-validez basada en secuentes

El procedimiento para decidir si una fórmula es $\mathbb{K}$-válida es el mismo que el de la semántica clásica. La función `DAT-secuentes`, presentada en 92, implementa este procedimiento. Los resultados de corrección y completitud son los siguientes:

103

```
(defthm correccion-DAT-secuentes
  (implies (and (es-proposicional F)
                (DAT-secuentes F)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-DAT-secuentes
  (implies
    (and (es-proposicional F)
         (not (DAT-secuentes F)))
    (no-modelo-K (genera-no-modelo-secuente-K
                  (CONTRAMOD-secuentes (cons nil (list F)))) F)))
```

Como se indicó en la sección 3.3.4 el resultado de corrección tiene una hipótesis adicional en la que se afirma que la asignación `sigma` asigna un valor de verdad a la fórmula `F`. En el resultado de completitud la conclusión afirma la existencia de una $\mathbb{K}$-asignación que es no–modelo de la fórmula `F`.

5.1.7.3 Decisión de $\mathbb{K}$-satisfacibilidad basada en secuentes

El algoritmo para decidir la $\mathbb{K}$-satisfacibilidad es el mismo que el de la semántica clásica. La función que lo implementa es `SAT-secuentes`, presentada en 94. La formalización y prueba de sus propiedades de corrección y completitud es similar a las de la semántica clásica. Los eventos son los siguientes:

104
```
(defthm correccion-SAT-secuentes
  (implies (and (es-proposicional F)
                (SAT-secuentes F))
           (modelo-K (genera-no-modelo-secuente-K
                       (SAT-secuentes F)) F)))

(defthm completitud-SAT-secuentes
  (implies (and (es-proposicional F)
                (modelo-K sigma F))
           (SAT-secuentes F)))
```

5.1.7.4 Decisión de consecuencia $\mathbb{K}$-lógica basada en secuentes

Al igual que en los dos casos anteriores, el algoritmo para decidir la consecuencia lógica en $\mathbb{K}$ es el mismo que para la semántica clásica. La función que lo implementa, `DEDUC-secuentes`, está definida en *96*. Las resultados de corrección y completitud se han construido siguiendo las indicaciones de la sección 3.3.4.

105
```
(defthm correccion-DEDUC-secuentes
  (implies (and (lista-formulas lista-H)
                (es-proposicional F)
                (DEDUC-secuentes lista-H F)
                (modelo-conjuncion-K sigma lista-H)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-DEDUC-secuentes
  (let ((sigma (genera-no-modelo-secuente-K
                (CONTRAMOD-secuentes (cons lista-H (list F))))))
    (implies (and (lista-formulas lista-H)
                  (es-proposicional F)
                  (not (DEDUC-secuentes lista-H F)))
             (and (modelo-conjuncion-K sigma lista-H)
                  (no-modelo-K sigma F)))))
```

5.2 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, presentando datos sobre su evaluación en algunas fórmulas de Urquhart [83] y fórmulas representando el problema de las N reinas [62]. En la sección 4.3 se indica como se cargan en ACL2 los ficheros en los que se definen estas familias de fórmulas.

Una vez certificados los libros, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `5-secuentes`:

```
../calculos-proposicionales/5-secuentes> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/5-secuentes".

ACL2 !>
```

A continación incluimos el libro que contiene la teoría desarrollada en este capítulo. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "secuentes")
```

Para poder utilizar los procedimientos de decisión desarrollados en este capítulo, tenemos que proporcionar una definición concreta de la función que selecciona una fórmula de un secuente. El símbolo asociado a esta función ya existe en el sistema por lo que tendremos que redefinirla. Las propiedades de los procedimientos de decisión de satisfacibilidad, validez y consecuencia lógica se mantienen siempre que la nueva definición de la función `seleccion` tenga las propiedades presentadas en *86*.

```
ACL2 !>(set-ld-redefinition-action '(:warn . :overwrite) state)

ACL2 !>(defun seleccion (lista)
         (cond ((endp lista) nil)
               ((es-atomica (car lista)) (seleccion (cdr lista)))
               (t (car lista))))
```

En la tabla 5.1 se muestran los tiempos de evaluación de la función `DAT-secuentes` para las 15 primeras fórmulas de Urquhart. En la tabla 5.2 se muestra la misma información sobre la evaluación de la función `SAT-secuentes` para las fórmulas correspondientes al problema de las N reinas, para algunos valores de N.

Fórmula	tiempo	Fórmula	tiempo	Fórmula	tiempo
U_1	0.000	U_6	0.010	U_{11}	1.380
U_2	0.000	U_7	0.050	U_{12}	3.170
U_3	0.000	U_8	0.110	U_{13}	7.370
U_4	0.000	U_9	0.230	U_{14}	16.650
U_5	0.010	U_{10}	0.560	U_{15}	37.490

Tabla 5.1: Tiempos de evaluación de las fórmulas de Urquhart

Tamaño del tablero	2	3	4	5	6
Tiempo	0.000	0.050	0.420	1.900	170.450

Tabla 5.2: Tiempos de evaluación para el problema de las N reinas

Sumario

En este capítulo:

- Se ha formalizado el cálculo de secuentes. Basándonos en este cálculo se ha desarrollado un procedimiento que implementa un algoritmo para decidir la validez de un secuente, mediante la búsqueda de un contramodelo. Se han demostrado las propiedades de corrección y completitud en el sentido clásico de este procedimiento.

- Basándonos en el procedimiento de búsqueda de un contramodelo de un secuente se han desarrollado procedimientos para decidir la validez, satisfacibilidad y consecuencia lógica. Se han demostrado las propiedades de corrección y completitud en el sentido clásico de estos procedimientos.

- Se ha demostrado que los algoritmos desarrollados también pueden ser utilizados en la semántica de Kleene, presentado los eventos que establecen sus propiedades de corrección y completitud en $\mathbb{K}$.

Capítulo 6

Extensiones de ACL2

ACL2 es un sistema abierto en el que cualquier usuario puede desarrollar herramientas para facilitar su trabajo. Desde cierto punto de vista, estas herramientas son extensiones del sistema que pueden llegar a integrarse con él. Nos separamos momentáneamente del desarrollo de la teoría sobre cálculos proposicionales, para presentar dos de estas herramientas que hemos desarrollado y que serán de utilidad en los capítulos posteriores.

En la primera sección presentamos una herramienta para facilitar las pruebas de terminación de funciones recursivas compleja. Esta herramienta proporciona una determinada extensión de una relación bien fundamentada definida sobre un conjunto A, que es de utilidad en la prueba de terminación de funciones que hacen recursión sobre una lista de elementos de dicho conjunto. El desarrollo de esta herramienta reveló un pequeño error en el sistema subsanado en la versión actual del mismo. La herramienta fue presentada en [67] y actualmente se distribuye junto con el sistema.

En la segunda sección se presenta una herramienta que facilita la reutilización de resultados de segundo orden. Para establecer un resultado de segundo orden en el sistema, se consideran determinadas funciones para las que se asumen ciertas propiedades, a partir de las que se demuestra el resultado. Este resultado es cierto para cualquier conjunto de funciones que cumplan las propiedades asumidas y se puede establecer utilizando instanciación funcional. La herramienta que se desarrolla en esta sección realiza de forma automática este proceso de instanciación funcional, facilitando así la labor del usuario. De nuevo el desarrollo de la herramienta reveló un error en el sistema, más grave que el anterior, en el proceso de instanciación funcional[1]. La herramienta ha sido presentada en [52].

[1]Este error está corregido en la copia del sistema que aparece en el CD que acompaña a esta memoria. En la sección "Recent changes to this page" de la página en internet del sistema [42] se puede encontrar más información sobre la forma de corregir este error.

6.1 Pruebas de terminación basadas en multiconjuntos

Como se ha comentado en la sección 2.2.3, para que el sistema admita la definición de una función en la lógica, es necesario demostrar que dicha función termina. Para ello, se ha de probar que una determinada medida de los argumentos de la función decrece en las llamadas recursivas, con respecto a cierta relación bien fundamentada. Para funciones recursivas simples, esta prueba de terminación se efectúa automáticamente por el sistema. Sin embargo, existen esquemas recursivos complicados para los que el sistema no es capaz de probar la terminación sin ayuda por parte del usuario, que debe indicar la relación bien fundamentada y/o la función de medida con la que se ha de intentar la prueba.

Una herramienta para demostrar la terminación de funciones recursivas son los multiconjuntos. Informalmente, un multiconjunto es un conjunto con elementos repetidos. Dershowitz y Manna [20] demostraron que toda relación bien fundamentada sobre un conjunto A induce una relación bien fundamentada sobre el conjunto de multiconjuntos finitos cuyos elementos están en A. Este resultado es útil para demostrar la terminación de funciones entre cuyos argumentos aparece una lista de elementos del conjunto A de los que algunos, en las llamadas recursivas, son reemplazados por otros más pequeños con respecto a la relación bien fundamentada en A.

En esta sección se define una relación entre multiconjuntos inducida por una relación binaria y se demuestra que si esta última es bien fundamentada entonces la relación entre multiconjuntos también lo es. Este resultado se demuestra en un marco genérico, permitiendo así la posterior definición en el sistema de relaciones bien fundamentadas entre multiconjuntos, inducidas por relaciones bien fundamentadas. Para facilitar la definición de la relación entre multiconjuntos hemos desarrollado un comando llamado `defmul` que, a partir de una relación bien fundamentada, genera de forma automática la relación entre multiconjuntos y demuestra que también es bien fundamentada. Este comando es un buen ejemplo de cómo el demostrador puede ser extendido mediante herramientas específicas que hacen más fácil determinadas tareas. Finalmente se muestra un ejemplo de uso de la relación entre multiconjuntos inducida por una relación bien fundamentada.

En esta memoria se ha utilizado esta herramienta para demostrar la terminación de ciertas funciones en las secciones 7.1.2 y 8.1.2.1.

Los eventos relacionados con los multiconjuntos y la prueba de la buena fundamentación de la relación entre multiconjuntos inducida, se encuentran en el libro `multiconjuntos.lisp`. La definición del comando `defmul` se encuentra en el libro `defmul.lisp`. El ejemplo desarrollado en la última parte de esta sección se encuentra en el libro `arboles-binarios.lisp`.

6.1.1 Relaciones bien fundamentadas entre multiconjuntos

Definición 6.1 *Un* **multiconjunto** *sobre un conjunto A es una función $M : A \longrightarrow \mathbb{N}$, donde $\mathbb{N}$ es el conjunto de los número naturales. Decimos que M es un multiconjunto* **finito** *si hay un número finito de elementos de A tales que $M(x) > 0$. El conjunto de todos los multiconjuntos finitos sobre A lo notaremos por $\mathcal{M}(A)$.*

Intuitivamente, un multiconjunto sobre A es una función que asigna a cada elemento de A el número de veces que dicho elemento aparece en el multiconjunto. Usaremos una notación similar a la de conjuntos para representar los multiconjuntos finitos: representaremos como $\{\!\{a, a, b, b, b, c, e, e\}\!\}$, el multiconjunto definido sobre el conjunto $\{a, b, c, d, e\}$ en el que $M(a) = 2$, $M(b) = 3$, $M(c) = 1$, $M(d) = 0$ y $M(e) = 2$. De esta forma, si $M(x) = n > 0$ entonces el valor x aparecerá n veces en la representación de M. Obsérvese que el orden de los elementos en la representación del multiconjunto M no es importante, $\{\!\{b, b, a, a, b, e, c, e\}\!\}$ representa el mismo multiconjunto que $\{\!\{a, a, b, b, b, c, e, e\}\!\}$.

En la formalización representaremos los multiconjuntos finitos sobre un conjunto A, como listas de elementos de dicho conjunto, en las que pueden aparecer repeticiones. De esta forma la lista `(a a b b b c e e)` representa el multiconjunto $\{\!\{a, a, b, b, b, c, e, e\}\!\}$. Al igual que antes, el orden de los elementos en estas listas no es importante. Si `mp` es un predicado que caracteriza la pertenencia al conjunto A, entonces la pertenencia al conjunto $\mathcal{M}(A)$ queda caracterizada por el siguiente predicado:

106
```
(defun mul-mp (M)
  (if (atom M)
      (equal M nil)
      (and (mp (car M)) (mul-mp (cdr M)))))
```

Los conceptos y operaciones habituales sobre multiconjuntos son similares a las de conjuntos, teniendo en cuenta las múltiples ocurrencias de los elementos. Los que son necesarios a lo largo de esta sección son:

Definición 6.2 *Sean $M_1, M_2 \in \mathcal{M}(A)$ y $x \in A$*

- *x* **pertenece** *a M_1, y lo notaremos $x \in M_1$, si $M_1(x) > 0$.*
- *M_1 está* **incluido** *en M_2, y lo notaremos $M_1 \subseteq M_2$, si para cada $x \in A$ se tiene $M_1(x) \leq M_2(x)$.*
- *El multiconjunto* **vacío***, que notaremos $\emptyset$, es la función que a cada $x \in A$ le asigna 0.*

- *La* **unión** *de M_1 y M_2, que notaremos $M_1 \cup M_2$, es la función que a cada $x \in A$ le asigna $M_1(x) + M_2(x)$.*

- *La* **diferencia** *de M_1 y M_2, que notaremos $M_1 \setminus M_2$, es la función que a cada $x \in A$ le asigna $M_1(x) - M_2(x)$, si $M_1(x) \geq M_2(x)$ ó 0 en caso contrario.*

Puesto que representamos los multiconjuntos utilizando listas con repeticiones, algunos de los conceptos anteriores se formalizan utilizando funciones predefinidas en el sistema. De esta forma la función `member-equal` formaliza el concepto de pertenencia, la función `subsetp-equal` el de contención, la función `append` la unión y la función `endp` sirve para distinguir el multiconjunto vacío.

La única operación que es necesario formalizar es la diferencia de multiconjuntos. Para ello consideramos la función `elimina-una`, presentada en *58*, que elimina una ocurrencia de un elemento de una lista. Para calcular la diferencia entre dos multiconjuntos `M-1` y `M-2` hacemos lo siguiente: para cada ocurrencia de un elemento en `M-2`, eliminamos una ocurrencia de dicho elemento de `M-1`. La función que resulta es:

107
```
(defun mul-diferencia (M-1 M-2)
  (if (endp M-2)
      M-1
      (mul-diferencia (elimina-una (car M-2) M-1)
                      (cdr M-2))))
```

La siguiente definición proporciona una forma de extender una relación definida sobre un conjunto A al conjunto de multiconjuntos finitos sobre A:

Definición 6.3 *Sea $\lhd$ una relación binaria definida sobre un conjunto A. La* **relación entre multiconjuntos**[2] *inducida por $\lhd$ sobre $\mathcal{M}(A)$, notada por $\lhd_{\mathcal{M}}$, se define de la siguiente manera: $N \lhd_{\mathcal{M}} M$ si y sólo si $M \setminus N \neq \emptyset$ y para todo $y \in N \setminus M$, existe $x \in M \setminus N$ tal que $y \lhd x$.*

Una visión intuitiva de la relación entre multiconjuntos inducida por una relación binaria es proporcionada por Smullyan en [77]. Se dispone de una bolsa llena de bolas etiquetadas con números enteros positivos, no importa cuantas bolas hay con la misma etiqueta ni que números etiquetan las bolas. Esta bolsa es un multiconjunto finito de números enteros. La relación entre multiconjuntos inducida por el orden usual entre los números enteros, consiste en tomar de la bolsa una bola con etiqueta n y reemplazarla por tantas bolas como queramos etiquetadas con cualquier valor menor que n.

[2]La definición original de Dershowitz y Manna [20] es más amplia, pero la presentada aquí es más adecuada para su formalización en ACL2. Ambas definiciones son equivalentes cuando se trata de una relación de orden parcial.

En la formalización definimos la relación entre multiconjuntos de la siguiente forma: supongamos que `rel` es una función binaria que representa una relación definida sobre el conjunto A. Para definir la relación entre dos multiconjuntos M y N, consideramos las diferencias $N \setminus M$ y $M \setminus N$, y comprobamos que para cada elemento de la primera, existe en la segunda un elemento mayor con respecto a la relación `rel`. La función que implementa este proceso es `mul-rel`:

108

```
(defun mul-rel (N M)
  (let ((M-N (mul-diferencia M N))
        (N-M (mul-diferencia N M)))
    (and (consp M-N)
         (paratodo-existe-rel-mayor N-M M-N))))

(defun paratodo-existe-rel-mayor (N M)
  (if (endp N)
      t
      (and (existe-rel-mayor (car N) M)
           (paratodo-existe-rel-mayor (cdr N) M))))

(defun existe-rel-mayor (x M)
  (cond ((endp M) nil)
        ((rel x (car M)) t)
        (t (existe-rel-mayor x (cdr M)))))
```

Como se ha comentado al principio de esta sección, nuestro objetivo es demostrar que, dada una relación bien fundamentada, la relación entre multiconjuntos inducida por ella también es bien fundamentada. Como se indicó en la sección 2.2.2, el concepto de relación bien fundamentada es equivalente al de relación noetheriana. A lo largo de esta sección utilizaremos indistintamente ambos términos. De esta forma, el resultado que queremos demostrar lo enunciamos utilizando el término "relación noetheriana", que es el que aparece en el artículo original de Dersowitz y Manna, [20]. La prueba de este resultado utiliza el lema de König para el que son necesarios determinados conceptos sobre árboles.

Definición 6.4 *Un árbol es un par formado por un elemento, al que llamamos etiqueta del árbol, y una lista posiblemente vacía y no necesariamente finita de árboles, a los que llamamos hijos del árbol. Dado un árbol* $\mathbf{T}$*, notaremos por* $e(\mathbf{T})$ *a su etiqueta. Un árbol* $\mathbf{T}'$ *es descendiente de otro árbol* $\mathbf{T}$ *si* $\mathbf{T}'$ *es un hijo de* $\mathbf{T}$ *o es descendiente de alguno de sus hijos. Un árbol es finito si tiene un número finito de descendientes, en otro caso se llama infinito. Un árbol es finitamente ramificado si tiene una cantidad finita de hijos. Una rama de un árbol* $\mathbf{T}$ *es una secuencia de árboles* $\{\mathbf{T}_i\}_{i\geq 0}$ *tales que* $\mathbf{T}_0 = \mathbf{T}$ *y para todo* i*,* $\mathbf{T}_{i+1}$ *es un hijo de* $\mathbf{T}_i$*. Una rama es infinita si la secuencia de árboles que la define es infinita. Una*

hoja es un árbol cuya lista de hijos es vacía. Dado un árbol $\mathbf{T}$*, el conjunto de sus hojas es el formado por todos sus descendientes que sean hojas.*

Teorema 6.5 (Lema de König) *Todo árbol infinito y finitamente ramificado tiene una rama infinita.*

Demostración:

Sea $\mathbf{T}$ un árbol infinito y finitamente ramificado. Sea $\mathbf{T}_0 = \mathbf{T}$. Dado un árbol infinito y finitamente ramificado $\mathbf{T}_i$, el número de sus descendientes es igual a la suma del número de descendientes de cada uno de sus hijos más el número de hijos que tiene. Cómo $\mathbf{T}_i$ es infinito y tiene un número finito de hijos, alguno de ellos ha de tener una cantidad infinita de descendientes. Sea $\mathbf{T}_{i+1}$ dicho hijo.

De esta forma estamos construyendo una secuencia infinita de árboles $\{\mathbf{T}_i\}_{i\geq 0}$ tales que $\mathbf{T}_0 = \mathbf{T}$ y para todo i, $\mathbf{T}_{i+1}$ es un hijo de $\mathbf{T}_i$, es decir, una rama infinita de $\mathbf{T}$.

□

El principal resultado de esta sección es el siguiente:

Teorema 6.6 (Dersowitz y Manna, [20]) *Sea* $\lhd$ *una relación noetheriana sobre un conjunto* A*. Entonces* $\lhd_{\mathcal{M}}$ *es noetheriana.*

Demostración:

La prueba es por reducción al absurdo. Supongamos que $\lhd_{\mathcal{M}}$ no es noetheriana sobre $\mathcal{M}(A)$. Entonces existe una sucesión infinita de elementos de $\mathcal{M}(A)$ tal que: $\cdots \lhd_{\mathcal{M}} M_4 \lhd_{\mathcal{M}} M_3 \lhd_{\mathcal{M}} M_2 \lhd_{\mathcal{M}} M_1$.

Vamos a construir un árbol infinito $\mathbf{T}$ tal que para cada descendiente $\mathbf{T}'$ de $\mathbf{T}$ y para cada hijo $\mathbf{T}''$ de $\mathbf{T}'$ se tiene que $e(\mathbf{T}'') \lhd e(\mathbf{T}')$:

1. La etiqueta de $\mathbf{T}$ es un elemento cualquiera $a \in A$ y sus hijos son árboles (en principio hoja) cuyas etiquetas son los elementos de M_1. Así, para cada ocurrencia de un elemento x en M_1 hay una hoja de $\mathbf{T}$ cuya etiqueta es x.

2. Como $M_2 \lhd_{\mathcal{M}} M_1$, para cada $y \in M_2 \setminus M_1$ existe $x \in M_1 \setminus M_2$ tal que $y \lhd x$. En este caso añadimos el árbol hoja cuya etiqueta es y como hijo de una de las hojas de $\mathbf{T}$ etiquetadas con x. De nuevo se verifica que para cada ocurrencia de un elemento x en M_2 hay una hoja de $\mathbf{T}$ cuya etiqueta es x. Además, como $M_2 \setminus M_1 \neq \emptyset$, al menos se ha añadido un descendiente nuevo al árbol $\mathbf{T}$.

3. Supongamos que ya hemos modificado $\mathbf{T}$ hasta llegar al multiconjunto M_i. De esta forma, para cada ocurrencia de un elemento x en M_i hay una hoja de $\mathbf{T}$ cuya etiqueta es x. Como $M_{i+1} \lhd_{\mathcal{M}} M_i$, para cada $y \in M_{i+1} \setminus M_i$ existe $x \in M_i \setminus M_{i+1}$ tal que $y \lhd x$. En este caso añadimos el árbol hoja cuya etiqueta es y como hijo de una de las hojas de $\mathbf{T}$ etiquetadas con x.

De nuevo se verifica que para cada ocurrencia de un elemento x en M_{i+1} hay una hoja de $\mathbf{T}$ cuya etiqueta es x. Además, como $M_{i+1} \setminus M_i \neq \emptyset$, al menos se ha añadido un descendiente nuevo al árbol $\mathbf{T}$.

Finalizada la construcción del árbol $\mathbf{T}$, se verifica que en cada paso se ha añadido al menos un descendiente nuevo y, como la sucesión $\{M_i\}_{i\geq 1}$ es infinita, el árbol $\mathbf{T}$ es infinito. Por otro lado, si $\mathbf{T}'$ es un descendiente de $\mathbf{T}$ y $\mathbf{T}''$ es hijo de $\mathbf{T}'$, entonces $\mathbf{T}''$ se ha añadido en alguno de los pasos de construcción de $\mathbf{T}$ de forma que $e(\mathbf{T}'') \lhd e(\mathbf{T}')$.

Por el lema de König, $\mathbf{T}$ tiene una rama infinita: $\{\mathbf{T}_i\}_{i\geq 0}$, tal que $\mathbf{T}_0 = \mathbf{T}$ y para todo i, $\mathbf{T}_{i+1}$ es un hijo de $\mathbf{T}_i$. De esta forma, la sucesión infinita de elementos de A: $\{e(\mathbf{T}_i)\}_{i\geq 1}$, verifica que $e(\mathbf{T}_{i+1}) \lhd e(\mathbf{T}_i)$, por la construcción de $\mathbf{T}$, y esto está en contradicción con el hecho de que $\lhd$ es noetheriana sobre A. Por tanto $\lhd_{\mathcal{M}}$ es noetheriana sobre $\mathcal{M}(A)$.

□

Consideremos de nuevo el ejemplo de Smullyan sobre la bolsa llena de bolas etiquetadas con número enteros positivos. Supongamos que en cada paso se toma de la bolsa una bola con etiqueta n y se reemplaza por tantas bolas como queramos etiquetadas con cualquier valor menor que n. El teorema anterior nos asegura que en algún momento la bolsa se queda vacía.

Veamos a continuación como se formaliza el teorema anterior en ACL2. El teorema trata sobre una relación noetheriana cualquiera $\lhd$. En la sección 2.2.2 hemos visto cómo se caracterizan este tipo de relaciones. De esta forma asumimos en un encapsulado de ACL2 la existencia de una función binaria `rel`, que representa la relación noetheriana, un predicado `mp`, que caracteriza la pertenencia al conjunto A y una función de inmersión en los ordinales `fn`. La única propiedad que se exige a estas funciones es que `rel` represente una relación de buena fundamentación en el conjunto caracterizado por `mp` con función de inmersión `fn`:

109
```
(defthm rel-es-bien-fundamentada-en-mp
   (and (implies (mp x)
                 (e0-ordinalp (fn x)))
        (implies (and (mp x)
                      (mp y)
                      (rel x y))
                 (e0-ord-< (fn x) (fn y)))))
```

Con respecto a `rel` y `mp` consideramos las funciones `mul-mp`, *106*, que caracteriza la pertenencia a $\mathcal{M}(A)$, y `mul-rel`, *108*, que implementa la relación entre multiconjuntos inducida por `rel`. La demostración del teorema consiste en definir una función de inmersión `mul-fn` y establecer el siguiente evento:

110
```
(defthm mul-rel-es-bien-fundamentada-en-mul-mp
  (and (implies (mul-mp M)
                (e0-ordinalp (mul-fn M)))
       (implies (and (mul-mp M)
                     (mul-mp N)
                     (mul-rel N M))
                (e0-ord-< (mul-fn N) (mul-fn M)))))
```

Puesto que la formulación del resultado es distinta a la original, la demostración también es diferente. Nuestra prueba se basa en el siguiente resultado de la teoría de ordinales (véase [36]): 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.

El isomorfismo H sugiere la siguiente definición de la función `mul-fn`: dado un multiconjunto de elementos que cumplen `mp`, se aplica `fn` a cada elemento para obtener un multiconjunto de ordinales. Luego se utiliza H para obtener un ordinal menor que ε_0. Si los ordinales están representados en notación ACL2 (véase la sección 2.2.1), 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 `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.

111
```
(defun mul-fn (M)
  (if (consp M)
      (insercion-ordenada (fn1 (car M)) (mul-fn (cdr M)))
      0))

(defmacro fn1 (x) `(suma-1-si-es-entero (fn ,x)))

(defun suma-1-si-es-entero (x) (if (integerp x) (1+ x) x))

(defun insercion-ordenada (x M)
  (cond ((atom M) (cons x M))
        ((not (e0-ord-< x (car M)))  (cons x M))
        (t (cons (car M) (insercion-ordenada x (cdr M))))))
```

En estas condiciones, la primera parte del evento que asegura la buena fundamentación de la relación `mul-rel`, mostrado en *110*, es bastante fácil de demostrar:

```
(implies (mul-mp M)
         (e0-ordinalp (mul-fn M)))
```

Nos centraremos en la prueba de la otra parte de dicho evento:

```
(implies (and (mul-mp M)
              (mul-mp N)
              (mul-rel N M))
         (e0-ord-< (mul-fn N) (mul-fn M))))
```

La prueba se desarrolla 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` un elemento de `M` y `v` un elemento de `N` tales que `(fn1 u)` y `(fn1 v)` son los mayores elementos (respecto del orden entre ordinales) de los multiconjuntos $\{\!\{$`(fn1 x)` : `x` es un elemento de `M`$\}\!\}$ y $\{\!\{$`(fn1 y)` : `y` es un elemento de `N`$\}\!\}$, respectivamente. Estos elementos se obtienen con la función `fn1-maximo`:

112
```
(defun fn1-maximo (M)
  (cond ((atom M) nil)
        ((atom (cdr M)) (car M))
        (t (let ((max-cdr (fn1-maximo (cdr M))))
             (if (e0-ord-< (fn1 max-cdr) (fn1 (car M)))
                 (car M)
                 max-cdr)))))
```

De esta forma `u` es `(fn1-maximo M)` y `v` es `(fn1-maximo N)`. Veamos los distintos casos que se pueden dar en función de los valores `(fn1 u)` y `(fn1 v)`:

Si `(fn1 v)` es menor que `(fn1 u)` entonces, por definición del orden entre los ordinales se tiene que `(mul-fn N)` es menor que `(mul-fn M)`. Este hecho se formaliza con el siguiente evento:

113
```
(defthm induccion-caso-1
  (implies (and (not (atom N))
                (not (atom M))
                (mul-mp N)
                (mul-mp M)
                (mul-rel N M)
                (e0-ord-< (fn1 (fn1-maximo N))
                          (fn1 (fn1-maximo M))))
           (e0-ord-< (mul-fn N) (mul-fn M))))
```

Si `(fn1 u)` es menor que `(fn1 v)` entonces, como `(fn1 u)` es el mayor ordinal de $\{\!\{$`(fn1 x)` : `x` es un elemento de `M`$\}\!\}$, `v` está en `N\M` y, por la definición de `mul-rel`, existe `z` en `M\N` tal que `(rel v z)`. Por tanto `(fn1 v)` es menor que `(fn1 z)`. Luego `(fn1 u)` es menor que `(fn1 z)`, lo que contradice el hecho de que `u` es el elemento de `M` tal que `(fn1 u)` es el mayor ordinal de $\{\!\{$`(fn1 x)` : `x` es un elemento de `M`$\}\!\}$. Este hecho se formaliza con el siguiente evento:

114
```
(defthm induccion-caso-2
  (implies (and (not (atom N))
                (not (atom M))
                (mul-mp N)
                (mul-mp M)
                (mul-rel N M))
           (not (e0-ord-< (fn1 (fn1-maximo M))
                          (fn1 (fn1-maximo N))))))
```

Si `(fn1 u)` es igual a `(fn1 v)` entonces `v` está en `M`, ya que en otro caso existiría un elemento `z` en `M\N` tal que `(rel v z)` y se tendría la misma contradicción que en el caso anterior. En este caso consideramos `M-aux` y `N-aux` los resultados de eliminar de `M` y `N`, respectivamente, una ocurrencia de `v`. Por la definición de `mul-rel` se sigue verificando `(mul-rel N-aux M-aux)` y el número de elementos de `N-aux` es menor que el de `N`. Luego, por hipótesis de inducción, se tiene que `(mul-fn N-aux)` es menor que `(mul-fn M-aux)`. Finalmente, y gracias a las propiedades demostradas acerca de `mul-fn` y la función que elimina un elemento de un multiconjunto, se obtiene que `(mul-fn N)` es menor que `(mul-fn M)`. Este último resultado se formaliza de la siguiente forma:

115
```
(defthm induccion-caso-3
  (Implies (and (not (atom N))
                (not (atom M))
                (mul-mp N)
                (mul-mp M)
                (equal (fn1 (fn1-maximo M))
                       (fn1 (fn1-maximo N)))
                (e0-ord-< (mul-fn (elimina-una (fn1-maximo N) N))
                          (mul-fn (elimina-una (fn1-maximo N) M)))
                (mul-rel N M))
           (e0-ord-< (mul-fn N) (mul-fn M))))
```

Este 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 presentado:

116
```
(defun induccion-mul-rel (N M)
  (cond ((or (atom N) (atom M)) t)
        (t (let* ((Max-M (fn1-maximo M))
                  (Max-N (fn1-maximo N))
                  (Fn1-Max-M (fn1 Max-M))
                  (Fn1-Max-N (fn1 Max-N)))
             (if (equal Fn1-Max-M Fn1-Max-N)
                 (induccion-mul-rel (elimina-una Max-N N)
                                    (elimina-una Max-N M))
                 t)))))
```

Una vez definida la función que proporciona el esquema de inducción y demostrados los distintos casos del mismo, se demuestra sin dificultad la segunda parte del evento `rel-es-bien-fundamentada-en-mp`.

6.1.2 El comando `defmul`

En la sección anterior hemos formalizado y demostrado la buena fundamentación de la relación en un marco genérico, ya que no se han asumido propiedades particulares de las funciones `rel`, `mp` y `fn`, excepto la que asegura la buena fundamentación de `rel`. Esto nos permite usar instancias funcionales del teorema `mul-rel-es-bien-fundamentada-en-mul-mp`, 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 una relación $relx$, de la cual se sabe que cumple la propiedad de buena fundamentación sobre un conjunto de objetos caracterizados por la propiedad mpx, con función de inmersión fnx. Es decir, se ha probado el siguiente teorema:

```
(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)))))
```

Para definir en la lógica de ACL2 la relación entre multiconjuntos inducida por la relación $relx$ y probar su buena fundamentación, hemos de establecer, en principio, los siguientes eventos en ACL2:

- Las definiciones que se necesitan para implementar la relación entre multiconjuntos inducida por $relx$: las funciones `existe-`$relx$`-mayor`, `paratodo-existe-`$relx$`-mayor` y `mul-`$relx$ de manera totalmente análoga a las dadas en *108*.

- La definición de la función que caracteriza los multiconjuntos, `mul-`mpx, de forma similar la mostrada en *106*.
- La definición de la función de inmersión `mul-`fnx, similar a la presentada en *111*.
- El teorema de buena fundamentación para `mul-`$relx$, `mul-`mpx y `mul-`fnx. Este teorema puede ser probado directamente usando una instancia funcional del teorema `mul-rel-es-bien-fundamentada-en-mul-mp`. La sustitución funcional que hay que usar debe asignar a los símbolos `x`, `y`, `rel`, `fn`, `mp`, `existe-rel-mayor`, `paratodo-existe-rel-mayor`, `mul-rel`, `mul-mp` y `mul-fn` los correspondientes a la nueva relación definida: x, y, $relx$, fnx, mpx, `existe-`$relx$`-mayor`, `paratodo-existe-`$relx$`-mayor`, `mul-`$relx$, `mul-`mpx y `mul-`fnx, 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`, que nos proporciona una manera automática de realizar esta tarea. En este caso, para definir la relación `mul-`$relx$, inducida por $relx$, y probar su buena fundamentación, 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. Estos 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.

6.1.3 Ejemplo

El siguiente ejemplo está inspirado en [76], donde Slind formaliza varios esquemas de transformación de programas utilizando el sistema HOL. Uno de sus ejemplos es la transformación de un esquema recursivo general sobre árboles binarios, a una forma iterativa. El ejemplo es original de Wand [84] y también ha sido desarrollado en el sistema PVS por Shankar [73].

Definición 6.7 *Un árbol binario es un árbol hoja o un árbol que tiene únicamente dos hijos. Dado un árbol binario* $\mathbf{T}$ *que no sea una hoja, llamaremos hijo izquierdo, y lo notaremos* $i(\mathbf{T})$*, a su primer hijo e hijo derecho, y lo notaremos* $d(\mathbf{T})$*, a su segundo hijo.*

Es habitual definir funciones sobre árboles binarios con el siguiente esquema recursivo: La función `base` comprueba si estamos en el caso base de la recursión.

En ese caso se utiliza la función `f-base` para obtener un valor. Si no estamos en el caso base, se hacen llamadas recursivas sobre los hijos izquierdo y derecho (obtenidos respectivamente con las funciones `i` y `d`) y se combinan los resultados con la función `combina`. La función `f-recursiva` representa este esquema recursivo:

117
```
(defun f-recursiva (TT)
  (declare (xargs :measure (medida TT)
                  :well-founded-relation rel-bin))
  (if (base TT)
      (f-base TT)
      (combina (f-recursiva (i TT))
               (f-recursiva (d TT)))))
```

Para que el sistema admita esta función se proporciona una función de medida cuyo valor en los argumentos de las llamadas recursivas es menor, con respecto a una relación bien fundamentada, que el valor en el argumento original. La función de medida es `medida` y la relación bien fundamentada `rel-bin`. Estas funciones han de verificar las siguientes propiedades:

1. La función `rel-bin` es una relación bien fundamentada en el conjunto caracterizado por `mp-bin` con función de inmersión `fn-bin`:

   118
   ```
   (defthm rel-bin-es-bien-fundamentada-en-mp-bin
     (and (implies (mp-bin x)
                   (e0-ordinalp (fn-bin x)))
          (implies (and (mp-bin x)
                        (mp-bin y)
                        (rel-bin x y))
                   (e0-ord-< (fn-bin x) (fn-bin y)))))
   ```

2. El valor de la función `medida` está en el conjunto caracterizado por `mp-bin`:

   119
   ```
   (defthm medida-es-mp-bin
     (mp-bin (medida TT)))
   ```

3. El valor de la función `medida` en los hijos izquierdo y derecho de un árbol binario es menor, con respecto a `rel-bin`, que el valor en el árbol original.

   120
   ```
   (defthm hijos-rel-bin
     (implies (not (base TT))
              (and (rel-bin (medida (i TT)) (medida TT))
                   (rel-bin (medida (d TT)) (medida TT)))))
   ```

En estas condiciones se puede definir un esquema recursivo de cola o iterativo que, cuando la función `combina` es asociativa y tiene elemento neutro dado por la función constante `neutro`, es equivalente al recursivo:

121

```
(defun f-iterativa (TT)
  (f-iterativa-aux (list TT) (neutro)))

(defun f-iterativa-aux (Lista-TT v)
  (declare (xargs :measure (medida-lista Lista-TT)
                  :well-founded-relation mul-rel-bin))
  (cond ((endp Lista-TT) v)
        ((base (car Lista-TT))
         (f-iterativa-aux (cdr Lista-TT)
                          (combina v (f-base (car Lista-TT)))))
        (t (f-iterativa-aux (list* (i (car Lista-TT))
                                   (d (car Lista-TT))
                                   (cdr Lista-TT))
                            v))))
```

La función `f-iterativa` hace una llamada a la función auxiliar `f-iterativa-aux` a la que se pasan como argumento la lista unitaria formada por el árbol binario para el se quiere realizar determinado cálculo y el elemento neutro de la función `combina`. El primer argumento de la función `f-iterativa-aux` se interpreta como la lista de árboles binarios para los que todavía no se ha realizado el cálculo. El segundo argumento es un acumulador donde se almacena el valor obtenido para los árboles a los que ya se ha realizado el cálculo. Esta función comprueba si el primer árbol de `Lista-TT` está en el caso base, en cuyo caso realiza una llamada recursiva sobre el resto de la lista `lista-TT` y el resultado de combinar el valor del acumulador con el valor obtenido para dicho árbol. Si el primer árbol de `Lista-TT` no está en el caso base, se realiza una llamada recursiva sobre el resultado de reemplazar dicho árbol por sus hijos izquierdo y derecho, sin modificar el acumulador.

La terminación de la función `f-iterativa-aux` no es trivial pues la longitud de su primer argumento aumenta en la segunda llamada recursiva. Si establecemos una analogía con el ejemplo de Smullyan, los árboles binarios son las bolas de una bolsa, que es el primer argumento de `f-iterativa-aux`. En cada llamada recursiva se toma de la bolsa una bola, es decir un árbol binario, etiquetada con la medida de dicho árbol (dada por la función `medida`) y, o se elimina (primera llamada recursiva), o se reemplaza por las bolas que representan sus hijos (segunda llamada recursiva) etiquetadas con sus medidas correspondientes, que son menores que la de la bola eliminada, con respecto a una relación bien fundamentada (la dada por la función `rel-bin`).

Es decir, la terminación de la función `f-iterativa-aux` se basa en que el multiconjunto de las medidas de los árboles de su primer argumento, decrece con respecto a la relación entre multiconjuntos inducida por `rel-bin`. Para construir el multiconjunto de las medidas consideramos la función `medida-lista`:

122
```
(defun medida-lista (Lista-TT)
  (if (endp Lista-TT)
      nil
      (cons (medida (car Lista-TT))
            (medida-lista (cdr Lista-TT)))))
```

La relación entre multiconjuntos inducida por `rel-bin`, junto con el evento que asegura que es una relación bien fundamentada, se obtiene al evaluar la siguiente expresión:

```
(defmul (REL-BIN REL-BIN-ES-BIEN-FUNDAMENTADA-EN-MP-BIN
         MP-BIN FN-BIN X Y))
```

En la definición de la función `f-iterativa-aux`, *121*, se han indicado la medida de los argumentos que se ha de considerar para demostrar su terminación, `(medida-lista Lista-TT)` y la relación bien fundamentada con respecto a la que esta medida disminuye, `mul-rel-bin`. Esta última función es la que se obtiene al evaluar la expresión anterior.

Finalmente, cuando la función `combina` es asociativa y el valor dado por la función constante `neutro` es un elemento neutro de la misma, se puede demostrar que las funciones `f-recursiva` y `f-iterativa` son equivalentes.

123
```
(defthm f-recursiva-igual-f-iterativa
  (equal (f-iterativa TT)
         (f-recursiva TT)))

(defthm combina-asociativa
  (equal (combina (combina v-1 v-2) v-3)
         (combina v-1 (combina v-2 v-3))))

(defthm combina-neutro
  (equal (combina (neutro) v) v))
```

Otros ejemplos del uso de la relación entre multiconjuntos inducida por una relación dada se pueden encontrar en [67] y en [51].

6.2 Teorías genéricas

Como se ha visto en la sección anterior, en ACL2 se pueden probar resultados sobre funciones para las cuales únicamente se asumen determinadas propiedades. Estos resultados son generales en el sentido de que son ciertos para cualquier conjunto de funciones que cumpla las propiedades asumidas, y se pueden reutilizar mediante el proceso de instanciación funcional. De esta forma se pueden establecer y utilizar resultados de segundo orden en ACL2. Sin embargo, el proceso de instanciación funcional es tedioso cuando se trata de gran cantidad de eventos, o hay que definir nuevas funciones a partir de aquellas para las que se asumen determinadas propiedades.

En esta sección definimos un conjunto de herramientas que facilitan la reutilización de resultados de segundo orden en ACL2. De esta forma, dado un conjunto de funciones que cumplan las condiciones de un resultado de segundo orden, estas herramientas construyen de forma automática las funciones y teoremas necesarios para demostrar la propiedad que establece dicho resultado. En la primera parte de esta sección se describe el tipo de situación en el que se pueden utilizar estas herramientas, utilizando como ejemplo el teorema de buena fundamentación de la relación entre multiconjuntos inducida por una relación bien fundamentada. En la segunda parte se describe cómo funcionan las herramientas de instanciación genérica y en la tercera se proporciona un ejemplo de uso.

En esta memoria se ha utilizado las herramientas de instanciación genérica para obtener versiones concretas de teoremas de segundo orden en las secciones 7.2, 7.3 y 9.3. Los eventos correspondientes a las herramientas de instanciación funcional se encuentran en el libro `teorias-genericas.lisp` y el ejemplo desarrollado al final de esta sección en el libro `insercion-ordenada.lisp`.

6.2.1 Teorías genéricas en ACL2

Una *teoría* en ACL2 es una lista de eventos de tipo definición y teorema, admitidos por el sistema, tal y como podrían aparecer en un libro cualquiera. Una *teoría genérica* es una teoría que depende de la existencia de un conjunto de funciones, a las que llamaremos *funciones genéricas*, que verifican ciertas propiedades.

Por ejemplo, en la sección anterior se ha supuesto la existencia de tres funciones: una relación bien fundamentada `rel`, una función que caracteriza la pertenencia a un conjunto `mp` y una función de inmersión en los ordinales `fn`, verificando la siguiente propiedad:

124

```
(defthm rel-es-bien-fundamentada-en-mp
   (and (implies (mp x)
                 (e0-ordinalp (fn x)))
        (implies (and (mp x) (mp y) (rel x y))
                 (e0-ord-< (fn x) (fn y)))))
```

A partir de estas funciones se han desarrollado las definiciones de las funciones `mul-mp`, `existe-rel-mayor`, `paratodo-existe-rel-mayor`, `mul-rel` y `mul-fn`, y se ha demostrado el siguiente teorema:

125
```
(defthm mul-rel-es-bien-fundamentada-en-mul-mp
  (and (implies (mul-mp M)
                (e0-ordinalp (mul-fn M)))
       (implies (and (mul-mp M)
                     (mul-mp N)
                     (mul-rel N M))
                (e0-ord-< (mul-fn N) (mul-fn M)))))
```

En este caso, los eventos correspondientes a las definiciones de la funciones `mul-mp`, `existe-rel-mayor`, `paratodo-existe-rel-mayor`, `mul-rel` y `mul-fn` y el evento `mul-rel-es-bien-fundamentada-en-mul-mp` forman una teoría genérica definida para las funciones `rel`, `mp` y `fn`.

La característica principal de las teorías genéricas es que formalizan resultados de segundo orden en ACL2. En el caso de las relaciones de multiconjuntos, la teoría genérica formaliza el siguiente teorema:

Teorema 6.8 *Sea A un conjunto, $\lhd : A \times A \longrightarrow \mathbb{B}$ un relación binaria y $f : A \longrightarrow \mathcal{O}rd$ una función, tales que para todo par de elementos $u, v \in A$ se tiene $u \lhd v \Longrightarrow f(u) < f(v)$. Entonces existe una función $f^* : \mathcal{M}(A) \longrightarrow \mathcal{O}rd$ tal que para todo par de multiconjuntos $M, N \in \mathcal{M}(A)$ se tiene $N \lhd_{\mathcal{M}} M \Longrightarrow f^*(N) < f^*(M)$.*

Para utilizar estos resultados de segundo orden, debemos proporcionar versiones concretas de las funciones para las que se define la teoría genérica, demostrando que tienen las mismas propiedades asumidas para ellas. Una vez hecho esto, se pueden construir las versiones concretas de los eventos que proporciona la teoría genérica, sustituyendo en sus definiciones las funciones genéricas por las concretas. Para demostrar las versiones concretas de los teoremas que proporciona la teoría genérica, se utiliza una instancia funcional en la que las funciones genéricas son sustituidas por las concretas.

Esto es precisamente lo que hace, entre otras cosas, la macro `defmul` a partir de versiones concretas de las funciones `rel`, `mp` y `fn`. Consideremos la relación de orden entre ordinales definida por la función `e0-ord-<`, la función `e0-ordinalp` que caracteriza los ordinales de ACL2 y la función identidad `id`, como función de inmersión. Estas funciones verifican la propiedad exigida para las funciones `rel`, `mp` y `fn`:

126
```
(defthm e0-ord-<-es-bien-fundamentada-en-e0-ordinalp
   (and (implies (e0-ordinalp x)
                 (e0-ordinalp (id x)))
        (implies (and (e0-ordinalp x)
                      (e0-ordinalp y)
                      (e0-ord-< x y))
                 (e0-ord-< (id x) (id y)))))
```

Por tanto, se pueden construir versiones concretas de las funciones que proporciona la teoría genérica, sin más que sustituir las funciones `rel`, `mp` y `fn` por `e0-ord-<`, `e0-ordinalp` e `id`, respectivamente. Estas versiones concretas son:

127
```
(defun mul-mp-ord (M)
  (if (atom M)
      (equal M nil)
      (and (e0-ordinalp (car M)) (mul-mp-ord (cdr M)))))

(defun mul-rel-ord (N M)
  (let ((M-N (mul-diferencia M N))
        (N-M (mul-diferencia N M)))
    (and (consp M-N)
         (paratodo-existe-rel-mayor-ord N-M M-N))))

(defun paratodo-existe-rel-mayor-ord (N M)
  (if (endp N)
      t
      (and (existe-rel-mayor-ord (car N) M)
           (paratodo-existe-rel-mayor-ord (cdr N) M))))

(defun existe-rel-mayor-ord (x M)
  (cond ((endp M) nil)
        ((e0-ord-< x (car M)) t)
        (t (existe-rel-mayor-ord x (cdr M)))))

(defun mul-fn-ord (M)
  (if (consp M)
      (insercion-ordenada (fn1-ord (car M))
                          (mul-fn-ord (cdr M)))
      0))

(defmacro fn1-ord (x) `(suma-1-si-es-entero (id ,x)))
```

Hemos indicado en negrita los usos de las funciones concretas que han sustituido a las generales. Obsérvese que, como estos eventos ya existían en el sistema, es necesario asignarles un nuevo nombre. En este caso les hemos añadido el sufijo "*-ord*" y hemos indicado con letra cursiva todos los usos de estos nuevos nombres.

El teorema de buena fundamentación para la función `mul-rel-ord` se obtiene como una instancia funcional del teorema de buena fundamentación para `mul-rel`:

128

```
(defthm mul-rel-ord-es-bien-fundamentada-en-mul-mp-ord
  (and (implies (mul-mp-ord x)
                (e0-ordinalp (mul-fn-ord x)))
       (implies (and (mul-mp-ord x)
                     (mul-mp-ord y)
                     (mul-rel-ord x y))
                (e0-ord-< (mul-fn-ord x) (mul-fn-ord Y))))
  :hints (("Goal"
           :by (:functional-instance
                  mul-rel-es-bien-fundamentada-en-mul-mp
                  (mul-mp mul-mp-ord)
                  (exite-rel-mayor existe-rel-mayor-ord)
                  (paratodo-existe-rel-mayor
                              paratodo-existe-rel-mayor-ord)
                  (mul-rel mul-rel-ord)
                  (mul-fn mul-fn-ord)
                  (rel e0-ord-<)
                  (mp e0-ordinalp)
                  (fn id)))))
```

Al conjunto de versiones concretas de los eventos proporcionados por una teoría genérica lo denominamos *instancia* de la teoría, y al proceso por el que se han obtenido, *instanciación genérica*.

6.2.2 Herramienta de instanciación genérica

Para poder automatizar el proceso de instanciación genérica, necesitamos disponer de todos los eventos de una teoría genérica para poder construir las versiones concretas a partir de ellos. Para ello utilizamos una constante en la que se almacenan los eventos en el mismo orden en que son admitidos por el sistema. De esta forma, un desarrollo habitual de una teoría genérica tendría el siguiente aspecto:

```
(encapsulate
  (...)
  (propiedades de las funciones genéricas)
 )

(defun f1 (...) ...)     \
                         |
(defthm thm1 ...)        |     Estos eventos dependen
                         |    de las propiedades de las
(defun f2 (...) ...)     |    funciones genéricas.
                         |
(defthm thm2 ...)        |
                         |
(defthm thm3 ...)        /

(defconst *<teoria>*
  '((defun f1 (...) ...)  \
    (defthm thm1 ...)     |    Eventos de la teoría genérica
    (defthm thm3 ...)))   /
```

El primer evento es un encapsulado en el que se asume la existencia de las funciones genéricas con determinadas propiedades. A continuación se desarrollan los eventos que dependen de estas funciones genéricas y sus propiedades. Finalmente utilizamos `defconst` para almacenar la secuencia de eventos que forman la teoría genérica en una constante `*<teoria>*`. Una vez que se ha asignado un valor a una constante, éste no puede ser modificado. Las constantes en ACL2 son meras abreviaturas de expresiones más complejas, de esta forma, para cada teoría genérica se ha de utilizar una constante distinta. Aquí `<teoria>` representa el nombre único con el que se denomina la teoría.

Obsérvese que no todos los eventos que dependen de las propiedades de las funciones genéricas aparecen en la teoría genérica. Esto se debe a que es habitual en ACL2 utilizar funciones y teoremas auxiliares que ayudan en la prueba de un teorema, pero que no sirven para nada más. Este es el caso de la función `maximo-fn1` y los eventos `induccion-caso-1`, `induccion-caso-2` e `induccion-caso-3` para el caso de las relaciones de multiconjuntos.

Una vez construida la constante con la secuencia de eventos que hay que instanciar, para poder realizar el proceso de instanciación funcional, necesitamos un conjunto de funciones concretas que sustituyan a las funciones genéricas para las que se define la teoría. Estas funciones, y su relación con las genéricas se proporciona a través de una lista de asociación. Por ejemplo la siguiente es una lista de asociación que establece dicha relación para las funciones genéricas y concretas del ejemplo de las relaciones de multiconjuntos:

```
((rel e0-ord-<)
 (mp  e0-ordinalp)
 (fn  id))
```

Para instanciar una definición de la teoría genérica, sustituimos en la misma todas las ocurrencias de las funciones genéricas por las funciones concretas, según la relación dada por dicha lista de asociación. Como hemos visto antes, es necesario escoger un nuevo nombre para la instancia obtenida. El par formado por el nombre de la función genérica y el nuevo nombre escogido para la instancia pasan a formar parte de la lista de asociación, para ser utilizados en la siguiente instancia. En el ejemplo de las relaciones de multiconjuntos, al generar la instancia de la función `mul-fn`, la lista de asociación se amplia dando lugar a la siguiente:

```
((mul-fn mul-fn-ord)
 (rel    e0-ord-<)
 (mp     e0-ordinalp)
 (fn     id))
```

Para instanciar un teorema de la teoría genérica, sustituimos en el mismo todas las ocurrencias de las funciones genéricas por las funciones concretas, según la relación dada por la lista de asociación. Al igual que para las definiciones, es necesario escoger un nuevo nombre para la instancia obtenida. Además se incluye un consejo para demostrar la versión concreta del teorema, como una instancia funcional del teorema genérico en el que las funciones genéricas se sustituyen por las concretas tal y como se indica en la lista de asociación.

Este proceso es realizado por la macro `definstancia-*<teoria>*` (donde `<teoria>` es el nombre con el que se denomina a la teoría genérica que se desea instanciar) cuyos argumentos son la lista de asociación inicial, que relaciona las funciones para las que se define la teoría genérica con las concretas, y una cadena que se utiliza como sufijo para construir los nuevos nombres de los eventos instanciados. Por ejemplo, si `*multiconjuntos*` es la constante que almacena la teoría genérica de las relaciones de multiconjuntos, lo siguiente es una instancia de dicha teoría que construye la relación entre multiconjuntos de ordinales inducida por `e0-ord-<`:

```
(definstancia-*multiconjuntos*
 '((rel e0-ord-<)
   (mp  e0-ordinalp)
   (fn  id))
 "-ord")
```

Debido a restricciones del sistema, no podemos utilizar como argumento la constante que almacena la teoría genérica. Así, es necesario construir la macro `definstancia-*<teoria>*` para cada teoría genérica definida. Esto la hacemos con la macro `construye-teoria-generica` de la siguiente forma:

```
(construye-teoria-generica *<teoria>*)
```

Un último paso en la automatización de este proceso consiste en la macro `define-teoria-generica`, que se encarga de almacenar los eventos indicados con la palabra reservada `exportar` en la constante que representa la teoría y realiza la construcción de la macro `definstancia-*<teoria>*`:

```
(define-teoria-generica *<teoria>*
  (encapsulate
    (...)
    (propiedades de las funciones genéricas)
   )

  (exportar
    (defun f1 (...) ...))

  (exportar
    (defthm thm1 ...))

  (defun f2 (...) ...)

  (defthm thm2 ...)

  (exportar
    (defthm thm3 ...))

 )
```

Esta última macro sólo se puede utilizar cuando la teoría genérica se desarrolla en un único libro. Cuando los eventos que forman la teoría genérica se encuentran desarrollados en distintos libros, hay que crear explícitamente la constante que almacena la teoría genérica y evaluar la macro `construye-teoria-generica` sobre ella.

6.2.3 Ejemplo

Presentamos a continuación un ejemplo de uso de la herramienta de instanciación genérica. A partir de una relación de orden total en un conjunto dado, definimos un procedimiento de ordenación por inserción sobre listas de elementos de dicho conjunto. En el desarrollo genérico se demuestra que con dicho procedimiento se obtienen listas ordenadas.

Primero asumimos en un encapsulado la existencia de las funciones para las que se define la teoría genérica, junto con sus propiedades:

129

```
(encapsulate
  (((menor * *) => *)
   ((dominio *) => *))

  (local (defun menor (x y)
           (declare (ignore x y))
           t))

  (local (defun dominio (x)
           (declare (ignore x))
           t))

  (defthm menor-es-transitivo
    (implies (and (dominio x) (dominio y) (dominio z)
                  (menor x y) (menor y z))
             (menor x z)))

  (defthm menor-es-total
    (implies (and (dominio x) (dominio y)
                  (not (menor x y)))
             (menor y x)))

 )
```

Una vez hecho esto, establecemos el conjunto de resultados (definiciones y teoremas) que forma la teoría genérica. En este caso definimos una función de ordenación por inserción y demostramos que devuelve listas ordenadas. Esta función es `ordena-insercion` y para definirla utilizamos la función `inserta-en-orden`, que añade un elemento a una lista ordenada, de forma que el resultado sea también ordenado.

130
```
(defun ordena-insercion (x)
  (if (endp x)
      nil
      (inserta-en-orden (car x)
                        (ordena-insercion (cdr x)))))

(defun inserta-en-orden (e x)
  (if (or (endp x) (menor e (car x)))
      (cons e x)
      (cons (car x)
            (inserta-en-orden e (cdr x)))))
```

Para demostrar que la función `ordena-insercion` ordena listas de elementos del conjunto de partida, definimos la función `lista-en-dominio`, que caracteriza las listas de elementos de dicho conjunto, y `lista-ordenada`, que comprueba si todos los elementos de una lista están ordenados:

131
```
(defun lista-en-dominio (x)
  (or (endp x)
      (and (dominio (car x))
           (lista-en-dominio (cdr x)))))

(defun lista-ordenada (x)
  (or (endp x)
      (endp (cdr x))
      (and (menor (car x) (cadr x))
           (lista-ordenada (cdr x)))))
```

Antes de demostrar las propiedades de `ordena-insercion` probamos que la función `inserta-en-orden` se comporta adecuadamente con respecto a las funciones anteriores: `lista-en-dominio` y `lista-ordenada`:

132
```
(defthm inserta-en-orden-preserva-el-dominio
  (implies (and (dominio e)
                (lista-en-dominio x))
           (lista-en-dominio (inserta-en-orden e x))))

(defthm inserta-en-orden-es-ordenada
  (implies (and (lista-en-dominio x)
                (lista-ordenada x)
                (dominio e))
           (lista-ordenada (inserta-en-orden e x))))
```

Finalmente se demuestra que la función `ordena-insercion` devuelve una lista ordenada:

133
```
(defthm ordena-insercion-preserva-el-dominio
  (implies (lista-en-dominio x)
           (lista-en-dominio (ordena-insercion x))))

(defthm ordena-insercion-es-ordenada
  (implies (lista-en-dominio x)
           (lista-ordenada (ordena-insercion x))))
```

A continuación se define la teoría genérica `*insercion-ordenada*` para las funciones `dominio` y `menor`. Las propiedades asumidas para estas funciones son las establecidas por los eventos `menor-es-transitivo` y `menor-es-total` en *129*. Los eventos que proporciona esta teoría son las definiciones presentadas en *130* y *131* y el teorema `ordena-insercion-es-ordenada`. Se asigna a la constante `*insercion-ordenada*` la lista de los eventos que se quieren instanciar y se evalúa la macro `construye-teoria-generica` sobre esta constante:

```
(defconst *insercion-ordenada*
  '((defun ordena-insercion (x)
      ...)
    (defun inserta-en-orden (e x)
      ...)
    (defun lista-en-dominio (x)
      ...)
    (defun lista-ordenada (x)
      ...)
    (defthm ordena-insercion-es-ordenada
      ...))
 )

(construye-teoria-generica *insercion-ordenada*)
```

Ahora disponemos de la macro `definstancia-*insercion-ordenada*` con la que podemos instanciar la teoría genérica construida. A continuación se muestra un uso de esta macro, para construir un procedimiento de ordenación por inserción para listas de números enteros:

134
```
(defun orden-enteros (x y)
  (<= x y))

(definstancia-*insercion-ordenada*
  ((dominio   integerp)
   (menor     orden-enteros))
  "-enteros")
```

Cuando esta herramienta sea utilizada en algún punto de esta memoria, omitiremos el proceso de creación de la constante que almacena la teoría genérica, si bien se indicarán las funciones para las que se define, sus propiedades asumidas y los eventos que proporciona. Otros ejemplos del uso de la herramienta de instanciación genérica se pueden encontrar en [52].

Sumario

En este capítulo:

- Se ha probado formalmente que la relación entre multiconjuntos inducida por una relación bien fundamentada es bien fundamentada.
- El resultado anterior se ha demostrado de una manera general, lo que nos ha permitido definir un comando que genera de forma automática la relación bien fundamentada entre multiconjuntos a partir de una relación bien fundamentada concreta.
- Se ha desarrollado el concepto de teoría genérica, definida para un conjunto de funciones con determinadas propiedades, en la que se demuestran resultados de segundo orden sobre dichas funciones. Estos resultados pueden ser reutilizados para cualquier conjunto de funciones concretas cumpliendo las mismas propiedades que las funciones sobre las que se define la teoría genérica.
- Se ha desarrollado unas herramientas que facilitan la labor de reutilizar los resultados que proporcionan las teorías genéricas.

Capítulo 7

Sistemas de transformación proposicionales

En los capítulos 4 y 5 se han presentado dos métodos para decidir la satisfacibilidad de una fórmula proposicional. En estos métodos se construyen ciertas estructuras a partir de las fórmulas, se define la semántica de estas estructuras y se buscan asignaciones en las que tengan un valor distinguido. En este capítulo presentamos un marco genérico en el que se pueden expresar como casos particulares este tipo de procedimientos para decidir la satisfacibilidad de una fórmula.

Este marco genérico es interesante por dos razones. Por un lado, el formalismo teórico abstrae determinadas propiedades de los distintos métodos, a partir de las cuales todos ellos funcionan de la misma forma. Estas propiedades son la base para demostrar la corrección y completitud de los procedimientos de decisión de satisfacibilidad. Por otro lado, la formalización del marco genérico y el uso de las herramientas de instanciación genérica nos permiten obtener de forma automática funciones verificadas que implementan los procedimientos de decisión de satisfacibilidad.

En la primera sección se define el marco genérico que se obtiene como resultado de abstraer ciertas propiedades en los métodos para decidir la satisfacibilidad. Este márco generico se desarrolla tanto para la semántica clásica como para la semántica de Kleene. En ambos casos se formaliza la abstracción y se define una teoría genérica que, gracias a las herramientas de instanciación funcional, permite instanciar dicha abstracción.

En la segunda y tercera sección se presentan respectivamente el método de los tableros semánticos y el de secuentes como casos particulares del marco genérico. En estas secciones se presentan los eventos necesarios para instanciar con éxito la teoría asociada al marco genérico.

7.1 Un marco genérico para la decisión de satisfacibilidad

Analizando los métodos basados en tableros semánticos y secuentes presentados en los capítulos anteriores, se puede observar un comportamiento común. Estos métodos no trabajan directamente con fórmulas sino con unos objetos construidos a partir de ellas. Los objetos se modifican repetidamente usando reglas de expansión que reducen su complejidad. Estas reglas preservan el significado de los objetos. Eventualmente, a partir de cierto tipo de objetos simples, se pueden obtener asignaciones que prueban la satisfacibilidad de la fórmula original. A estas asignaciones las llamaremos *asignaciones distinguidas*. Si no se encuentra ninguno de estos objetos simples entonces la fórmula original es insatisfacible.

De esta forma, el método de los tableros semánticos puede ser visto como la aplicación de un conjunto de reglas de expansión actuando sobre las ramas de árboles etiquetados con fórmulas. Estas ramas son los objetos sobre los que trabaja el método. Las reglas son aplicadas hasta obtener una rama sin fórmulas complementarias en la que todas las fórmulas no literales hayan sido expandidas. Para esta rama se puede construir fácilmente una asignación distinguida, definida de forma que todos los literales positivos de la rama sean ciertos y todos los negativos falsos. Si no se obtiene dicha rama, entonces todas las ramas contendrán fórmulas complementarias. Esto indica que la fórmula original es insatisfacible.

En el método basado en secuentes, los objetos sobre los que se trabaja son los secuentes. Estos objetos son modificados de acuerdo con las reglas de expansión hasta obtener un secuente atómico que no sea un axioma. A partir de este secuente se puede construir fácilmente una asignación distinguida, definida de forma que todos los átomos de la parte izquierda del secuente sean ciertos y los de la parte derecha falsos. Esta asignación distinguida hace falso el secuente a partir del que se construye y, por las propiedades de las reglas de expansión, también hace falso el secuente considerado inicialmente. Si este secuente inicial se construye de forma que sea lógicamente equivalente a la negación de una fórmula, estaremos demostrando que dicha fórmula es satisfacible.

En esta sección formalizamos un marco genérico que refleja el comportamiento común de estos métodos. Los eventos relacionados con la semántica clásica presentados en esta sección se encuentran en `SAT-generico.lisp`, y los relacionados con la semántica de Kleene en `SAT-generico-K.lisp`.

7.1.1 Un algoritmo genérico de decisión de satisfacibilidad: SAT_g

Definición 7.1 *Un* **sistema de transformación proposicional** *en Σ (STP) es una tripleta $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, donde $\mathcal{O}$, $\mathcal{R}$ y $\mathcal{V}$ son conjuntos tales que $\mathcal{R} \subseteq \mathcal{O} \times (\mathcal{O}^* \cup \{\mathbf{t}\})$ y $\mathcal{V} \subseteq \mathcal{O} \times \mathbb{V}_\Sigma$. Donde $\mathbb{V}_\Sigma = \{\sigma : \Sigma \longrightarrow \mathbb{B}\}$ y $\mathcal{O}^*$ es el conjunto*

de las secuencias finitas de elementos de $\mathcal{O}$.

Diremos que $\mathcal{O}$ es el conjunto de los *objetos proposicionales* (o simplemente *objetos*) y $\mathcal{R}$ el conjunto de las *reglas de expansión*. Utilizaremos la notación $O \rightsquigarrow_{\mathcal{G}} L$ para indicar que (O, L) es un elemento de $\mathcal{R}$. Si $(O, \sigma) \in \mathcal{V}$ diremos que σ es una *asignación distinguida* de O y lo notaremos de la siguiente forma, $\sigma \models_{\mathcal{G}} O$.

Nótese que se permite la existencia de reglas de la forma $O \rightsquigarrow_{\mathcal{G}} \langle\rangle$ y reglas de la forma $O \rightsquigarrow_{\mathcal{G}} \mathbf{t}$. Las primeras son reglas que sirven para eliminar objetos a partir de los cuales es imposible obtener una asignación distinguida. Las segundas sirven para indicar que a partir del objeto O se puede obtener una asignación distinguida.

Para el método de los tableros semánticos presentado en el capítulo 4 los objetos son las ramas, las asignaciones distinguidas de una rama son aquellas que la hacen cierta y las reglas de expansión se construyen a partir de las reglas de expansión de tableros mostradas en la figura 4.1, junto con reglas que eliminan ramas cerradas y reglas que indican que, a partir de ramas no cerradas con todas las fórmulas no literales expandidas, se puede obtener un modelo.

Definición 7.2 *Dado un STP* $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$*:*

1. *Una* **regla de computación** *es una función* $r : \mathcal{O} \longrightarrow \mathcal{O}^* \cup \{\mathbf{t}\}$ *tal que* $r \subseteq \mathcal{R}$.

2. *Una* **función de representación** *es una función* $i : \mathbb{P}(\Sigma) \longrightarrow \mathcal{O}$.

3. *Una* **función de medida** *es una función* $\mu : \mathcal{O} \longrightarrow \mathcal{O}rd$.

4. *Una* **función modelo** *es una función* $\sigma : \mathcal{O}_{\mathbf{t}} \longrightarrow \mathbb{V}_{\Sigma}$, *donde* $\mathcal{O}_{\mathbf{t}} = \{O \in \mathcal{O} : O \rightsquigarrow_{\mathcal{G}} \mathbf{t}\}$.

Una regla de computación establece un criterio de utilización de las reglas de expansión. En el método de los tableros semánticos el criterio utilizado elimina primero las ramas cerradas, después intenta aplicar las regla de expansión para las dobles negaciones, α-fórmulas y β-fórmulas, en este orden, y finalmente, detecta la existencia de ramas no cerradas con todas sus fórmulas no literales expandidas.

Una función de representación establece una relación entre las fórmulas y los objetos proposicionales, proporcionando así el objeto inicial para el que se va a buscar una asignación distinguida. En el caso de los tableros semánticos, la función de representación construye una lista unitaria a partir de una fórmula dada.

La función de medida sirve para justificar la terminación del procedimiento de búsqueda de una asignación distinguida. La función modelo proporciona dicha asignación distinguida a partir de cierto tipo de objetos básicos. En el método de los tableros semánticos, la función de medida es la medida uniforme de una

rama y la función modelo es la que construye una asignación en la que todos los literales de una rama son ciertos.

Definición 7.3 *Decimos que un STP es* **exitoso** *con respecto a una regla de computación r, una función de representación i, una función de medida μ y una función modelo σ si se verifican las siguientes propiedades:*

$\mathcal{P}_1$: $O_i \in r(O) \Longrightarrow \mu(O_i) < \mu(O)$

$\mathcal{P}_2$: $F \in \mathbb{P}(\Sigma) \Longrightarrow (\sigma \models F \Longleftrightarrow \sigma \models_{\mathcal{G}} i(F))$

$\mathcal{P}_3$: $O \in \mathcal{O} \wedge r(O) \neq \mathbf{t} \Longrightarrow (\sigma \models_{\mathcal{G}} O \Longleftrightarrow \exists O_i \in r(O), \sigma \models_{\mathcal{G}} O_i)$

$\mathcal{P}_4$: $O \in \mathcal{O} \wedge r(O) = \mathbf{t} \Longrightarrow \sigma(O) \models_{\mathcal{G}} O$

Más adelante se definirá el algoritmo $SAT_{\mathcal{G}}$ para comprobar la satisfacibilidad de una fórmula proposicional y se demostrará que, para todo STP exitoso $\mathcal{G}$, dicho algoritmo (basado en la regla de computación y la función de representación que hacen que $\mathcal{G}$ sea exitoso) termina para cualquier fórmula proposicional que reciba como entrada y es correcto y completo. La formalización del algoritmo y el establecimiento de sus propiedades se basan en que $\mathcal{G}$ es exitoso. Veamos como se formaliza este hecho.

La propiedad de ser exitoso se basa en la existencia de cuatro funciones con determinadas propiedades. Para poder formalizar este tipo de situación en ACL2 utilizamos el mecanismo de encapsulado. De esta forma se introducen símbolos de función para los cuales se asumen un conjunto de propiedades. Las funciones y sus interpretaciones en la formalización son las siguientes:

`(gen-objeto O)`	$O \in \mathcal{O}$
`(gen-asignacion-distinguida sigma O)`	$\sigma \models_{\mathcal{G}} O$
`(gen-regla-computacion O)`	$r(O)$
`(gen-representacion F)`	$i(F)$
`(gen-medida O)`	$\mu(O)$
`(gen-modelo O)`	$\sigma(O)$

Algunas propiedades de estas funciones vienen dadas por sus dominios de definición y sus valores. Así se han de verificar:

1. $r : \mathcal{O} \longrightarrow \mathcal{O}^* \cup \{\mathbf{t}\}$, cuya formalización es:

135
```
(defthm member-gen-regla-computacion-es-gen-objeto
  (implies (and (gen-objeto O-1)
                (member-equal O-2
                              (gen-regla-computacion O-1)))
           (gen-objeto O-2)))
```

2. $i : \mathbb{P}(\Sigma) \longrightarrow \mathcal{O}$, cuya formalización es:

```
(defthm gen-representacion-es-gen-objeto                    136
  (implies (es-proposicional F)
           (gen-objeto (gen-representacion F))))
```

3. $\mu : \mathcal{O} \longrightarrow \mathcal{O}rd$, cuya formalización es:

```
(defthm gen-medida-es-e0-ordinalp                           137
  (e0-ordinalp (gen-medida 0)))
```

Obsérvese que en la tercera propiedad no se exige que el argumento de la función `gen-medida` sea un elemento de $\mathcal{O}$. La propiedad mostrada en 137 es más general y siempre se puede obtener sin más que definir la función μ igual a 0 para cualquier elemento que no se encuentre en $\mathcal{O}$.

Las propiedades $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ y $\mathcal{P}_4$ definen el concepto de STP exitoso. Su formalización es la siguiente:

```
(defthm P1                                                  138
  (implies (member-equal O-2 (gen-regla-computacion O-1))
           (e0-ord-< (gen-medida O-2)
                     (gen-medida O-1))))
```

Obsérvese que no se exige que el objeto `O-1` sea un elemento de $\mathcal{O}$. Esta propiedad es más general y siempre se puede obtener sin más que definir la función r igual a $\langle\rangle$ para cualquier elemento que no se encuentre en $\mathcal{O}$.

```
(defthm P2                                                  139
  (implies (es-proposicional F)
           (iff (gen-asignacion-distinguida
                   sigma (gen-representacion F))
                (modelo sigma F))))
```

Para poder expresar la propiedad $\exists O_i \in r(O), \sigma \models_{\mathcal{G}} O_i$, definimos la función `gen-asignacion-distinguida-lista`, que comprueba si una asignación es distinguida para alguno de los objetos de una lista:

140
```
(defthm P3
  (implies (and (gen-objeto O)
                (not (equal (gen-regla-computacion O) t)))
           (iff (gen-asignacion-distinguida-lista
                  sigma (gen-regla-computacion O))
                (gen-asignacion-distinguida sigma O))))

(defun gen-asignacion-distinguida-lista (sigma lista-O)
  (cond ((endp lista-O) nil)
        (t (or (gen-asignacion-distinguida
                 sigma (car lista-O))
               (gen-asignacion-distinguida-lista
                 sigma (cdr lista-O))))))
```

141
```
(defthm P4
  (implies (and (gen-objeto O)
                (equal (gen-regla-computacion O) t))
           (gen-asignacion-distinguida (gen-modelo O) O)))
```

Dado un STP $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, una regla de computación r y una función de representación i, definimos el algoritmo $SAT_{\mathcal{G}}$ para demostrar la satisfacibilidad de una fórmula proposicional:

Algoritmo 7.4 ($SAT_{\mathcal{G}}$) *El dato de entrada de este algoritmo es una fórmula F, y actúa de la siguiente forma:*

1. *Inicialmente se considera la lista de objetos $\langle i(F) \rangle$*
2. *Dada una lista de objetos $\langle O_1, \ldots, O_n \rangle$, se selecciona un elemento O_j*
   (a) *Si $r(O_j) = \mathbf{t}$, entonces el algoritmo termina y devuelve $\langle O_j \rangle$.*
   (b) *Si $r(O_j) = \langle O'_1, \ldots, O'_m \rangle$, entonces el algoritmo vuelve al paso 2 con la lista $\langle O'_1, \ldots, O'_m, O_1, \ldots, O_{j-1}, O_{j+1}, \ldots, O_n \rangle$.*
3. *Si la lista de objetos se queda vacía, el algoritmo termina y devuelve $\mathbf{f}$.*

La idea intuitiva es la siguiente: dada F, el algoritmo comienza con el objeto $i(F)$ y repetidamente aplica las reglas de $\mathcal{R}$ hasta que se obtiene $\mathbf{t}$ o no quedan más objetos. La terminación de este proceso está garantizada por una función de medida μ con respecto a la cual los objetos expandidos sean menores que los originales. Si el proceso termina al obtener $\mathbf{t}$, entonces existe un objeto O tal que $r(O) = \mathbf{t}$ y, a partir de éste, se puede obtener una asignación distinguida.

Esta asignación es un modelo de la fórmula original. Si el proceso termina al no quedar objetos, la fórmula original es insatisfacible.

Nótese que en el punto 2 se escoge un elemento cualquiera O_j. Esta elección incorpora cierta heurística al algoritmo. En la formalización se considera la existencia de una función `gen-seleccion` que selecciona un elemento de una lista.

142
```
(defthm gen-seleccion-member
  (implies (consp lista-O)
           (member-equal (gen-seleccion lista-O) lista-O)))
```

Las funciones anteriores y sus propiedades son asumidas en un encapsulado. Una vez hecho esto, se define la función `SAT-generico` que implementa el algoritmo $SAT_{\mathcal{G}}$. Esta función se basa en `OBJ-SAT-generico`, que implementa los puntos 2 y 3 de la descripción del algoritmo.

143
```
(defun SAT-generico (F)
  (OBJ-SAT-generico (list (gen-representacion F))))

(defun OBJ-SAT-generico (lista-O)
  (declare (xargs :measure (gen-medida-lista lista-O)
                  :well-founded-relation mul-e0-ord-<))
  (if (endp lista-O)
     nil
     (let* ((O (gen-seleccion lista-O))
            (resto (elimina-una (gen-seleccion lista-O) lista-O))
            (expansion (gen-regla-computacion O)))
       (cond ((equal expansion t)
              (list O))
             (t (OBJ-SAT-generico (append expansion resto)))))))
```

Para que el sistema admita la función `OBJ-SAT-generico` es necesario que se pueda demostrar su terminación. Para ello hemos proporcionado una medida de los argumentos, `gen-medida-lista`, definida como la extensión a listas de la medida μ. La prueba de la terminación se detalla en la siguiente sección.

La función `OBJ-SAT-generico` busca un objeto a partir del cuál se pueda obtener una asignación distinguida. Esta es la versión genérica de la función que implementa el algoritmo $MOD_{\mathcal{R}}$ para el método de los tableros semánticos y $CONTRAMOD_{\mathcal{S}}$ para el método basado en secuentes. De esta forma, si la función de representación verifica la propiedad $\mathcal{P}_2$, la función `SAT-generico` proporciona información suficiente para construir un modelo de una fórmula satisfacible F. Para ello basta con evaluar la función modelo σ sobre el objeto devuelto

por $SAT_{\mathcal{G}}$, cuando éste es aplicado con éxito sobre F. Notaremos $MOD_{\mathcal{G}}$ a esta construcción, implementada por la función `MOD-generico`.

144
```
(defun MOD-generico (F)
  (gen-modelo (car (SAT-generico F))))
```

7.1.2 Terminación de $SAT_{\mathcal{G}}$

Para demostrar la terminación del algoritmo $SAT_{\mathcal{G}}$ basta con demostrar que la función `OBJ-SAT-generico` termina. El argumento de esta función es una lista de objetos proposicionales. Para demostrar su terminación hemos considerado una extensión de la medida μ a listas de objetos:

Definición 7.5 *Dada una función de medida $\mu : \mathcal{O} \longrightarrow \mathcal{O}rd$, la extensión de μ a $\mathcal{O}^*$ es la función $\mu^* : \mathcal{O}^* \longrightarrow \mathcal{M}(\mathcal{O}rd)$ definida como sigue, $\mu^*(\langle O_1, \ldots, O_n\rangle) = \{\!\{\mu(O_1), \ldots, \mu(O_n)\}\!\}$.*

La función `gen-medida-lista` formaliza este concepto:

145
```
(defun gen-medida-lista (lista-O)
  (if (endp lista-O)
      nil
      (cons (gen-medida (car lista-O))
            (gen-medida-lista (cdr lista-O)))))
```

Estas medidas se comparan con respecto a la extensión a multiconjuntos definida en 6.3 de la relación de orden entre los ordinales. Notaremos $<_{\mathcal{M}}$ a esta extensión. La función que la formaliza es `mul-eO-ord-<`. Su definición y propiedades se obtienen al evaluar el siguiente evento:

146
```
(defmul (EO-ORD-< NIL EO-ORDINALP EO-ORD-<-FN NIL NIL))
```

Teorema 7.6 *Dados un STP $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V}\rangle$, una regla de computación r, una función de representación i y una función de medida μ verificando $\mathcal{P}_1$, entonces el algoritmo $SAT_{\mathcal{G}}$ termina para cualquier fórmula inicial.*

Demostración:

Para demostrar la terminación de $SAT_{\mathcal{G}}$, tendremos que probar que el punto 2 es un bucle finito. Supongamos que $\langle O_1, \ldots, O_n\rangle$ es la lista de objetos en dicho punto, O_j el objeto seleccionado y $r(O_j) = \langle O'_1, \ldots, O'_m\rangle$.

Por la propiedad $\mathcal{P}_1$ se tiene que para todo k, $\mu(O'_k) < \mu(O_j)$. Por tanto se verifica $\mu^*(\langle O'_1, \ldots, O'_m\rangle) <_{\mathcal{M}} \mu^*(\langle O_j\rangle)$. Y, por la definición de la extensión a

multiconjuntos dada en 6.3, se tiene que

$$\mu^*(\langle O'_1, \ldots, O'_m, O_1, \ldots, O_{j-1}, O_{j+1}, \ldots, O_n\rangle) <_{\mathcal{M}} \mu^*(\langle O_1, \ldots, O_n\rangle)$$

De esta forma, en cada iteración, la medida μ^* de la lista considerada en el punto 2 es menor, con respecto a $<_{\mathcal{M}}$, que en el paso anterior. Luego el punto 2 es un bucle finito, puesto que $<_{\mathcal{M}}$ es una relación bien fundamentada.

□

En la formalización este resultado queda como sigue:

147

```
(defthm OBJ-SAT-generico-terminacion
  (let* ((O (gen-seleccion lista-O))
         (resto (remove-one (gen-seleccion lista-O) lista-O))
         (expansion (gen-regla-computacion O)))
    (implies
      (and (consp lista-O)
           (not (equal expansion t)))
      (mul-e0-ord-< (gen-medida-lista (append expansion resto))
                    (gen-medida-lista lista-O)))))
```

7.1.3 Corrección y completitud de $SAT_{\mathcal{G}}$

Teorema 7.7 *Sea $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V}\rangle$ un STP exitoso con respecto a una regla de computación r, una función de representación i, una función de medida μ y una función modelo σ. Entonces:*

1. *$SAT_{\mathcal{G}}$ es correcto: Dada una fórmula F, si $SAT_{\mathcal{G}}(F) \neq \mathbf{f}$, entonces F es satisfacible. Además, si $SAT_{\mathcal{G}}(F) = \langle O\rangle$, entonces $\sigma(O) \models F$.*

2. *$SAT_{\mathcal{G}}$ es completo: Dada una fórmula F, si F es satisfacible entonces $SAT_{\mathcal{G}}(F) \neq \mathbf{f}$.*

Demostración:

Antes de pasar a demostrar el teorema, obsérvese que si $\langle O_1, \ldots, O_n\rangle$ es la lista de objetos en el punto 2 del algoritmo, O_j es el elemento seleccionado y $r(O_j) = \langle O'_1, \ldots, O'_m\rangle$ entonces, por la propiedad $\mathcal{P}_3$, se tiene que para cualquier asignación σ, existe O en $\langle O_1, \ldots, O_n\rangle$ tal que $\sigma \models_{\mathcal{G}} O$ si y sólo si existe O' en $\langle O'_1, \ldots, O'_m, O_1, \ldots, O_{j-1}, O_{j+1}, \ldots, O_n\rangle$ tal que $\sigma \models_{\mathcal{G}} O'$.

$SAT_{\mathcal{G}}$ **es correcto**: Si $SAT_{\mathcal{G}}(F) = \langle O\rangle$ entonces $r(O) = \mathbf{t}$ y, por $\mathcal{P}_4$, $\sigma(O) \models_{\mathcal{G}} O$. Así, por la observación anterior, en toda lista considerada en el punto 2 existe O' tal que $\sigma(O) \models_{\mathcal{G}} O'$. Por lo tanto, esto también ocurre en la lista inicial, $\langle i(F)\rangle$, es decir, $\sigma(O) \models_{\mathcal{G}} i(F)$, y, por $\mathcal{P}_2$, $\sigma(O) \models F$.

$SAT_{\mathcal{G}}$ **es completo**: Sea $\sigma \models F$, entonces por $\mathcal{P}_2$, $\sigma \models_{\mathcal{G}} i(F)$. Así, por la observación indicada al principio de esta demostración, en toda lista considerada en el punto 2 existe O' tal que $\sigma \models_{\mathcal{G}} O'$. Por lo tanto, la lista en el punto 2 no

puede quedarse vacía y, como el algoritmo termina, en algún paso se considerará un objeto O tal que $r(O) = \mathbf{t}$. Por tanto, $SAT_{\mathcal{G}}(F) = \langle O \rangle \neq \mathbf{f}$.

□

Los siguientes teoremas establecen los resultados de corrección y completitud de $SAT_{\mathcal{G}}$. La condición acerca de la satisfacibilidad de la fórmula `F` se establece como se indicó en la sección 3.2.4.

148
```
(defthm correccion-SAT-generico
  (implies (and (es-proposicional F)
                (SAT-generico F))
           (modelo (MOD-generico F) F)))

(defthm completitud-SAT-generico
  (implies (and (es-proposicional F)
                (modelo sigma F))
           (SAT-generico F)))
```

Previamente se han demostrado propiedades de corrección y completitud similares acerca de `OBJ-SAT-generico` para una lista de objetos proposicionales `lista-O`:

149
```
(defthm correccion-OBJ-SAT-generico
  (implies (and (gen-objeto-lista lista-O)
                (OBJ-SAT-generico lista-O))
           (gen-asignacion-distinguida-lista
            (gen-modelo (car (OBJ-SAT-generico lista-O)))
            lista-O)))

(defthm completitud-OBJ-SAT-generico
  (implies (and (gen-objeto-lista lista-O)
                (gen-asignacion-distinguida-lista sigma lista-O))
           (OBJ-SAT-generico lista-O)))
```

En estos eventos la función `gen-objeto-lista` comprueba que su argumento es una lista de elementos de $\mathcal{O}$:

150
```
(defun gen-objeto-lista (lista-O)
  (cond ((endp lista-O) t)
        (t (and (gen-objeto (car lista-O))
                (gen-objeto-lista (cdr lista-O))))))
```

Los eventos mostrados en *148* se obtienen como una instancia de los presentados en *149* cuando el valor `lista-O` es `(list (gen-representacion F))`.

7.1.4 Un algoritmo genérico de decisión de validez: $DAT_{\mathcal{G}}$

El algoritmo anterior se puede utilizar para decidir la validez de una fórmula F, para ello basta con comprobar que el algoritmo $SAT_{\mathcal{G}}$ devuelve **f** cuando se aplica a $\neg F$.

Algoritmo 7.8 ($DAT_{\mathcal{G}}$) *Dada una fórmula $F \in \mathbb{P}(\Sigma)$, $DAT_{\mathcal{G}}(F)$ es* **t** *si y sólo si $SAT_{\mathcal{G}}(\neg F)$ es* **f**.

La función que implementa este algoritmo es la siguiente:

151
```
(defun DAT-generico (F)
  (not (SAT-generico (negacion F))))
```

Teorema 7.9 *Sea $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$ un STP exitoso con respecto a una regla de computación r, una función de representación i, una función de medida μ y una función modelo σ. Dada una fórmula $F \in \mathbb{P}(\Sigma)$, F es válida si y sólo si $DAT_{\mathcal{G}}(F) = \mathbf{t}$.*

La prueba de este teorema se tiene gracias a las propiedades de corrección y completitud de $SAT_{\mathcal{G}}$. Los siguientes eventos establecen los resultados de corrección y completitud de la función que implementa el algoritmo $DAT_{\mathcal{G}}$:

152
```
(defthm correccion-DAT-generico
  (implies (and (es-proposicional F)
                (DAT-generico F))
           (modelo sigma F)))

(defthm completitud-DAT-generico
  (implies (and (es-proposicional F)
                (not (DAT-generico F)))
           (not (modelo (MOD-generico (negacion F)) F))))
```

La prueba de estos resultados se obtiene como una instancia de los resultados presentados en *149* cuando el valor de la variable `lista-O` es `(list (gen-representacion (negacion F))`.

7.1.5 La teoría genérica `*sat-generico*`

Utilizando las herramientas de construcción de teorías genéricas presentadas en la sección 6.2, se ha definido la teoría genérica `*sat-generico*` para facilitar la reutilización de los resultados anteriores. Las funciones para las que se define esta teoría son las siguientes:

```
gen-objeto
gen-asignacion-distinguida
gen-asignacion-distinguida-lista
gen-regla-computacion
gen-representacion
gen-medida
gen-modelo
gen-seleccion
```

Las propiedades asumidas para estas funciones son las presentadas en *135*, *136*, *137*, *142* y los eventos correspondientes a $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ y $\mathcal{P}_4$, mostrados en *138*, *139*, *140* y *141* respectivamente.

Esta teoría proporciona, entre otros eventos, las definiciones de las funciones que implementan $SAT_{\mathcal{G}}$, $DAT_{\mathcal{G}}$ y $MOD_{\mathcal{G}}$, el teorema de terminación de $SAT_{\mathcal{G}}$ mostrado en *147* y los resultados de corrección y completitud mostrados en *149*, *148* y *152*.

Para utilizar esta teoría bastará con proporcionar versiones concretas de las funciones para las que está definida y demostrar que cumplen las propiedades asumidas en la misma. Una vez hecho esto se puede utilizar la herramienta de instanciación genérica presentada en la sección 6.2, para obtener algoritmos verificados para decidir la satisfacibilidad y la validez de una fórmula proposicional.

7.1.6 Sistemas de transformación proposicionales en $\mathbb{K}$

Un desarrollo similar al presentado en las secciones anteriores se puede realizar en el contexto de la semántica de Kleene, basta con reconsiderar los conceptos relacionados con la semántica. El primero de estos conceptos es el de sistema de transformación proposicional:

Definición 7.10 *Un* **$\mathbb{K}$-sistema de transformación proposicional** *en Σ ($\mathbb{K}$-STP) es una tripleta $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, donde $\mathcal{O}$, $\mathcal{R}$ y $\mathcal{V}$ son conjuntos tales que $\mathcal{R} \subseteq \mathcal{O} \times (\mathcal{O}^* \cup \{\mathbf{t}\})$ y $\mathcal{V} \subseteq \mathcal{O} \times \mathbb{K}_\Sigma$. Donde $\mathbb{K}_\Sigma = \{\sigma : \Sigma \longrightarrow \mathbb{K}\}$.*

Los elementos de un $\mathbb{K}$-STP se interpretan de igual forma que en el caso clásico, la única diferencia es que los elementos de $\mathcal{V}$ son pares formados por objetos proposicionales y $\mathbb{K}$-asignaciones. Otro concepto a reconsiderar es el de función modelo:

Definición 7.11 *Dado un $\mathbb{K}$-STP $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, una función $\mathbb{K}$-modelo es una función $\sigma : \mathcal{O}_{\mathbf{t}} \longrightarrow \mathbb{K}_\Sigma$, donde $\mathcal{O}_{\mathbf{t}} = \{O \in \mathcal{O} : O \rightsquigarrow_{\mathcal{G}} \mathbf{t}\}$.*

Finalmente definimos el concepto de $\mathbb{K}$-STP exitoso, basta con reformular la propiedad $\mathcal{P}_2$ con respecto al concepto de modelo en $\mathbb{K}$.

Definición 7.12 *Decimos que un* $\mathbb{K}$*-STP es* **exitoso** *con respecto a una regla de computación* r*, una función de representación* i*, una función de medida* μ *y una función* $\mathbb{K}$*-modelo* σ *si se verifican las propiedades* $\mathcal{P}_1$*,* $\mathcal{P}_3$*,* $\mathcal{P}_4$ *y*

$\mathcal{P}_2'$*:* $F \in \mathbb{P}(\Sigma) \Longrightarrow (\sigma \models_{\mathbb{K}} F \Longleftrightarrow \sigma \models_{\mathcal{G}} i(F))$

La formalización de la propiedad $\mathcal{P}_2'$ es la siguiente:

153
```
(defthm P2-K
  (implies (es-proposicional F)
           (iff (gen-asignacion-distinguida
                  sigma (gen-representacion F))
                (modelo-K sigma F))))
```

Los procedimientos $SAT_{\mathcal{G}}$, $MOD_{\mathcal{G}}$ y $DAT_{\mathcal{G}}$ se definen en la semántica de Kleene de igual forma que en la semántica clásica, por tanto, consideramos las mismas funciones para implementarlos. Esta funciones se han definido en *143*, *144* y *151*. La prueba de la terminación de `OBJ-SAT-generico` es independiente de la semántica considerada.

Las propiedades de corrección y completitud en $\mathbb{K}$ de los algoritmos $SAT_{\mathcal{G}}$ y $DAT_{\mathcal{G}}$ se formalizan con respecto al concepto de modelo en $\mathbb{K}$, siguiendo las indicaciones dadas en la sección 3.3.4. Los eventos correspondientes son los siguientes:

154
```
(defthm correccion-SAT-generico
  (implies (and (es-proposicional F)
                (SAT-generico F))
           (modelo-K (MOD-generico F) F)))

(defthm completitud-SAT-generico
  (implies (and (es-proposicional F)
                (modelo-K sigma F))
           (SAT-generico F)))

(defthm correccion-DAT-generico
  (implies (and (es-proposicional F)
                (DAT-generico F)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-DAT-generico
  (implies (and (es-proposicional F)
                (not (DAT-generico F)))
           (no-modelo-K (MOD-generico (negacion F)) F)))
```

Todos estos resultados, junto con una teoría genérica que facilita su reutilización, se encuentran en el libro `SAT-generico-K.lisp`.

7.2 Un STP exitoso basado en tableros semánticos

El algoritmo basado en tableros semánticos descrito en el capítulo 4 se puede interpretar como una versión particular del algoritmo $SAT_{\mathcal{G}}$. Los objetos proposicionales son las ramas de los tableros semánticos. Las asignaciones distinguidas de una rama son aquellas que son modelo de dicha rama. El objeto inicial es una rama formada únicamente por la fórmula cuya satisfacibilidad se quiere comprobar. Estas ramas son expandidas de acuerdo a las reglas de la figura 4.1 para obtener nuevas ramas. Las ramas cerradas no son escogidas para expandir pues no poseen modelos. Esto se corresponde con una regla de expansión que "elimina" las ramas cerradas, es decir, cuya parte derecha es $\langle\rangle$. Si se encuentra una rama no cerrada con todas sus fórmulas no literales expandidas, el algoritmo termina y devuelve dicha rama. Esto se corresponde con una regla de expansión que indica que a partir de dicha rama se puede obtener un modelo, es decir, cuya parte derecha es **t**. Si todas las ramas están cerradas el algoritmo termina y devuelve **f**. Esto se corresponde con el hecho de que no queden ramas por procesar, ya que las ramas cerradas son eliminadas.

En esta sección desarrollamos un sistema de transformación proposicional basado en esta descripción. Primero describimos formalmente este sistema junto con una regla de computación, una función de representación, una función de medida y una función modelo adecuadas para demostrar que es exitoso. A continuación presentamos su formalización, relacionándola con la desarrollada en el capítulo 4. Finalmente comentamos cómo se hace este mismo proceso en la semántica de Kleene. Los eventos relacionados con la semántica clásica presentados en esta sección se encuentran en el libro `SAT-tableros.lisp`, los relacionados con la semántica de Kleene se encuentran en el libro `SAT-tableros-K.lisp`.

7.2.1 Descripción del STP exitoso $\mathcal{T}$

Consideramos el siguiente sistema de transformación proposicional basado en el método de los tableros semánticos:

Definición 7.13 *$\mathcal{T} = \langle \mathcal{O}_{\mathcal{T}}, \mathcal{R}_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}} \rangle$ es el STP basado en tableros semánticos en el que $\mathcal{O}_{\mathcal{T}}$ es el conjunto de listas finitas de fórmulas $\mathbb{P}(\Sigma)^*$, representando las ramas de los tableros, $\mathcal{V}_{\mathcal{T}}$ es el conjunto de pares (θ, σ) tales que σ es modelo de la rama θ, y $\mathcal{R}_{\mathcal{T}}$ es el conjunto de reglas de expansión representado por el siguiente conjunto de esquemas de regla:*

$\mathcal{R}_{\mathcal{T}1}$:	$\langle \Gamma_1, G, \Gamma_2, \neg G, \Gamma_3 \rangle \rightsquigarrow_{\mathcal{T}} \langle\rangle$
$\mathcal{R}_{\mathcal{T}2}$:	$\langle \Gamma_1, \neg\neg G, \Gamma_2 \rangle \rightsquigarrow_{\mathcal{T}} \langle \langle \Gamma_1, G, \Gamma_2 \rangle \rangle$
$\mathcal{R}_{\mathcal{T}3}$:	$\langle \Gamma_1, \alpha, \Gamma_2 \rangle \rightsquigarrow_{\mathcal{T}} \langle \langle \Gamma_1, \alpha_1, \alpha_2, \Gamma_2 \rangle \rangle$
$\mathcal{R}_{\mathcal{T}4}$:	$\langle \Gamma_1, \beta, \Gamma_2 \rangle \rightsquigarrow_{\mathcal{T}} \langle \langle \Gamma_1, \beta_1, \Gamma_2 \rangle, \langle \Gamma_1, \beta_2, \Gamma_2 \rangle \rangle$
$\mathcal{R}_{\mathcal{T}5}$:	$\Gamma \rightsquigarrow_{\mathcal{T}} \mathbf{t}$ *si* Γ *no contiene fórmulas no literales ni pares de fórmulas complementarias*

donde Γ, Γ_1, Γ_2 *y* Γ_3 *son listas finitas de fórmulas.*

Utilizando el STP anterior, junto con una función de representación $i_{\mathcal{T}}$ y una regla de computación $r_{\mathcal{T}}$ adecuadas, podemos expresar el algoritmo $SAT_{\mathcal{T}}$ como un caso particular del algoritmo $SAT_{\mathcal{G}}$. Además, para garantizar las propiedades de terminación, corrección y completitud, tenemos que proporcionar una función de medida $\mu_{\mathcal{T}}$ y una función modelo $\sigma_{\mathcal{T}}$, de forma que $\mathcal{T}$ sea exitoso con respecto a $r_{\mathcal{T}}$, $i_{\mathcal{T}}$, $\mu_{\mathcal{T}}$ y $\sigma_{\mathcal{T}}$.

Definición 7.14 *Para el STP* $\mathcal{T}$ *definimos:*

1. *La función de representación* $i_{\mathcal{T}}$ *tal que para toda* $F \in \mathbb{P}(\Sigma)$, $i_{\mathcal{T}}(F) = \langle F \rangle$.

2. *La regla de computación* $r_{\mathcal{T}}$ *que actúa de la siguiente forma. Dada una lista de fórmulas* θ*, si* θ *es una rama cerrada, se le aplica una regla del tipo* $\mathcal{R}_{\mathcal{T}1}$*. En otro caso se selecciona una fórmula* F *no literal en* θ*. Si* F *es una doble negación,* α*-fórmula o* β*-fórmula, se aplica a* θ *una regla del tipo* $\mathcal{R}_{\mathcal{T}2}$*,* $\mathcal{R}_{\mathcal{T}3}$ *o* $\mathcal{R}_{\mathcal{T}4}$*, según el caso, descomponiendo* F*. Si* θ *no contiene fórmulas no literales, se aplica a* θ *una regla del tipo* $\mathcal{R}_{\mathcal{T}5}$*.*

3. *La función de medida* $\mu_{\mathcal{T}}$*, que es la medida uniforme de una lista de fórmulas,* u^+*. Así, para toda lista de fórmulas* θ *se tiene:* $\mu_{\mathcal{T}}(\theta) = u^+(\theta)$.

4. *La función modelo* $\sigma_{\mathcal{T}}$ *tal que, para toda lista de fórmulas* θ*,* $\sigma_{\mathcal{T}}(\theta)$ *es la asignación definida de forma que* $\sigma_{\mathcal{T}}(\theta) \models p$ *si y sólo si* p *es un literal positivo que aparece en* θ*.*

Teorema 7.15 *El STP* $\mathcal{T}$ *es exitoso con respecto a la regla de computación* $r_{\mathcal{T}}$*, la función de representación* $i_{\mathcal{T}}$*, la función de medida* $\mu_{\mathcal{T}}$ *y la función modelo* $\sigma_{\mathcal{T}}$*.*

Demostración:

Para demostrar el teorema tendremos que probar las propiedades $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ y $\mathcal{P}_4$ para las funciones $r_{\mathcal{T}}$, $i_{\mathcal{T}}$, $\mu_{\mathcal{T}}$ y $\sigma_{\mathcal{T}}$.

$\mathcal{P}_1$: Dada $\theta_i \in r_{\mathcal{T}}(\theta)$, entonces θ_i es una rama obtenida al aplicar una regla de expansión de tableros a θ, luego por el teorema 4.18 se verifica que $\mu_{\mathcal{T}}(\theta_i) = u^+(\theta_i) < u^+(\theta) = \mu_{\mathcal{T}}(\theta)$.

$\mathcal{P}_2$: Dada la fórmula F, $\sigma \models_{\mathcal{T}} \langle F \rangle = i_{\mathcal{T}}(F)$ si y sólo si σ es modelo de algún elemento de $\langle F \rangle$, es decir, si y sólo si σ es modelo de F.

$\mathcal{P}_3$: Dada la rama θ tal que $r_{\mathcal{T}}(\theta) \neq \mathbf{t}$, entonces θ es una rama cerrada, en cuyo caso $r_{\mathcal{T}}(\theta) = \langle\rangle$, o tiene alguna fórmula no literal, en cuyo caso $r_{\mathcal{T}}(\theta)$ es la lista de las ramas obtenidas al aplicar una regla de expansión de tableros a θ.

Si θ es una rama cerrada, entonces $\sigma \not\models_{\mathcal{T}} \theta$ por el teorema 4.14 y además no existe ninguna rama en $r_{\mathcal{T}}(\theta)$. Por tanto se tiene la equivalencia $\sigma \models_{\mathcal{T}} \theta \Longleftrightarrow \exists\theta_i \in r_{\mathcal{T}}(\theta), \sigma \models_{\mathcal{T}} \theta_i$

Si θ tiene alguna fórmula no literal, por el teorema 4.11 se tiene que una asignación σ es modelo de θ ($\sigma \models_{\mathcal{T}} \theta$) si y sólo si es modelo de alguna de las ramas que se obtienen al aplicar a θ una regla de expansión de tableros ($\exists\theta_i \in r_{\mathcal{T}}(\theta), \sigma \models_{\mathcal{T}} \theta_i$).

$\mathcal{P}_4$: Dada la rama θ tal que $r_{\mathcal{T}}(\theta) = \mathbf{t}$, entonces θ es una rama no cerrada con todas sus fórmulas no literales expandidas y, por el teorema 4.15, la asignación σ definida de forma que $\sigma(p) = \top$ si y sólo si p es un literal positivo que aparece en θ, es un modelo de θ. Esta asignación es $\sigma_{\mathcal{T}}(\theta)$, por tanto $\sigma_{\mathcal{T}}(\theta) \models_{\mathcal{T}} \theta$

□

7.2.2 Formalización de $\mathcal{T}$

Para formalizar el conjunto de objetos proposicionales en el que se basa $\mathcal{T}$, tenemos que definir una función que caracterice dichos objetos. En la descripción de $\mathcal{T}$ se define el conjunto $\mathcal{O}_{\mathcal{T}}$ como el formado por las listas finitas de fórmulas. De esta forma, para caracterizar la pertenencia a dicho conjunto, basta con comprobar que todos los elementos de una lista son fórmulas. En *54* se definió la función `es-rama-tablero` que realiza dicha comprobación. Para mantener la correspondencia con los nombres de las funciones utilizadas en la formalización del marco genérico, llamamos a esta función `tab-objeto`:

155
```
(defun tab-objeto (rama)
  (or (endp rama)
      (and (es-proposicional (car rama))
           (tab-objeto (cdr rama)))))
```

Las asignaciones distinguidas de una rama θ son los modelos de dicha rama. En *55* se definió la función `modelo-rama` que comprueba si una asignación es modelo de una rama. De esta forma, la propiedad $\sigma \models_{\mathcal{T}} O$ se caracteriza utilizando dicha función. Como antes, para mantener la correspondencia con la formalización del marco genérico, la llamamos `tab-asignacion-distinguida`:

156
```
(defun tab-asignacion-distinguida (sigma rama)
  (cond ((endp rama) t)
        (t (and (modelo sigma (car rama))
                (tab-asignacion-distinguida sigma (cdr rama))))))
```

La regla de computación $r_{\mathcal{T}}$ no está completamente determinada puesto que, si la rama a la que hay que aplicarla no está cerrada, se ha de escoger una fórmula de dicha rama y no se indica el criterio que se utiliza para hacer esto. De hecho, esto también ocurría en el procedimiento de búsqueda de un modelo de una rama. Como entonces, esta indeterminación se resuelve considerando una función `seleccion` que escoge una fórmula no literal de una rama, si es que existe. La existencia de esta función se asume en un encapsulado ACL2, caracterizándola por las propiedades formalizadas en *60*.

La función que implementa la regla de computación es la siguiente:

157
```
(defun tab-regla-computacion (rama)
  (if (rama-cerrada rama)
      nil
      (let ((F (seleccion rama)))
        (cond ((doble-negacion F)
               (list (añade-elemento (componente-neg-neg F)
                                     (elimina-una F rama))))
              ((alfa-formula F)
               (list (añade-elemento
                      (componente-1 F)
                      (añade-elemento (componente-2 F)
                                      (elimina-una F rama)))))
              ((beta-formula F)
               (list (añade-elemento (componente-1 F)
                                     (elimina-una F rama))
                     (añade-elemento (componente-2 F)
                                     (elimina-una F rama))))
              (t t)))))
```

En esta definición hemos utilizado la función `rama-cerrada`, definida en *56*, que comprueba si una lista de fórmulas tiene literales complementarios, así como las funciones que caracterizan las dobles negaciones, α-fórmulas y β-fórmulas, definidas en *47* y *45* y las que calculan sus componentes, definidas en *47* y *46*. Las funciones `elimina-una` y `añade-elemento` son las definidas en *58*, la primera elimina un elemento de una lista y la segunda añade un elemento a una lista, si es que no estaba ya en ella.

Obsérvese que el resultado devuelto por la función anterior es **t** o una lista de ramas. La formalización de esta propiedad es la siguiente:

158
```
(defthm member-tab-regla-computacion-tab-objeto
  (implies (and (tab-objeto rama-1)
                (member-equal rama-2
                              (tab-regla-computacion rama-1)))
           (tab-objeto rama-2))
```

La función de representación $i_{\mathcal{T}}$ es bastante simple, consiste en construir una lista formada por una única fórmula. La función `tab-representacion` realiza esta construcción. Fácilmente se demuestra que el resultado que devuelve es un elemento de $\mathcal{O}_{\mathcal{T}}$.

159
```
(defun tab-representacion (F)
  (list F))

(defthm tab-representacion-es-tab-objeto
  (implies (es-proposicional F)
           (tab-objeto (tab-representacion F))))
```

La función de medida $\mu_{\mathcal{T}}$ es la medida uniforme de una lista de fórmulas. Esta función está implementada por `medida-uniforme-rama`, definida en *62*. Para mantener la correspondencia con la formalización del marco genérico, la llamaremos `tab-medida`. El resultado devuelto por esta función es un ordinal.

160
```
(defun tab-medida (rama)
  (cond ((endp rama) 0)
        (t (+ (medida-uniforme (car rama))
              (tab-medida (cdr rama))))))

(defthm tab-medida-es-e0-ordinalp
  (e0-ordinalp (tab-medida rama)))
```

La función modelo $\sigma_{\mathcal{T}}$ construye una asignación en la que todos los literales positivos de una rama son ciertos. Esta construcción es la que realiza la función `genera-modelo-literales`, definida en *63*. Como en casos anteriores utilizaremos el nombre `tab-modelo` para mantener la correspondencia con los nombres de funciones utilizados en la formalización del marco genérico.

161
```
(defun tab-modelo (lista-L)
  (cond ((endp lista-L) nil)
        ((es-simbolo-proposicional (car lista-L))
         (asume-valor (car lista-L) *V*
                      (tab-modelo (cdr lista-L))))
        (t (tab-modelo (cdr lista-L)))))
```

Veamos a continuación la formalización de las propiedades que garantizan que $\mathcal{T}$ es un STP exitoso. La propiedad $\mathcal{P}_1$ está formalizada por el evento `P1-tab` y se demuestra fácilmente a partir de los eventos presentados en 53 y al hecho de que para toda α-fórmula F, $u(\alpha_1) + u(\alpha_2) < u(\alpha)$.

162
```
(defthm P1-tab
  (implies (member-equal rama-2 (tab-regla-computacion rama-1))
           (e0-ord-< (tab-medida rama-2)
                     (tab-medida rama-1))))
```

La prueba de la propiedad $\mathcal{P}_2$ se obtiene inmediatamente a partir de las definiciones de modelo de una rama y la función de representación. Su formalización es la siguiente:

163
```
(defthm P2-tab
  (implies (es-proposicional F)
           (iff (tab-asignacion-distinguida
                  sigma (tab-representacion F))
                (modelo sigma F))))
```

Para formalizar la propiedad $\mathcal{P}_3$ necesitamos definir una función que compruebe si una asignación es modelo de alguna de las ramas de una lista. Esta función es `tab-asignacion-distinguida-lista`. La formalización de esta función y de la propiedad $\mathcal{P}_3$ es la siguiente:

164
```
(defthm P3-tab
  (implies (and (tab-objeto rama)
                (not (equal (tab-regla-computacion rama) t)))
           (iff (tab-asignacion-distinguida-lista
                  sigma (tab-regla-computacion rama))
                (tab-asignacion-distinguida sigma rama))))

(defun tab-asignacion-distinguida-lista (sigma tablero)
  (cond ((atom tablero) nil)
        (t (or (tab-asignacion-distinguida
                 sigma (car tablero))
               (tab-asignacion-distinguida-lista
                 sigma (cdr tablero))))))
```

La prueba del evento `P3-tab` se obtiene a partir de resultados similares a los presentados en [59], en los que se utiliza la función `tab-asignacion-distinguida` en lugar de `modelo-rama`.

Finalmente la propiedad $\mathcal{P}_4$ se demuestra fácilmente a partir de la definición de `tab-modelo` y el hecho de que la rama sobre la que se evalúa sólo contiene literales y no tiene pares de fórmulas complementarias. Su formalización es la siguiente:

165
```
(defthm P4-tab
  (implies (and (tab-objeto rama)
                (equal (tab-regla-computacion rama) t))
           (tab-asignacion-distinguida (tab-modelo rama) rama))
```

El siguiente evento realiza una instancia de la teoría genérica `*sat-generico*` utilizando las funciones que describen el STP exitoso $\mathcal{T}$. Para seleccionar la rama a procesar en cada paso hemos utilizado la función `car`, que devuelve el primer elemento de una lista.

166
```
(definstancia-*sat-generico*
  ((gen-objeto                  tab-objeto)
   (gen-representacion          tab-representacion)
   (gen-asignacion-distinguida  tab-asignacion-distinguida)
   (gen-asignacion-distinguida-lista
                                tab-asignacion-distinguida-lista)
   (gen-regla-computacion       tab-regla-computacion)
   (gen-seleccion               car)
   (gen-medida                  tab-medida)
   (gen-modelo                  tab-modelo))
  "-tableros")
```

Una vez evaluado este evento, se obtiene automáticamente la función que implementa el algoritmo $SAT_{\mathcal{T}}$ como una instancia de $SAT_{\mathcal{G}}$. También se obtienen versiones basadas en tableros del algoritmo de decisión de validez $DAT_{\mathcal{G}}$ y del procedimiento de construcción de modelos $MOD_{\mathcal{G}}$. Estas funciones son las siguientes:

```
(DEFUN SAT-GENERICO-TABLEROS (F)
  (OBJ-SAT-GENERICO-TABLEROS (LIST (TAB-REPRESENTACION F))))

(DEFUN MOD-GENERICO-TABLEROS (F)
  (TAB-MODELO (CAR (SAT-GENERICO-TABLEROS F))))

(DEFUN DAT-GENERICO-TABLEROS (F)
  (NOT (SAT-GENERICO-TABLEROS (NEGACION F))))

(DEFUN OBJ-SAT-GENERICO-TABLEROS (LISTA-O)
  (IF (ENDP LISTA-O)
      NIL
      (LET* ((O (CAR LISTA-O))
             (RESTO (ELIMINA-UNA (CAR LISTA-O) LISTA-O))
             (EXPANSION (TAB-REGLA-COMPUTACION O)))
        (COND ((EQUAL EXPANSION T) (LIST O))
              (T (OBJ-SAT-GENERICO-TABLEROS
                   (APPEND EXPANSION RESTO)))))))
```

Además de generar automáticamente estas funciones, al instanciar la teoría genérica `*sat-generico*`, también se obtienen de forma automática los eventos que establecen sus propiedades de corrección y completitud:

```
(DEFTHM CORRECCION-SAT-GENERICO-TABLEROS
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (SAT-GENERICO-TABLEROS F))
           (MODELO (MOD-GENERICO-TABLEROS F) F)))

(DEFTHM COMPLETITUD-SAT-GENERICO-TABLEROS
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (MODELO SIGMA F))
           (SAT-GENERICO-TABLEROS F)))

(DEFTHM CORRECCION-DAT-GENERICO-TABLEROS
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (DAT-GENERICO-TABLEROS F))
           (MODELO SIGMA F)))

(DEFTHM COMPLETITUD-DAT-GENERICO-TABLEROS
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (NOT (DAT-GENERICO-TABLEROS F)))
           (NOT (MODELO (MOD-GENERICO-TABLEROS (NEGACION F))
                        F))))
```

7.2.3 Un $\mathbb{K}$-STP exitoso basado en tableros semánticos

A partir del método de los tableros semánticos también se puede construir un $\mathbb{K}$-STP exitoso, de forma que los procedimientos de decisión de validez y satisfacibilidad presentados en la sección 4.2.7, se puedan obtener como casos particulares de los procedimientos genéricos.

El $\mathbb{K}$-STP exitoso basado en tableros semánticos se diferencia del desarrollado para la semántica clásica en el concepto de $\mathbb{K}$-asignación distinguida y en la función $\mathbb{K}$-modelo. Las $\mathbb{K}$-asignaciones distinguidas se definen con respecto al concepto de modelo en $\mathbb{K}$. La función que las caracteriza es `modelo-rama-K`, presentada en 72. La función $\mathbb{K}$-modelo tiene que construir una $\mathbb{K}$-asignación en la que todos los literales de una rama sean ciertos, esta construcción se obtiene con la función `genera-modelo-literales-K`, presentada en 74.

La prueba de la propiedad $\mathcal{P}_1$ es la misma que para el caso de la semántica clásica. La propiedad $\mathcal{P}_2$ es trivial a partir del concepto de modelo en $\mathbb{K}$ y la función de representación. Las reglas de expansión preservan los $\mathbb{K}$-modelos como se expuso en la sección 4.2.7, de aquí que se verifique la propiedad $\mathcal{P}_3$. Finalmente, la propiedad $\mathcal{P}_4$ se cumple gracias a que la función $\mathbb{K}$-modelo no deja indeterminado ningún literal de la rama a la que se aplica.

7.3 Un STP exitoso basado en secuentes

El procedimiento para decidir la satisfacibilidad basado en secuentes descrito en el capítulo 5, también se puede interpretar como una versión particular del algoritmo $SAT_{\mathcal{G}}$. En esta ocasión, los objetos proposicionales son los secuentes. El objeto inicial asociado a una fórmula F es el secuente $F \Rightarrow$. Se verifica que una asignación es modelo de F si y sólo si es contramodelo de $F \Rightarrow$, de aquí que las asignaciones distinguidas de un secuente sean las que lo hacen falso. Los secuentes son modificados de acuerdo con las reglas de expansión mostradas en la definición 5.5. Los secuentes axioma no son escogidos para expandir pues no poseen contramodelos. Esto se corresponde con una regla de expansión que "elimina" dichos secuentes, es decir, cuya parte derecha es $\langle\rangle$. Si se encuentra un secuente atómico que no sea axioma, el algoritmo termina y devuelve dicho secuente. Esto se corresponde con una regla de expansión que indica que a partir de dicho secuente se puede obtener un contramodelo, es decir, una regla cuya parte derecha es $\mathbf{t}$. Si no se obtiene ninguno de estos secuentes, el algoritmo termina y devuelve $\mathbf{f}$.

En esta sección desarrollamos un sistema de transformación proposicional basado en esta descripción. Primero describimos formalmente este sistema junto con una regla de computación, una función de representación, una función de medida y una función modelo adecuadas para demostrar que es exitoso. A continuación presentamos su formalización, relacionándola con la desarrollada en el capítulo 5. Finalmente comentamos cómo se hace este mismo proceso en la semántica de

Kleene. Los eventos relacionados con la semántica clásica presentados en esta sección se encuentran en el libro `SAT-secuentes.lisp`, los relacionados con la semántica de Kleene se encuentran en el libro `SAT-secuentes-K.lisp`.

7.3.1 Descripción del STP exitoso $\mathcal{S}$

Consideramos el siguiente sistema de transformación proposicional basado en el cálculo de secuentes:

Definición 7.16 *$\mathcal{S} = \langle \mathcal{O}_s, \mathcal{R}_s, \mathcal{V}_s \rangle$ es el STP basado en secuentes en el que $\mathcal{O}_s$ es el conjunto de los secuentes, $\mathcal{V}_s$ es el conjunto de pares (S, σ) tales que σ es contramodelo del secuente S, y $\mathcal{R}_s$ es el conjunto de reglas de expansión representado por el siguiente conjunto de esquemas de regla:*

$\mathcal{RS}_1$:	$\langle \Gamma_1, F, \Gamma_2 \rangle \Rightarrow \langle \Delta_1, F, \Delta_2 \rangle \rightsquigarrow_s \langle\rangle$
$\mathcal{RS}_2$:	$\langle \Gamma_1, \neg F, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle \rangle$
$\mathcal{RS}_3$:	$\langle \Gamma_1, F \wedge G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_4$:	$\langle \Gamma_1, F \vee G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_5$:	$\langle \Gamma_1, F \rightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_6$:	$\langle \Gamma_1, F \leftrightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_s \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, G, \Delta \rangle \rangle$
$\mathcal{RS}_7$:	$\Gamma \Rightarrow \langle \Delta_1, \neg F, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_8$:	$\Gamma \Rightarrow \langle \Delta_1, F \wedge G, \Delta_2 \rangle \rightsquigarrow_s \langle \Gamma \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle, \Gamma \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_9$:	$\Gamma \Rightarrow \langle \Delta_1, F \vee G, \Delta_2 \rangle \rightsquigarrow_s \langle \Gamma \Rightarrow \langle F, G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{10}$:	$\Gamma \Rightarrow \langle \Delta_1, F \rightarrow G, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{11}$:	$\Gamma \Rightarrow \langle \Delta_1, F \leftrightarrow G, \Delta_2 \rangle \rightsquigarrow_s \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle, \langle G, \Gamma \rangle \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{12}$:	$\Gamma \Rightarrow \Delta \rightsquigarrow_s \mathbf{t}$ *si* $\Gamma \Rightarrow \Delta$ *es un secuente atómico que no es un axioma*

donde Γ, Γ_1, Γ_2, Δ, Δ_1 y Δ_2 son listas finitas de fórmulas.

Utilizando el STP anterior, junto con una función de representación i_s y una regla de computación r_s adecuadas, podemos expresar el algoritmo SAT_s como un caso particular del algoritmo SAT_g. Además, para garantizar las propiedades de terminación, corrección y completitud tenemos que proporcionar una función de medida μ_s y una función modelo σ_s, de forma que $\mathcal{S}$ sea exitoso con respecto a r_s, i_s, μ_s y σ_s.

Definición 7.17 *Para el STP $\mathcal{S}$ definimos:*

1. *La función de representación i_s tal que para toda $F \in \mathbb{P}(\Sigma)$, $i_s(F) = F \Rightarrow$.*
2. *La regla de computación r_s que actúa de la siguiente forma. Dada un secuente $\Gamma \Rightarrow \Delta$, si dicho secuente es una axioma, se le aplica una regla del tipo $\mathcal{RS}_1$. En otro caso se selecciona una fórmula F no atómica de Γ. En función del tipo de dicha fórmula se le aplica una de las reglas $\mathcal{RS}_2$, $\mathcal{RS}_3$, $\mathcal{RS}_4$, $\mathcal{RS}_5$ o $\mathcal{RS}_6$, descomponiendo F. Si en Γ no hay fórmulas no*

atómicas, se selecciona una fórmula F no atómica de Δ. En función del tipo de dicha fórmula se le aplica una de las reglas $\mathcal{RS}_7$, $\mathcal{RS}_8$, $\mathcal{RS}_9$, $\mathcal{RS}_{10}$ o $\mathcal{RS}_{11}$, descomponiendo F. Si el secuente $\Gamma \Rightarrow \Delta$ es atómico, se le aplica una regla del tipo $\mathcal{RS}_{12}$.

3. *La función de medida $\mu_{\mathcal{S}}$, que es el número de conectivas de un secuente, $r_{\mathcal{S}}$. Así, para todo secuente S se tiene $\mu_{\mathcal{S}}(S) = r_{\mathcal{S}}(S)$.*

4. *La función modelo $\sigma_{\mathcal{S}}$ tal que, para todo secuente S, $\sigma_{\mathcal{S}}(S)$ es la asignación definida de forma que $\sigma(p) = \top$ si y sólo si p es un símbolo que aparece en la parte izquierda de S.*

Teorema 7.18 *El STP $\mathcal{S}$ es exitoso con respecto a la regla de computación $r_{\mathcal{S}}$, la función de representación $i_{\mathcal{S}}$, la función de medida $\mu_{\mathcal{S}}$ y la función modelo $\sigma_{\mathcal{S}}$.*

Demostración:

Para demostrar el teorema tendremos que probar las propiedades $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ y $\mathcal{P}_4$ para las funciones $r_{\mathcal{S}}$, $i_{\mathcal{S}}$, $\mu_{\mathcal{S}}$ y $\sigma_{\mathcal{S}}$.

$\mathcal{P}_1$: Dado $S_i \in r_{\mathcal{S}}(S)$, entonces S_i es un secuente obtenido como premisa al utilizar una regla de inferencia para concluir S, luego por el teorema 5.11 se verifica que $\mu_{\mathcal{S}}(S_i) = r_{\mathcal{S}}(S_i) < r_{\mathcal{S}}(S) = \mu_{\mathcal{S}}(S)$.

$\mathcal{P}_2$: Dada la fórmula F, $\sigma \models_{\mathcal{S}} F \Rightarrow = i_{\mathcal{S}}(F)$ si y sólo si σ es contramodelo de $F\Rightarrow$, es decir, si y sólo si σ es modelo de F.

$\mathcal{P}_3$: Dado el secuente S tal que $r_{\mathcal{S}}(S) \neq \mathbf{t}$, entonces S es un secuente axioma, en cuyo caso $r_{\mathcal{S}}(S) = \langle\rangle$, o existe una fórmula no atómica en su parte izquierda o su parte derecha, en cuyo caso $r_{\mathcal{S}}(S)$ es la lista de premisas de una de las reglas de inferencia del cálculo G'.

Si S es un secuente axioma entonces, por el teorema 5.4, cualquier asignación σ es modelo de S, luego $\sigma \not\models_{\mathcal{S}} S$. Por otro lado, $r_{\mathcal{S}}(S)$ es la lista vacía. Por tanto se tiene la equivalencia $\sigma \models_{\mathcal{S}} S \Longleftrightarrow \exists S_i \in r_{\mathcal{S}}(S), \sigma \models_{\mathcal{S}} S_i$

Si existe una fórmula no atómica en la parte izquierda o derecha de S entonces, por el teorema 5.6, se tiene que una asignación σ es modelo de S si y sólo si σ es modelo de todas las premisas de una regla de inferencia cuya conclusión sea S. Por tanto, σ es contramodelo de S ($\sigma \models_{\mathcal{S}} S$) si y sólo si es contramodelo de alguna de las premisas ($\exists S_i \in r_{\mathcal{S}}(S), \sigma \models_{\mathcal{S}} S_i$).

$\mathcal{P}_4$: Dado el secuente S tal que $r_{\mathcal{S}}(S) = \mathbf{t}$, entonces S es un secuente atómico que no es un axioma y, por el teorema 5.8, la asignación σ definida de forma que $\sigma(p) = \top$ si y sólo si p aparece en la parte izquierda de S, no es modelo de S. Esta asignación es $\sigma_{\mathcal{S}}(S)$, por tanto $\sigma_{\mathcal{S}}(S) \models_{\mathcal{S}} S$

□

7.3.2 Formalización de $\mathcal{S}$

Para formalizar el conjunto de objetos proposicionales en el que se basa $\mathcal{S}$, tenemos que definir una función que caracterice dichos objetos. Esta función ya ha sido definida en 79, para mantener la correspondencia con los nombres de las funciones utilizadas en la formalización del marco genérico, la llamaremos `sec-objeto`:

167
```
(defun sec-objeto (S)
  (and (consp S)
       (lista-formulas (car S))
       (lista-formulas (cdr S))))
```

Las asignaciones distinguidas de un secuente son sus contramodelos, es decir, aquellas asignaciones en las que todos los elementos de la parte izquierda del secuente son ciertos y todos los de la parte derecha son falsos. La función que comprueba esta propiedad es `sec-asignacion-distinguida`. Para comprobar que la asignación hace ciertos todos los elementos de la parte izquierda del secuente utilizamos la función `modelo-conjuncion`, definida en 80. De forma análoga definimos `no-modelo-conjuncion` para comprobar que la asignación hace falsos todos los elementos de la parte derecha del secuente.

168
```
(defun sec-asignacion-distinguida (sigma S)
  (and (modelo-conjuncion sigma (car S))
       (no-modelo-conjuncion sigma (cdr S))))

(defun no-modelo-conjuncion (sigma Gamma)
  (cond ((endp Gamma) t)
        ((not (modelo sigma (car Gamma)))
         (no-modelo-conjuncion sigma (cdr Gamma)))
        (t nil)))
```

El comportamiento de la función `sec-asignacion-distinguida` es el contrario al de la función `modelo-secuente` definida en 80. El valor devuelto por `sec-asignacion-distinguida` es `T` si y sólo si el valor de `modelo-secuente` es `nil`.

La regla de computación r_s no está completamente determinada puesto que, si el secuente al que hay que aplicarla no es un axioma, se ha de escoger una fórmula no atómica de su parte izquierda o de su parte derecha, y no se indica el criterio a seguir para realizar esta selección. De hecho, esto también ocurría en el procedimiento de búsqueda de un contramodelo de un secuente. Como entonces, esta indeterminación se resuelve considerando una función `seleccion` que escoge una fórmula no atómica de una lista, si es que existe. La existencia de esta

función se asume en un encapsulado ACL2, caracterizándola por las propiedades formalizadas en 86.

La función que implementa la regla de computación es la siguiente:

169
```
(defun sec-regla-computacion (S)
  (let ((Gamma (car S))
        (Delta (cdr S)))
    (if (secuente-axioma S)
        nil
        (let ((F-1 (seleccion Gamma)))
          (if F-1
              (secuentes-expansion-izq F-1 Gamma Delta)
              (let ((F-2 (seleccion Delta)))
                (if F-2
                    (secuentes-expansion-der F-2 Gamma Delta)
                    t)))))))
```

En esta definición hemos utilizado la función `secuente-axioma`, definida en 81, que comprueba si hay algún elemento en común en las partes izquierda y derecha de un secuente. También se han utilizado `secuentes-expansion-izq` y `secuentes-expansion-der`, definidas respectivamente en 83 y 84, para obtener las premisas de las reglas de inferencia correspondientes al tipo de las fórmulas `F-1` y `F-2`.

El resultado devuelto por esta función es **t** o una lista de secuentes:

170
```
(defthm member-sec-regla-computacion-es-sec-objeto
   (implies (and (sec-objeto S-1)
                 (member-equal S-2 (sec-regla-computacion S-1)))
            (sec-objeto S-2)))
```

La función de representación i_S es muy simple, consiste en construir el secuente $F \Rightarrow$. La función `sec-representacion` realiza esta construcción. El resultado que devuelve es un secuente.

171
```
(defun sec-representacion (F)
  (list (list F)))

(defthm sec-representacion-es-sec-objeto
  (implies (es-proposicional formula)
           (sec-objeto (sec-representacion formula))))
```

La función de medida μ_s es el número de conectivas de un secuente. Esta función está implementada por `medida-secuente`, definida en 88. Para mantener la correspondencia con la formalización del marco genérico, la llamaremos `sec-medida`. El resultado que devuelve es un ordinal.

172
```
(defun sec-medida (S)
  (+ (medida-lista-formulas (car S))
     (medida-lista-formulas (cdr S))))

(defthm sec-medida-es-e0-ordinalp
  (e0-ordinalp (sec-medida S)))
```

La función modelo σ_s construye una asignación en la que todos los símbolos de la parte izquierda de un secuente son ciertos. Esta construcción es la que realiza la función `genera-contramodelo-secuente`, definida en 89. Como en casos anteriores utilizamos el nombre `sec-modelo` para mantener la correspondencia con los nombres de funciones utilizados en la formalización del marco genérico.

173
```
(defun sec-modelo (S)
  (genera-asignacion-positiva (car S)))
```

Veamos a continuación la formalización de las propiedades que garantizan que $\mathcal{S}$ es un STP exitoso. La propiedad $\mathcal{P}_1$ está formalizada por el evento `P1-sec` y se demuestra fácilmente dado que el número de conectivas de las subfórmulas de una fórmula compuesta es menor que el de la original.

174
```
(defthm P1-sec
  (implies (member-equal S-2 (sec-regla-computacion S-1))
           (e0-ord-< (sec-medida S-2)
                     (sec-medida S-1))))
```

La prueba de la propiedad $\mathcal{P}_2$ se obtiene inmediatamente a partir de las definiciones de contramodelo de un secuente y la función de representación. Su formalización es la siguiente:

175
```
(defthm P2-sec
  (implies (es-proposicional F)
           (iff (se-asignacion-distinguida
                   sigma (sec-representacion F))
                (modelo sigma F))))
```

Para formalizar la propiedad $\mathcal{P}_3$ necesitamos definir una función que compruebe si una asignación es contramodelo de alguno de los secuentes de una lista. Esta función es `sec-asignacion-distinguida-lista`. La formalización de esta función y de la propiedad $\mathcal{P}_3$ es la siguiente:

176

```
(defthm P3-sec
  (implies (and (sec-objeto S)
                (not (equal (sec-regla-computacion S) t)))
           (iff (sec-asignacion-distinguida-lista
                  sigma (sec-regla-computacion S))
                (sec-asignacion-distinguida sigma S))))

(defun sec-asignacion-distinguida-lista (sigma lista-S)
  (cond ((endp lista-S) nil)
        (t (or (sec-asignacion-distinguida
                 sigma (car lista-S))
               (sec-asignacion-distinguida-lista
                 sigma (cdr lista-S))))))
```

La prueba del evento `P3-sec` se obtiene a partir de resultados similares para `secuentes-expansion-izq` y `secuentes-expansion-der`.

Finalmente, la propiedad $\mathcal{P}_4$ se demuestra fácilmente a partir de la definición de `sec-modelo` y el hecho de que el secuente sobre el que se evalúa es atómico y no es un axioma. Su formalización es la siguiente:

177

```
(defthm P4-sec
  (implies (and (sec-objeto S)
                (equal (sec-regla-computacion S) t))
           (sec-asignacion-distinguida-K (sec-modelo-K S) S)))
```

El siguiente evento realiza una instancia de la teoría genérica `*sat-generico*` utilizando las funciones que describen el STP exitoso $\mathcal{S}$. Para seleccionar el secuente a procesar en cada paso hemos utilizado la función `car`, que devuelve el primer elemento de una lista.

178
```
(definstancia-*sat-generico*
  ((gen-objeto                  sec-objeto)
   (gen-representacion          sec-representacion)
   (gen-asignacion-distinguida  sec-asignacion-distinguida)
   (gen-asignacion-distinguida-lista
                                sec-asignacion-distinguida-lista)
   (gen-regla-computacion       sec-regla-computacion)
   (gen-seleccion               car)
   (gen-medida                  sec-medida)
   (gen-modelo                  sec-modelo))
  "-secuentes")
```

Una vez evaluado este evento, se obtiene automáticamente la función que implementa el algoritmo $SAT_{\mathcal{S}}$, como una instancia de $SAT_{\mathcal{G}}$. También se obtienen versiones basadas en secuentes del algoritmo de decisión de validez $DAT_{\mathcal{G}}$ y del procedimiento de construcción de modelos $MOD_{\mathcal{G}}$. Estas funciones son las siguientes:

```
(DEFUN SAT-GENERICO-SECUENTES (F)
  (OBJ-SAT-GENERICO-SECUENTES (LIST (SEC-REPRESENTACION F))))

(DEFUN MOD-GENERICO-SECUENTES (F)
  (SEC-MODELO (CAR (SAT-GENERICO-SECUENTES F))))

(DEFUN DAT-GENERICO-SECUENTES (F)
  (NOT (SAT-GENERICO-SECUENTES (NEGACION F))))

(DEFUN OBJ-SAT-GENERICO-SECUENTES (LISTA-O)
  (IF (ENDP LISTA-O)
      NIL
      (LET* ((O (CAR LISTA-O))
             (RESTO (ELIMINA-UNA (CAR LISTA-O) LISTA-O))
             (EXPANSION (SEC-REGLA-COMPUTACION O)))
        (COND ((EQUAL EXPANSION T) (LIST O))
              (T (OBJ-SAT-GENERICO-SECUENTES
                   (APPEND EXPANSION RESTO)))))))
```

Además de generar automáticamente estas funciones, al instanciar la teoría genérica `*sat-generico*`, también se obtienen de forma automática los eventos que establecen sus propiedades de corrección y completitud.

```
(DEFTHM CORRECCION-SAT-GENERICO-SECUENTES
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (SAT-GENERICO-SECUENTES F))
           (MODELO (MOD-GENERICO-SECUENTES F) F)))

(DEFTHM COMPLETITUD-SAT-GENERICO-SECUENTES
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (MODELO SIGMA F))
           (SAT-GENERICO-SECUENTES F)))

(DEFTHM CORRECCION-DAT-GENERICO-SECUENTES
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (DAT-GENERICO-SECUENTES F))
           (MODELO SIGMA F)))

(DEFTHM COMPLETITUD-DAT-GENERICO-SECUENTES
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (NOT (DAT-GENERICO-SECUENTES F)))
           (NOT (MODELO (MOD-GENERICO-SECUENTES (NEGACION F))
                        F))))
```

7.3.3 Un $\mathbb{K}$-STP exitoso basado en secuentes

A partir del cálculo de secuentes también se puede construir un $\mathbb{K}$-STP exitoso, de forma que los procedimientos de decisión de validez y satisfacibilidad presentados en la sección 5.1.7, se puedan obtener como casos particulares de los procedimientos genéricos.

El $\mathbb{K}$-STP exitoso basado en secuentes se diferencia del desarrollado para la semántica clásica en el concepto de $\mathbb{K}$-asignación distinguida y en la función $\mathbb{K}$-modelo. Las $\mathbb{K}$-asignaciones distinguidas se definen con respecto al concepto de modelo en $\mathbb{K}$. La función que las caracteriza es `no-modelo-secuente-K`, presentada en *98*. La función $\mathbb{K}$-modelo tiene que construir una $\mathbb{K}$-asignación en la que todos las fórmulas atómicas de la parte izquierda de un secuente sean ciertas, y las de la parte derecha falsas. Esta construcción se obtiene con la función `genera-no-modelo-secuente-K`, presentada en *102*.

La prueba de la propiedad $\mathcal{P}_1$ es la misma que para el caso de la semántica clásica. La propiedad $\mathcal{P}_2$ es trivial a partir del concepto de no-modelo en $\mathbb{K}$ y la función de representación. Las reglas de expansión preservan los $\mathbb{K}$-no-modelos como se expuso en la sección 5.1.7, de aquí que se verifique la propiedad $\mathcal{P}_3$. Finalmente, la propiedad $\mathcal{P}_4$ se cumple gracias a que la función $\mathbb{K}$-modelo no deja indeterminada ninguna fórmula atómica del secuente al que se aplica.

7.4 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, presentando datos sobre su evaluación en algunas fórmulas de Urquhart [83] y fórmulas representando el problema de las N reinas [62]. En la sección 4.3 se indica como se cargan en ACL2 los ficheros en los que se definen estas familias de fórmulas.

Una vez certificados los libros, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `7-generico`:

```
../calculos-proposicionales/7-generico> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/7-generico".

ACL2 !>
```

Veamos primero los procedimientos de decisión obtenidos al instanciar el marco genérico con el STP basado en tableros semánticos. Incluimos el libro que contiene los eventos correspondientes. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "SAT-tableros")
```

Al igual que ocurre con los procedimientos de decisión desarrollados en el capítulo 4, tenemos que proporcionar una definición concreta de la función que selecciona una fórmula de una rama.

```
ACL2 !>(set-ld-redefinition-action '(:warn . :overwrite) state)

ACL2 !>(defun seleccion (lista)
         (cond ((endp lista) nil)
               ((es-literal (car lista)) (seleccion (cdr lista)))
               (t (car lista))))
```

En la tabla 7.1[1] se muestran los tiempos de evaluación de la función `DAT-generico-tableros` para las 15 primeras fórmulas de Urquhart. En la tabla 7.2[1] se muestra la misma información sobre la evaluación de la función `SAT-generico-tableros` para las fórmulas correspondientes al problema de las N reinas, para algunos valores de N.

Fórmula	tiempo	Fórmula	tiempo	Fórmula	tiempo
U_1	0.000	U_6	0.160	U_{11}	19.720
U_2	0.000	U_7	0.550	U_{12}	49.050
U_3	0.010	U_8	1.410	U_{13}	118.590
U_4	0.020	U_9	3.770	U_{14}	249.060
U_5	0.130	U_{10}	8.150	U_{15}	538.960

Tabla 7.1: Tiempos de evaluación de las fórmulas de Urquhart (tableros)

Tamaño del tablero	2	3	4	5	6
Tiempo	0.000	0.150	1.290	5.920	543.390

Tabla 7.2: Tiempos de evaluación para el problema de las N reinas (tableros)

Uno de los procedimientos que se obtienen como resultado de instanciar el marco genérico es el que proporciona un modelo de una fórmula. Podemos utilizar dicho procedimiento para obtener modelos para las fórmulas que describen el problema de las N reinas[1]. Estos modelos se corresponden con soluciones de dicho problema:

```
ACL2 !>(MOD-generico-tableros (n-reinas 4))
((15 1) (9 1) (8 1) (2 1))
ACL2 !>(MOD-generico-tableros (n-reinas 5))
((24 1) (17 1) (15 1) (8 1) (1 1))
```

En la codificación que se ha hecho del problema de las N reinas, la solución obtenida para un tablero de tamaño 4 es la que tiene reinas situadas en las casillas con los números 2 (primera fila, segunda columna), 8 (segunda fila, cuarta columna), 9 (tercera fila, primera columna) y 15 (cuarta fila, tercera columna). La solución obtenida para un tablero de tamaño 5 es la que tiene reinas situadas en las casillas con los números 1 (primera fila, primera columna), 8 (segunda

[1]En el fichero `SAT-tableros.lisp` hay tres STP basados en tableros semánticos, el descrito en este capítulo, con respecto al cual se proporcionan los datos, es el segundo.

fila, tercera columna), 15 (tercera fila, quinta columna), 17 (cuarta fila, segunda columna) y 24 (quinta fila, cuarta columna).

Para utilizar los procedimientos de decisión obtenidos al instanciar el marco genérico con el STP basado en secuentes, actuamos de manera similar:

```
ACL2 !>(include-book "SAT-secuentes")
```

Al igual que ocurre con los procedimientos de decisión desarrollados en el capítulo 5, tenemos que proporcionar una definición concreta de la función que selecciona una fórmula de una secuente.

```
ACL2 !>(set-ld-redefinition-action '(:warn . :overwrite) state)

ACL2 !>(defun seleccion (lista)
         (cond ((endp lista) nil)
               ((es-atomica (car lista)) (seleccion (cdr lista)))
               (t (car lista))))
```

En la tabla 7.3 se muestran los tiempos de evaluación de la función `DAT-secuentes` para las 15 primeras fórmulas de Urquhart. En la tabla 7.4 se muestra la misma información para las fórmulas correspondientes al problema de las N reinas, para algunos valores de N.

Fórmula	tiempo	Fórmula	tiempo	Fórmula	tiempo
U_1	0.000	U_6	0.010	U_{11}	1.530
U_2	0.000	U_7	0.040	U_{12}	3.570
U_3	0.000	U_8	0.100	U_{13}	8.230
U_4	0.000	U_9	0.310	U_{14}	18.540
U_5	0.010	U_{10}	0.630	U_{15}	43.290

Tabla 7.3: Tiempos de evaluación de las fórmulas de Urquhart (secuentes)

Tamaño del tablero	2	3	4	5	6
Tiempo	0.000	0.040	0.430	1.900	189.990

Tabla 7.4: Tiempos de evaluación para el problema de las N reinas (secuentes)

Sumario

En este capítulo:

- Se ha definido un marco genérico del que diversos métodos para decidir la satisfacibilidad de una fórmula son casos particulares. Se ha formalizado esta abstracción, demostrando sus propiedades, y se ha definido una teoría genérica para facilitar su reutilización.
- Se ha presentado el método de los tableros semánticos como un caso particular del marco genérico. Se han formalizado las funciones y propiedades necesarias para obtener algoritmos verificados de decisión de satisfacibilidad y validez, como una instancia de la teoría asociada al marco genérico.
- Se ha realizado el mismo desarrollo para el método de los secuentes. También se han obtenido algoritmos verificados de decisión de satisfacibilidad y validez, como instancia de la teoría asociada al marco genérico.

Capítulo 8

Procedimiento de Davis y Putnam

El procedimiento de Davis y Putnam fue introducido en 1960 [19] como una técnica de decisión de validez adecuada para la automatización. Este procedimiento es un sistema de refutación: para probar la validez de una fórmula F se busca un modelo de su negación $\neg F$. Si no se encuentra dicho modelo entonces la fórmula original es válida. Desde un punto de vista constructivo, se trata de un procedimiento para buscar un modelo de una fórmula; es decir, determinar su satisfacibilidad. Este método es la base de algunos de los algoritmos más eficientes para determinar la satisfacibilidad de una fórmula como Chaff [61], Relsat [5], SATO [88], SATZ [57], C–SAT [22] o POSIT [27].

Este procedimiento no trabaja directamente con una fórmula, sino con una forma clausal (una lista de listas de literales) que guarda una relación de equivalencia lógica con la fórmula de partida. Por ello, en la primera sección de este capítulo, se definen los conceptos de cláusula y forma clausal y se demuestra que cualquier fórmula se puede transformar en una forma clausal lógicamente equivalente a ella. Para ello se proporciona un algoritmo de transformación basado en la notación uniforme. Se discute la prueba de terminación de este algoritmo y que efectivamente construye formas clausales lógicamente equivalentes a la fórmula de partida. Otra propiedad importante de este algoritmo es que sirve para decidir la validez de una fórmula. Tanto los conceptos de cláusula y forma clausal como el algoritmo de transformación y sus propiedades son establecidos en la semántica clásica y en la semántica de Kleene.

En la segunda sección se presenta el procedimiento de Davis y Putnam como un método para decidir la satisfacibilidad de una forma clausal. Este procedimiento se basa en una serie de reglas de transformación que son analizadas en esta sección. Utilizando estas reglas de transformación se definen dos algoritmos para determinar la satisfacibilidad de una forma clausal. Se demuestran las propiedades de corrección y completitud de estos algoritmos tanto en la semántica clásica como en la de Kleene.

En la última sección se analiza el procedimiento de Davis y Putnam como un caso particular del marco genérico presentado en el capitulo 7. En esta sección se presentan lo eventos necesarios para instanciar con éxito la teoría asociada al marco genérico.

8.1 Cláusulas y formas clausales

En esta sección se presenta la formalización de los conceptos de cláusula y forma clausal. Ambos se representan utilizando listas, una cláusula es una lista de literales y una forma clausal es una lista de cláusulas. Las cláusulas se interpretan como la disyunción generalizada de los literales que en ella aparecen y las formas clausales como la conjunción generalizada de las cláusulas que la forman. Los eventos correspondientes a esta formalización para la semántica clásica se encuentran en el libro `clausulas.lisp`, los correspondientes para la semántica de Kleene se encuentran en el libro `clausulas-K.lisp`.

Un resultado importante en la lógica proposicional clásica es que toda fórmula puede ser transformada en una forma clausal lógicamente equivalente a ella. En la segunda parte de esta sección se presenta un algoritmo que realiza dicha transformación, basado en la descripción dada en [26]. Este algoritmo es más eficiente que el clásico basado en la transformación a forma normal conjuntiva (vía la forma normal negativa) y, además, puede ser utilizado como un procedimiento de decisión de validez de fórmulas. La principal dificultad encontrada en la definición de este algoritmo ha sido la prueba de su terminación, que se ha realizado mediante órdenes de multiconjuntos. Finalmente se demuestra que el algoritmo devuelve una forma clausal lógicamente equivalente a la fórmula que recibe como argumento y que el valor devuelto es la forma clausal vacía si y sólo si la fórmula original es válida. Los eventos correspondientes a esta parte para la semántica clásica se encuentran en el libro `forma-clausal.lisp` y los correspondientes a la semántica de Kleene en el libro `forma-clausal-K.lisp`.

8.1.1 Sintaxis y semántica

Definición 8.1 *Una* **cláusula** *es una lista de literales,* $\langle L_1, \ldots, L_n \rangle$.

Definimos el predicado `es-clausula` para comprobar si un objeto `C` es una cláusula.

179
```
(defun es-clausula (C)
  (or (endp C)
      (and (es-literal (car C))
           (es-clausula (cdr C)))))
```

Las cláusulas se interpretan como la disyunción de los literales que las forman. La semántica de las cláusulas se establece en las siguientes definiciones:

Definición 8.2 *El valor de una cláusula* $\langle L_1, \ldots, L_n \rangle$ *en una asignación* σ *es:* $\tilde{\sigma}(\langle L_1, \ldots, L_n \rangle) = \mathcal{F}_\vee(\tilde{\sigma}(L_1), \mathcal{F}_\vee(\ldots \mathcal{F}_\vee(\tilde{\sigma}(L_{n-1}), \tilde{\sigma}(L_n)) \ldots))$.

La función que implementa este concepto es `valor-clausula`:

180
```
(defun valor-clausula (C sigma)
  (cond ((endp C) *F*)
        (t (funcion-de-verdad-de-disyuncion
            (valor (car C) sigma)
            (valor-clausula (cdr C) sigma)))))
```

Definición 8.3 *Una asignación* σ *es modelo de una cláusula* C*, y lo notaremos* $\sigma \models C$*, si y sólo si es modelo de algún literal de* C*.*

Con la representación que hemos escogido, las cláusulas no son fórmulas reconocidas por el predicado `es-proposicional`. Por ello, no podemos utilizar la función `modelo` para comprobar si una asignación es modelo de una cláusula. La versión de esta función para cláusulas es `modelo-clausula`.

181
```
(defun modelo-clausula (sigma C)
  (if (consp C)
      (or (modelo sigma (car c))
          (modelo-clausula sigma (cdr C)))
      nil))
```

Definición 8.4 *Una cláusula* C *es válida, si para toda asignación* σ*, el valor de* C *en* σ *es* $\top$*.*

El concepto de cláusula válida no se puede formalizar en ACL2 pues supondría trabajar con todas las asignaciones y éste es un conjunto infinito. Este concepto se incorpora a los teoremas de igual forma a como se hace para las fórmulas (ver detalles en la sección 3.2.4).

Se verifica que una asignación es modelo de una cláusula según la definición anterior si y sólo si el valor de la cláusula en dicha asignación es $\top$:

182
```
(defthm modelo-clausula-valor-clausula
  (iff (modelo-clausula sigma C)
       (equal (valor-clausula C sigma) *V*)))
```

Obsérvese que la cláusula vacía no tiene modelos, puesto que no tiene ningún literal que pueda ser cierto en una asignación dada.

El siguiente teorema permite comprobar la validez de una cláusula sin más que analizar sus elementos:

Teorema 8.5 *Una cláusula es válida si y sólo si contiene literales complementarios.*

Para demostrar una de las implicaciones de este teorema necesitamos construir una asignación en la que una cláusula que no contenga literales complementarios sea falsa. Esta asignación se construye teniendo en cuenta que los complementarios de los literales que aparecen en la cláusula han de ser ciertos. Definimos a continuación la asignación $\sigma[L/\top]$ como una extensión de σ en la que el literal L es cierto, y por tanto $\overline{L}$ falso.

Definición 8.6 *Sea σ una asignación y L un literal, $\sigma[L/\top]$ es la asignación definida por:*

$$\sigma[L/\top](y) = \begin{cases} \top & \text{si } L = y \\ \bot & \text{si } L = \neg y \\ \sigma(x) & \text{en otro caso} \end{cases}$$

Este concepto se formaliza mediante la función `asume-literal`:

183
```
(defun asume-literal (L sigma)
  (cond ((es-simbolo-proposicional L)
         (asume-valor L *V* sigma))
        (t (asume-valor (arg1 L) *F* sigma))))
```

Teorema 8.7 *Dada una asignación σ y un literal L, $\sigma[L/\top] \models L$, $\sigma[L/\top] \not\models \overline{L}$ y para todo literal L' distinto de L y $\overline{L}$ se tiene $\sigma[L/\top] \models L'$ si y sólo si $\sigma \models L'$.*

La formalización de este resultado es la siguiente:

184
```
(defthm asume-literal-es-modelo
  (implies (es-literal L)
           (modelo (asume-literal L sigma) L)))

(defthm asume-literal-no-es-modelo-c
  (implies (es-literal L)
           (not (modelo (asume-literal L sigma)
                        (complementario L)))))

(defthm literal-mantiene-valor
  (implies (and (es-literal L-1)
                (es-literal L-2)
                (not (equal L-1 L-2))
                (not (equal (complementario L-1) L-2)))
           (iff (modelo (asume-literal L-1 sigma) L-2)
                (modelo sigma L-2))))
```

Para establecer el teorema 8.5 en la formalización, hemos definido la función `tiene-literales-complementarios`, que comprueba si una cláusula tiene literales complementarios, y la función `contramodelo-clausula`, que construye una asignación en la que una cláusula sin literales complementarios es falsa.

185
```
(defthm tiene-literales-complementarios-es-clausula-valida
  (implies (and (tiene-literales-complementarios C)
                (es-clausula C))
           (modelo-clausula sigma C)))

(defthm clausula-valida-tiene-literales-complementarios
  (implies (and (not (tiene-literales-complementarios C))
                (es-clausula C))
           (not (modelo-clausula (contramodelo-clausula C) C))))

(defun tiene-literales-complementarios (C)
  (cond ((endp C) nil)
        ((member-equal (complementario (car C)) (cdr C)) t)
        (t (tiene-literales-complementarios (cdr C)))))

(defun contramodelo-clausula (C)
  (cond ((endp C) nil)
        (t (asume-literal (complementario (car C))
                          (contramodelo-clausula (cdr C))))))
```

Definición 8.8 *Una* **forma clausal** *es una lista de cláusulas,* $\langle C_1, \ldots, C_n \rangle$.

Definimos el predicado `es-forma-clausal` para comprobar si un objeto `Fc` es una forma clausal.

186
```
(defun es-forma-clausal (Fc)
  (or (endp Fc)
      (and (es-clausula (car Fc))
           (es-forma-clausal (cdr Fc)))))
```

Definición 8.9 *El valor de una forma clausal* $\langle C_1, \ldots, C_n \rangle$ *en una asignación* σ *es:* $\tilde{\sigma}(\langle C_1, \ldots, C_n \rangle) = \mathcal{F}_\wedge(\tilde{\sigma}(C_1), \mathcal{F}_\wedge(\ldots \mathcal{F}_\wedge(\tilde{\sigma}(C_{n-1}), \tilde{\sigma}(C_n)) \ldots))$.

La función que implementa este concepto es `valor-forma-clausal`:

187
```
(defun valor-forma-clausal (Fc sigma)
  (cond ((endp Fc) *V*)
        (t (funcion-de-verdad-de-conjuncion
            (valor-clausula (car Fc) sigma)
            (valor-forma-clausal (cdr Fc) sigma)))))
```

Definición 8.10 *Una asignación σ es modelo de una forma clausal S, y lo notaremos $\sigma \models S$, si y sólo si σ es modelo de todas las cláusulas de S. Una forma clausal S es satisfacible, si existe una asignación σ tal que $\sigma \models S$.*

Al igual que ocurre con la representación de las cláusulas, las formas clausales tal y como están representadas no son fórmulas reconocidas por el predicado `es-proposicional`. Por ello, tampoco podemos utilizar la función `modelo` para comprobar si una asignación es modelo de una forma clausal. La versión de esta función para formas clausales es `modelo-forma-clausal`.

188
```
(defun modelo-forma-clausal (sigma Fc)
  (if (consp Fc)
      (and (modelo-clausula sigma (car Fc))
           (modelo-forma-clausal sigma (cdr Fc)))
      t))
```

Definición 8.11 *Una forma clausal S es válida, si para toda asignación σ, el valor de S en σ es $\top$. Una forma clausal S es satisfacible si existe una asignación σ modelo de S. Una forma clausal S es insatisfacible si ninguna asignación σ es modelo de S.*

Al igual que ocurre con el concepto de cláusula válida, los conceptos de forma clausal satisfacible y forma clausal válida no se pueden definir en la formalización. Estos conceptos se incorporan a los teoremas de igual forma a como se hace para las fórmulas (ver detalles en la sección 3.2.4).

Se verifica que una asignación es modelo de una forma clausal según la definición anterior si y sólo si el valor de la forma clausal en dicha asignación es $\top$:

189
```
(defthm modelo-forma-clausal-valor-forma-clausal
  (iff (modelo-forma-clausal sigma C)
       (equal (valor-forma-clausal C sigma) *V*)))
```

Obsérvese que, por la definición de modelo de una forma clausal, una forma clausal es válida si y sólo si lo son todas las cláusulas que en ella aparecen. Por tanto, la forma clausal vacía es válida.

Teorema 8.12 *Si una forma clausal contiene la cláusula vacía, entonces es insatisfacible.*

La prueba de este resultado es inmediata, dado que para que una forma clausal sea satisfacible, tiene que existir una asignación modelo de todas las cláusulas que la forman y, la cláusula vacía no tiene modelos. Para formalizarlo utilizamos la función `contiene-clausula-vacia`, que comprueba si la cláusula vacía pertenece a una forma clausal:

```
                                                                    190
(defthm contiene-clausula-vacia-es-insatisfacible
  (implies (contiene-clausula-vacia Fc)
           (not (modelo-forma-clausal sigma Fc))))

(defun contiene-clausula-vacia (Fc)
  (cond ((endp Fc) nil)
        ((consp (car Fc))
         (contiene-clausula-vacia (cdr Fc)))
        (t t)))
```

8.1.2 Transformación a forma clausal: $\mathcal{FC}$

Teorema 8.13 *Existen algoritmos para transformar cualquier fórmula proposicional en una forma clausal lógicamente equivalente.*

Para demostrar este teorema implementaremos un algoritmo que realiza dicha transformación, basado en la notación uniforme. El algoritmo comienza con la lista cuyo único elemento es la lista unitaria formada por F, $\langle\langle F\rangle\rangle$. En cada paso aplica una regla que elimina una α-fórmula, β-fórmula o doble negación, reemplazándola por sus componentes. Cuando las reglas no se puedan aplicar obtenemos una lista de listas de literales, es decir, una forma clausal.

Algoritmo 8.14 *($\mathcal{FC}$) El dato de entrada de este algoritmo es una fórmula F y actúa como se describe a continuación:*

1. *Inicialmente se considera:* $\langle\langle F\rangle\rangle$

2. *Dada una lista de listas de fórmulas:* $\langle\Gamma_1,\ldots,\Gamma_n\rangle$, *se aplican repetidamente, hasta que no se pueda más, las siguientes reglas:*

   $\mathcal{R}_{\mathcal{FC}1}$ $\langle\Gamma_1,\ldots,\Gamma_n\rangle \Longrightarrow \langle\Gamma_1,\ldots,\Gamma_{i-1},\Gamma_{i+1},\ldots,\Gamma_n\rangle$, *si* Γ_i *contiene fórmulas complementarias.*

   $\mathcal{R}_{\mathcal{FC}2}$ $\langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},\neg\neg G,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle \Longrightarrow$
   $\Longrightarrow \langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},G,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle$

   $\mathcal{R}_{\mathcal{FC}3}$ $\langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},\beta,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle \Longrightarrow$
   $\Longrightarrow \langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},\beta_1,\beta_2,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle$

   $\mathcal{R}_{\mathcal{FC}4}$ $\langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},\alpha,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle \Longrightarrow$
   $\Longrightarrow \langle\Gamma_1,\ldots,\Gamma_{i-1},\langle F_1,\ldots,F_{j-1},\alpha_1,F_{j+1},...,F_m\rangle,$
   $\langle F_1,\ldots,F_{j-1},\alpha_2,F_{j+1},...,F_m\rangle,\Gamma_{i+1},\ldots,\Gamma_n\rangle$

3. *Si a la lista* $\langle\Gamma_1,\ldots,\Gamma_n\rangle$ *no se le puede aplicar ninguna de las reglas anteriores, el algoritmo termina y devuelve* $\langle\Gamma_1,\ldots,\Gamma_n\rangle$.

Este algoritmo se puede mejorar de dos formas. Puesto que la lista inicial no contiene listas con literales complementarios, la aplicación de la regla $\mathcal{R}_{\mathcal{FC}_1}$ se puede componer con las restantes, comprobando que el complementario del elemento o elementos que se añaden no aparezcan ya en la lista de fórmulas. Otra forma de mejorar las reglas es evitando que aparezcan elementos repetidos. Ambas mejoras se incorporan en el proceso de añadir una fórmula a una lista de fórmulas.

Son varias las funciones que intervienen en la implementación de este algoritmo. La función principal es `forma-clausal`, que recibe como argumento una fórmula F y realiza una llamada a `forma-clausal-aux` con la lista $\langle\langle F\rangle\rangle$.

191
```
(defun forma-clausal (F)
  (forma-clausal-aux (list (list F))))
```

La función `forma-clausal-aux` se encarga de aplicar recursivamente las reglas en las que se basa el algoritmo. Esta función recibe como argumento una lista de listas de fórmulas $\langle\Gamma_1, \ldots, \Gamma_n\rangle$. Si la lista está vacía el proceso termina. Si Γ_1 no tiene fórmulas no literales, entonces se trata de una cláusula y pasará a formar parte de la forma clausal que estamos construyendo. En este caso el proceso continúa con $\langle\Gamma_2, \ldots, \Gamma_n\rangle$. Si Γ_1 tiene fórmulas no literales, entonces se aplica una de las reglas $\mathcal{R}_{\mathcal{FC}_2}$, $\mathcal{R}_{\mathcal{FC}_3}$ o $\mathcal{R}_{\mathcal{FC}_4}$ (teniendo en cuenta la regla $\mathcal{R}_{\mathcal{FC}_1}$ a la hora de añadir las componentes) y se continúa el proceso.

192
```
(defun forma-clausal-aux (S)
  (declare (xargs :measure (medida-uniforme-S S)
                  :well-founded-relation mul-mul-e0-ord-<))
  (if (endp S)
      nil
      (let ((F (selecciona-no-literal (car S))))
        (if F
            (forma-clausal-aux
              (append (forma-clausal-procesa-una F (car S))
                      (cdr S)))
            (cons (car S) (forma-clausal-aux (cdr S)))))))
```

Obsérvese que el argumento de la función `forma-clausal-aux` no siempre disminuye en las llamadas recursivas, de hecho, en algunas ocasiones aumenta de tamaño ($\mathcal{R}_{\mathcal{FC}_3}$). Para demostrar la terminación del algoritmo ha sido necesario proporcionar una medida del argumento de `forma-clausal-aux` que disminuya en las llamadas recursivas, con respecto a una relación bien fundamentada. Los detalles sobre la prueba de terminación se encuentran en la página 198.

La función `selecciona-no-literal` recibe como argumento una lista de fórmulas y devuelve el primer elemento de la misma que no sea un literal. En caso de no existir ninguna fórmula no literal, devuelve `nil`.

193
```
(defun selecciona-no-literal (Gamma)
  (cond ((endp Gamma) nil)
        ((es-literal (car Gamma))
         (selecciona-no-literal (cdr Gamma)))
        (t (car Gamma))))
```

La función `forma-clausal-procesa-una` procesa una lista de fórmulas con un elemento no literal, aplicándole una de las reglas en las que se basa el algoritmo $\mathcal{FC}$. Obsérvese que los casos de esta función se corresponden con las reglas $\mathcal{R}_{\mathcal{FC}_2}$, $\mathcal{R}_{\mathcal{FC}_3}$ y $\mathcal{R}_{\mathcal{FC}_4}$ respectivamente. La regla $\mathcal{R}_{\mathcal{FC}_1}$ es considerada en el proceso de añadir las fórmulas componentes, implementado por `añade-formula` y `añade-formulas`. El último caso de esta función sirve para eliminar aquellas situaciones en las que los datos de entrada no son correctos.

194
```
(defun forma-clausal-procesa-una (F Gamma)
  (let ((Gamma-F (elimina-una F Gamma)))
    (cond ((doble-negacion F)
           (añade-formula (componente-neg-neg F) Gamma-F))
          ((beta-formula F)
           (añade-formulas (list (componente-1 F)
                                 (componente-2 F)) Gamma-F))
          ((alfa-formula F)
           (append (añade-formula (componente-1 F) Gamma-F)
                   (añade-formula (componente-2 F) Gamma-F)))
          (t '(nil)))))
```

La función `añade-formula` se encarga de aplicar la regla $\mathcal{R}_{\mathcal{FC}_1}$ y de evitar que aparezcan fórmulas repetidas al añadir las componentes. Esta función recibe como argumentos una fórmula F y una lista de fórmulas Γ. Si F aparece en Γ, devuelve una lista unitaria con la lista sin modificar, $\langle\Gamma\rangle$. Si el complementario de F aparece en Γ, devuelve `nil` como resultado de aplicar la regla $\mathcal{R}_{\mathcal{FC}_1}$. En cualquier otro caso devuelve la lista unitaria con el resultado de añadir F a Γ. De esta forma el valor devuelto por la función `añade-formula` es una lista cuyos elementos son listas de fórmulas. Si dicho valor es `nil` entonces la lista de fórmulas se tiene que eliminar como consecuencia de aplicar $\mathcal{R}_{\mathcal{FC}_1}$.

195
```
(defun añade-formula (F Gamma)
  (cond ((member-equal F Gamma) (list Gamma))
        ((member-equal (complementario F) Gamma) nil)
        (t (list (cons F Gamma)))))
```

Para añadir más de una fórmula a una lista siguiendo el criterio anterior definimos la función `añade-formulas`. Esta función actúa recursivamente sobre la lista de fórmulas que hay que añadir. Obsérvese que, después de realizar la llamada recursiva, se comprueba si el resultado es la lista vacía. Esta situación indicaría que en el proceso de la llamada recursiva se ha utilizado en algún momento la regla $\mathcal{R}_{\mathcal{FC}_1}$, por tanto el resultado final ha de ser la lista vacía.

196
```
(defun añade-formulas (Fs Gamma)
  (cond ((endp Fs) (list Gamma))
        (t (let ((añade-resto (añade-formulas (cdr Fs) Gamma)))
             (if añade-resto
                 (añade-formula (car Fs) (car añade-resto))
                 nil)))))
```

En el siguiente ejemplo, hemos utilizado la función `forma-clausal` para obtener las formas clausales asociadas a las fórmulas $(p \leftrightarrow q) \leftrightarrow (q \leftrightarrow p)$ y $(p \vee q) \rightarrow (r \wedge s)$. Obsérvese que el primer ejemplo es una fórmula válida y que la forma clausal devuelta es la vacía.

197
```
ACL2 !>(forma-clausal '(<-> (<-> p q) (<-> q p)))
NIL
ACL2 !>(forma-clausal '(-> (\/ p q) (& r s)))
((R (- P)) (S (- P)) (R (- Q)) (S (- Q)))
```

8.1.2.1 Prueba de terminación de $\mathcal{FC}$

La prueba de terminación de la función que implementa el algoritmo $\mathcal{FC}$ consiste en demostrar que la función `forma-clausal-aux` termina para cualquier argumento. Para ello, se ha proporcionado una medida de los argumentos que disminuye en las llamadas recursivas, con respecto a una relación bien fundamentada. El argumento de `forma-clausal-aux` es una lista de listas de fórmulas, de ahí que se haya considerado una medida que genera multiconjuntos de multiconjuntos de ordinales.

Definición 8.15 *La medida uniforme de una lista de fórmulas* $\langle F_1, \ldots, F_n\rangle$ *es el multiconjunto* $u^*(\langle F_1, \ldots, F_n\rangle) = \{\!\{u(F_1), \ldots, u(F_n)\}\!\}$.

La formalización de este concepto viene dada por la siguiente función:

198
```
(defun medida-uniforme-lista (Gamma)
  (cond ((endp Gamma) nil)
        (t (cons (medida-uniforme (car Gamma))
                 (medida-uniforme-lista (cdr Gamma))))))
```

Estas medidas se comparan con respecto a la extensión a multiconjuntos definida en 6.3 de la relación de orden entre los ordinales. Notaremos $<_{\mathcal{M}}$ a esta extensión. La función que la formaliza, `mul-e0-ord-<`, y sus propiedades se obtienen al evaluar el siguiente evento:

```
(defmul (E0-ORD-< NIL E0-ORDINALP E0-ORD-<-FN NIL NIL))
```

Teorema 8.16 *Sea $\langle F_1, \ldots, F_n\rangle$ una lista de fórmulas y F_i una fórmula no literal. Entonces:*

Si $F_i = \neg\neg G$ se verifica:

$$u^*(\langle F_1, \ldots, F_{j-1}, G, F_{j+1}, ..., F_m\rangle) <_{\mathcal{M}} u^*(\langle F_1, \ldots, F_m\rangle)$$

Si F es una α-fórmula se verifica:

$$u^*(\langle F_1, \ldots, F_{j-1}, \alpha_i, F_{j+1}, ..., F_m\rangle) <_{\mathcal{M}} u^*(\langle F_1, \ldots, F_m\rangle)$$

para $i = 1, 2$

Si F es una β-fórmula se verifica:

$$u^*(\langle F_1, \ldots, F_{j-1}, \beta_1, \beta_2, F_{j+1}, ..., F_m\rangle) <_{\mathcal{M}} u^*(\langle F_1, \ldots, F_m\rangle)$$

Para formalizar este resultado se usa la función `forma-clausal-procesa-una`, *194*, que se encarga de procesar una lista de fórmulas con un elemento no literal, aplicándole una de las reglas $\mathcal{R}_{\mathcal{FC}_2}$, $\mathcal{R}_{\mathcal{FC}_3}$ y $\mathcal{R}_{\mathcal{FC}_4}$.

199

```
(defthm medida-uniforme-Gamma-forma-clausal-procesa-una
  (implies
    (and (selecciona-no-literal Gamma)
         (member-equal N-Gamma
                       (forma-clausal-procesa-una
                         (selecciona-no-literal Gamma) Gamma)))
    (mul-e0-ord-< (medida-uniforme-Gamma N-Gamma)
                  (medida-uniforme-Gamma Gamma))))
```

La función `forma-clausal-procesa-una` también utiliza de forma implícita la regla $\mathcal{R}_{\mathcal{FC}_1}$ al añadir las fórmulas componentes. Puesto que entre las hipótesis del teorema se afirma la existencia de `N-Gamma` en el resultado devuelto por `forma-clausal-procesa-una`, podemos estar seguros de que a dicha lista de fórmulas no se le ha aplicado, ni se puede, la regla $\mathcal{R}_{\mathcal{FC}_1}$. La prueba de este teorema se obtiene fácilmente a partir de *53*.

Definición 8.17 *La medida uniforme de una lista de listas de fórmulas $\langle \Gamma_1, \ldots, \Gamma_n\rangle$ es el multiconjunto $u^{**}(\langle \Gamma_1, \ldots, \Gamma_n\rangle) = \{\!\{u^*(\Gamma_1), \ldots, u^*(\Gamma_n)\}\!\}$*

La formalización de este concepto viene dada por la siguiente función:

```
(defun medida-uniforme-S (S)                                        200
  (cond ((endp S) nil)
        (t (cons (medida-uniforme-Gamma (car S))
                 (medida-uniforme-S (cdr S))))))
```

Estas medidas se comparan con respecto a la extensión a multiconjuntos definida en 6.3 de la relación de orden $<_{\mathcal{M}}$ definida para multiconjuntos de ordinales. Notaremos $\ll_{\mathcal{M}}$ a esta extensión. La función que la formaliza, `mul-mul-e0-ord-<`, y sus propiedades se obtienen al evaluar el siguiente evento:

```
(defmul (MUL-E0-ORD-< MULTISET-EXTENSION-OF-E0-ORD-<-WELL-FOUNDED
                      E0-ORDINALP-TRUE-LISTP
                      MAP-E0-ORD-<-FN-E0-ORD X Y))
```

Teorema 8.18 *Si $\langle \Gamma_1, \ldots, \Gamma_n \rangle$ es una lista de listas de fórmulas a la que se puede aplicar alguna de las reglas $\mathcal{R}_{\mathcal{FC}_1}$, $\mathcal{R}_{\mathcal{FC}_2}$, $\mathcal{R}_{\mathcal{FC}_3}$ o $\mathcal{R}_{\mathcal{FC}_4}$, y $\langle \Gamma'_1, \ldots, \Gamma'_m \rangle$ es el resultado de aplicar dicha regla, entonces*

$$u^{**}(\langle \Gamma'_1, \ldots, \Gamma'_m \rangle) \ll_{\mathcal{M}} u^{**}(\langle \Gamma_1, \ldots, \Gamma_n \rangle)$$

y, por tanto, el algoritmo $\mathcal{FC}$ termina.

Este teorema es automáticamente generado al evaluar el evento *192*. Su demostración se obtiene gracias al teorema *199*.

A continuación se muestra un ejemplo de cómo la medida uniforme de una lista de listas de fórmulas disminuye al aplicar las reglas $\mathcal{R}_{\mathcal{FC}_1}$, $\mathcal{R}_{\mathcal{FC}_2}$, $\mathcal{R}_{\mathcal{FC}_3}$ o $\mathcal{R}_{\mathcal{FC}_4}$, para obtener una forma clausal para la fórmula $(p \leftrightarrow \neg q)$. En la primera columna se muestra la lista de listas de fórmulas en cada paso, en la segunda columna su medida uniforme y en la tercera, la regla aplicada de dicho paso.

Lista de listas de fórmulas	u^{**}	Regla
$\langle\langle p \leftrightarrow \neg q\rangle\rangle$	$\langle\langle 6\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_3}$
$\langle\langle p \wedge q, \neg p \wedge \neg\neg q\rangle\rangle$	$\langle\langle 3, 5\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_4}$
$\langle\langle p, \neg p \wedge \neg\neg q\rangle, \langle \neg q, \neg p \wedge \neg\neg q\rangle\rangle$	$\langle\langle 0, 5\rangle, \langle 1, 5\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_4}$
$\langle\langle p, \neg p\rangle, \langle p, \neg\neg q\rangle, \langle \neg q, \neg p \wedge \neg\neg q\rangle\rangle$	$\langle\langle 0, 1\rangle, \langle 0, 2\rangle, \langle 1, 5\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_1}$
$\langle\langle p, \neg\neg q\rangle, \langle \neg q, \neg p \wedge \neg\neg q\rangle\rangle$	$\langle\langle 0, 2\rangle, \langle 1, 5\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_2}$
$\langle\langle p, q\rangle, \langle \neg q, \neg p \wedge \neg\neg q\rangle\rangle$	$\langle\langle 0, 0\rangle, \langle 1, 5\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_4}$
$\langle\langle p, q\rangle, \langle \neg q, \neg p\rangle, \langle \neg q, \neg\neg q\rangle\rangle$	$\langle\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 1, 2\rangle\rangle$	$\mathcal{R}_{\mathcal{FC}_1}$
$\langle\langle p, q\rangle, \langle \neg q, \neg p\rangle\rangle$	$\langle\langle 0, 0\rangle, \langle 1, 1\rangle\rangle$	

Como se puede observar, en cada paso la medida de la lista de listas de fórmulas se reduce, ya sea porque se elimina una de sus sublistas, o porque se reemplaza una de ellas por una o varias sublistas menores que la original con respecto a $<_{\mathcal{M}}$, es decir en las que algún valor numérico de la eliminada se ha reemplazado por valores numéricos menores. Así en el primer paso la lista $\langle 6 \rangle$ se reemplaza por $\langle 3, 5 \rangle$ en la que todos los valores numéricos son menores que 6. En el segundo caso se reemplaza la lista $\langle 3, 5 \rangle$ por las listas $\langle 0, 5 \rangle$ y $\langle 1, 5 \rangle$, en las que el valor 3 se ha substituido respectivamente por 0 y 1, ambos menores, etc.

8.1.2.2 Propiedades de $\mathcal{FC}$

Teorema 8.19 *Si $F \in \mathbb{P}(\Sigma)$ entonces $\mathcal{FC}(F)$ es una forma clausal*

201
```
(defthm forma-clausal-es-forma-clausal
  (implies (es-proposicional F)
           (es-forma-clausal (forma-clausal F))))
```

La prueba de este evento es inmediata puesto que cuando `forma-clausal-aux` termina, no quedan más fórmulas no literales por procesar y, por tanto, el resultado es una forma clausal.

Teorema 8.20 *Si $F \in \mathbb{P}(\Sigma)$ entonces, para toda asignación σ, el valor de $\mathcal{FC}(F)$ en σ es igual al valor de F en σ.*

202
```
(defthm valor-forma-clausal-forma-clausal
  (implies (es-proposicional F)
           (equal (valor F sigma)
                  (valor-forma-clausal (forma-clausal F)
                                       sigma))))
```

La prueba de este teorema requiere demostrar previamente resultados similares sobre las funciones `forma-clausal-aux`, `forma-clausal-procesa-una`, `añade-formula`, `añade-formulas` y `elimina-una`.

Una consecuencia de este teorema es que, si disponemos de un procedimiento para decidir la satisfacibilidad de una forma clausal, podemos combinarlo con el algoritmo $\mathcal{FC}$ para obtener un procedimiento para decidir la satisfacibilidad de una fórmula. En la sección 8.2 se estudian dos procedimientos de decisión de satisfacibilidad para formas clausales, para cada uno de ellos se demuestran los resultados de corrección y completitud con respecto al concepto de modelo de una forma clausal. El teorema anterior nos asegura que la combinación de estos procedimientos con el algoritmo $\mathcal{FC}$, proporciona un proceso correcto y completo de decisión de satisfacibilidad para fórmulas.

Teorema 8.21 *Si $F \in \mathbb{P}(\Sigma)$ entonces $\mathcal{FC}(F)$ no contiene cláusulas con literales complementarios.*

La prueba de este resultado es simple: en el valor devuelto por $\mathcal{FC}$ no puede haber cláusulas con literales complementarios pues la regla $\mathcal{R}_{\mathcal{FC}_1}$ las elimina. Para formalizarlo utilizamos la función `tiene-clausulas-con-complementarios`, que comprueba si en una forma clausal aparecen cláusulas con literales complementarios.

203
```
(defthm forma-clausal-no-tiene-clausulas-con-complementarios
  (implies (es-proposicional F)
           (not (tiene-clausulas-con-complementarios
                  (forma-clausal F)))))

(defun tiene-clausulas-con-complementarios (Fc)
  (cond ((endp Fc) nil)
        ((tiene-literales-complementarios (car Fc)) t)
        (t (tiene-clausulas-con-complementarios (cdr Fc)))))
```

Como consecuencia de este teorema se obtiene un procedimiento para decidir la validez de una fórmula. Basta con aplicar a una fórmula el algoritmo $\mathcal{FC}$ para obtener una forma clausal, si se obtiene la forma clausal vacía entonces la fórmula original es válida.

Teorema 8.22 *Sea $F \in \mathbb{P}(\Sigma)$ entonces F es válida si y sólo si $\mathcal{FC}(F)$ es la forma clausal vacía.*

Demostración:

Si F es una fórmula válida entonces, por el teorema 8.20, $\mathcal{FC}(F)$ es una forma clausal válida y, por la definición de modelo de una forma clausal, todas las cláusulas de $\mathcal{FC}(F)$ son válidas. Por el teorema 8.5 sabemos que una cláusula es válida si y sólo si tiene literales complementarios y, por el teorema 8.21, $\mathcal{FC}(F)$ no tiene cláusulas con literales complementarios. Por tanto $\mathcal{FC}(F)$ es la forma clausal vacía.

Supongamos que $\mathcal{FC}(F)$ es la forma clausal vacía. Por la definición de modelo de una forma clausal sabemos que la forma clausal vacía es válida y por el teorema 8.20 que $\mathcal{FC}(F)$ y F tienen el mismo valor en cualquier asignación. Por tanto, F es válida.

□

La formalización de este teorema es la siguiente:

204

```
(defthm correccion-validez-forma-clausal
  (implies (and (es-proposicional F)
                (equal (forma-clausal F) nil))
           (modelo sigma F)))

(defthm completitud-validez-forma-clausal
  (implies (and (es-proposicional F)
                (forma-clausal F))
           (not (modelo (contramodelo-clausula
                         (car (forma-clausal F))) F))))
```

Obsérvese que en el resultado de completitud hemos utilizado la función `contramodelo-clausula` para obtener un contramodelo de una fórmula cuando al aplicar sobre ella el algoritmo $\mathcal{FC}$ no obtenemos la forma clausal vacía.

8.1.3 Cláusulas y formas clausales en $\mathbb{K}$

Los conceptos de cláusula y forma clausal se definen de la misma forma en la semántica de Kleene que en la semántica clásica. Las cláusulas se interpretan como la disyunción en $\mathbb{K}$ de los literales que las forman y las formas clausales como la conjunción en $\mathbb{K}$ de las cláusulas que las forman.

Definición 8.23 *El valor de una cláusula* $\langle L_1, \ldots, L_n\rangle$ *en una* $\mathbb{K}$*-asignación* σ *es:* $\tilde{\sigma}_{\mathbb{K}}(\langle L_1, \ldots, L_n\rangle) = \mathcal{K}_\vee(\tilde{\sigma}_{\mathbb{K}}(L_1), \mathcal{K}_\vee(\ldots \mathcal{K}_\vee(\tilde{\sigma}_{\mathbb{K}}(L_{n-1}), \tilde{\sigma}_{\mathbb{K}}(L_n))\ldots))$. *Una cláusula* C *es válida en* $\mathbb{K}$ *o* $\mathbb{K}$*-válida, si para toda* $\mathbb{K}$*-asignación* σ*, el valor de* C *en* σ *es* $\top$ *o* $\bot$.

Una $\mathbb{K}$*-asignación* σ *es modelo de una cláusula* C*, y lo notaremos* $\sigma \models_{\mathbb{K}} C$*, si y sólo si es modelo de algún literal de* C.

El valor de una cláusula en una $\mathbb{K}$-asignación se determina en la formalización con la función `valor-clausula-K`. El concepto de modelo en $\mathbb{K}$ de una cláusula se formaliza con la función `modelo-clausula-K`. Ambas funciones se definen como en el caso clásico pero utilizando las funciones `valor-K` y `modelo-K` en lugar de `valor` y `modelo`. También se verifica que una $\mathbb{K}$-asignación es modelo de una cláusula si y sólo si el valor de la cláusula en dicha $\mathbb{K}$-asignación es $\top$.

Teorema 8.24 *Una cláusula es válida en* $\mathbb{K}$ *si y sólo si contiene literales complementarios.*

Este teorema hace referencia al concepto de validez en $\mathbb{K}$ que, como en el caso clásico, no podemos definir. Para formalizarlo, tenemos en cuenta las observaciones hechas en la sección 3.3.4, acerca del concepto de $\mathbb{K}$-validez de una fórmula.

205
```
(defthm tiene-literales-complementarios-es-clausula-valida
  (implies (and (tiene-literales-complementarios C)
                (valor-clausula-K C sigma)
                (es-clausula C))
           (modelo-clausula-K sigma C)))

(defthm clausula-valida-tiene-literales-complementarios
  (implies (and (not (tiene-literales-complementarios C))
                (es-clausula C))
           (not (modelo-clausula-K
                  (contramodelo-clausula C) C))))
```

Definición 8.25 *El valor de una forma clausal* $\langle C_1, \ldots, C_n\rangle$ *en una* $\mathbb{K}$*-asignación* σ *es:* $\tilde{\sigma}_{\mathbb{K}}(\langle C_1, \ldots, C_n\rangle) = \mathcal{K}_\wedge(\tilde{\sigma}_{\mathbb{K}}(C_1), \mathcal{K}_\wedge(\ldots \mathcal{K}_\wedge(\tilde{\sigma}_{\mathbb{K}}(C_{n-1}), \tilde{\sigma}_{\mathbb{K}}(C_n))\ldots))$. *Una forma clausal* S *es válida en* $\mathbb{K}$ *o* $\mathbb{K}$*-válida, si para toda* $\mathbb{K}$*-asignación* σ*, el valor de* S *en* σ *es* $\top$ *o* $\perp\!\!\!\perp$.

Una $\mathbb{K}$*-asignación* σ *es modelo en* $\mathbb{K}$ *de una forma clausal* S*, y lo notaremos* $\sigma \models_{\mathbb{K}} S$*, si y sólo si* σ *es modelo en* $\mathbb{K}$ *de todas las cláusulas de* S*. Una forma clausal* S *es satisfacible en* $\mathbb{K}$ *o* $\mathbb{K}$*-satisfacible, si existe una* $\mathbb{K}$*-asignación* σ *tal que* $\sigma \models_{\mathbb{K}} S$.

El valor de una forma clausal en una $\mathbb{K}$-asignación se determina mediante la función `valor-forma-clausal-K`, que se define de igual forma que en el caso clásico pero utilizando `valor-clausula-K` en lugar de `valor-clausula`. La función `modelo-forma-clausal-K` formaliza el concepto de modelo en $\mathbb{K}$ de una forma cláusula, su definición es igual a la correspondiente en la lógica clásica pero utilizando `modelo-clausula-K` en lugar de `modelo-clausula`. También se verifica que una $\mathbb{K}$-asignación es modelo de una forma clausal si y sólo si el valor de la forma clausal en dicha $\mathbb{K}$-asignación es $\top$.

El algoritmo de transformación a forma clausal es aplicable en el caso de la semántica de Kleene y mantiene sus propiedades, entre las que destacamos dos:

1. El algoritmo $\mathcal{FC}$ construye formas clausales lógicamente equivalentes a la fórmula original: Si $F \in \mathbb{P}(\Sigma)$ entonces, para toda $\mathbb{K}$-asignación σ, el valor de $\mathcal{FC}(F)$ en σ es igual al valor de F en σ.

206
```
(defthm valor-forma-clausal-K-forma-clausal
  (implies (es-proposicional F)
           (equal (valor-K F sigma)
                  (valor-forma-clausal-K (forma-clausal F)
                                         sigma))))
```

2. El algoritmo $\mathcal{FC}$ proporciona un procedimiento para decidir la validez en $\mathbb{K}$ de una fórmula: Sea $F \in \mathbb{P}(\Sigma)$ entonces F es $\mathbb{K}$-válida si y sólo si $\mathcal{FC}(F)$ es la forma clausal vacía.

207
```
(defthm correccion-validez-forma-clausal
  (implies (and (es-proposicional F)
                (equal (forma-clausal F) nil)
                (valor-K F sigma))
           (modelo-K sigma F)))

(defthm completitud-validez-forma-clausal
  (implies (and (es-proposicional F)
                (forma-clausal F))
           (not (modelo-K (contramodelo-clausula
                           (car (forma-clausal F))) F))))
```

De nuevo hemos utilizado la función `contramodelo-clausula` para obtener un contramodelo en $\mathbb{K}$ de la fórmula `F`.

8.2 Davis y Putnam

En esta sección presentamos el procedimiento de Davis y Putnam como un método para decidir la satisfacibilidad de formas clausales. Para determinar la satisfacibilidad de una fórmula F, aplicaremos este procedimiento a la forma clausal obtenida al evaluar el algoritmo $\mathcal{FC}$ sobre F. El procedimiento consiste en aplicar repetidamente reglas de transformación a una forma clausal. Dichas reglas preservan la satisfacibilidad. Si como consecuencia de aplicarlas se obtiene la forma clausal vacía, entonces la forma clausal original es satisfacible. Si, a lo largo del proceso, todas las formas clausales obtenidas contienen la cláusula vacía, entonces la forma clausal original es insatisfacible.

En esta sección no presentamos ningún procedimiento para decidir la validez de una fórmula basado en el de Davis y Putnam, esto se debe a que utilizamos el algoritmo $\mathcal{FC}$ para obtener una forma clausal asociada a una fórmula y, como se ha visto en la sección anterior, este algoritmo sirve para caracterizar las fórmulas válidas.

En la primera parte de esta sección se analizan las reglas de transformación en las que se basa el procedimiento de Davis y Putnam y se demuestran sus propiedades. Estas transformaciones son: eliminación de cláusulas con literales complementarios, bifurcación, eliminación de cláusulas unitarias y eliminación de literales puros. Utilizando estas reglas de transformación presentamos dos algoritmos que permiten decidir la satisfacibilidad de una forma clausal. El primero de ellos se basa únicamente en la regla de bifurcación. El segundo utiliza las

reglas de bifurcación, eliminación de cláusulas unitarias y eliminación de literales puros y se desarrolla como un refinamiento del primero. La regla de eliminación de cláusulas con literales complementarios no es utilizada para formalizar estos algoritmos, ya que, si se utiliza el procedimiento $\mathcal{FC}$ para calcular la forma clausal asociada a una fórmula, entonces dicha forma clausal no tiene cláusulas con literales complementarios. Finalmente, se comenta cómo estos algoritmos se pueden utilizar en la semántica de Kleene y se indican las principales diferencias con los de la semántica clásica. Los eventos relativos a las semántica clásica se encuentran en el libro `davis-putnam.lisp` y los relativos a la semántica de Kleene en el libro `davis-putnam-K.lisp`.

8.2.1 Transformaciones de formas clausales

Como se ha comentado antes, el procedimiento de Davis y Putnam consiste en aplicar reglas de transformación a formas clausales. Veamos a continuación cuáles son estas reglas y qué propiedades tienen.

8.2.1.1 Eliminación de cláusulas con literales complementarios

La primera transformación consiste en eliminar las cláusulas con literales complementarios. Estas cláusulas son válidas, por tanto, son prescindibles a la hora de comprobar si una asignación es modelo de una forma clausal. Esta idea está reflejada en el siguiente teorema:

Teorema 8.26 *Dada una forma clausal S, si S' es el resultado de eliminar de S todas las cláusulas con literales complementarios, entonces dada una asignación σ, $\sigma \models S$ si y sólo si $\sigma \models S'$.*

Hemos definido la función `elimina-clausulas-con-complementarios` para implementar el proceso de eliminación de las cláusulas con literales complementarios. Una vez hecho esto podemos formalizar este teorema.

208
```
(defthm elimina-clausulas-con-complementarios-preserva-modelos
  (implies
    (es-forma-clausal Fc)
    (iff (modelo-forma-clausal
           sigma (elimina-clausulas-con-complementarios Fc))
         (modelo-forma-clausal sigma Fc))))

(defun elimina-clausulas-con-complementarios (Fc)
  (cond
    ((endp Fc) nil)
    ((tiene-literales-complementarios (car Fc))
     (elimina-clausulas-con-complementarios (cdr Fc)))
    (t (cons (car Fc)
             (elimina-clausulas-con-complementarios (cdr Fc))))))
```

El proceso de eliminación de cláusulas con literales complementarios es un paso preliminar al resto de las reglas del procedimiento de Davis y Putnam, ya que, cuando estas reglas son aplicadas a una forma clausal en la que no hay cláusulas con literales complementarios, no se generan nuevas cláusulas con literales complementarios. Como veremos más adelante, esta transformación no es necesaria para implementar un procedimiento de decisión de satisfacibilidad para formas clausales.

Una propiedad importante de esta transformación es que sirve para caracterizar las formas clausales válidas.

Teorema 8.27 *Sea S una forma clausal entonces, S es válida si y sólo si al eliminar de S todas las cláusulas con literales complementarios se obtiene la forma clausal vacía.*

La demostración de este teorema se debe a que una forma clausal es válida si y sólo si todas las cláusulas que la forman también lo son y, por el teorema 8.5, esto ocurre si y sólo si contienen literales complementarios. Por tanto, si eliminamos todas las cláusulas con literales complementarios, quedará la forma clausal vacía.

Este teorema se formaliza con los siguientes eventos:

209
```
(defthm elimina-clausulas-con-complementarios-nil-fc-valida
  (implies
    (and (es-forma-clausal Fc)
         (equal (elimina-clausulas-con-complementarios Fc) nil))
    (modelo-forma-clausal sigma Fc)))

(defthm fc-valida-elimina-clausulas-con-complementarios-nil
  (implies
    (and (es-forma-clausal Fc)
         (not (equal (elimina-clausulas-con-complementarios Fc)
                     nil)))
    (not (modelo-forma-clausal
           (contramodelo-clausula
             (car (elimina-clausulas-con-complementarios Fc)))
           Fc))))
```

La prueba de estos resultados se obtiene a partir de las propiedades que caracterizan las cláusulas válidas, mostradas en *185*.

8.2.1.2 Reducción de una forma clausal por un literal

El resto de las reglas de transformación del procedimiento de Davis y Putnam se basan en el proceso de reducción de una forma clausal por un literal. De forma intuitiva, este proceso reduce todo lo posible una forma clausal, asumiendo que un determinado literal es cierto:

Definición 8.28 *Sean S un conjunto de cláusulas y L un literal. La reducción de S por L es el conjunto $S_L = \{C - \{\overline{L}\} : C \in S \text{ y } L \notin C\}$. Análogamente, la reducción de S por $\overline{L}$ es el conjunto $S_{\overline{L}} = \{C - \{L\} : C \in S \text{ y } \overline{L} \notin C\}$.*

Por comodidad a la hora de expresar determinados eventos, hemos utilizado dos funciones para implementar el proceso de reducción de una forma clausal por un literal. La primera, `reduce-forma-clausal-literal`, sirve para obtener S_L, la otra, `reduce-forma-clausal-literal-C`, para obtener $S_{\overline{L}}$.

210
```
(defun reduce-forma-clausal-literal (L Fc)
  (cond ((endp Fc) nil)
        ((member-equal L (car Fc))
         (reduce-forma-clausal-literal L (cdr Fc)))
        (t (cons (elimina-literal (complementario L) (car Fc))
                 (reduce-forma-clausal-literal L (cdr Fc))))))

(defun reduce-forma-clausal-literal-C (L Fc)
  (cond ((endp Fc) nil)
        ((member-equal (complementario L) (car Fc))
         (reduce-forma-clausal-literal-C L (cdr Fc)))
        (t (cons (elimina-literal L (car Fc))
                 (reduce-forma-clausal-literal-C L (cdr Fc))))))

(defun elimina-literal (L C)
  (cond ((endp C) C)
        ((equal L (car C))
         (elimina-literal L (cdr C)))
        (t (cons (car C) (elimina-literal L (cdr C))))))
```

Existe una relación entre estas funciones que resulta de utilidad a la hora de establecer algunos eventos: el resultado de evaluar la primera de ellas sobre el complementario de un literal `L` es igual al resultado devuelto por la segunda. El evento que formaliza esta propiedad es el siguiente:

211
```
(defthm reduce-forma-clausal-literal-C-equivalente
  (implies (es-literal L)
           (equal (reduce-forma-clausal-literal-C L Fc)
                  (reduce-forma-clausal-literal
                    (complementario L) Fc))))
```

Como se ha comentado antes, si el proceso de reducción de una forma clausal por un literal es aplicado a una forma clausal en la que no haya cláusulas con literales complementarios, el resultado tampoco las tiene. La formalización de este hecho utiliza la función `tiene-clausulas-con-complementarios` que comprueba si una forma clausal tiene cláusulas con literales complementarios:

212

```
(defthm reduce-fc-literal-sin-clausulas-con-complementarios
  (implies (not (tiene-clausulas-con-complementarios Fc))
           (not (tiene-clausulas-con-complementarios
                 (reduce-forma-clausal-literal L Fc)))))

(defthm reduce-fc-literal-C-sin-clausulas-con-complementarios
  (implies (not (tiene-clausulas-con-complementarios Fc))
           (not (tiene-clausulas-con-complementarios
                 (reduce-forma-clausal-literal-C L Fc)))))

(defun tiene-clausulas-con-complementarios (Fc)
  (cond ((endp Fc) nil)
        ((tiene-literales-complementarios (car Fc)) t)
        (t (tiene-clausulas-con-complementarios (cdr Fc)))))
```

El siguiente teorema establece la idea intuitiva de que S_L es el resultado de reducir todo lo posible una forma clausal S asumiendo que un literal L es cierto:

Teorema 8.29 *Sea S una forma clausal y L un literal.*

1. *Si $\sigma \models S$ y $\sigma \models L$ entonces $\sigma \models S_L$.*
2. *Si $\sigma \models S_L$ entonces $\sigma[L/\top] \models S$.*

Demostración:

Sean S una forma clausal y L un literal,

1. Supongamos que $\sigma \models S$ y $\sigma \models L$, para demostrar que $\sigma \models S_L$ basta con probar que σ es modelo de cualquier elemento de S_L. Dada $C' \in S_L$, existe $C \in S$ tal que $C' = C - \{\overline{L}\}$. Como $\sigma \models L$ entonces $\sigma \not\models \overline{L}$, por tanto σ es modelo de un literal de C distinto de $\overline{L}$, luego $\sigma \models C - \{\overline{L}\} = C'$.
2. Supongamos que $\sigma \models S_L$, para demostrar que $\sigma[L/\top] \models S$ basta con probar que $\sigma[L/\top]$ es modelo de cualquier elemento de S. Dada $C \in S$:
   - Si $L \in C$ entonces $\sigma[L/\top] \models C$, puesto que, por el teorema 8.7, $\sigma[L/\top]$ es modelo de L.
   - Si $L \notin C$ entonces $C - \{\overline{L}\} \in S_L$ y por tanto $\sigma \models C - \{\overline{L}\}$. Como $L, \overline{L} \notin C - \{\overline{L}\}$, entonces existe un literal $L' \in C$ tal que $L' \neq L$, $L' \neq \overline{L}$ y $\sigma \models L'$, luego, por el teorema 8.7, $\sigma[L/\top] \models L'$, y de aquí que $\sigma[L/\top] \models C$.

□

La formalización de este resultado para `reduce-forma-clausal-literal` es la siguiente:

213
```
(defthm completitud-reduce-forma-clausal-literal
  (implies (and (modelo-forma-clausal sigma Fc)
                (modelo sigma L)
                (es-literal L))
           (modelo-forma-clausal
             sigma (reduce-forma-clausal-literal L Fc))))

(defthm correccion-reduce-forma-clausal-literal
  (implies (and (es-literal L)
                (es-forma-clausal Fc)
                (modelo-forma-clausal
                  sigma (reduce-forma-clausal-literal L Fc)))
           (modelo-forma-clausal (asume-literal L sigma) Fc)))
```

Obsérvese que el teorema es igualmente cierto para el literal $\overline{L}$, es decir:

1. Si $\sigma \models S$ y $\sigma \models \overline{L}$ entonces $\sigma \models S_{\overline{L}}$.
2. Si $\sigma \models S_{\overline{L}}$ entonces $\sigma[\overline{L}/\top] \models S$.

En este caso la formalización es la siguiente:

214
```
(defthm completitud-reduce-forma-clausal-literal-C
  (implies (and (modelo-forma-clausal sigma Fc)
                (not (modelo sigma L))
                (es-literal L))
           (modelo-forma-clausal
             sigma (reduce-forma-clausal-literal-C L Fc))))

(defthm correccion-reduce-forma-clausal-literal-C
  (implies (and (es-literal L)
                (es-forma-clausal Fc)
                (modelo-forma-clausal
                 sigma (reduce-forma-clausal-literal-C L Fc)))
           (modelo-forma-clausal
             (asume-literal (complementario L) sigma) Fc)))
```

8.2.1.3 Regla de bifurcación

La regla de bifurcación consiste en dividir una forma clausal S en dos, en función de si un literal L es cierto o falso. Para cada una de estas dos posibilidades se consideran, respectivamente, los conjuntos S_L y $S_{\overline{L}}$. En este caso diremos que se ha aplicado la regla de bifurcación en el literal L. La satisfacibilidad de S quedará garantizada si al menos uno de los dos conjuntos es satisfacible.

Teorema 8.30 *Sea S una forma clausal y L un literal, entonces S es satisfacible si y sólo si S_L o $S_{\overline{L}}$ también lo es.*

Demostración:

Supongamos que S es satisfacible. Sea σ una asignación modelo de S. Si $\sigma \models L$ entonces, por el teorema 8.29-1, $\sigma[L/\top]$ es modelo de S_L. Si $\sigma \not\models L$ entonces $\sigma \models \overline{L}$ y, de nuevo por el teorema 8.29-1, $\sigma[\overline{L}/\top]$ es modelo de $S_{\overline{L}}$. Luego S_L o $S_{\overline{L}}$ es satisfacible.

Supongamos ahora que S_L es satisfacible. Sea σ una asignación modelo de S_L. Por el teorema 8.29-2, $\sigma[L/\top] \models S$, luego S es satisfacible.

Supongamos finalmente que $S_{\overline{L}}$ es satisfacible. Sea σ una asignación modelo de $S_{\overline{L}}$. Por el teorema 8.29-2, $\sigma[\overline{L}/\top] \models S$, luego S es satisfacible.

□

La formalización de este teorema consiste en los eventos mostrados en *213* y *214*.

Como se verá en la sección 8.2.2, la regla de bifurcación por sí sola es suficiente para decidir la satisfacibilidad de una forma clausal. La forma de actuar de esta regla es similar a la de las tablas de verdad: se analizan las dos situaciones que surgen al asignarle un valor de verdad a cada variable. Sin embargo, en determinadas ocasiones no es necesario analizar las dos posibilidades a las que da lugar la regla de bifurcación, ya que una de las dos, o es insatisfacible, o su satisfacibilidad es consecuencia de la satisfacibilidad de la otra posibilidad.

8.2.1.4 Eliminación de cláusulas unitarias

La regla de eliminación de cláusulas unitarias es un caso particular de la regla de bifurcación, en el que una de las dos formas clausales obtenidas es insatisfacible y, por tanto, se puede prescindir de su análisis.

Definición 8.31 *Una cláusula C es unitaria si contiene un único elemento.*

215
```
(defun clausula-unitaria (C)
  (and (consp C)
       (not (consp (elimina-literal (car C) C)))))
```

Si la cláusula unitaria $C = \{L\}$ está en una forma clausal S, entonces el conjunto $S_{\overline{L}}$ contiene la cláusula vacía y, por el teorema 8.12 es insatisfacible. Luego, al aplicar la regla de bifurcación en L, se puede prescindir del análisis de una de las dos formas clausales obtenidas. En esta situación la satisfacibilidad de S quedará garantizada si S_L es satisfacible.

Teorema 8.32 *Sea S una forma clausal y $C = \{L\}$ una cláusula de S, entonces S es satisfacible si y sólo si S_L también lo es.*

Demostración:

Por el teorema 8.30 S es satisfacible si y sólo si S_L o $S_{\overline{L}}$ también lo es. Como $C = \{L\}$ es una cláusula unitaria de S, entonces $C - \{L\} = \emptyset \in S_{\overline{L}}$ y, por el teorema 8.12, $S_{\overline{L}}$ es insatisfacible. Luego S es satisfacible si y sólo si S_L también lo es.

□

En la formalización, sólo hemos necesitado añadir un evento afirmando que la reducción de una forma clausal por el complementario de un literal de una cláusula unitaria, contiene la cláusula vacía:

216
```
(defthm reduce-fc-literal-C-clausula-unitaria-contiene-nula
  (implies (and (member-equal C Fc)
                (clausula-unitaria C)
                (member-equal L C))
           (contiene-clausula-vacia
             (reduce-forma-clausal-literal-C L Fc))))
```

8.2.1.5 Eliminación de literales puros

La regla de eliminación de literales puros es otro caso particular de la regla de bifurcación, en el que se puede prescindir del análisis de una de las dos formas clausales obtenidas. En este caso la satisfacibilidad de una de ellas conlleva la satisfacibilidad de la otra y, por tanto, sólo será necesaria analizar la segunda.

Definición 8.33 *Dada una forma clausal S, un literal puro en S es un literal cuyo complementario no aparece en ninguna cláusula de S.*

La función `es-literal-puro` formaliza este concepto. Para ello hemos utilizado la función `literal-en-forma-clausal` que comprueba si un literal aparece en alguna cláusula de una forma clausal.

217
```
(defun es-literal-puro (L Fc)
  (not (literal-en-forma-clausal (complementario L) Fc)))

(defun literal-en-forma-clausal (L Fc)
  (cond ((endp Fc) nil)
        (t (or (member-equal L (car Fc))
               (literal-en-forma-clausal L (cdr Fc))))))
```

Teorema 8.34 *Si L es un literal puro en S, entonces $S_L \subseteq S_{\overline{L}}$.*

Demostración:

Dado $C \in S_L$, existe $C' \in S$ tal que $L \notin C'$ y $C = C' - \{\overline{L}\}$. Como L es un literal puro en S, $\overline{L}$ no puede aparecer en C', luego $C = C'$ y $L, \overline{L} \notin C$. Por tanto $C - \{L\} = C \in S_{\overline{L}}$.

□

La formalización de este resultado es la siguiente:

218
```
(defthm reduce-forma-clausal-literal-C-literal-puro
  (implies (es-literal-puro L Fc)
           (subsetp-equal (reduce-forma-clausal-literal L Fc)
                          (reduce-forma-clausal-literal-C L Fc))))
```

Teorema 8.35 *Sean S y S' dos formas clausales tales que $S \subseteq S'$. Para toda asignación σ, si $\sigma \models S'$ entonces $\sigma \models S$.*

Este resultado es trivial a partir del concepto de modelo de una forma clausal. Su formalización es la siguiente:

219
```
(defthm modelo-forma-clausal-subsetp-equal
  (implies (and (subsetp-equal Fc-1 Fc-2)
                (modelo-forma-clausal sigma Fc-2))
           (modelo-forma-clausal sigma Fc-1)))
```

Los dos resultados anteriores permiten prescindir del análisis de $S_{\overline{L}}$ al aplicar la regla de bifurcación a un literal L puro en la forma clausal S.

Teorema 8.36 *Sea S una forma clausal y L un literal puro en S, entonces S es satisfacible si y sólo si S_L también lo es.*

Demostración:

Supongamos que S es satisfacible, entonces, por el teorema 8.30, S_L o $S_{\overline{L}}$ es satisfacible. Si $S_{\overline{L}}$ es satisfacible entonces, por el teorema 8.35, S_L también lo es pues $S_L \subseteq S_{\overline{L}}$ (teorema 8.34). Por tanto, en cualquier caso S_L es satisfacible.

Si S_L es satisfacible entonces, por el teorema 8.30, se tiene que S también lo es.

□

No ha sido necesario añadir ningún evento formalizando este resultado, es suficiente con los mostrados en *218* y *219*.

8.2.2 Decisión de satisfacibilidad por bifurcación: $SAT_{\mathcal{B}}$

A continuación desarrollamos un algoritmo para decidir la satisfacibilidad de una forma clausal basado únicamente en la regla de bifurcación. En cada paso se toma un literal que aparezca en alguna cláusula de una forma clausal y se consideran las dos situaciones que se pueden dar, en función de si ese literal es cierto o falso. Si se realiza este proceso para todos los literales que aparezcan en alguna cláusula de una forma clausal, estaremos analizando todas las asignaciones definidas en el conjunto de variables que aparecen en dicha forma clausal; es decir, estaremos desarrollando un proceso similar al basado en tablas de verdad. De aquí que el resultado sea un procedimiento de decisión de satisfacibilidad.

Puesto que las reglas de eliminación de cláusulas unitarias y de literales puros son casos particulares de la regla de bifurcación, los resultados de corrección y completitud para el procedimiento de Davis y Putnam se obtendrán como instancias de los mismos resultados para el procedimiento basado en bifurcación.

Algoritmo 8.37 (*$SAT_{\mathcal{B}}$*) *El dato de entrada de este algoritmo es una forma clausal S, y actúa como se describe a continuación:*

1. *Si S contiene la cláusula vacía, el algoritmo termina y devuelve* **f**.

2. *Se selecciona un literal L que aparezca en alguna cláusula de S. A continuación se evalúa el algoritmo sobre S_L y, si devuelve* **f**, *entonces se evalúa sobre $S_{\overline{L}}$.*

3. *Si no se puede escoger un literal L que aparezca en alguna cláusula de S, es decir, S está vacía, entonces el algoritmo termina y devuelve* **t**.

Este algoritmo no está completamente determinado puesto que no se indica cómo se escoge el literal L. Para resolver esta indeterminación hemos considerado una función `seleccion` que escoge un literal de una de las cláusulas de S. Si S está vacía devuelve `nil`. Se ha asumido la existencia de esta función en un encapsulado ACL2, caracterizándola por las propiedades:

- Si la función `seleccion` devuelve un valor, entonces dicho valor es un literal.

- Si `(seleccion Fc)` devuelve un valor, entonces dicho valor aparece en alguna cláusula de `Fc`.

- Si la forma clausal `Fc` tiene alguna cláusula no vacía, entonces `(seleccion Fc)` devuelve algún valor.

La formalización de estas propiedades es la siguiente:

220
```
(defthm seleccion-esta-en-conjunto
  (implies (seleccion Fc)
           (literal-en-forma-clausal (seleccion Fc) Fc)))

(defthm seleccion-es-literal
  (implies (and (es-forma-clausal Fc)
                (seleccion Fc))
           (es-literal (seleccion Fc))))

(defthm seleccion-existe
  (implies (and (es-forma-clausal Fc)
                (consp Fc)
                (consp (car Fc)))
           (seleccion Fc)))
```

La función que implementa el algoritmo $SAT_{\mathcal{B}}$ es la siguiente:

221
```
(defun SAT-bifurcacion (Fc)
  (declare (xargs :measure (medida-forma-clausal Fc)))
  (if (contiene-clausula-vacia Fc)
      nil
      (let ((L (seleccion Fc)))
        (if L
            (or (SAT-bifurcacion
                  (reduce-forma-clausal-literal L Fc))
                (SAT-bifurcacion
                  (reduce-forma-clausal-literal-C L Fc)))
            t))))
```

Para una función de selección concreta, por ejemplo la que devuelve el primer literal de la primera cláusula no vacía de una forma clausal, podemos utilizar la función anterior para determinar la satisfacibilidad de un conjunto de cláusulas. Si la forma clausal es satisfacible, la función devuelve `T`, en caso contrario devuelve `nil`.

222
```
ACL2 !>(SAT-bifurcacion '( (p q) ((- p) (- q)) ))
T
ACL2 !>(SAT-bifurcacion '( (q) ((- p) (- q)) (p (- q)) ))
NIL
```

En el primer caso la forma clausal $\langle\langle p, q\rangle, \langle\neg p, \neg q\rangle\rangle$ es satisfacible (un modelo es cualquier asignación en la que p sea cierta y q falsa). En el segundo caso, la forma clausal $\langle\langle q\rangle, \langle\neg p, \neg q\rangle, \langle p, \neg q\rangle\rangle$ es insatisfacible.

Para que la función `SAT-bifurcacion` sea admitida por ACL2, se ha de demostrar que termina para cualquier dato de entrada. Para ello hemos proporcionado una medida de su argumento, `medida-forma-clausal` que decrece en las llamadas recursivas.

Definición 8.38 *La longitud de una forma clausal, $l(S)$, es la suma del número de literales que aparecen en las cláusulas que la forman.*

La función `medida-forma-clausal` formaliza este concepto. Para ello se utiliza la función `len` que devuelve el número de elementos de una lista:

223
```
(defun medida-forma-clausal (Fc)
  (cond ((endp Fc) 0)
        (t (+ (len (car Fc))
              (medida-forma-clausal (cdr Fc))))))
```

Teorema 8.39 *Sean S una forma clausal y L un literal que aparece en alguna cláusula de S entonces $l(S_L), l(S_{\overline{L}}) < l(S)$.*

Los eventos que formalizan este teorema son los siguientes:

224
```
(defthm reduce-forma-clausal-literal-disminuye-medida
  (implies (literal-en-forma-clausal L Fc)
           (< (medida-forma-clausal
                (reduce-forma-clausal-literal L Fc))
              (medida-forma-clausal Fc))))

(defthm reduce-forma-clausal-literal-C-disminuye-medida
  (implies (literal-en-forma-clausal L Fc)
           (< (medida-forma-clausal
                (reduce-forma-clausal-literal-C L Fc))
              (medida-forma-clausal Fc))))
```

Teorema 8.40 (Completitud de $SAT_{\mathcal{B}}$) *Dada una forma clausal S, si S es satisfacible entonces $SAT_{\mathcal{B}}(S) = \mathbf{t}$.*

Demostración:
La prueba se desarrolla por inducción en el número total de literales que aparecen en las cláusulas de S.

- Si $l(S) = 0$ y S es satisfacible entonces S no puede contener la cláusula vacía y no se puede seleccionar ningún literal de una cláusula de S, por tanto $SAT_{\mathcal{B}}(S) = \mathbf{t}$.

- Supongamos que para toda forma clausal S' satisfacible tal que $l(S') < l(S)$, se tiene $SAT_{\mathcal{B}}(S') = \mathbf{t}$. Si S es satisfacible entonces no contiene la cláusula vacía. Consideremos L un literal cualquiera que aparezca en una cláusula de S.

  Como S es satisfacible, por el teorema 8.30 se tiene que S_L o $S_{\overline{L}}$ también lo es. Por el teorema 8.39 se tiene que $l(S_L), l(S_{\overline{L}}) < l(S)$. Luego, por la hipótesis de inducción, $SAT_{\mathcal{B}}(S_L) = \mathbf{t}$ o $SAT_{\mathcal{B}}(S_{\overline{L}}) = \mathbf{t}$, es decir alguno de los dos casos del punto 2 de la descripción del algoritmo $SAT_{\mathcal{B}}$ tiene éxito, por tanto, $SAT_{\mathcal{B}}(S) = \mathbf{t}$.

□

La formalización de este teorema es la siguiente:

225
```
(defthm completitud-SAT-bifurcacion
  (implies (and (modelo-forma-clausal sigma Fc)
                (es-forma-clausal Fc))
           (SAT-bifurcacion Fc)))
```

La demostración de este resultado en ACL2 se obtiene fácilmente a partir de los eventos `completitud-reduce-forma-clausal-literal` y `completitud-reduce-forma-clausal-literal-C`, mostrados respectivamente en *213* y *214*.

Teorema 8.41 (Corrección de $SAT_{\mathcal{B}}$) *Dada una forma clausal S, si $SAT_{\mathcal{B}}(S) = \mathbf{t}$ entonces S es satisfacible.*

Demostración:

La prueba se desarrolla por inducción en el número total de literales que aparecen en las cláusulas de S.

- Si $l(S) = 0$ y $SAT_{\mathcal{B}}(S) = \mathbf{t}$, entonces S es la forma clausal vacía, que es cierta en cualquier asignación. Luego S es satisfacible.

- Supongamos que toda forma clausal S' tal que $l(S') < l(S)$ y $SAT_{\mathcal{B}} = \mathbf{t}$, es satisfacible. Puesto que $SAT_{\mathcal{B}}(S) = \mathbf{t}$, entonces S no contiene la cláusula vacía. Como $l(S) > 0$, existe un literal L en una cláusula de S. Como $SAT_{\mathcal{B}}(S) = \mathbf{t}$, entonces $SAT_{\mathcal{B}}(S_L) = \mathbf{t}$ o $SAT_{\mathcal{B}}(S_{\overline{L}}) = \mathbf{t}$.

  Si $SAT_{\mathcal{B}}(S_L) = \mathbf{t}$ entonces, por hipótesis de inducción se tiene que S_L es satisfacible, ya que $l(S_L) < l(S)$ por el teorema 8.39. Por tanto, el teorema 8.30 asegura que S es satisfacible.

  Si $SAT_{\mathcal{B}}(S_{\overline{L}}) = \mathbf{t}$ entonces, por hipótesis de inducción se tiene que $S_{\overline{L}}$ es satisfacible, ya que $l(S_{\overline{L}}) < l(S)$ por el teorema 8.39. Por tanto, el teorema 8.30 asegura que S es satisfacible.

□

Al igual que ocurre con los resultados de corrección de los procedimientos de decisión de satisfacibilidad de una fórmula (sección 3.2.4), para formalizar este teorema se necesita proporcionar explícitamente una asignación modelo de la forma clausal. La función que construye dicho modelo es `MOD-bifurcacion`. Esta función actúa exactamente igual que `SAT-bifurcacion`, pero tiene un argumento adicional donde se va construyendo una asignación, en la que todos los literales por los que se ha hecho alguna reducción son ciertos. Cuando la función `MOD-bifurcacion` termina devuelve dicha asignación.

226
```
(defun MOD-bifurcacion (Fc sigma)
  (declare (xargs :measure (medida-forma-clausal Fc)))
  (if (contiene-clausula-vacia Fc)
      nil
      (let ((L (seleccion Fc)))
        (if L
            (or (MOD-bifurcacion
                  (reduce-forma-clausal-literal L Fc)
                  (asume-literal L sigma))
                (MOD-bifurcacion
                  (reduce-forma-clausal-literal-C L Fc)
                  (asume-literal (complementario L) sigma)))
            sigma))))
```

Al igual que `SAT-bifurcacion`, esta función puede ser evaluada una vez se proporciona una función de selección concreta. En los siguientes ejemplos se ha usado la función de selección que devuelve el primer literal de la primera cláusula no vacía de una forma clausal.

227
```
ACL2 !>(MOD-bifurcacion '( (p q) ((- p) (- q)) ) nil)
((Q 0) (P 1))
ACL2 !>(MOD-bifurcacion '( (q) ((- p) (- q)) (p (- q)) ) nil)
NIL
```

En el primer caso la forma clausal $\langle\langle p, q\rangle, \langle\neg p, \neg q\rangle\rangle$ es satisfacible y un modelo es cualquier asignación en la que p sea cierto y q falso. En el segundo caso, la forma clausal $\langle\langle q\rangle, \langle\neg p, \neg q\rangle, \langle p, \neg q\rangle\rangle$ es insatisfacible y, por tanto, no tiene modelos.

Esta función guarda una relación obvia con la función `SAT-bifurcacion`: `MOD-bifurcacion` devuelve un valor distinto de `nil` si y sólo si `SAT-bifurcacion` también lo hace.

228
```
(defthm SAT-bifurcacion-MOD-bifurcacion
  (implies (consp sigma)
           (iff (MOD-bifurcacion Fc sigma)
                (SAT-bifurcacion Fc))))
```

Gracias a esta relación y a otras propiedades que relacionan las funciones `asume-literal` y `MOD-bifurcacion`, se demuestra el siguiente evento que formaliza el teorema de corrección de $SAT_{\mathcal{B}}$:

229
```
(defthm correccion-SAT-bifurcacion
  (implies (and (es-forma-clausal Fc)
                (SAT-bifurcacion Fc))
           (modelo-forma-clausal (MOD-bifurcacion Fc sigma) Fc)))
```

8.2.3 Decisión de satisfacibilidad por Davis-Putnam: $SAT_{\mathcal{D}}$

El procedimiento de Davis y Putnam combina las reglas de transformación mostradas en la sección 8.2.1, para decidir la satisfacibilidad de una forma clausal. La regla de eliminación de cláusulas con literales complementarios es utilizada una única vez, al comienzo del proceso, ya que el resto de las reglas de transformación no genera nuevas cláusulas con literales complementarios. Como se ha visto en la sección anterior, esta regla no es necesaria para decidir la satisfacibilidad de una forma clausal.

El algoritmo que presentamos basado en el procedimiento de Davis y Putnam utiliza únicamente las reglas de eliminación de cláusulas unitarias, eliminación de literales puros y bifurcación, dando prioridad a las dos primeras frente a la última, para así disminuir el número de ramificaciones.

Algoritmo 8.42 ($SAT_{\mathcal{D}}$) *El dato de entrada de este algoritmo es una forma clausal S, y actúa como se describe a continuación:*

1. *Si S contiene la cláusula vacía, el algoritmo termina y devuelve* **f**.

2. *Si S contiene cláusulas unitarias, se selecciona una de ellas $C = \{L\}$ y se evalúa el algoritmo sobre S_L.*

3. *Si S tiene literales puros, se selecciona uno de ellos L y se evalúa el algoritmo sobre S_L.*

4. *En otro caso, se selecciona un literal L que aparezca en alguna cláusula de S. A continuación se evalúa el algoritmo sobre S_L y, si devuelve* **f**, *entonces se evalúa sobre $S_{\overline{L}}$.*

5. *Si no se puede escoger un literal L que aparezca en alguna cláusula de S, es decir, S está vacía, entonces el algoritmo termina y devuelve* $\mathbf{t}$.

Teorema 8.43 *Dada una forma clausal S, $SAT_{\mathcal{D}}(S) = \mathbf{t}$ si y sólo si S es satisfacible.*

Demostración:

Este algoritmo es equivalente a $SAT_{\mathcal{B}}$ para una función de selección que, en primer lugar busca literales en cláusulas unitarias, si no los encuentra busca literales puros y en otro caso devuelve un literal cualquiera:

- Si el literal escogido pertenece a una cláusula unitaria, por el teorema 8.32 sabemos que S es satisfacible si y sólo si lo es S_L y, por tanto, no hay que evaluar el algoritmo sobre $S_{\overline{L}}$.

- Si el literal escogido es puro, por el teorema 8.36 sabemos que S es satisfacible si y sólo si lo es S_L y, por tanto, tampoco hay que evaluar el algoritmo sobre $S_{\overline{L}}$.

- En otro caso el algoritmo $SAT_{\mathcal{D}}$ funciona igual que $SAT_{\mathcal{B}}$.

De esta forma, por los teoremas de corrección y completitud de $SAT_{\mathcal{B}}$, se tiene que $SAT_{\mathcal{D}}(S) = \mathbf{t}$ si y sólo si S es satisfacible.

□

La función de selección mencionada en la demostración del teorema es la siguiente:

230
```
(defun seleccion-instancia (Fc)
  (let ((C (busca-clausula-unitaria Fc)))
    (if C
        (car C)
        (let ((L (busca-literal-puro Fc)))
          (or L
              (if (endp Fc)
                  nil
                  (seleccion Fc)))))))
```

Donde `busca-clausula-unitaria` devuelve la primera cláusula unitaria de la forma clausal `Fc` y `busca-literal-puro` el primer literal puro en `Fc`.

Esta función tiene las propiedades formalizadas en *220* y por tanto el algoritmo $SAT_{\mathcal{B}}$ construido a partir de ella conserva las propiedades de corrección y completitud. La versión de la función que implementa $SAT_{\mathcal{B}}$ para la función de selección `seleccion-instancia` es:

231
```
(defun SAT-bifurcacion-instancia (Fc)
  (declare (xargs :measure (medida-forma-clausal Fc)))
  (if (contiene-clausula-vacia Fc)
      nil
      (let ((L (seleccion-instancia Fc)))
        (if L
            (or (SAT-bifurcacion-instancia
                  (reduce-forma-clausal-literal L Fc))
                (SAT-bifurcacion-instancia
                  (reduce-forma-clausal-literal-C L Fc)))
            t))))
```

De manera análoga, construimos la función `MOD-bifurcacion-instancia` como un caso particular de la función `MOD-bifurcacion` cuando la función de selección utilizada es `seleccion-instancia`:

232
```
(defun MOD-bifurcacion-instancia (Fc sigma)
  (declare (xargs :measure (medida-forma-clausal Fc)))
  (if (contiene-clausula-vacia Fc)
      nil
      (let ((L (seleccion-instancia Fc)))
        (if L
            (or (MOD-bifurcacion-instancia
                  (reduce-forma-clausal-literal L Fc)
                  (asume-literal L sigma))
                (MOD-bifurcacion-instancia
                  (reduce-forma-clausal-literal-C L Fc)
                  (asume-literal (complementario L) sigma)))
            sigma))))
```

Los eventos que establecen las propiedades de corrección y completitud de la función `SAT-bifurcacion-instancia` se obtienen como instancias funcionales (ver sección 2.2.6) de los correspondientes para la función `SAT-bifurcacion`, sustituyendo las funciones `SAT-bifurcacion`, `MOD-bifurcacion` y `seleccion` por las funciones `SAT-bifurcacion-instancia`, `MOD-bifurcacion-instancia` y `seleccion-instancia`:

233
```
(defthm completitud-SAT-bifurcacion-instancia
  (implies (and (es-forma-clausal Fc)
                (modelo-forma-clausal sigma Fc))
           (SAT-bifurcacion-instancia Fc)))

(defthm correccion-SAT-bifurcacion-instancia
  (implies (and (es-forma-clausal Fc)
                (SAT-bifurcacion-instancia Fc))
           (modelo-forma-clausal
             (MOD-bifurcacion-instancia Fc sigma) Fc)))
```

La función que implementa $SAT_{\mathcal{D}}$ no es `SAT-bifurcacion-instancia`, ya que esta función evalúa las formas clausales $S_{\overline{L}}$ para los literales L en cláusulas unitarias y puros. La función que implementa el algoritmo $SAT_{\mathcal{D}}$ es la siguiente:

234
```
(defun SAT-davis-putnam (Fc)
  (declare (xargs :measure (medida-forma-clausal Fc)))
  (cond
   ((contiene-clausula-vacia Fc) nil)
   (t (let ((C (busca-clausula-unitaria Fc)))
       (if C
           (SAT-davis-putnam
             (reduce-forma-clausal-literal (car C) Fc))
           (let ((L (busca-literal-puro Fc)))
            (if L
                (SAT-davis-putnam
                  (reduce-forma-clausal-literal L Fc))
                (let ((L (seleccion Fc)))
                 (if L
                     (or (SAT-davis-putnam
                           (reduce-forma-clausal-literal L Fc))
                         (SAT-davis-putnam
                           (reduce-forma-clausal-literal-C L Fc)))
                     t)))))))))
```

Las funciones `SAT-bifurcacion-instancia` y `SAT-davis-putnam` son equivalentes. Esto se debe a que, para los literales en cláusulas unitarias y los literales puros, no es necesario evaluar `SAT-bifurcacion-instancia` sobre la forma clausal $S_{\overline{L}}$. La formalización de estas afirmaciones es la siguiente:

235
```
(defthm SAT-bifurcacion-instancia-clausula-unitaria
  (implies (and (es-forma-clausal Fc)
                (member-equal C Fc)
                (clausula-unitaria C))
           (not (SAT-bifurcacion-instancia
                  (reduce-forma-clausal-literal-C (car C) Fc)))))

(defthm SAT-bifurcacion-instancia-literal-puro
  (implies (and (es-forma-clausal Fc)
                (es-literal-puro L Fc)
                (SAT-bifurcacion-instancia
                  (reduce-forma-clausal-literal-C L Fc)))
           (SAT-bifurcacion-instancia
             (reduce-forma-clausal-literal L Fc)))
```

La prueba de estos resultados se obtiene a partir de las propiedades de las reducciones por literales en cláusulas unitarias y literales puros, mostradas en *216*, *218* y *219*.

Utilizando los eventos anteriores se demuestra que `SAT-davis-putnam` y `SAT-bifurcacion-instancia` son equivalentes y, por tanto, `SAT-davis-putnam` es un proceso correcto y completo de decisión de satisfacibilidad.

236
```
(defthm SAT-davis-putnam-y-SAT-bifurcacion-equivalentes
  (implies (es-forma-clausal Fc)
           (iff (SAT-davis-putnam Fc)
                (SAT-bifurcacion-instancia Fc))))

(defthm completitud-SAT-davis-putnam
  (implies (and (es-forma-clausal Fc)
                (modelo-forma-clausal sigma Fc))
           (SAT-davis-putnam Fc)))

(defthm correccion-SAT-davis-putnam
  (implies (and (es-forma-clausal Fc)
                (SAT-davis-putnam Fc))
           (modelo-forma-clausal
             (MOD-bifurcacion-instancia Fc sigma) Fc)))
```

8.2.4 Procedimiento de Davis y Putnam en $\mathbb{K}$

Los procedimientos desarrollados en la sección previa pueden ser utilizados para decidir la satisfacibilidad de una forma clausal en $\mathbb{K}$. Los algoritmos son los mismos, puesto que no se basan en aspectos semánticos, pero no ocurre lo mismo con los conceptos y propiedades que aseguran su corrección y completitud. Estas propiedades se basan principalmente en que la eliminación de cláusulas con literales complementarios preserva el conjunto de modelos y que la reducción de un conjunto de cláusulas por un literal mantiene la satisfacibilidad:

Teorema 8.44 *Dada una forma clausal S, si S' es el resultado de eliminar de S todas las cláusulas con literales complementarios, entonces dada una $\mathbb{K}$-asignación σ, $\sigma \models_{\mathbb{K}} S$ si y sólo si $\sigma \models_{\mathbb{K}} S'$.*

La formalización de este teorema se basa, al igual que para la semántica clásica, en la función `elimina-clausulas-con-complementarios`, presentada en *208*. Se ha añadido una condición adicional que asegura que el valor de la forma clausal S no queda indeterminado.

237
```
(defthm elimina-clausulas-con-complementarios-preserva-modelos
  (implies
    (and (valor-forma-clausal-K Fc sigma)
         (es-forma-clausal Fc))
    (iff (modelo-forma-clausal-K
           sigma (elimina-clausulas-con-complementarios Fc))
         (modelo-forma-clausal-K sigma Fc)))
```

Al igual que en el caso clásico, esta regla de transformación sirve para caracterizar las formas clausales válidas.

Teorema 8.45 *Sea S una forma clausal entonces, S es $\mathbb{K}$-válida si y sólo si al eliminar de S todas las cláusulas con literales complementarios se obtiene la forma clausal vacía.*

La formalización es similar a la del caso clásico salvo que en uno de los eventos hay que incluir una condición adicional que asegura que el valor de la forma clausal original no queda indeterminado.

238

```
(defthm elimina-clausulas-con-complementarios-nil-fc-valida
  (implies
    (and (es-forma-clausal Fc)
         (equal (elimina-clausulas-con-complementarios Fc) nil)
         (valor-forma-clausal-K Fc sigma))
    (modelo-forma-clausal-K sigma Fc)))

(defthm fc-valida-elimina-clausulas-con-complementarios-nil
  (implies
    (and (es-forma-clausal Fc)
         (not (equal (elimina-clausulas-con-complementarios Fc)
                     nil)))
    (not (modelo-forma-clausal-K
           (contramodelo-clausula
             (car (elimina-clausulas-con-complementarios Fc)))
           Fc))))
```

Las operaciones de reducción de una forma clausal por un literal también mantienen la satisfacibilidad en la semántica de Kleene:

Teorema 8.46 *Sea S una forma clausal y L un literal, entonces:*

1. *Si $\sigma \models_{\mathbb{K}} S$ y $\sigma \models_{\mathbb{K}} L$ entonces $\sigma \models_{\mathbb{K}} S_L$.*

2. *Si $\sigma \models_{\mathbb{K}} S_L$ entonces $\sigma[L/\top] \models_{\mathbb{K}} S$.*

La formalización de estos resultados es similar a la presentada en *213* y *214* para la semántica clásica, salvo que se utilizan las funciones `modelo-K` y `modelo-forma-clausal-K`.

Los algoritmos $SAT_{\mathcal{B}}$ y $SAT_{\mathcal{D}}$ se definen en la semántica de Kleene de igual forma a como se ha hecho en la semántica clásica. Las propiedades de corrección y completitud de $SAT_{\mathcal{B}}$ se basan en las propiedades de la regla de bifurcación, que se siguen verificando en $\mathbb{K}$.

Teorema 8.47 *Dada una forma clausal S, S es $\mathbb{K}$-satisfacible si y sólo si $SAT_{\mathcal{B}}(S) = \mathbf{t}$.*

La formalización es similar, utilizando la función `modelo-forma-clausal-K` para comprobar que una $\mathbb{K}$-asignación es modelo de una forma clausal:

239
```
(defthm completitud-SAT-bifurcacion
  (implies (and (modelo-forma-clausal-K sigma Fc)
                (es-forma-clausal Fc))
           (SAT-bifurcacion Fc)))

(defthm correccion-SAT-bifurcacion
  (implies (and (es-forma-clausal Fc)
                (SAT-bifurcacion Fc))
           (modelo-forma-clausal-K
             (MOD-bifurcacion Fc sigma) Fc))
```

Donde el valor devuelto por la función `MOD-bifurcacion` es utilizado como $\mathbb{K}$-asignación en la que la forma clausal es cierta.

El desarrollo del algoritmo $SAT_{\mathcal{D}}$ y la demostración de sus propiedades se ha realizado siguiendo la misma técnica utilizada en el caso clásico.

8.3 Un STP exitoso basado en Davis-Putnam

El procedimiento de Davis y Putnam se puede interpretar como un caso particular del algoritmo $SAT_{\mathcal{G}}$ desarrollado en el capítulo 7. La idea del procedimiento es aplicar reglas de transformación a las formas clausales hasta obtener una forma clausal vacía. En este caso se puede afirmar que la forma clausal de partida es satisfacible. Como función de representación utilizamos la que proporciona el algoritmo de transformación a forma clausal $\mathcal{FC}$, con la ventaja adicional de que en la forma clausal obtenida no hay cláusulas con literales complementarios y se puede prescindir de la regla de eliminación de este tipo de cláusulas. Los objetos proposicionales son las formas clausales sobre las que se aplican las reglas, junto con una lista de los literales a los que ya se ha aplicado alguna regla. Las reglas de expansión son las derivadas de la regla de bifurcación junto con reglas que eliminan los objetos asociados a formas clausales en las que aparezca la cláusula vacía y una regla que termina el proceso de expansión para la forma clausal vacía. Las asignaciones distinguidas serán aquellas que hagan cierta una forma clausal y todos los literales cuya certeza se ha asumido al aplicar las reglas de transformación.

En esta sección desarrollamos un sistema de transformación proposicional basado en el procedimiento de Davis y Putnam. Primero describimos formalmente este sistema junto con una regla de computación, una función de representación, una función de medida y una función modelo adecuadas para demostrar que es exitoso. A continuación presentamos su formalización, relacionándola con la desarrollada en la sección anterior. Los eventos desarrollados en esta sección se encuentran en el libro `SAT-davisputnam.lisp`.

8.3.1 Descripción del STP exitoso $\mathcal{D}'$

Una característica fundamental del algoritmo $SAT_{\mathcal{G}}$ es que proporciona información suficiente para construir un modelo de una fórmula satisfacible. En el procedimiento de Davis y Putnam se ha definido una función que construye un modelo para formas clausales satisfacibles, a partir de los literales cuya certeza se ha asumido al aplicar las reglas de transformación. Esta información tiene que ser almacenada junto con el objeto proposicional, para poder construir un modelo de la fórmula inicial a partir del resultado devuelto por la instancia del algoritmo $SAT_{\mathcal{G}}$. De esta forma, los objetos proposicionales que consideramos son pares formados por la forma clausal que queda por transformar y la lista de literales cuya certeza se ha asumido al aplicar las reglas de transformación.

Definición 8.48 *$\mathcal{D}' = \langle \mathcal{O}_{\mathcal{D}'}, \mathcal{R}_{\mathcal{D}'}, \mathcal{V}_{\mathcal{D}'} \rangle$ es el STP basado en el procedimiento de Davis y Putnam en el que $\mathcal{O}_{\mathcal{D}'}$ es el conjunto de pares $\langle S, M \rangle$ donde S es una forma clausal y M es una lista de literales en la que no aparecen literales complementarios y tal que ningún elemento de M ni su complementario aparecen en S, $\mathcal{V}_{\mathcal{D}'}$ es el conjunto de pares $(\langle S, M \rangle, \sigma)$ tales que σ es modelo de la forma clausal S y de todos los elementos de M, y $\mathcal{R}_{\mathcal{D}'}$ es el conjunto de reglas de expansión representado por el siguiente conjunto de esquemas de regla:*

$\mathcal{R}_{\mathcal{D}'1}$:	$\langle S, M \rangle \rightsquigarrow_{\mathcal{D}'} \langle\rangle$ *si* $\langle\rangle \in S$
$\mathcal{R}_{\mathcal{D}'2}$:	$\langle S, \langle L_1, \ldots, L_n \rangle\rangle \rightsquigarrow_{\mathcal{D}'} \langle\langle S_L, \langle L, L_1, \ldots, L_n \rangle\rangle\rangle$ *si* $\langle L \rangle \in S$
$\mathcal{R}_{\mathcal{D}'3}$:	$\langle S, \langle L_1, \ldots, L_n \rangle\rangle \rightsquigarrow_{\mathcal{D}'} \langle\langle S_L, \langle L, L_1, \ldots, L_n \rangle\rangle, \langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n \rangle\rangle\rangle$ *donde L es un literal que aparece en algún elemento de S*
$\mathcal{R}_{\mathcal{D}'4}$:	$\langle\langle\rangle, M \rangle \rightsquigarrow_{\mathcal{D}'} \mathbf{t}$

donde S es una forma clausal, $\langle S, M \rangle$ y $\langle S, \langle L_1, \ldots, L_N \rangle\rangle$ son elementos de $\mathcal{O}_{\mathcal{D}'}$.

Las reglas del tipo $\mathcal{R}_{\mathcal{D}'1}$ se encargan de eliminar los objetos asociados a formas clausales en las que aparece la cláusula vacía. Las reglas del tipo $\mathcal{R}_{\mathcal{D}'2}$ se corresponden con la regla de eliminación de cláusulas unitarias y las reglas del tipo $\mathcal{R}_{\mathcal{D}'3}$, con la regla de bifurcación. Las reglas del tipo $\mathcal{R}_{\mathcal{D}'4}$ detectan el final del proceso.

La regla de eliminación de literales puros no ha sido añadida pues lleva a una regla de computación que no conserva las asignaciones distinguidas tal y como se exige en la propiedad $\mathcal{P}_3$:

$$\langle S, M \rangle \in \mathcal{O} \wedge r_{\mathcal{D}'}(\langle S, M \rangle) \neq \mathbf{t} \Longrightarrow \\ \Longrightarrow (\sigma \models_{\mathcal{D}'} \langle S, M \rangle \Longleftrightarrow \exists \langle S_i, M_i \rangle \in r_{\mathcal{D}'}(\langle S, M \rangle), \sigma \models_{\mathcal{D}'} \langle S_i, M_i \rangle)$$

Si consideramos la regla de eliminación de literales puros, entonces a partir del objeto proposicional $\langle\langle\langle p, q \rangle, \langle \neg q, r \rangle\rangle, \langle\rangle\rangle$, se obtendrá el objeto $\langle\langle\langle \neg q, r \rangle\rangle, \langle p \rangle\rangle$. En esta situación, la asignación que hace p falsa y, q y r ciertos, es una asignación distinguida del primer objeto proposicional, pero no del segundo.

Utilizando el STP anterior, junto con una función de representación $i_{\mathcal{D}'}$ y una regla de computación $r_{\mathcal{D}'}$ adecuadas, obtenemos un procedimiento de decisión de satisfacibilidad basado en bifurcación y eliminación de cláusulas unitarias, como un caso particular del algoritmo $SAT_{\mathcal{G}}$. Además, para garantizar las propiedades de terminación, corrección y completitud, tenemos que proporcionar una función de medida $\mu_{\mathcal{D}'}$ y una función modelo $\sigma_{\mathcal{D}'}$, de forma que $\mathcal{D}'$ sea exitoso con respecto a $r_{\mathcal{D}'}$, $i_{\mathcal{D}'}$, $\mu_{\mathcal{D}'}$ y $\sigma_{\mathcal{D}'}$.

Definición 8.49 *Para el STP $\mathcal{D}'$ definimos:*

1. *La función de representación $i_{\mathcal{D}'}$ tal que para toda $F \in \mathbb{P}(\Sigma)$, $i_{\mathcal{D}'}(F) = \langle \mathcal{FC}(F), \langle\rangle \rangle$.*

2. *La regla de computación $r_{\mathcal{D}'}$ que actúa de la siguiente forma. Dado un objeto proposicional $\langle S, M\rangle$, si S contiene la cláusula vacía, se le aplica una regla del tipo $\mathcal{R}_{\mathcal{D}'1}$. En otro caso se selecciona un literal de una cláusula unitaria en S, si es que existe. En este caso se le aplica una regla del tipo $\mathcal{R}_{\mathcal{D}'2}$. Si en S no hay cláusulas unitarias, se selecciona un literal cualquiera que aparezca en algún elemento de S, si es que existe, y se le aplica una regla del tipo $\mathcal{R}_{\mathcal{D}'3}$. Si S es la forma clausal vacía, se aplica una regla del tipo $\mathcal{R}_{\mathcal{D}'4}$.*

3. *La función de medida $\mu_{\mathcal{D}'}$ tal que, para todo $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}'}$, $\mu_{\mathcal{D}'}(\langle S, M\rangle)$ es la longitud de la forma clausal S, $l(S)$.*

4. *La función modelo $\sigma_{\mathcal{D}'}$ tal que, para todo $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}'}$, $\mu_{\mathcal{D}'}(\langle S, M\rangle) \models p$ si y sólo si p es un literal positivo que aparece en M.*

Teorema 8.50 *El STP $\mathcal{D}'$ es exitoso con respecto a la regla de computación $r_{\mathcal{D}'}$, la función de representación $i_{\mathcal{D}'}$, la función de medida $\mu_{\mathcal{D}'}$ y la función modelo $\sigma_{\mathcal{D}'}$.*

Demostración:

Para demostrar el teorema tendremos que probar las propiedades $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ y $\mathcal{P}_4$ para las funciones $r_{\mathcal{D}'}$, $i_{\mathcal{D}'}$, $\mu_{\mathcal{D}'}$ y $\sigma_{\mathcal{D}'}$.

$\mathcal{P}_1$: Dado $\langle S_i, M_i\rangle \in r_{\mathcal{D}'}(\langle S, M\rangle)$, entonces existe un literal L tal que $S_i = S_L$ o $S_i = S_{\overline{L}}$, luego por el teorema 8.39 se verifica que $\mu_{\mathcal{D}'}(\langle S_i, M_i\rangle) = l(S_i) < l(S) = \mu_{\mathcal{D}'}(\langle S, M\rangle)$.

$\mathcal{P}_2$: Dada la fórmula F, $\sigma \models_{\mathcal{D}'} \langle \mathcal{FC}(F), \langle\rangle\rangle$ si y sólo si $\sigma \models \mathcal{FC}(F)$ y, por el teorema 8.20, esto ocurre si y sólo si $\sigma \models F$.

$\mathcal{P}_3$: Dado $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}'}$ tal que $r_{\mathcal{D}'}(\langle S, M\rangle) \neq \mathbf{t}$, entonces se tienen tres situaciones distintas:

1. $r_{\mathcal{D}'}(\langle S, M\rangle) = \langle\rangle$. En este caso la cláusula vacía está en S y, por el teorema 8.12, S no tiene modelos. Por otro lado, no existe ningún elemento en $r_{\mathcal{D}'}(\langle S, M\rangle)$. Por tanto se tiene la equivalencia $\sigma \models_{\mathcal{D}'} \langle S, M\rangle \iff \exists\langle S_i, M_i\rangle \in r_{\mathcal{D}'}(\langle S, M\rangle), \sigma \models_{\mathcal{D}'} \langle S_i, M_i\rangle$
2. $r_{\mathcal{D}'}(\langle S, \langle L_1, \ldots, L_n\rangle\rangle) = \langle\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle\rangle$ para algún literal L tal que $\langle L\rangle \in S$:
   (a) Sea σ una asignación distinguida de $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$, es decir, $\sigma \models S$ y $\sigma \models L_i$ para todo i. Como $\langle L\rangle \in S$ y $\sigma \models S$ entonces $\sigma \models L$ y, por el teorema 8.29-1, $\sigma \models S_L$. Luego σ es una asignación distinguida de $\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle$.
   (b) Sea σ una asignación distinguida de $\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle$, es decir, $\sigma \models S_L$, $\sigma \models L$ y $\sigma \models L_i$ para todo i. Por el teorema 8.29-2, $\sigma[L/\top] \models S$. Como $\sigma \models L$ se tiene $\sigma = \sigma[L/\top]$ y por tanto $\sigma \models S$. Luego σ es una asignación distinguida de $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$.
3. $r_{\mathcal{D}'}(\langle S, \langle L_1, \ldots, L_n\rangle\rangle) = \langle\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle, \langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n\rangle\rangle\rangle$ para algún literal L que aparece en un elemento de S.
   (a) Sea σ una asignación distinguida de $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$, es decir, $\sigma \models S$ y $\sigma \models L_i$ para todo i. Se tiene que $\sigma \models L$ o $\sigma \models \overline{L}$. Si $\sigma \models L$ la situación es igual que para el caso 2 y por tanto σ es una asignación distinguida de $\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle$. Si $\sigma \models \overline{L}$, por el teorema 8.29-1, $\sigma \models S_{\overline{L}}$. Luego σ es una asignación distinguida de $\langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n\rangle\rangle$.
   (b) Si σ es una asignación distinguida de $\langle S_L, \langle L, L_1, \ldots, L_n\rangle\rangle$, entonces al igual que en el caso 2, se puede concluir que σ es una asignación distinguida de $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$.
   Si σ una asignación distinguida de $\langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n\rangle\rangle$, es decir, $\sigma \models S_{\overline{L}}$, $\sigma \models \overline{L}$ y $\sigma \models L_i$ para todo i. Por el teorema 8.29-2, $\sigma[\overline{L}/\top] \models S$. Como $\sigma \models \overline{L}$ se tiene $\sigma = \sigma[\overline{L}/\top]$ y por tanto $\sigma \models S$. Luego σ es una asignación distinguida de $\langle S, \langle L_1, \ldots, L_n\rangle\rangle$.

$\mathcal{P}_4$: Sea $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}'}$ tal que $r_{\mathcal{D}'}(\langle S, M\rangle) = \mathbf{t}$, entonces S es la forma clausal vacía, que es cierta en cualquier asignación. Por tanto $\sigma_{\mathcal{D}'}(\langle S, M\rangle) \models S$. Por otro lado, la construcción de $\sigma_{\mathcal{D}'}(\langle S, M\rangle)$ y el hecho de que M no contiene literales complementarios nos asegura que para cualquier literal L en M, $\sigma_{\mathcal{D}'}(\langle S, M\rangle) \models L$. Por tanto, $\sigma_{\mathcal{D}'}(\langle S, M\rangle)$ es una asignación distinguida de $\langle S, M\rangle$.

□

8.3.2 Formalización de $\mathcal{D}'$

Para formalizar el conjunto de objetos proposicionales en los que se basa $\mathcal{D}'$, tenemos que definir una función que caracterice dichos objetos. El conjunto $\mathcal{O}_{\mathcal{D}'}$

está compuesto por pares de la forma $\langle S, M\rangle$, tales que S es una forma clausal y M es una lista de literales en la que no aparecen literales complementarios y tal que para todo L en M, ni L ni $\overline{L}$ aparecen en ningún elemento de S.

La función que caracteriza el conjunto $\mathcal{O}_{\mathcal{D}'}$ es `d&p-objeto`. En su definición hemos utilizado la función `es-forma-clausal` para comprobar que S es una forma clausal, la función `tiene-literales-complementarios` para comprobar que M no tiene literales complementarios y, para comprobar que dado un elemento L de M, ni L ni $\overline{L}$ están en S, definimos la función `lista-literales-no-esta-en-fc`:

240
```
(defun d&p-objeto (S-M)
  (and (es-forma-clausal (car S-M))
       (es-clausula (cdr S-M))
       (not (tiene-literales-complementarios (cdr S-M)))
       (lista-literales-no-esta-en-fc (cdr S-M) (car S-M))))

(defun lista-literales-no-esta-en-fc (lista-L Fc)
  (cond
    ((endp lista-L) t)
    (t (and (not (literal-en-forma-clausal (car lista-L) Fc))
            (not (literal-en-forma-clausal
                  (complementario (car lista-L)) Fc))
            (lista-literales-no-esta-en-fc (cdr lista-L) Fc)))))
```

Las asignaciones distinguidas de un par $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}'}$ son aquellas que son modelo de S y hacen cierto todo literal L en M. La función `modelo-forma-clausal` sirve para comprobar lo primero y para lo segundo definimos la función `modelo-lista-literales`. La función `d&p-asignacion-distinguida` comprueba que una asignación es distinguida para un objeto proposicional:

241
```
(defun d&p-asignacion-distinguida (sigma S-M)
  (and (modelo-forma-clausal sigma (car S-M))
       (modelo-lista-literales sigma (cdr S-M))))

(defun modelo-lista-literales (sigma lista-L)
  (cond ((endp lista-L) t)
        ((modelo sigma (car lista-L))
         (modelo-lista-literales sigma (cdr lista-L)))
        (t nil)))
```

La regla de computación $r_{\mathcal{D}'}$ no está completamente determinada puesto que, si tenemos que aplicar reglas del tipo $\mathcal{R}_{\mathcal{D}'2}$ o $\mathcal{R}_{\mathcal{D}'3}$, no se indica un criterio para la elección del literal L. En el primer caso escogeremos el literal de la primera

cláusula unitaria de S, en el segundo caso se considera una función `seleccion` que escoge un literal de alguna de las cláusulas de S, si es que existe. La existencia de esta función se asume en un encapsulado ACL2, caracterizándola por las propiedades formalizadas en *220*.

La función que implementa la regla de computación es la siguiente:

242
```
(defun d&p-regla-computacion (S-M)
  (let ((Fc (car S-M))
        (lista-L (cdr S-M)))
   (if (contiene-clausula-vacia Fc)
       nil
       (let ((C (busca-clausula-unitaria Fc)))
        (if C
            (list (cons (reduce-forma-clausal-literal (car C) Fc)
                        (cons (car C) lista-L)))
            (let ((L (seleccion Fc)))
             (if L
                (list (cons (reduce-forma-clausal-literal L Fc)
                            (cons L lista-L))
                      (cons (reduce-forma-clausal-literal-c L Fc)
                            (cons (complementario L) lista-L)))
                t)))))))
```

En esta definición hemos utilizado la función `contiene-clausula-vacia`, definida en *190*, que comprueba si una forma clausal contiene la cláusula vacía, así como `reduce-forma-clausal-literal` y `reduce-forma-clausal-literal-C`, definidas en *210*, que, respectivamente, reducen una forma clausal por un literal `L` y por su complementario. La función `busca-clausula-unitaria` devuelve la primera cláusula unitaria de una forma clausal.

Obsérvese que el resultado devuelto por esta función es **t** o una lista de objetos proposicionales. La formalización de esta propiedad es la siguiente:

243
```
(defthm d&p-objeto-d&p-regla-computacion
  (implies (and (d&p-objeto S-M-1)
                (member-equal S-M-2
                              (d&p-regla-computacion S-M-1)))
           (d&p-objeto S-M-2)))
```

La función de representación $i_{\mathcal{D}'}$ es bastante simple, consiste en construir un par cuyo primer elemento sea el resultado de evaluar el algoritmo $\mathcal{FC}$ sobre una fórmula F y cuyo segundo elemento es la lista vacía. Esta construcción la realiza la función `d&p-representacion`. El valor que devuelve es un elemento de $\mathcal{O}_{\mathcal{D}'}$.

244
```
(defun d&p-representacion (F)
  (list (forma-clausal F)))

(defthm d&p-representacion-es-d&p-objeto
  (implies (es-proposicional F)
           (d&p-objeto (d&p-representacion F))))
```

La función de medida $\mu_{\mathcal{D}'}$ de un objeto $\langle S, M\rangle$, es la longitud de la forma clausal S. La función `medida-forma-clausal`, definida en *223*, calcula dicho valor. La función que implementa la medida $\mu_{\mathcal{D}'}$ es `d&p-medida`. El resultado que devuelve es un ordinal.

245
```
(defun d&p-medida (S-M)
  (medida-forma-clausal (car S-M)))

(defthm e0-ordinalp-d&p-media
  (e0-ordinalp (d&p-medida secuente)))
```

La función modelo $\sigma_{\mathcal{D}'}$ aplicada a un objeto $\langle S, M\rangle$, construye una asignación en la que un símbolo p es cierto si y sólo si es un literal positivo que aparece en M. De esta forma, todos los literales positivos de M son ciertos y, como M no contiene literales complementarios, todos los literales negativos también lo son. Esta asignación se construye extendiendo la asignación parcial vacía, de forma que todos los elementos de M sean ciertos. Para extender una asignación haciendo un literal cierto utilizamos la función `asume-literal`, presentada en *183*.

246
```
(defun d&p-modelo (S-M)
  (modelo-literales (cdr S-M)))

(defun modelo-literales (lista-L)
  (cond ((endp lista-L) nil)
        (t (asume-literal (car lista-L)
                          (modelo-literales (cdr lista-L))))))
```

Veamos a continuación la formalización de las propiedades que garantizan que $\mathcal{D}'$ es un STP exitoso. La propiedad $\mathcal{P}_1$ está formalizada por el evento `P1-d&p` y se demuestra fácilmente a partir de los eventos presentados en *224*.

247
```
(defthm P1-d&p
  (implies (member-equal S-M-2 (d&p-regla-computacion S-M-1))
           (e0-ord-< (d&p-medida S-M-2)
                     (d&p-medida S-M-1))))
```

La prueba de la propiedad $\mathcal{P}_2$ es inmediata a partir de las definiciones de asignación distinguida y la equivalencia entre el valor de una fórmula y el de la forma clausal obtenida con el algoritmo $\mathcal{FC}$, formalizada en *202*. Su formalización es la siguiente:

248
```
(defthm P2-d&p
  (implies (es-proposicional F)
           (iff (d&p-asignacion-distinguida
                  sigma (d&p-representacion F))
                (modelo sigma F))))
```

Para formalizar la propiedad $\mathcal{P}_3$ necesitamos definir una función que compruebe si una asignación es distinguida para algún objeto de una lista. Esta función es `d&p-asignacion-distinguida-lista`. La formalización de esta función y de la propiedad $\mathcal{P}_3$ es la siguiente:

249
```
(defthm P3-d&p
  (implies (and (d&p-objeto S-M)
                (not (equal (d&p-regla-computacion S-M) t)))
           (iff (d&p-asignacion-distinguida-lista
                  sigma (d&p-regla-computacion S-M))
                (d&p-asignacion-distinguida sigma S-M))))

(defun d&p-asignacion-distinguida-lista (sigma d&p-obj-lst)
  (cond ((endp d&p-obj-lst) nil)
        (t (or (d&p-asignacion-distinguida
                 sigma (car d&p-obj-lst))
               (d&p-asignacion-distinguida-lista
                 sigma (cdr d&p-obj-lst))))))
```

La prueba del evento `P3-d&p` se obtiene a partir de propiedades de las funciones de reducción similares a las mostradas en *213* y *214*.

Finalmente, la propiedad $\mathcal{P}_4$ se demuestra fácilmente a partir de la definición de `d&p-modelo` y el hecho de que, para todo objeto $\langle S, M\rangle$, M no contiene literales complementarios.

250
```
(defthm P4-d&p
  (implies (and (d&p-objeto S-M)
                (equal (d&p-regla-computacion S-M) t))
           (d&p-asignacion-distinguida (d&p-modelo S-M) S-M)))
```

El siguiente evento realiza una instancia de la teoría genérica `*sat-generico*` utilizando las funciones que describen el STP exitoso $\mathcal{D}'$. Para seleccionar el objeto a procesar en cada paso hemos utilizado la función `car`, que devuelve el primer elemento de una lista.

251

```
(definstancia-*sat-generico*
  ((gen-objeto                  d&p-objeto)
   (gen-representacion          d&p-representacion)
   (gen-asignacion-distinguida  d&p-asignacion-distinguida)
   (gen-asignacion-distinguida-lista
                                d&p-asignacion-distinguida-lista)
   (gen-regla-computacion       d&p-regla-computacion)
   (gen-seleccion               car)
   (gen-medida                  d&p-medida)
   (gen-modelo                  d&p-modelo))
  "-secuentes")
```

Una vez evaluado este evento, se obtiene automáticamente la función que implementa el algoritmo $SAT_{\mathcal{D}'}$, como una instancia de $SAT_{\mathcal{G}}$. También se obtienen versiones basadas en el procedimiento de transformación de formas clausales, del algoritmo de decisión de validez $DAT_{\mathcal{G}}$ y del procedimiento de construcción de modelos $MOD_{\mathcal{G}}$. Estas funciones son las siguientes:

```
(DEFUN SAT-GENERICO-D&P (F)
  (OBJ-SAT-GENERICO-D&P (LIST (D&P-REPRESENTACION F))))

(DEFUN MOD-GENERICO-D&P (F)
  (D&P-MODELO (FIRST (SAT-GENERICO-D&P F))))

(DEFUN DAT-GENERICO-D&P (F)
  (NOT (SAT-GENERICO-D&P (NEGACION F))))

(DEFUN OBJ-SAT-GENERICO-D&P (LISTA-O)
  (IF (ENDP LISTA-O)
      NIL
      (LET* ((O (CAR LISTA-O))
             (RESTO (ELIMINA-UNA (CAR LISTA-O) LISTA-O))
             (EXPANSION (D&P-REGLA-COMPUTACION O)))
        (COND ((EQUAL EXPANSION T) (LIST O))
              (T (OBJ-SAT-GENERICO-D&P
                   (APPEND EXPANSION RESTO)))))))
```

Además de generar automáticamente estas funciones, al instanciar la teoría genérica `*sat-generico*`, también se obtienen de forma automática los eventos que establecen sus propiedades corrección y completitud.

```
(DEFTHM CORRECCION-SAT-GENERICO-D&P
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (SAT-GENERICO-D&P F))
           (MODELO (MOD-GENERICO-D&P F) F)))

(DEFTHM COMPLETITUD-SAT-GENERICO-D&P
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (MODELO SIGMA F))
           (SAT-GENERICO-D&P F)))

(DEFTHM CORRECCION-DAT-GENERICO-D&P
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (DAT-GENERICO-D&P F))
           (MODELO SIGMA F)))

(DEFTHM COMPLETITUD-DAT-GENERICO-D&P
  (IMPLIES (AND (ES-PROPOSICIONAL F)
                (NOT (DAT-GENERICO-D&P F)))
           (NOT (MODELO (MOD-GENERICO-D&P (NEGACION F))
                        F))))
```

8.4 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, presentando datos sobre su evaluación en algunas fórmulas representando el problema de las N reinas [62] y algunos de los problemas de la colección DIMACS [82].

Una vez certificados los libros, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `8-davisputnam`:

```
../calculos-proposicionales/8-davisputnam> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/8-davisputnam".

ACL2 !>
```

A continación incluimos el libro que contiene la teoría desarrollada en este capítulo. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "davis-putnam")
```

Para poder utilizar los procedimientos de decisión desarrollados en este capítulo, tenemos que proporcionar una definición concreta de la función que selecciona un literal de una forma clausal. El símbolo asociado a esta función ya existe en el sistema por lo que tendremos que redefinirla. Las propiedades de los procedimientos de decisión se mantienen siempre que la nueva definición de la función `seleccion` tenga las propiedades presentadas en *220*.

```
ACL2 !>(set-ld-redefinition-action '(:warn . :overwrite) state)

ACL2 !>(defun seleccion (Fc)
         (cond ((endp Fc) nil)
               ((endp (car Fc))
                (seleccion (cdr Fc)))
               (t (car (car Fc)))))
```

En la tabla 8.1 se muestra información sobre el tiempo de comprobación de la satisfacibilidad de las fórmulas correspondientes al problema de las N reinas, para algunos valores de N. Estos tiempos incluyen la transformación a forma clausal y la evaluación del procedimiento de decisión `SAT-davis-putnam`. En la sección 4.3 se indica como se carga en ACL2 el fichero en el que se define esta familia.

Tamaño del tablero	2	3	4	5	6	7	8
Tiempo	0.010	0.020	0.090	0.290	0.690	2.140	3.220

Tabla 8.1: Tiempos de evaluación para el problema de las N reinas

La familia de ejemplos para la que presentamos los datos a continuación está tomada de la colección DIMACS, que se encuentra disponible en `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`. Esta colección consiste en distintas familias de problemas expresados en forma clausal, en las que los literales positivos se representan con números enteros positivos y los negativos con números enteros negativos. Para poder evaluar los procedimientos de decisión de satisfacibilidad para conjuntos de cláusulas sobre estos ejemplos, hemos modificado la formalización de la sintaxis de la lógica proposicional.

Problema	Satisfacible	Variables	Clausulas	Tiempo
aim-50-1_6-no-1	No	50	80	6.530
aim-50-1_6-no-2	No	50	80	2.140
aim-50-1_6-no-3	No	50	80	3.730
aim-50-1_6-no-4	No	50	80	1.230
aim-50-1_6-yes1-1	Sí	50	80	0.550
aim-50-1_6-yes1-2	Sí	50	80	0.280
aim-50-1_6-yes1-3	Sí	50	80	0.180
aim-50-1_6-yes1-4	Sí	50	80	0.120
aim-50-2_0-no-1	No	50	100	13.260
aim-50-2_0-no-2	No	50	100	3.030
aim-50-2_0-no-3	No	50	100	2.650
aim-50-2_0-no-4	No	50	100	1.730
aim-50-2_0-yes1-1	Sí	50	100	0.710
aim-50-2_0-yes1-2	Sí	50	100	0.460
aim-50-2_0-yes1-3	Sí	50	100	0.140
aim-50-2_0-yes1-4	Sí	50	100	0.360
aim-50-3_4-yes1-1	Sí	50	170	1.280
aim-50-3_4-yes1-2	Sí	50	170	7.800
aim-50-3_4-yes1-3	Sí	50	170	3.210
aim-50-3_4-yes1-4	Sí	50	170	1.900
aim-50-6_0-yes1-1	Sí	50	300	1.650
aim-50-6_0-yes1-2	Sí	50	300	0.630
aim-50-6_0-yes1-3	Sí	50	300	1.190
aim-50-6_0-yes1-4	Sí	50	300	1.170

Tabla 8.2: Datos sobre la evaluación de la familia AIM-50

Destacamos el hecho de que sólo ha sido necesario modificar ligeramente el archivo `sintaxis.lisp`. La teoría desarrollada en el resto de los archivos se mantiene exactamente igual. En el directorio `calculos-proposicionales/0-ejemplos` se encuentra el subdirectorio `dimacs`, donde están los libros necesarios para evaluar estos ejemplos. Para certificar estos libros y evaluar los ejemplos, basta con ejecutar la orden `make` en dicho directorio:

```
../calculos-proposicionales/0-ejemplos/dimacs> make
```

En la tabla 8.2 se muestran los tiempos de evaluación obtenidos para la familia AIM-50. En esta tabla también indicamos el número de variables y el número de cláusulas de cada problema y si se trata de un conjunto de cláusulas satisfacible o no. En el directorio `dimacs/0-ejemplos` hay otras familias tomadas de la colección DIMACS.

Sumario

En este capítulo:

- Se han presentado los conceptos de cláusula y forma clausal, demostrando resultados que caracterizan las cláusulas válidas y las formas clausales insatisfacibles. La formalización correspondiente se ha realizado tanto en la semántica clásica como en la de Kleene.

- Se ha desarrollado un algoritmo para transformar una fórmula proposicional en una forma clausal lógicamente equivalente a ella. Se ha demostrado que este algoritmo termina y que calcula una forma clausal lógicamente equivalente a la fórmula inicial. Se ha probado que la validez de una forma clausal queda asegurada cuando este algoritmo devuelve la forma clausal vacía. Se ha formalizado el algoritmo y sus propiedades tanto en la semántica clásica como en la de Kleene.

- Se ha analizado el conjunto de reglas de transformación en las que se basa el procedimiento de Davis y Putnam y se ha demostrado que estas reglas preservan la satisfacibilidad de las formas clausales. Se han construido dos procedimientos de decisión de satisfacibilidad para formas clausales basados en estas reglas, presentando el segundo de ellos como un refinamiento del primero. Se han probado las propiedades de corrección y completitud de estos algoritmos. El desarrollo se ha realizado tanto para la semántica clásica como para la de Kleene.

- Se ha definido un sistema de transformación proposicional basado en el procedimiento de Davis y Putnam. Se han demostrado las propiedades que garantizan que dicho sistema es exitoso. Una vez formalizados los elementos y propiedades que describen este STP, se ha mostrado como se obtienen automáticamente algoritmos verificados de decisión de satisfacibilidad y validez, como una instancia de la teoría asociada al marco genérico.

Capítulo 9

Resolución

El cálculo de resolución fué inventado por Robinson en 1965 [65]. Desde entonces el procedimiento básico de resolución ha sido refinado dando lugar a distintas variantes. Quizá la más conocida sea la SLD-resolución [46], en la que se basa la programación lógica. Estas variantes también se han utilizado en el desarrollo de sistemas de deducción, entre los que destaca OTTER [56] y su variante EQP, utilizado con éxito en la obtención de una demostración de la conjetura de Robbins [55].

En este capítulo presentamos el cálculo de resolución desde un punto de vista genérico: el proceso básico de resolución, el cálculo de una resolvente, se hace depender de una condición predeterminada. La consideración de distintas condiciones da lugar a distintas variantes de resolución: positiva, negativa, semántica, con conjunto soporte, ... Un desarrollo más amplio acerca de los procedimientos de decisión basados en resolución y sus variantes puede encontrarse en [4].

El capítulo está formado por tres secciones. En la primera se presenta el concepto clásico de resolvente binaria y el de resolvente condicionada. En este último el cálculo de las resolventes se hace depender de una condición predeterminada a la que llamaremos *condición de resolución*. Varios refinamientos de resolución se pueden obtener considerando distintas condiciones de resolución. Se presentan algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución condicionada y se demuestran sus propiedades de terminación y corrección. En esta sección se establecen los eventos relacionados con la formalización de estos algoritmos y la demostración de sus propiedades.

La segunda sección está dedicada a la prueba de Bezem de la completitud de la resolución condicionada presentada en [8]. Esta prueba ha sido formalizada en ACL2 y se utiliza para demostrar la completitud del algoritmo de decisión de insatisfacibilidad cuando la condición de resolución verifica cierta propiedad con respecto a una asignación predeterminada. También se presentan y formalizan conceptos y propiedades relativas a conjuntos de representantes de colecciones de conjuntos, necesarios en el desarrollo de la prueba.

En la tercera sección se define una teoría genérica que facilita la reutilización

de los algoritmos desarrollados en la primera sección, preservando sus propiedades de corrección y completitud. Se presentan ejemplos de instancias de esta teoría genérica para la resolución binaria, positiva, negativa, semántica y con conjunto soporte.

9.1 Resolución condicionada

Los procedimientos basados en el principio de resolución consisten en, a partir de un conjunto de cláusulas, generar todas las posibles cláusulas que se pueden obtener aplicando una regla de inferencia básica. Esta regla consiste en, bajo ciertas condiciones, obtener una nueva cláusula, llamada resolvente, a partir de de otras dos, llamadas cláusulas padre, de forma que la satisfacibilidad de las cláusulas padre asegura la satisfacibilidad de la resolvente. Así, el conjunto de cláusulas inicial se va incrementando con las nuevas cláusulas generadas hasta que llega a ser saturado por resolución, es decir, no se puede obtener ninguna resolvente nueva. Si, a lo largo del proceso, se obtiene la cláusula vacía, entonces se puede afirmar que el conjunto original es insatisfacible. Si, por el contrario, se llega a obtener un conjunto saturado por resolución en el que no está la cláusula vacía, entonces se puede afirmar que el conjunto original es satisfacible. De forma intuitiva, el procedimiento de resolución genera todas las cláusulas que son consecuencia lógica de un conjunto de cláusulas de partida. Si se obtiene la cláusula vacía, que es insatisfacible, entonces el conjunto de cláusulas inicial no puede ser satisfacible.

A lo largo de este capítulo, las cláusulas se consideran conjuntos en lugar de listas. Esto se hace para evitar la generación de cláusulas con los mismos literales, lo que haría innecesariamente largo el proceso de saturación. De la misma forma, hablamos de conjuntos de cláusulas en lugar de formas clausales, es decir, listas de cláusulas. Esto no supone ningún cambio en la representación de las cláusulas, que siguen siendo listas de literales, ni en la de las formas clausales, que siguen siendo listas de cláusulas. El cambio aparece cuando se realizan operaciones sobre cláusulas, como comparación o unión, en estos casos tenemos que tener en cuenta que los elementos pueden aparecer en cualquier orden. Las funciones que actúan sobre conjuntos serán comentadas conforme vayan apareciendo en el texto.

9.1.1 Conjuntos saturados por resolución condicionada

La operación básica en la que se basan los procedimientos de resolución, es la que calcula una nueva cláusula, llamada resolvente, a partir de otras dos, llamadas cláusulas padres, de forma que la satisfacibilidad de las cláusulas padre asegura la satisfacibilidad de la resolvente. Las diferencias entre estos procedimientos consisten en exigir distintas propiedades a las cláusulas padre para poder obtener la resolvente. Por tanto, la situación más general es aquella en la que no se

exige ninguna propiedad a las cláusulas padre. En este caso el procedimiento de resolución se denomina resolución binaria.

Definición 9.1 *Sean C_1 y C_2 dos cláusulas y L un literal tal que $L \in C_1$ y $\overline{L} \in C_2$. La resolvente binaria, o simplemente resolvente, de C_1 y C_2 con respecto al literal L es la cláusula $(C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$.*

En la formalización del concepto de resolvente hemos prescindido de las comprobaciones $L \in C_1$ y $\overline{L} \in C_2$. Esto es debido a que está función únicamente será evaluada para determinados literales tomados de C_1 cuyo complementario aparece en C_2. De esta forma, se evita una sobrecarga de comprobaciones que hace que el código final sea un poco más eficiente. Por otro lado, aún prescindiendo de las comprobaciones $L \in C_1$ y $\overline{L} \in C_2$, la satisfacibilidad del resultado queda garantizada si las cláusulas padre son satisfacibles.

252
```
(defun resolvente (L C-1 C-2)
  (union-equal (elimina-elemento L C-1)
               (elimina-elemento (complementario L) C-2)))
```

Donde la función `elimina-elemento` elimina todas las ocurrencias de un elemento de una cláusula y la función `union-equal` agrupa todos los elementos de dos listas eliminando los repetidos.

Una característica de la función anterior es que, si los argumentos son cláusulas, entonces el resultado sigue siendo una cláusula. Esto se debe a que las funciones `union-equal` y `elimina-elemento` tienen la misma propiedad:

253
```
(defthm resolvente-es-clausula
  (implies (and (es-clausula C-1)
                (es-clausula C-2))
           (es-clausula (resolvente L C-1 C-2))))
```

Teorema 9.2 *Si C es la resolvente de las cláusulas C_1 y C_2 en el literal L y σ una asignación tal que $\sigma \models C_1$ y $\sigma \models C_2$, entonces $\sigma \models C$.*

Demostración:

En la situación descrita en el enunciado del teorema, se tienen dos posibilidades, $\sigma \models L$ o $\sigma \models \overline{L}$ o, de otra forma, $\sigma \not\models \overline{L}$ o $\sigma \not\models L$. Si $\sigma \not\models \overline{L}$ entonces, como $\sigma \models C_2$, se tiene $\sigma \models (C_2 - \{\overline{L}\})$ y por tanto $\sigma \models C$. Si $\sigma \not\models L$ entonces, como $\sigma \models C_1$, se tiene $\sigma \models (C_1 - \{L\})$ y por tanto $\sigma \models C$.

□

Obsérvese que en la demostración del resultado no se ha utilizado en ningún momento el hecho de que el literal L pertenezca a C_1 y el $\overline{L}$ a C_2. La formalización

del teorema refleja esta situación puesto que en las hipótesis no se exige que el literal o su complementerio pertenezcan a las cláusulas padre ni la función resolvente lo comprueba:

254
```
(defthm correccion-resolvente
  (implies (and (es-clausula C-1)
                (es-clausula C-2)
                (es-literal L)
                (modelo-clausula sigma C-1)
                (modelo-clausula sigma C-2))
           (modelo-clausula sigma (resolvente L C-1 C-2)))))
```

Como se ha comentado antes, la función `resolvente` sólo se evaluará para literales tomados de la primera cláusula padre cuyo complementario aparece en la segunda. Para obtener uno de estos literales consideramos la función `literal-resolucion-clausulas`:

255
```
(defun literal-resolucion-clausulas (C-1 C-2)
  (cond ((endp C-1) nil)
        ((member-equal (complementario (car C-1)) C-2)
         (car C-1))
        (t (literal-resolucion-clausulas (cdr C-1) C-2))))
```

Nuestra intención es definir un procedimiento de resolución lo más general posible, dentro de las limitaciones del sistema. Para ello tenemos en cuenta que la mayoría de las estrategias de resolución se basan en comprobar cierta propiedad sobre las cláusulas padre. Por ejemplo, en la resolución positiva (negativa) una de las cláusulas padre ha de tener todos sus literales positivos (negativos), en la resolución unidad una de las cláusulas padre ha de ser unitaria, en la resolución semántica una de las cláusulas padre ha de ser cierta en una asignación predeterminada y la otra falsa, etc. La propia resolución binaria tiene asociada una propiedad cuyo valor es siempre cierto.

Definición 9.3 *Sean C_1 y C_2 dos cláusulas, L un literal y P un predicado binario conmutativo tales que $L \in C_1$, $\overline{L} \in C_2$ y $P(C_1, C_2)$. La* **resolvente condicionada** *de C_1 y C_2 con respecto al literal L es la cláusula $(C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$. Llamaremos a P la condición de resolución.*

A lo largo de este capítulo consideraremos un predicado binario conmutativo predeterminado al que llamaremos `condicion`, y siempre que usemos el término "resolvente", estaremos haciendo referencia al concepto de "resolvente condicionada". Este predicado se ha incorporado a la formalización utilizando un encapsulado de ACL2. De esta forma se ha asumido la existencia de la función de dos argumentos `condicion` con las siguientes propiedades:

- El valor devuelto por la función `condicion` es un booleano en el sistema.
- La función `condicion` es conmutativa.
- La función `condicion` es congruente (ver sección 2.2.7) con respecto a la igualdad de cláusulas como conjuntos.

La formalización de estas propiedades es la siguiente:

256
```
(defthm condicion-boolean-p
  (booleanp (condicion C-1 C-2)))

(defthm condicion-conmutativa
  (iff (condicion C-1 C-2)
       (condicion C-2 C-1)))

(defcong igual-conjunto iff (condicion C-1 C-2) 1)

(defcong igual-conjunto iff (condicion C-1 C-2) 2)
```

Donde la función `igual-conjunto` comprueba que dos listas contiene los mismos elementos, es decir, son iguales como conjuntos.

Un aspecto importante de los procedimientos basados en resolución es que las cláusulas válidas son prescindibles, ya que la satisfacibilidad o insatisfacibilidad de un conjunto de cláusulas no depende de ellas. Esta afirmación es consecuencia de los resultados de corrección y completitud que se demuestran en este capítulo acerca de un procedimiento basado en resolución que no considera cláusulas válidas.

Otra apreciación importante es que, si hay dos literales distintos con respecto a los cuales se pueda obtener una resolvente entre dos cláusulas, entonces las resolventes obtenidas son válidas y, según lo anterior, se puede prescindir de ellas en el procedimiento de resolución.

Teorema 9.4 *Dadas dos cláusulas C_1 y C_2, tales que $P(C_1, C_2)$ y dos literales distintos L_1 y L_2 tales que $L_1, L_2 \in C_1$ y $\overline{L}_1, \overline{L}_2 \in C_2$, entonces la resolvente de C_1 y C_2 con respecto a L_1 es válida. Por tanto, la resolvente con respecto a L_2 también es válida.*

Demostración:

Puesto que L_2 es distinto de L_1, entonces se tiene que $L_2 \in C_1 - \{L_1\}$. De la misma forma, como $\overline{L}_2$ es distinto de $\overline{L}_1$ se tiene que $\overline{L_2} \in C_2 - \{\overline{L}_1\}$. Luego la resolvente de C_1 y C_2 con respecto a L_1 contiene literales complementarios y, por el teorema 8.5, es válida.

□

Esto quiere decir que la resolvente entre dos cláusulas es única salvo que se obtengan cláusulas válidas, de las que se prescinde. Por tanto, para obtener una resolvente sin literales complementarios basta con, comprobar si las cláusulas padre cumplen la propiedad P y buscar el primer literal L, de la primera cláusula padre C_1, cuyo complementario aparezca en la segunda cláusula padre C_2. Si la resolvente de C_1 y C_2 con respecto a L contiene literales complementarios, entonces cualquier resolvente entre dichas cláusulas los tendrá y se prescinde de todas. Si, por el contrario, la resolvente de C_1 y C_2 con respecto a L no contiene literales complementarios, entonces L es el único literal con respecto al que se puede calcular una resolvente entre C_1 y C_2. De aquí en adelante llamaremos resolvente de dos cláusulas a la única resolvente sin literales complementarios entre dichas cláusulas, si es que existe.

La función `resolucion-cond-clausulas` calcula una resolvente entre dos cláusulas, si es que existe, teniendo en cuenta lo anterior. Esta función utiliza `nil` para indicar que no ha sido posible obtener una resolvente sin literales complementarios entre las cláusulas padre. Sin embargo `nil` es la lista vacía y podría ser obtenida como resolvente entre dichas cláusulas. Por eso esta función devuelve una lista unitaria formada por la resolvente obtenida. De esta forma, el resultado `nil` indica que no hay resolventes sin literales complementarios y el resultado `(nil)`, que la resolvente obtenida es la cláusula vacía. Para comprobar si la resolvente tiene literales complementarios hemos utilizado la función `tiene-literales-complementarios`, presentada en 185

257
```
(defun resolucion-cond-clausulas (C-1 C-2)
  (if (condicion C-1 C-2)
      (let ((L (literal-resolucion-clausulas C-1 C-2)))
        (if L
            (let ((res (resolvente L C-1 C-2)))
              (if (tiene-literales-complementarios res)
                  nil
                  (list res)))
            nil))
      nil))
```

Como se ha comentado al comienzo de esta sección, el objetivo de los procedimientos basados en resolución es generar todas las cláusulas que sean consecuencia lógica de un conjunto de cláusulas de partida. Cuando se llega a esta situación, diremos que hemos obtenido un conjunto de cláusulas saturado por resolución.

Definición 9.5 *Diremos que un conjunto de cláusulas S está saturado por resolución (condicionada) si para cualesquiera $C_1, C_2 \in S$ tales que $P(C_1, C_2)$, entonces la resolvente de C_1 y C_2, si es que existe, aparece en S.*

La formalización de este concepto en ACL2 es compleja puesto que el sistema no dispone de un cuantificador. Sin embargo, la cuantificación a la que hace referencia la definición se realiza sobre los elementos del conjunto de cláusulas inicial y este tipo de cuantificación se puede formalizar analizando recursivamente la lista que representa dicho conjunto. De esta forma, para cada elemento C de S, comprobaremos que las resolventes entre C y algún otro elemento de S aparecen en S. Para hacer esto tendemos que volver a analizar recursivamente la lista que representa el conjunto S.

258

```
(defun saturado-cond-conjunto (S)
  (saturado-cond-conjunto-aux S S))

(defun saturado-cond-conjunto-aux (S-1 S-2)
  (cond
    ((endp S-1) t)
    (t (and (saturado-cond-clausula-conjunto (car S-1) S-1 S-2)
            (saturado-cond-conjunto-aux (cdr S-1) S-2)))))

(defun saturado-cond-clausula-conjunto (C S-1 S-2)
  (cond
    ((endp S-1) t)
    (t (let ((res (resolucion-cond-clausulas C (car S-1))))
        (if (consp res)
           (and (conjunto-miembro (car res) S-2)
               (saturado-cond-clausula-conjunto C (cdr S-1) S-2))
           (saturado-cond-clausula-conjunto C (cdr S-1) S-2))))))
```

La función `saturado-cond-clausula-conjunto` comprueba que las resolventes entre una cláusula `C` y algún elemento del conjunto `S-1` aparecen en `S-2`. Para ello utiliza la función `conjunto-miembro` que comprueba si en una colección de conjuntos existe alguno igual a uno dado. Para comprobar que las resolventes entre dos elementos cualesquiera de `S-1` aparecen en `S-2` definimos la función `saturado-cond-conjunto-aux`. Obsérvese que, como la resolvente entre dos cláusulas C_1 y C_2, si es que existe, es la misma, es decir el mismo conjunto, que la resolvente entre C_2 y C_1, esta función evita realizar la doble comprobación calculando sólo las resolventes entre un elemento de `S-1` y los que le siguen. Finalmente, la función `saturado-cond-conjunto` comprueba si el conjunto `S` es saturado por resolución.

Se verifica que, si un conjunto de cláusulas es saturado por resolución según la función anterior entonces, la resolvente, si es que existe, de dos cláusulas cualesquiera que aparezcan en dicho conjunto también aparece en él:

259
```
(defthm saturado-cond-conjunto-contiene-resolventes
  (implies
    (and (saturado-cond-conjunto S)
         (conjunto-miembro C1 S)
         (conjunto-miembro C2 S)
         (consp (resolucion-cond-clausulas C1 C2))
         (es-forma-clausal S))
    (conjunto-miembro (car (resolucion-cond-clausulas C1 C2))
                      S)))
```

Para comprobar que un conjunto de cláusulas es saturado por resolución, es más comodo utilizar la definición 9.5 como un teorema ACL2 de caracterización: Si suponemos la existencia de ciertas hipótesis `(saturado-cond-H)` y un conjunto de clausulas `(saturado-cond-S)` de forma que se pueda demostrar el siguiente evento:

260
```
(defthm resolvente-cond-miembro-saturado
  (implies
    (and (saturado-cond-H)
         (conjunto-miembro C1 (saturado-cond-S))
         (conjunto-miembro C2 (saturado-cond-S))
         (consp (resolucion-cond-clausulas C1 C2)))
    (conjunto-miembro (car (resolucion-cond-clausulas C1 C2))
                      (saturado-cond-S))))
```

Entonces se puede demostrar este otro:

261
```
(defthm saturado-cond-conjunto-extension
  (implies (saturado-cond-H)
           (saturado-cond-conjunto (saturado-cond-S))))
```

De esta forma, la prueba de que un conjunto de cláusulas `S` es saturado por resolución, consiste en hacer una instancia funcional (ver sección 2.2.6) de este último evento en la que la función `saturado-cond-S` sea la función constante igual a `S` y la función `saturado-cond-H` un conjunto de hipótesis, demostrando previamente un resultado similar al mostrado en *260* sobre `S` y dichas hipótesis. En la sección 9.1.4 hay un ejemplo de uso de esta técnica.

9.1.2 Decisión de insatisfacibilidad por saturación: $INSAT_{\mathcal{R}}$

Como se ha comentado en la sección anterior, los procedimientos basados en resolución se basan en, a partir de un conjunto de cláusulas, obtener otro saturado

por resolución. El conjunto original será satisfacible si y sólo si en el proceso de saturación no se obtiene la cláusula vacía. Por tanto, es importante asegurarse de que se generan todas las resolventes posibles, mediante un proceso que, en el caso de la lógica proposicional, termina para cualquier conjunto de cláusulas inicial.

Para conseguir esto es habitual considerar dos conjuntos, en el primero, S_1, se encuentran las cláusulas para las que todavía se pueden generar resolventes nuevas, mientras que en el segundo, S_2, se encuentran aquellas para las que ya se han generado todas las resolventes posibles. El proceso de saturación consiste en tomar una cláusula C del primer conjunto y obtener todas las resolventes entre C y algún elemento de S_2. Estas resolventes son añadidas a S_1, de donde se elimina C, que es añadido a S_2. El proceso continúa hasta que S_1 se queda vacío. Si el proceso comienza con S_2 igual al conjunto vacío, entonces al terminar se obtiene en S_2 la saturación por resolución de S_1. Si a lo largo del proceso se obtiene la cláusula vacía entonces el conjunto de cláusulas original es insatisfacible.

Algoritmo 9.6 ($\mathcal{SR}$) *El dato de entrada de este algoritmo es un conjunto de cláusulas S. Inicialmente se consideran los conjuntos $S_1 = S$ y $S_2 = \emptyset$ y se actúa como sigue:*

1. *Si S_1 está vacío, el algoritmo termina y devuelve S_2.*

2. *Se selecciona la primera cláusula C de S_1 y se obtienen todas las resolventes entre C y los elementos de S_2 que no aparezcan en S_1 ni en S_2. Llamaremos a este conjunto S_n.*

3. *Si S_n contiene la cláusula vacía, el algoritmo termina y devuelve* $\mathbf{t}$.

4. *Si S_n no contiene la cláusula vacía, se vuelve al paso 1 con los conjuntos $(S_1 - \{C\}) \cup S_n$ y $S_2 \cup \{C\}$.*

En la siguiente tabla se muestra un ejemplo de aplicación de este proceso de saturación. Comenzando con $S_1 = \langle\langle p, q\rangle, \langle p, \neg q\rangle, \langle \neg q, p\rangle\rangle$ y $S_2 = \langle\rangle$. La primera cláusula que se toma de S_1 es $\langle p, q\rangle$, entre esta cláusula y S_2 no se obtiene ninguna resolvente, así que eliminamos dicha cláusula de S_1 y la añadimos a S_2. En el segundo paso se escoge la cláusula $\langle p, \neg q\rangle$. Ahora hay una resolvente entre esta cláusula y las del conjunto $S_2 = \langle\langle p, q\rangle\rangle$, la cláusula $\langle p\rangle$, que es añadida a S_1. En este punto $S_1 = \langle\langle p\rangle, \langle \neg q, p\rangle\rangle$ y $S_2 = \langle\langle p, q\rangle, \langle p, \neg q\rangle\rangle$. La siguiente cláusula que se toma de S_1 es $\langle p\rangle$. Entre esta cláusula y las de S_2 no se generan resolventes, por tanto añadimos $\langle p\rangle$ a S_2 y lo eliminamos de S_1. La siguiente cláusula de S_1 es $\langle \neg p, q\rangle$. Entre ésta y las de S_2 se obtienen $\langle q\rangle$, que se añade a S_1, y $\langle p\rangle$, de la que se prescinde pues no es nueva. Finalmente se toma $\langle q\rangle$ de S_1. Entre esta cláusula y las de S_2 no aparecen nuevas resolventes (de nuevo se obtiene $\langle p\rangle$ que ya había aparecido), con lo que S_1 queda vacío y el proceso termina.

S_1	S_2	C	Resolventes
$\langle\langle p,q\rangle,\langle p,\neg q\rangle,\langle\neg q,p\rangle\rangle$	$\langle\rangle$	$\langle p,q\rangle$	
$\langle\langle p,\neg q\rangle,\langle\neg q,p\rangle\rangle$	$\langle\langle p,q\rangle\rangle$	$\langle p,\neg q\rangle$	$\langle p\rangle$
$\langle\langle p\rangle,\langle\neg q,p\rangle\rangle$	$\langle\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle p\rangle$	
$\langle\langle\neg q,p\rangle\rangle$	$\langle\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle\neg q,p\rangle$	$\langle p\rangle,\langle q\rangle$
$\langle\langle q\rangle\rangle$	$\langle\langle\neg q,p\rangle,\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle q\rangle$	$\langle p\rangle$
$\langle\rangle$	$\langle\langle q\rangle,\langle\neg q,p\rangle,\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$		

El algoritmo $\mathcal{SR}$ se puede utilizar para decidir la insatisfacibilidad de un conjunto de cláusulas S, basta con comprobar que el resultado de aplicar dicho algoritmo a S es $\mathbf{t}$.

Algoritmo 9.7 ($INSAT_{\mathcal{R}}$) *Dado un conjunto de cláusulas S, $INSAT_{\mathcal{R}}(S) = \mathbf{t}$ si y sólo si $\mathcal{SR}(S) = \mathbf{t}$.*

La función `INSAT-resolucion` implementa el algoritmo $INSAT_{\mathcal{R}}$. Esta función se define comprobando que el resultado de evaluar `saturacion-resolucion` sobre el conjunto de cláusulas inicial es `t`.

262
```
(defun INSAT-resolucion (S)
  (equal (resolucion-saturacion S) t))
```

La función `saturacion-resolucion` implementa el algoritmo $\mathcal{SR}$. Esta función hace una llamada a la función `resolucion-cond-saturacion` con el primer argumento igual al conjunto de cláusulas inicial y el segundo igual al conjunto vacío.

263
```
(defun saturacion-resolucion (S)
  (resolucion-cond-saturacion S nil))
```

La función `resolucion-cond-saturacion` implementa los puntos 2, 3 y 4 del algoritmo de saturación $\mathcal{SR}$. Sus argumentos son los conjuntos de cláusulas S_1 y S_2.

264
```
(defun resolucion-cond-saturacion (S-1 S-2)
  (declare (xargs :measure (medida-saturacion S-1 S-2)))
  (if (endp S-1)
      S-2
      (let ((resolventes (resolucion-cond-clausula-conjunto
                          (car S-1) S-2 nil (append S-1 S-2))))
        (if (contiene-clausula-vacia resolventes)
            t
            (resolucion-cond-saturacion
              (append resolventes (cdr S-1))
              (cons (car S-1) S-2))))))
```

La función `resolucion-cond-clausula-conjunto` implementa el punto 2. Esta función recibe cuatro argumentos, el primero es la cláusula que se toma del conjunto S_1, el segundo es el conjunto S_2, el tercero es un acumulador donde se irán almacenando las resolventes nuevas y el cuarto es la concatenación de S_1 y S_2 y se utiliza para comprobar si una resolvente ya ha aparecido.

265
```
(defun resolucion-cond-clausula-conjunto (C S-2 S-N S-1-2)
  (if (endp S-2)
      S-N
      (let ((res (resolucion-cond-clausulas C (car S-2))))
        (if (or (not (consp res))
                (conjunto-miembro (car res) S-1-2)
                (conjunto-miembro (car res) S-N))
            (resolucion-cond-clausula-conjunto
              C (cdr S-2) S-N S-1-2)
            (resolucion-cond-clausula-conjunto
              C (cdr S-2) (cons (car res) S-N) S-1-2)))))
```

Utilizando un predicado `condicion` concreto, por ejemplo uno que siempre sea cierto (que se corresponde con la resolución binaria), podemos utilizar las funciones anteriores para comprobar la insatisfacibilidad de un conjunto de cláusulas y para obtener un conjunto saturado por resolución, si es que el conjunto de cláusulas inicial es satisfacible.

266
```
ACL2 !>(INSAT-resolucion '( (p q) (p (- q)) ((- p) q) ))
NIL
ACL2 !>(saturacion-resolucion '( (p q) (p (- q)) ((- p) q) ))
((Q) ((- P) Q) (P) (P (- Q)) (P Q))
ACL2 !>(INSAT-resolucion
          '( (p q) (p (- q)) ((- p) q) ((- p) (- q)) ))
T
ACL2 !>(saturacion-resolucion
          '( (p q) (p (- q)) ((- p) q) ((- p) (- q)) ))
T
```

Obsérvese que la función `saturacion-resolucion` devuelve `t` en el caso de que en el proceso se haya obtenido la cláusula vacía y en otro caso devuelve el conjunto de cláusulas saturado por resolución. Por otro lado la función `INSAT-resolucion` devuelve `t` si en el proceso de saturación se obtiene la cláusula vacía y `nil` en otro caso.

9.1.3 Terminación de $INSAT_{\mathcal{R}}$

La terminación del algoritmo $INSAT_{\mathcal{R}}$ estará garantizada si se demuestra que la función `resolucion-cond-saturacion` termina para cualquier dato de entrada. Para ello se proporciona una medida de los argumentos, `medida-saturacion`, que decrece en las llamadas recursivas. Veamos cómo se obtiene dicha medida.

La función `resolucion-cond-saturacion` tiene una única llamada recursiva en la que el primer argumento es `(append resolventes (cdr S-1))` y el segundo es `(cons (car S-1) S-2)`. De esta forma, el segundo argumento de la llamada recursiva siempre tiene un elemento más que al principio, `(car S-1)`, mientras que el primer argumento siempre tiene un elemento menos, `(car S-1)`, y, eventualmente, se le añaden los elementos de `resolventes`.

Se tienen dos situaciones distintas en función del valor de `resolventes`. Si este valor es `nil`, es decir no hay resolventes nuevas, entonces el primer argumento de la llamada recursiva tiene menos elementos que al principio y su longitud puede usarse como medida para garantizar la terminación. Si el valor de la variable `resolventes` es distinto de `nil` entonces los dos argumentos de la llamada recursiva tienen más elementos que al principio.

Los elementos de la lista almacenada en la variable `resolventes` son los obtenidos al evaluar la función `resolucion-cond-clausula-conjunto`, y, en el valor devuelto por esta función, no se incluye ninguna cláusula que se encuentre ya en `(append S-1 S-2)`. Luego los elementos de `resolventes` no aparecen en los valores iniciales de los argumentos y el número total de cláusulas distintas en los argumentos de la llamada recursiva aumenta. Este número está acotado y por tanto, la diferencia entre una cota y dicho número es un valor que disminuye en las llamadas recursivas en las que `resolventes` es distinto de `nil`.

La medida que se considera para demostrar la terminación de la función `resolucion-cond-saturacion` es un par en el que el primer elemento es la diferencia entre una cota del número total de cláusulas distintas que se pueden obtener y el número de cláusulas distintas que aparecen en los argumentos de la llamada, y el segundo elemento es la longitud del primer argumento. Estos valores se comparan de forma lexicográfica: primero se comparan los primeros elementos de los pares y, si son iguales, se comparan los segundos elementos. En la siguiente tabla mostramos cómo cambian estos valores en las llamadas recursivas del ejemplo de la página 250.

S_1	S_2	Medida
$\langle\langle p,q\rangle,\langle p,\neg q\rangle,\langle\neg q,p\rangle\rangle$	$\langle\rangle$	$\langle\ 9-3\ ,\ 3\ \rangle$
$\langle\langle p,\neg q\rangle,\langle\neg q,p\rangle\rangle$	$\langle\langle p,q\rangle\rangle$	$\langle\ 9-3\ ,\ 2\ \rangle$
$\langle\langle p\rangle,\langle\neg q,p\rangle\rangle$	$\langle\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle\ 9-4\ ,\ 2\ \rangle$
$\langle\langle\neg q,p\rangle\rangle$	$\langle\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle\ 9-4\ ,\ 1\ \rangle$
$\langle\langle q\rangle\rangle$	$\langle\langle\neg q,p\rangle,\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle\ 9-5\ ,\ 1\ \rangle$
$\langle\rangle$	$\langle\langle q\rangle,\langle\neg q,p\rangle,\langle p\rangle,\langle p,\neg q\rangle,\langle p,q\rangle\rangle$	$\langle\ 9-5\ ,\ 0\ \rangle$

En este caso, una cota del número total de cláusulas distintas que pueden aparecer es 9. Al iniciar el proceso la medida es $\langle\ 9-3\ ,\ 3\ \rangle$. En el primer paso no se generan resolventes nuevas y la longitud del primer argumento de la llamada recursiva disminuye en uno; la medida es $\langle\ 9-3\ ,\ 2\ \rangle$. En el segundo paso, se genera una resolvente nueva con lo que la medida pasa a ser $\langle\ 9-4\ ,\ 2\ \rangle$. En el tercer paso, tampoco se generan resolventes nuevas y la longitud del primer argumento de la siguiente llamada disminuye; la medida que resulta es $\langle\ 9-4\ ,\ 1\ \rangle$. En el cuarto paso, se obtiene una resolvente nueva con lo que la medida pasa a ser $\langle\ 9-5\ ,\ 1\ \rangle$. Finalmente, en el último paso no se generan resolventes nuevas, por lo que el primer argumento de la siguiente llamada se queda vacío; la medida es $\langle\ 9-5\ ,\ 0\ \rangle$. En todo paso el par obtenido es menor, en el sentido lexicográfico, que el par del paso anterior.

Teorema 9.8 *El número total de cláusulas distintas que se pueden obtener con n variables es 4^n.*

Demostración:

La prueba es por inducción en n.

1. Si $n = 0$, entonces sólo hay una cláusula que se pueda obtener con 0 variables, la cláusula vacía.

2. Supongamos que el resultado es cierto para $n-1$. Las cláusulas que se pueden obtener con n variables se pueden clasificar en cuatro grupos según lo que ocurra con una de esas variables v:

   - Las cláusulas que contienen el literal v y no contienen el literal $\neg v$.
   - Las cláusulas que contienen los literales v y $\neg v$.
   - Las cláusulas que no contienen el literal v y contienen el literal $\neg v$.
   - Las cláusulas que no contienen los literales v ni $\neg v$.

   En cualquiera de los cuatro casos, el número de cláusulas que se pueden obtener es igual al que se obtiene si la variable v no aparece, es decir, para $n-1$ variables. Así, en los cuatro casos, el número de cláusulas distintas que se pueden obtener es 4^{n-1}. Por tanto, el número de cláusulas distintas que se pueden obtener con n variables es 4^n.

□

La función `medida-resolucion-cota` calcula el valor de la cota a partir de los argumentos de la función `resolucion-cond-saturacion`. Para ello se utiliza la función `variables-forma-clausal`, que devuelve el conjunto de variables que aparecen en un conjunto de cláusulas (o una forma clausal), y la función `numero-clausulas-cota`, que calcula el valor 4^n, donde n es el número de elementos del conjunto obtenido con `variables-forma-clausal`.

267
```
(defun medida-resolucion-cota (S)
  (numero-clausulas-cota (variables-forma-clausal S)))
```

En la formalización ha sido necesario demostrar que el valor de la cota no cambia, al considerar los conjuntos de cláusulas de la llamada recursiva de la función `resolucion-cond-saturacion`:

268
```
(defthm medida-resolucion-cota-invariante
  (equal (medida-resolucion-cota
          (append (append (resolucion-cond-clausula-conjunto
                           (car S-1) S-2 nil (append S-1 S-2))
                          (cdr S-1))
                  (cons (car S-1) S-2))
         (medida-resolucion-cota (append S-1 S-2)))))
```

Para contar el número de cláusulas distintas que aparecen en un conjunto, hemos utilizado una técnica similar a la usada en la demostración del teorema 9.8. Para cada variable v que aparece en el conjunto de cláusulas sumamos el número de cláusulas en las que aparecen los literales v y $\neg v$, el número de cláusulas en las que aparece v pero no aparece $\neg v$, el número de cláusulas en las que no aparece v pero aparece $\neg v$ y el número de cláusulas en las que no aparecen ni v ni $\neg v$. La función que realiza este cálculo para un conjunto de variables dado es `numero-clausulas-variables`. El número de cláusulas distintas que aparecen en un conjunto `S` es el resultado de evaluar la función `numero-clausulas-variables` sobre el conjunto de variables que aparecen en `S`.

269
```
(defun medida-resolucion (S)
  (numero-clausulas-variables (variables-forma-clausal S) S))
```

Esta forma de calcular la cota y de contabilizar el número de cláusulas distintas que aparecen en un conjunto facilita la demostración en el sistema del siguiente teorema:

Teorema 9.9 *Sea S un conjunto de cláusulas en el que aparecen n variables distintas, entonces el número de elementos de S es menor o igual que 4^n.*

Este resultado es una consecuencia directa del teorema 9.8. Su formalización es la siguiente:

270
```
(defthm medida-resolucion-cota-es-maximo
  (<= (medida-resolucion S)
      (medida-resolucion-cota S)))
```

Por otro lado, el valor obtenido con la función `medida-resolucion` aumenta en las llamadas recursivas de la función `resolucion-cond-saturacion`, siempre que se generen nuevas resolventes y no haya aparecido la cláusula vacía:

271
```
(defthm medida-resolucion-aumenta
  (implies (and (not (contiene-clausula-vacia
                       (resolucion-cond-clausula-conjunto
                         (car S-1) S-2 nil (append S-1 S-2))))
                (consp (resolucion-cond-clausula-conjunto
                         (car S-1) S-2 nil (append S-1 S-2))))
   (< (medida-resolucion (cons c (append S-1 S-2)))
      (medida-resolucion
        (append (append (resolucion-cond-clausula-conjunto
                          (car S-1) S-2 nil (append S-1 S-2))
                        (cdr S-1))
                (cons (car S-1) S-2))))))
```

La prueba de este resultado requiere gran cantidad de lemas previos que relacionan la función `medida-resolucion` con las funciones `append`, `cons`, y `resolucion-cond-clausula-conjunto`.

Finalmente definimos la función `medida-saturacion` que devuelve el par cuyo primer elemento es uno más la diferencia entre el valor de la función `medida-resolucion-cota` y el de la función `medida-resolucion`, evaluadas sobre el total de cláusulas que aparecen en los conjuntos `S-1` y `S-2`. El segundo elemento del par es la longitud de `S-1`, obtenida con la función `len`:

272
```
(defun medida-saturacion (S-1 S-2)
  (cons (- (+ (medida-resolucion-cota (append S-1 S-2)) 1)
           (medida-resolucion (append S-1 S-2)))
        (len S-1)))
```

Se ha incrementado en una unidad el primer valor del par para que el resultado sea un ordinal en ACL2 (vease la sección 2.2.1 para obtener más detalles sobre la representación de los ordinales en ACL2) y se puedan comparar directamente con la función `e0-ord-<`.

Con los eventos mostrados en *268*, *270* y *271*, el sistema demuestra automáticamente el resultado de terminación de `resolucion-cond-saturación` con respecto a la medida de los argumentos dada por la función `medida-saturacion`.

9.1.4 Corrección de $INSAT_{\mathcal{R}}$ y otras propiedades

A continuación presentamos el teorema de corrección del algoritmo $INSAT_{\mathcal{R}}$, así como otras propiedades de $\mathcal{SR}$. Los eventos correspondientes a estos teoremas se encuentran en el libro `resolucion-aux.lisp`.

Teorema 9.10 (Corrección de $INSAT_{\mathcal{R}}$) *Dada un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) = \mathbf{t}$ entonces S es insatisfacible.*

Demostración:
La demostración es por reducción al absurdo. Supongamos que S tiene un modelo σ entonces, en cada paso del algoritmo $\mathcal{SR}$, σ es modelo de S_1, S_2 y S_n:

- Al principio del algoritmo $\mathcal{SR}$, σ es modelo de $S_1 = S$ y $S_2 = \emptyset$.
- Si σ es modelo de S_1 y S_2 entonces es modelo de la cláusula C tomada de S_1 y, por el teorema 9.2, es modelo de todos los elementos de S_n. Por tanto es modelo de los conjuntos $(S_1 - \{C\}) \cup S_n$ y $S_2 \cup \{C\}$, utilizados en la siguiente llamada de $\mathcal{SR}$.

Puesto que $INSAT_{\mathcal{R}}(S) = \mathbf{t}$, en algún momento la cláusula vacía aparece en S_n y esto contradice el hecho de que σ es modelo de S_n. Por tanto S es insatisfacible. □

El evento que formaliza este teorema se muestra en 273. Para probarlo se han establecido eventos que afirman que las funciones `resolucion-cond-clausulas` y `resolucion-cond-clausulas-conjunto` preservan la satisfacibilidad y se ha demostrado que si una asignación es modelo de los conjuntos S_1 y S_2 usados en el algoritmo $\mathcal{SR}$ entonces este algoritmo no devuelve $\mathbf{t}$, es decir, no se genera la cláusula vacía, lo que contradice la hipótesis `(INSAT-resolucion S)`.

273
```
(defthm correccion-INSAT-resolucion
  (implies (and (es-forma-clausal S)
                (INSAT-resolucion S))
           (not (modelo-forma-clausal sigma S))))
```

Los siguiente teoremas son propiedades sobre el algoritmo $\mathcal{SR}$, que serán necesarias para demostrar la completitud de $INSAT_{\mathcal{R}}$.

Teorema 9.11 *Dado un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces $\mathcal{SR}(S)$ es una forma clausal.*

Este resultado es trivial puesto que, por el resultado establecido en 253, la resolvente de dos cláusulas es una cláusula y, por tanto, el proceso de saturación por resolución condicionada genera formas clausales. Su formalización es la siguiente:

274
```
(defthm INSAT-resolucion-es-forma-clausal
  (implies (and (es-forma-clausal S)
                (not (INSAT-resolucion S)))
           (es-forma-clausal (saturacion-resolucion S))))
```

Teorema 9.12 *Dado un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces $\mathcal{SR}(S)$ no contiene la cláusula vacía.*

Este resultado también es obvio pues, si en el proceso de saturación por resolución aparece la cláusula vacía, entonces $INSAT_{\mathcal{R}}(S) = \mathbf{t}$. Su formalización es la siguiente:

275
```
(defthm INSAT-resolucion-no-contiene-clausula-vacia
  (implies (and (not (contiene-clausula-vacia S))
                (not (INSAT-resolucion S)))
           (not (contiene-clausula-vacia
                  (saturacion-resolucion S)))))
```

Teorema 9.13 *Dado un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces $S \subset \mathcal{SR}(S)$.*

De nuevo se trata de una propiedad fácil de demostrar puesto que, en cada iteración del algoritmo $\mathcal{SR}$ se elimina un elemento del conjunto S_1 que pasa a formar parte de S_2 en la siguiente iteración. Si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$, entonces el algoritmo $\mathcal{SR}$ devuelve el último valor de S_2 obtenido cuando S_1 es vacío. Como al principio $S_1 = S$, entonces cada elemento de S pasa, en alguna iteración, a formar parte del conjunto S_2 y, finalmente, del valor devuelto por $\mathcal{SR}$. Su formalización es la siguiente:

276
```
(defthm INSAT-resolucion-contiene-original
  (implies (not (INSAT-resolucion S))
           (subsetp-equal S (saturacion-resolucion S))))
```

Teorema 9.14 *Dado un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces $\mathcal{SR}(S)$ es un conjunto saturado por resolución condicionada.*

Demostración:

Supongamos que $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$, es decir, $\mathcal{SR}(S)$ devuelve el último valor del conjunto S_2, cuando S_1 es vacío. Sean C_1 y C_2 dos cláusulas de $\mathcal{SR}(S)$. En alguna iteración del proceso de saturación estas cláusulas se añaden al conjunto S_2, supongamos que C_1 es la primera en ser añadida y, unas iteraciones más tarde, es añadida C_2. En la iteración en la que C_2 es añadida a S_2, se calcula la resolvente condicionada entre C_1 y C_2 y, si existe, es añadida al conjunto S_1 de la siguiente iteración. Puesto que el proceso termina cuando el valor del conjunto S_1 es vacío, cada uno de sus elementos se pasa en alguna iteración al conjunto S_2, por tanto, la resolvente condicionada entre C_1 y C_2, si es que existe, se añade en alguna iteración a S_2 y, finalmente, aparece en el conjunto devuelto por $\mathcal{SR}$.

□

Para demostrar este teorema en la formalización hemos utilizado la caracterización dada en *260-261*. Hemos demostrado que, dadas dos cláusulas del conjunto `(saturacion-resolvente S)`, su resolvente condicionada, si es que existe, también está en dicho conjunto (`saturacion-resolucion-funciona`). El evento que afirma que el conjunto `(saturacion-resolucion S)` verifica el predicado `saturado-cond-conjunto` se obtiene como una instancia funcional (ver sección 2.2.6) del evento mostrado en *261* sustituyendo la función `saturado-cond-S` por la función constante igual al conjunto `(saturacion-resolucion S)` y la función `saturado-cond-H` por la función constante igual a la conjunción de las hipótesis `(es-forma-clausal S)` y `(not (INSAT-resolucion S))`. Su formalización es la siguiente:

277

```
(defthm INSAT-resolucion-saturado-cond-conjunto
  (implies (and (es-forma-clausal S)
                (not (INSAT-resolucion S)))
           (saturado-cond-conjunto (saturacion-resolucion S))))

(defthm saturacion-resolucion-funciona
  (implies
    (and (es-clausula C1)
         (es-clausula C2)
         (conjunto-miembro C1 (saturacion-resolucion S))
         (conjunto-miembro C2 (saturacion-resolucion S))
         (consp (resolucion-cond-clausulas C1 C2)))
    (conjunto-miembro (car (resolucion-cond-clausulas C1 C2))
                      (saturacion-resolucion S))))
```

9.2 Teorema de completitud de Bezem

Esta sección está dedicada a la formalización en ACL2 de la demostración de Bezem del teorema de completitud de la resolución condicionada [8]. Este teorema afirma que todo conjunto de cláusulas saturado por resolución condicionada que no contenga la cláusula vacía es satisfacible. La prueba es la misma para distintas estrategias de resolución para las que existe una relación entre la condición que las define y una asignación predeterminada.

El desarrollo de la prueba se hace de una manera similar a como se presenta en [21]. En la primera parte de esta sección se desarrollan resultados relativos a conjuntos que tienen intersección no vacía con todos los elementos de una colección de conjuntos. Estos conjuntos se denominan conjuntos representantes y son utilizados en la prueba de completitud. Los eventos desarrollados en esta parte se encuentran en el libro `representantes.lisp`.

En la segunda parte se presenta la prueba del teorema de completitud y su formalización, los eventos correspondientes se encuentran en el libro `resolucion-thm.lisp`. En la última parte se demuestra la completitud del algoritmo $INSAT_{\mathcal{R}}$, su formalización aparece en el libro `resolucion-aux.lisp`.

9.2.1 Conjuntos representantes

Definición 9.15 *Dada una colección de conjuntos $\mathcal{S}$. Un conjunto R es un*

1. *conjunto representante de $\mathcal{S}$ si $\forall C \in \mathcal{S}$ se tiene $R \cap C \neq \emptyset$, es decir, R intersecta todo elemento de $\mathcal{S}$.*

2. *conjunto representante minimal de $\mathcal{S}$, si R es un conjunto representante de $\mathcal{S}$, y no tiene ningún subconjunto propio que sea a su vez conjunto representante de $\mathcal{S}$.*

El concepto de conjunto representante lo hemos formalizado con la siguiente función, donde `conjunto-vacio` comprueba si su argumento es una lista sin elementos.

278
```
(defun conjunto-representante (R S)
  (cond ((endp S) t)
        ((not (conjunto-vacio (interseccion R (car S))))
         (conjunto-representante R (cdr S)))
        (t nil)))
```

El concepto de conjunto representante minimal se puede establecer calculando todos los subconjuntos propios de un conjunto dado y, para cada uno de ellos, comprobando que no son conjuntos representantes. Sin embargo, es más interesante, y fácil de expresar, una definición basada en el siguiente teorema de caracterización:

Teorema 9.16 *Sea R un conjunto representante de la colección de conjuntos $\mathcal{S}$. Las siguientes condiciones son equivalentes:*

1. *R es un conjunto representante minimal de $\mathcal{S}$.*

2. *$\forall L \in R\ \exists C \in \mathcal{S}$ tal que $R \cap C = \{L\}$. Donde C es el conjunto testigo asociado a L con respecto a R.*

Demostración:

Supongamos que R es un conjunto representante minimal de $\mathcal{S}$ entonces, dado $L \in R$, $R - \{L\}$ no es un conjunto representante de $\mathcal{S}$. Luego existe $C \in S$ tal que $C \cap (R - \{L\}) = \emptyset$, como $R \cap C \neq \emptyset$, se tiene $R \cap C = \{L\}$.

Supongamos que para todo $L \in R$ existe $C \in \mathcal{S}$ tal que $R \cap C = \{L\}$. Entonces, para todo $L \in R$, $R - \{L\}$ no es un conjunto representante pues

su intersección con C es vacía. Por tanto, ningún subconjunto propio de R es conjunto representante.

□

De esta forma, para definir el concepto de conjunto representante minimal, basta con comprobar que, para todo elemento L del candidato a conjunto representante minimal R, hay un elemento C en la colección $\mathcal{S}$ tal que, la intersección entre R y C es el conjunto unitario formado por L. La siguiente función recorre la colección $\mathcal{S}$ buscando un conjunto C que cumpla dicha condición con respecto a R y L:

279
```
(defun conjunto-testigo (R S L)
  (cond ((endp S) nil)
        ((and (member-equal L (car S))
              (conjunto-unitario (interseccion R (car S))))
         (car S))
        (t (conjunto-testigo R (cdr S) L))))
```

Donde la función `interseccion` devuelve la lista de los elementos que hay en común entre las listas que recibe como argumento y `conjunto-unitario` comprueba que no hay dos elementos distintos en la lista que recibe como argumento.

Para comprobar que un conjunto R es un conjunto representante minimal basta con comprobar que se puede determinar un conjunto testigo para todos los elementos de R. Esto se hace recursivamente sobre la lista de elementos de R. Este proceso está formalizado con la función `conjunto-representante-minimal-aux`. Esta función se utiliza en la definición de `conjunto-representante-minimal` para comprobar si un conjunto `R` es un conjunto representante minimal de una colección de conjuntos `S`.

280
```
(defun conjunto-representante-minimal (R S)
  (and (conjunto-representante minimal S)
       (conjunto-representante-minimal-aux R R S)))

(defun conjunto-representante-minimal-aux (R-aux R S)
  (cond ((endp R-aux) t)
        ((conjunto-testigo R S (car R-aux))
         (conjunto-representante-minimal-aux (cdr R-aux) R S))
        (t nil)))
```

Veamos a continuación cómo se formaliza el teorema 9.16. Por un lado, se verifica que, si `R` es un conjunto representante minimal de la colección `S` según la función anterior, entonces ningún subconjunto propio de `R` es conjunto representante de `S`:

281
```
(defthm conjunto-representante-minimal-equivalencia-2->1
  (implies (and (conjunto-representante-minimal R S)
                (subsetp-equal R-aux R)
                (not (subsetp-equal R R-aux)))
           (not (conjunto-representante R-aux S))))
```

Para formalizar la otra implicación hay que asumir la existencia de cierto conjunto `(hip-R)` y una colección de conjuntos `(hip-S)` tales que `(hip-R)` es un conjunto representante de `(hip-S)` y ningún subconjunto propio de `(hip-R)` es conjunto representante de `(hip-S)`:

282
```
(defthm hip-minimal-es-conjunto-representante
  (conjunto-representante (hip-R) (hip-S)))

(defthm hip-minimal-no-tiene-subconjuntos-propios-representantes
  (implies (and (subsetp-equal R-aux (hip-R))
                (not (subsetp-equal (hip-R) R-aux)))
           (not (conjunto-representante R-aux (hip-S)))))
```

En esta situación se puede demostrar:

283
```
(defthm conjunto-representante-minimal-equivalencia-1->2
  (conjunto-representante-minimal (hip-R) (hip-S)))
```

El siguiente teorema afirma que toda colección de conjuntos finitos y no vacíos tiene un conjunto representante minimal, y su demostración proporciona un método para construirlo:

Teorema 9.17 *Toda colección finita de conjuntos finitos y no vacíos tiene un conjunto representante minimal.*

Demostración:

Sea S una colección finita de conjuntos finitos y no vacíos. Sea R_0 la unión de todos los elementos de S. Como S no tiene conjuntos no vacíos, R_0 tiene intersección no vacía con todos los elementos de S y, por tanto, R_0 es un conjunto representante de S. Si R_0 no es un conjunto representante minimal entonces, por el teorema de caracterización 9.16, existe L en R_0 de forma que para todo $C \in S$ tal que $L \in C$, $R_0 \cap C$ tiene al menos un elemento distinto de L. Por tanto $R_0 - \{L\}$ sigue siendo un conjunto representante de S, llamemos R_1 a este conjunto. Este proceso de eliminación de elementos innecesarios continúa hasta obtener un R_n tal que para todo elemento L en R_n, existe $C \in S$ tal que $R_n \cap C = \{L\}$ y, por tanto, R_n es un conjunto representante minimal.

□

La prueba del teorema proporciona un método para obtener un conjunto representante de una colección $\mathcal{S}$, la unión de todos los elementos de dicha colección, y un procedimiento para reducirlo hasta obtener uno minimal.

Para obtener el primer conjunto representante utilizamos la función `union-coleccion`, que realiza la unión de todos los conjuntos de una colección. Para demostrar que es un conjunto representante tendremos que comprobar que la colección no contiene conjuntos vacíos, esto lo hace la función `coleccion-de-conjuntos-no-vacios`:

284
```
(defun union-coleccion (S)
  (cond ((endp S) nil)
        (t (union-equal (car S) (union-coleccion (cdr S))))))

(defun coleccion-de-conjuntos-no-vacios (S)
  (cond ((endp S) t)
        (t (and (not (conjunto-vacio (car S)))
                (coleccion-de-conjuntos-no-vacios (cdr S))))))

(defthm union-coleccion-es-conjunto-representante
  (implies (coleccion-de-conjuntos-no-vacios S)
           (conjunto-representante (union-coleccion S) S)))
```

El proceso de reducción consiste en eliminar de un conjunto representante los elementos para los que no existe un conjunto testigo. Esto lo hace la función `reduccion-conjunto-representante`. Para ello, esta función busca un elemento sin conjunto testigo en el conjunto representante `R` y lo elimina. Este proceso continúa hasta que no quedan elementos sin conjunto testigo. En este caso la función devuelve el conjunto reducido. Para buscar un elemento sin conjunto testigo se utiliza la función `elemento-sin-conjunto-testigo`.

285
```
(defun reduccion-conjunto-representante (R S)
  (if (elemento-sin-conjunto-testigo R R S)
      (reduccion-conjunto-representante
       (elimina-elemento
        (car (elemento-sin-conjunto-testigo R R S)) R)
       S)
      R))

(defun elemento-sin-conjunto-testigo (R-aux R S)
  (cond ((endp R-aux) nil)
        ((conjunto-testigo R S (car R-aux))
         (elemento-sin-conjunto-testigo (cdr R-aux) R S))
        (t (list (car R-aux)))))
```

Se verifica que la reducción utilizando este proceso, de un conjunto representante es un conjunto representante minimal:

286
```
(defthm reduccion-conjunto-representante-es-minimal
  (implies (conjunto-representante R S)
           (conjunto-representante-minimal
             (reduccion-conjunto-representante R S) S))
```

Finalmente, la función `construye-conjunto-representante-minimal` construye un conjunto representante minimal para una colección de conjuntos no vacíos. Gracias a los resultados mostrados en *284* y *286*, se puede demostrar que esta función devuelve un conjunto representante minimal:

287
```
(defun construye-conjunto-representante-minimal (S)
  (reduccion-conjunto-representante (union-coleccion S) S))

(defthm construye-conjunto-representante-minimal-es-minimal
  (implies (coleccion-de-conjuntos-no-vacios S)
           (conjunto-representante-minimal
            (construye-conjunto-representante-minimal S) S)))
```

9.2.2 Modelo de un conjunto saturado por resolución

La demostración de Bezem del teorema de completitud de la resolución condicionada considera una asignación σ_P tal que para todo par de cláusulas C, D con $\sigma_P \models C$ y $\sigma_P \not\models D$ se tiene que C y D cumplen la condición de resolución P. En estas condiciones se verifica el siguiente teorema:

Teorema 9.18 *Sea S un conjunto de cláusulas saturado por resolución condicionada que no contiene la cláusula vacía, entonces S es satisfacible.*

Demostración:

A partir del conjunto de cláusulas S y la asignación σ_P consideramos el conjunto $S_\perp(\sigma_P) = \{C \in S \text{ tales que } \sigma_P \not\models C\}$. Este conjunto no contiene la cláusula vacía, pues dicha cláusula no está en S, por tanto, por el teorema 9.17, existe un conjunto representante minimal de $S_\perp(\sigma_P)$. Sea R uno de estos representantes minimales.

Si para algún literal L se tiene $L, \overline{L} \in R$, entonces existen C_L y $C_{\overline{L}}$ en $S_\perp(\sigma_P)$ tales que $L \in C_L$ y $\overline{L} \in C_{\overline{L}}$. Se tiene que $\sigma_P \models L$ o $\sigma_P \models \overline{L}$ por tanto $\sigma_P \models C_L$ o $\sigma_P \models C_{\overline{L}}$ y esto contradice el hecho de que $C_L, C_{\overline{L}} \in S_\perp(\sigma_P)$. Por tanto R no contiene literales complementarios.

A partir de R y σ_P definimos la siguiente asignación:

$$\delta(x) = \begin{cases} \top & \text{si } x \in R & \text{(a)} \\ \top & \text{si } \neg x \notin R \text{ y } \sigma_P(x) = \top & \text{(b)} \\ \bot & \text{en otro caso} & \text{(c)} \end{cases}$$

Dado $C \in S$, demostraremos que $\delta \models C$ por inducción en el número de literales de C que son ciertos en σ_P:

1. Si el número de literales de C que son ciertos en σ_P es 0, entonces $\sigma_P \not\models C$ y $C \in S_\bot(\sigma_P)$. Como R es un conjunto representante minimal de $S_\bot(\sigma_P)$, existe $L \in C \cap R$:

   (a) Si L es un literal positivo, $L = x$, entonces, por la definición (a) de δ, se tiene que $\delta(x) = \top$ y, por tanto, $\delta \models L$ y $\delta \models C$.

   (b) Si L es un literal negativo, $L = \neg x$, entonces $\neg x \in R$ y, como R no contiene literales complementarios, $x \notin R$. Luego, por la definición (c) de δ se tiene que $\delta(x) = \bot$ y, por tanto, $\delta \models L$ y $\delta \models C$.

2. Supongamos que el número de literales de C que son ciertos en σ_P es $n > 0$ y que δ es modelo de cualquier cláusula en la que el número de literales que son ciertos en σ_P es menor que n.

   Sea $L \in C$ tal que $\sigma_P \models L$, este literal existe pues $n > 0$.

   (a) Si $\overline{L} \notin R$:

      i. Si L es un literal positivo, $L = x$, entonces, por la definición (b) de δ, se tiene que $\delta(x) = \top$ y, por tanto, $\delta \models L$ y $\delta \models C$.

      ii. Si L es un literal negativo, $L = \neg x$, entonces $x = \overline{L} \notin R$ y $\sigma_P(x) = \bot$ pues $\sigma_P \models L = \neg x$. Luego, por la definición (c) de δ, se tiene $\delta(x) = \bot$ y, por tanto, $\delta \models \neg x = L$ y $\delta \models C$.

   (b) Si $\overline{L} \in R$ entonces, como R es un conjunto representante minimal de $S_\bot(\sigma_P)$, existe $D \in S_\bot(\sigma_P)$ tal que $D \cap R = \{\overline{L}\}$.

   Se tienen $C, D \in S$ tales que $\sigma_P \models C$, pues $\sigma_P \models L$ y $L \in C$, y $\sigma_P \models D$. De esta forma, C y D cumplen la condición de resolución y, como S es saturado por resolución condicionada, $L \in C$ y $\overline{L} \in D$, se tiene que la resolvente de C y D con respecto a L está en S: $(C - \{L\}) \cup (D - \{\overline{L}\}) \in S$.

   Como $D \in S_\bot(\sigma_P)$ entonces ningún elemento de $D - \{\overline{L}\}$ es cierto en σ_P. Por tanto, el número de literales de $(C - \{L\}) \cup (D - \{\overline{L}\}) \in S$ que son ciertos en σ_P es menor que n, pues L es cierto en σ_P. Por hipótesis de inducción se tiene que $\delta \models (C - \{L\}) \cup (D - \{\overline{L}\})$.

   Supongamos que $\delta \models (D - \{\overline{L}\})$. Sea $L' \in (D - \{\overline{L}\})$ tal que $\delta \models L'$:

i. Si L' es un literal positivo, $L' = x$, entonces, como $D \cap R = \{\overline{L}\}$, se tiene que $x \notin R$. Además $\sigma(x) = \bot$, pues $x = L' \in D \in S_\bot(\sigma_P)$. De esta forma, por la definición (c) de δ, se tiene que $\delta(L') = \delta(x) = \bot$, lo que supone una contradicción.

ii. Si L' es un literal negativo, $L' = \neg x$, entonces, como $D \cap R = \{\overline{L}\}$, se tiene que $\neg x \notin R$. Además $\sigma(\neg x) = \bot$, pues $\neg x = L' \in D \in S_\bot(\sigma_P)$, por tanto $\sigma(x) = \top$. De esta forma, por la definición (b) de δ, se tiene que $\delta(x) = \top$ y $\delta(L') = \delta(\neg x) = \bot$, lo que supone una contradicción.

Luego $\delta \not\models (D - \{\overline{L}\})$ y, como $\delta \models (C - \{L\}) \cup (D - \{\overline{L}\})$, se verifica que $\delta \models (C - \{L\})$. Luego $\delta \models C$.

De esta forma, para todo $C \in S$, $\delta \models C$. Por tanto δ es un modelo de S y S es satisfacible.

□

La demostración se basa en la existencia de una asignación σ_P que guarda cierta relación con la condición de resolución P. En la formalización hemos asumido la existencia de una función constante que proporciona tal asignación. Dicha función es `asignacion` y su existencia se ha asumido en el mismo `encapsulado` en el que se asumió la existencia de la función `condicion`. La propiedad de `asignacion` con respecto a la condición de resolución es la siguiente:

288
```
(defthm propiedad-fundamental-condicion
  (implies (and (es-clausula D)
                (es-clausula C)
                (not (modelo-clausula (asignacion) D))
                (modelo-clausula (asignacion) C))
           (condicion C D)))
```

Aunque esta propiedad de la función `asignacion` es necesaria para demostrar el teorema de completitud, la mayor parte de la formalización se ha realizado para una asignación cualquiera, particularizando para la asignación proporcionada por la función `asignacion` sólo en aquellos eventos en donde es absolutamente necesario. De esta forma el conjunto $S_\bot$ se ha definido para cualquier asignación σ:

289
```
(defun coleccion-clausulas-falsas-asignacion (sigma S)
  (cond
    ((endp S) S)
    ((not (modelo-clausula sigma (car S)))
     (cons (car S)
           (coleccion-clausulas-falsas-asignacion sigma (cdr S))))
    (t (coleccion-clausulas-falsas-asignacion sigma (cdr S)))))
```

La construcción de la asignación δ se formaliza a partir de cualquier asignación σ y cualquier conjunto de literales R. El hecho de que sea un conjunto representante minimal de $S_\perp(\sigma)$ no afecta a la construcción. La función `nueva-asignacion` proporciona la asignación δ. Para ello obtiene los literales positivos del conjunto `R` (usando la función `literales-positivos`) y los literales positivos ciertos en la asignación `sigma` cuyo complementario no aparece en `R` (usando la función `literales-representante-asignacion`), y construye una asignación en la que todos estos literales son ciertos (usando la función `genera-asignacion-positiva` presentada en *89*)

290

```
(defun nueva-asignacion (R sigma)
  (genera-asignacion-positiva
    (append (literales-positivos R)
            (literales-representante-asignacion R sigma sigma))))

(defun literales-positivos (R)
  (cond ((endp R) nil)
        ((es-simbolo-proposicional (car R))
         (cons (car R) (literales-positivos (cdr R))))
        (t (literales-positivos (cdr R)))))

(defun literales-representante-asignacion (R alist sigma)
  (cond ((endp alist) nil)
        ((and (es-simbolo-proposicional (car (car alist)))
              (modelo sigma (car (car alist)))
              (not (member-equal
                     (complementario (car (car alist))) R)))
         (cons (car (car alist))
               (literales-representante-asignacion
                 R (cdr alist) sigma)))
        (t (literales-representante-asignacion
             R (cdr alist) sigma))))
```

Para desarrollar la prueba tenemos que definir una función cuyo esquema de inducción asociado se corresponda con el utilizado en la demostración. Al igual que en los casos anteriores esta función se puede definir para una asignación cualquiera σ. El caso base de la inducción se tiene para aquellas cláusulas que son falsas en σ, es decir, en las que no hay ningún literal cierto. En el caso inductivo, dada una cláusula C con un literal L cierto en σ, basta con suponer la propiedad para todas aquellas cláusulas de S que no contengan a L. Así, la demostración se desarrolla como sigue:

1. Primero se demuestra la propiedad para todas las cláusulas de $S_\perp(\sigma)$.

2. La propiedad para todas las cláusulas en las que el único literal cierto en σ sea L_1 se demuestra a partir del caso anterior.

3. La propiedad para todas las cláusulas en las que los únicos literales ciertos en σ sean L_1 y L_2 se demuestra a partir de los casos anteriores.

4. La propiedad para todas las cláusulas en las que los únicos literales ciertos en σ sean L_1, L_2 y L_3 se demuestra a partir de los casos anteriores, etc.

La inducción se realiza en función del número de literales de una cláusula que son ciertos en una asignación. Para ello definimos la función `induccion-literal-cierto`:

291
```
(defun induccion-literal-cierto (sigma S)
  (if (clausula-con-literal-cierto sigma S)
      (induccion-literal-cierto
       sigma (reduce-coleccion-literal
              S (literal-cierto-clausula
                  sigma (clausula-con-literal-cierto sigma S))))
      t))
```

En la llamada recursiva de esta función, se eliminan de un conjunto de cláusulas `S` todos los elementos en los que aparece un determinado literal cierto en la asignación `sigma`. La función `clausula-con-literal-cierto` busca en `S` una cláusula válida en la asignación `sigma`, la función `literal-cierto-clausula` devuelve un literal válido de esa cláusula y la función `reduce-coleccion-literal` elimina de `S` todos los elementos en los que aparece dicho literal.

292
```
(defun reduce-coleccion-literal (S L)
  (cond ((endp S) S)
        ((member-equal L (car S))
         (reduce-coleccion-literal (cdr S) L))
        (t (cons (car S) (reduce-coleccion-literal (cdr S) L)))))

(defun clausula-con-literal-cierto (sigma S)
  (cond ((endp S) nil)
        ((literal-cierto-clausula sigma (car S))
         (car S))
        (t (clausula-con-literal-cierto sigma (cdr S)))))

(defun literal-cierto-clausula (sigma C)
  (cond ((endp C) nil)
        ((modelo sigma (car C))
         (car C))
        (t (literal-cierto-clausula sigma (cdr C)))))
```

El esquema de inducción asociado a la función `induccion-literal-cierto` es el siguiente:

```
(IMPLIES (NOT (CLAUSULA-CON-LITERAL-CIERTO SIGMA S))
         (:P SIGMA S))

(IMPLIES (AND (CLAUSULA-CON-LITERAL-CIERTO SIGMA S)
               (:P SIGMA (REDUCE-COLECCION-LITERAL S L)))
         (:P SIGMA S))
```

Donde `:P` es la propiedad que se quiere demostrar y `L` es el literal cierto en `sigma` obtenido con las funciones `literal-cierto-clausula` y `clausula-con-literal-cierto`. El primer caso del esquema de inducción se corresponde con el caso base: en `S` no hay cláusulas con literales ciertos en `sigma`. El segundo caso asume la propiedad `:P` para el conjunto `(REDUCE-COLECCION-LITERAL S L)`, para demostrar la propiedad `:P` para el conjunto `S`.

Una vez definida la función que justifica el esquema de inducción, tenemos que establecer los dos casos de la demostración del teorema 9.18.

El caso base de la prueba se formaliza de la siguiente forma:

293
```
(defthm nueva-asignacion-clausula
  (implies (and (es-forma-clausal S)
                (conjunto-representante-minimal R S)
                (not (clausula-con-literal-cierto sigma S))
                (not (contiene-clausula-vacia S))
                (member-equal C S))
           (modelo-clausula (nueva-asignacion R sigma) C)))
```

La prueba de este evento se realiza siguiendo las líneas de la demostración del teorema 9.18: se obtiene un literal en la intersección de `C` y `R`, y se demuestra que, tanto si es un literal positivo como negativo, la asignación `sigma` es modelo de dicho literal. Obsérvese que este evento se establece para cualquier asignación `sigma`, por tanto, también es cierto para la asignación asociada a la condición de resolución.

El caso inductivo es más complicado, veamos los principales eventos relacionados con su automatización. El caso 2-(a) se formaliza con el siguiente evento:

294
```
(defthm modelo-nueva-asignacion-clausula-literal-C-not-member
  (implies (and (member-equal L C)
                (es-clausula C)
                (modelo sigma L)
                (not (member-equal (complementario L) R)))
           (modelo-clausula (nueva-asignacion R sigma) C)))
```

La prueba se basa en el hecho de que, en la situación descrita por las hipótesis del evento anterior, la asignación `(nueva-asignacion R sigma)` es modelo del literal `L` y, por tanto, modelo de la cláusula `C`. Al igual que en el caso base de la demostración, este evento se establece para cualquier asignación `sigma`, por tanto, también es cierto para la asignación asociada a la condición de resolución.

La parte final del caso 2-(b), es decir, la prueba de que δ no es modelo de $(D - \{\overline{L}\})$ y, si es modelo de la resolvente entre C y D, tiene que ser modelo de C, se formaliza con los siguientes eventos. El primero afirma que `(nueva-asignacion R sigma)` no es modelo de `(elimina-elemento L D)` y el segundo que, si `(nueva-asignacion R sigma)` es modelo de la resolvente entre `C` y `D` entonces es modelo de `C`:

295

```
(defthm modelo-clausula-nueva-asignacion-elimina-elemento
  (implies (and (es-literal L)
                (member-equal L D)
                (member-equal L R)
                (es-clausula D)
                (conjunto-unitario (interseccion R D))
                (not (modelo-clausula sigma D)))
           (not (modelo-clausula (nueva-asignacion R sigma)
                                 (elimina-elemento L D)))))

(defthm modelo-clausula-nueva-asignacion-caso-inductivo-aux
  (implies (and (conjunto-unitario (interseccion R D))
                (member-equal (complementario L) D)
                (member-equal (complementario L) R)
                (not (modelo-clausula sigma D))
                (es-clausula C)
                (es-clausula D)
                (member-equal L C)
                (condicion C D)
                (modelo-forma-clausal
                  (nueva-asignacion R sigma)
                  (resolucion-cond-clausulas C D)))
           (modelo-clausula (nueva-asignacion R sigma) C)))
```

Obsérvese que en el segundo evento se incluye la hipótesis `(condicion C D)`, para asegurar que se puede utilizar la resolución condicionada entre `C` y `D`. De esta forma el resultado es cierto para cualquier asignación `sigma`. La particularización para la asignación asociada a la condición de resolución permite eliminar dicha hipótesis. Nótese también que el valor devuelto por `resolucion-cond-clausulas` es la lista formada por la resolvente y por eso se utiliza `modelo-forma-clausal` para expresar la hipótesis acerca de su valor de verdad en la asignación

`(nueva-asignacion R sigma)`. Finalmente, nótese también que no se incluye ninguna hipótesis que afirme que la resolvente entre `C` y `D` existe, podría ser una cláusula con literales complementarios e incluso en ese caso el resultado es cierto.

Una vez demostrados estos resultados, el caso inductivo se formaliza de la siguiente forma:

296
```
(defthm nueva-asignacion-clausula-inductivo
  (implies (and (clausula-con-literal-cierto (asignacion) S)
                (modelo-forma-clausal
                  (nueva-asignacion R (asignacion))
                  (reduce-coleccion-literal
                    S (literal-cierto-clausula
                        (asignacion)
                        (clausula-con-literal-cierto
                          (asignacion) S))))
                (es-forma-clausal S)
                (saturado-cond-conjunto S)
                (not (contiene-clausula-vacia S))
                (conjunto-representante-minimal
                  R (coleccion-clausulas-falsas-asignacion
                      (asignacion) S))
                (member-equal C S))
           (modelo-clausula
             (nueva-asignacion R (asignacion)) C))
```

Obsérvese que para expresar el resultado se utiliza la asignación asociada a la condición de resolución, proporcionada por la función `asignacion`, y es necesario incluir una hipótesis, que asegura que `R` es un conjunto representante minimal de las cláusulas de `S` falsas en la asignación proporcionada por la función `asignacion`. Nótese también que la hipótesis de inducción es la segunda de las hipótesis de este evento.

Finalmente, el evento que formaliza el teorema 9.18 es el siguiente:

297
```
(defthm completitud-resolucion-bezem
  (implies (and (es-forma-clausal S)
                (saturado-cond-conjunto S)
                (not (contiene-clausula-vacia S))
                (conjunto-representante-minimal
                  R (coleccion-clausulas-falsas-asignacion
                      (asignacion) S)))
           (modelo-forma-clausal
             (nueva-asignacion R (asignacion)) S)))
```

Para su prueba en el sistema se ha indicado que se debe usar el esquema de inducción proporcionado por la función `induccion-literal-cierto` (ver sección 2.2.4).

9.2.3 Completitud de $INSAT_{\mathcal{R}}$

Una vez demostrado el teorema de completitud de la resolución condicionada, el resultado de completitud del algoritmo $INSAT_{\mathcal{R}}$ se obtiene de forma inmediata:

Teorema 9.19 (Completitud de $INSAT_{\mathcal{R}}$) *Dada un conjunto de cláusulas S, si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces S es satisfacible.*

Demostración:
Si $INSAT_{\mathcal{R}}(S) \neq \mathbf{t}$ entonces, por el teorema 9.11, $\mathcal{SR}(S)$ es un conjunto de cláusulas que, por el teorema 9.12, no contiene la cláusula vacía y, por el teorema 9.14, está saturado por resolución condicionada. Por tanto, por el teorema 9.18, $\mathcal{SR}(S)$ es satisfacible y, puesto que $S \subset \mathcal{SR}(S)$ por el teorema 9.13, S también lo es.

□

La formalización de este resultado es la siguiente:

298
```
(defthm completitud-INSAT-resolucion
  (implies (and (es-forma-clausal S)
                (not (contiene-clausula-vacia S))
                (not (INSAT-resolucion S)))
           (modelo-forma-clausal
             (nueva-asignacion
               (construye-conjunto-representante-minimal
                 (coleccion-clausulas-falsas-asignacion
                   (asignacion) (saturacion-resolucion S)))
               (asignacion))
            S))
```

La demostración del mismo se obtiene a partir de los eventos presentados en *274*, *275*, *277*, *297* y *276*. Obsérvese que la asignación que se utiliza para asegurar la satisfacibilidad de `S` es la que se obtiene aplicando la función `nueva-asignacion` a la asignación asociada a la condición de resolucion, `asignacion`, y un conjunto representante minimal de las cláusulas de `(saturacion-resolucion S)` falsas en dicha asignación.

9.3 Teoría genérica e instancias

En esta sección presentamos una teoría genérica para facilitar la reutilización de los resultados anteriores. Esta teoría se define para las funciones `condicion` y `asignacion`. A partir de estas funciones se construyen de forma automática las implementaciones de los algoritmos $\mathcal{SR}$ y $INSAT_{\mathcal{R}}$ y se demuestran las propiedades de corrección y completitud de este último. A continuación se desarrollan instancias de esta teoría genérica para la resolución binaria, positiva, negativa, resolución semántica y con soporte. Los eventos presentados en esta sección se encuentran en el libro `resolucion-gen.lisp`.

9.3.1 La teoría genérica `*resolucion*`

Utilizando las herramientas de construcción de teorías genéricas presentadas en la sección 6.2, se ha definido la teoría genérica `*resolucion*` para facilitar la reutilización de los resultados anteriores. Las funciones para las que se define esta teoría son `condicion` y `asignacion`.

Las propiedades asumidas para estas funciones son las presentadas en *256* y *288*. Esta teoría proporciona, entre otros eventos, las definiciones de las funciones que implementan el algoritmo de saturación por resolución condicionada $\mathcal{SR}$ y el algoritmo de decisión de insatisafibilidad basado en resolución $INSAT_{\mathcal{R}}$ y los resultados de corrección y completitud mostrados en *273* y *298*.

Para utilizar esta teoría bastará con proporcionar versiones concretas de las funciones para las que está definida y demostrar que cumplen las propiedades asumidas en la misma. Una vez hecho esto se puede utilizar la herramienta de instanciación genérica presentada en la sección 6.2 para obtener las versiones concretas y verificadas de los algoritmos $\mathcal{SR}$ y $INSAT_{\mathcal{R}}$.

9.3.2 Instancias

Veamos a continuación la condición de resolución y la asignación asociada a la misma, correspondientes a distintas estrategias de resolución. Para cada una de ellas se obtiene un procedimiento verificado de decisión de insatisfacibilidad.

9.3.2.1 Resolución binaria

La resolución binaria se basa en el concepto de resolvente presentado en la definición 9.1. En este caso la condición de resolución es siempre cierta y cualquier asignación sirve para instanciar la teoría genérica `*resolucion*`:

299
```
(defun cond-binaria (C-1 C-2)
  (declare (ignore C-1 C-2))
  t)

(defcong igual-conjunto iff (cond-binaria C-1 C-2) 1)

(defcong igual-conjunto iff (cond-binaria C-1 C-2) 2)

(defthm cond-binaria-conmuta
  (iff (cond-binaria C-1 C-2)
       (cond-binaria C-2 C-1)))

(defthm cond-binaria-boolean-p
  (booleanp (cond-binaria C-1 C-2)))

(defun asig-binaria ()
  nil)

(defthm propiedad-fundamental-cond-binaria
  (implies (and (es-clausula C-2)
                (es-clausula C-1)
                (not (modelo-clausula (asig-binaria) C-2)))
           (cond-binaria C-1 C-2)))
```

El siguiente evento realiza una instancia de la teoría genérica `*resolucion*` utilizando las funciones anteriores.

300
```
(definstancia-*resolucion*
  ((condicion   cond-binaria)
   (asignacion  asig-binaria))
  "-binaria")
```

Una vez evaluado este evento, se obtienen automáticamente las funciones que implementan los algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución binaria. Estas funciones son las siguientes:

```
(DEFUN SATURACION-RESOLUCION-BINARIA (S)
       (RESOLUCION-COND-SATURACION-BINARIA S NIL))

(DEFUN INSAT-RESOLUCION-BINARIA (S)
       (EQUAL (SATURACION-RESOLUCION-BINARIA S)
              T))

(DEFUN RESOLUCION-COND-SATURACION-BINARIA (S-1 S-2)
  (IF (ENDP S-1)
      S-2
      (LET ((RESOLVENTES
              (RESOLUCION-COND-CLAUSULA-CONJUNTO-BINARIA
                (CAR S-1) S-2 NIL (APPEND S-1 S-2))))
           (IF (CONTIENE-CLAUSULA-VACIA RESOLVENTES)
               T
               (RESOLUCION-COND-SATURACION-BINARIA
                 (APPEND RESOLVENTES (CDR S-1))
                 (CONS (CAR S-1) S-2))))))
```

Además de generar automáticamente estas funciones, al instanciar la teoría genérica `*resolucion*`, también se obtienen de forma automática los eventos que establecen las propiedades de corrección y completitud del algoritmo de decisión de insatisfacibilidad:

```
(DEFTHM CORRECCION-INSAT-RESOLUCION-BINARIA
  (IMPLIES (AND (ES-FORMA-CLAUSAL S)
                (INSAT-RESOLUCION-BINARIA S))
           (NOT (MODELO-FORMA-CLAUSAL SIGMA S))))

(DEFTHM COMPLETITUD-INSAT-RESOLUCION-BINARIA
  (IMPLIES (AND (ES-FORMA-CLAUSAL S)
                (NOT (CONTIENE-CLAUSULA-VACIA S))
                (NOT (INSAT-RESOLUCION-BINARIA S)))
           (MODELO-FORMA-CLAUSAL
             (NUEVA-ASIGNACION
               (CONSTRUYE-CONJUNTO-REPRESENTANTE-MINIMAL
                 (COLECCION-CLAUSULAS-FALSAS-ASIGNACION
                   (ASIG-BINARIA)
                   (SATURACION-RESOLUCION-BINARIA S)))
             (ASIG-BINARIA))
           S)))
```

9.3.2.2 Resolución positiva

Definición 9.20 *Una cláusula positiva es aquella en la que sólo aparecen literales positivos.*

```
(defun es-clausula-positiva (C)
  (if (endp C)
      t
      (and (es-literal-positivo (car C))
           (es-clausula-positiva (cdr C)))))
```
301

Definición 9.21 *Dadas dos cláusulas C_1 y C_2, diremos que C se obtiene por resolución positiva a partir de C_1 y C_2 si C es la resolvente entre C_1 y C_2 con respecto a algún literal L y una de las dos cláusulas padre es positiva.*

De esta forma la condición de resolución positiva comprueba que alguna de las dos cláusulas que recibe como argumento es positiva. Esta función verifica las propiedades presentadas en 256.

```
(defun cond-positiva (C-1 C-2)
  (or (es-clausula-positiva C-1)
      (es-clausula-positiva C-2)))

(defcong igual-conjunto iff (cond-positiva C-1 C-2) 1)

(defcong igual-conjunto iff (cond-positiva C-1 C-2) 2)

(defthm cond-positiva-conmuta
  (iff (cond-positiva C-1 C-2)
       (cond-positiva C-2 C-1)))

(defthm cond-positiva-boolean-p
  (booleanp (cond-positiva C-1 C-2)))
```
302

Asociada a la resolución positiva consideramos la asignación que hace falsos todos los símbolos proposicionales y, por tanto, todas las cláusulas positivas. De esta forma, si una cláusula es falsa en esta asignación, entonces se trata de una cláusula positiva y, junto con cualquier otra cláusula, cumple la condición de resolución positiva.

303
```
(defun asig-positiva ()
  nil)

(defthm propiedad-fundamental-cond-positiva
  (implies (and (es-clausula C-1)
                (es-clausula C-2)
                (not (modelo-clausula (asig-positiva) C-2)))
           (cond-positiva C-1 C-2)))
```

El siguiente evento realiza una instancia de la teoría genérica `*resolucion*` utilizando las funciones anteriores.

304
```
(definstancia-*resolucion*
  ((condicion   cond-positiva)
   (asignacion  asig-positiva))
  "-positiva")
```

Una vez evaluado este evento, se obtienen automáticamente las funciones que implementan los algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución positiva. También se obtienen los eventos que establecen las propiedades de corrección y completitud del algoritmo de decisión de insatisfacibilidad.

9.3.2.3 Resolución negativa

Definición 9.22 *Una cláusula negativa es aquella en la que sólo aparecen literales negativos.*

305
```
(defun es-clausula-negativa (C)
  (if (endp C)
      t
      (and (es-literal-negativo (car C))
           (es-clausula-negativa (cdr C)))))
```

Definición 9.23 *Dadas dos cláusulas C_1 y C_2, diremos que C se obtiene por resolución negativa a partir de C_1 y C_2 si C es la resolvente entre C_1 y C_2 con respecto a algún literal L y una de las dos cláusulas padre es negativa.*

De esta forma la condición de resolución negativa comprueba que alguna de las dos cláusulas que recibe como argumento es negativa. Esta función verifica las propiedades presentadas en 256.

306
```
(defun cond-negativa (C-1 C-2)
  (or (es-clausula-negativa C-1)
      (es-clausula-negativa C-2)))

(defcong igual-conjunto iff (cond-negativa C-1 C-2) 1)

(defcong igual-conjunto iff (cond-negativa C-1 C-2) 2)

(defthm cond-negativa-conmuta
  (iff (cond-negativa C-1 C-2)
       (cond-negativa C-2 C-1)))

(defthm cond-negativa-boolean-p
  (booleanp (cond-negativa C-1 C-2)))
```

Asociada a la resolución negativa consideramos la asignación que hace ciertos todos los símbolos proposicionales y, por tanto, hace falsas todas las cláusulas negativas. De esta forma, si una cláusula es falsa en esta asignación, entonces es un cláusula negativa y, junto con cualquier otra cláusula, cumple la condición de resolución negativa.

Debido a que las asignaciones se representan con listas de asociación finitas y a que el valor por defecto, para aquellos símbolos que no aparezcan explícitamente en una asignación, es $\bot$, no podemos representar la asignación que hace ciertos todos los símbolos proposicionales. Sin embargo, sí podemos asumir la existencia de dicha asignación, estableciendo esta propiedad como un axioma[1]. Una vez hecho esto se puede demostrar que, si una cláusula es falsa en esta asignación, entonces, junto con cualquier otra cláusula, cumple la condición de resolución negativa.

307
```
(defstub asig-negativa () => *)

(defaxiom asig-negativa-es-positiva
  (equal (valor F (asig-negativa)) *V*))

(defthm propiedad-fundamental-cond-negativa
  (implies (and (es-clausula C-1)
                (es-clausula C-2)
                (not (modelo-clausula (asig-negativa) C-2)))
           (cond-negativa C-1 C-2)))
```

[1]No se puede hacer en un encapsulado de ACL2 pues no existe una función testigo.

En la formalización anterior, el comando `defstub` sirve para introducir en el sistema un nuevo nombre de función constante, `asig-negativa`. Mediante el comando `defaxiom` asumimos que la función `asig-negativa` proporciona una asignación en la que todas las fórmulas son falsas. El uso de axiomas en ACL2 es peligroso pues puede hacer que el sistema quede inconsistente. Sin embargo, en este caso es la única posibilidad para poder utilizar la teoría genérica definida para las funciones `condicion` y `asignacion`.

El siguiente evento realiza una instancia de la teoría genérica `*resolucion*` utilizando las funciones anteriores.

308
```
(definstancia-*resolucion*
  ((condicion   cond-negativa)
   (asignacion  asig-negativa))
  "-negativa")
```

Una vez evaluado este evento, se obtienen automáticamente las funciones que implementan los algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución negativa. También se obtienen los eventos que establecen las propiedades de corrección y completitud del algoritmo de decisión de insatisfacibilidad. Nótese que la función `asig-negativa` sólo interviene en la prueba de la completitud y no es necesaria para utilizar los algoritmos.

9.3.2.4 Resolución semántica

Definición 9.24 *Dada una asignación σ y dos cláusulas C_1 y C_2, diremos que C se obtiene por resolución semántica a partir de C_1 y C_2 si C es la resolvente entre C_1 y C_2 con respecto a algún literal L, una de las dos cláusulas padre es cierta en σ y la otra es falsa.*

En esta estrategia de resolución se parte de una asignación fija, con respecto a la que se establece la condición de resolución. Consideramos en un encapsulado de ACL2, una asignación cualquiera proporcionada por la función constante `asig-semantica`. La única propiedad que hay que exigir a esta función es que el valor que devuelve es una lista de asociación.

309
```
(defthm alistp-asig-semantica
  (alistp (asig-semantica)))
```

Con respecto a la asignación proporcionada por la función `asig-semantica` definimos la siguiente condición de resolución:

310
```
(defun cond-semantica (C-1 C-2)
  (or (and (modelo-clausula (asig-semantica) C-1)
           (not (modelo-clausula (asig-semantica) C-2)))
      (and (modelo-clausula (asig-semantica) C-2)
           (not (modelo-clausula (asig-semantica) C-1)))))
```

Esta condición verifica las propiedades mostradas en *256* y, por su definición, se verifica que si una cláusula es falsa en la asignación proporcionada por la función `asig-semantica` y otra es cierta, entonces ambas cláusulas verifican la condición de resolución semántica.

311
```
(defcong igual-conjunto iff (cond-semantica C-1 C-2) 1)

(defcong igual-conjunto iff (cond-semantica C-1 C-2) 2)

(defthm cond-semantica-conmuta
  (iff (cond-semantica C-1 C-2)
       (cond-semantica C-2 C-1)))

(defthm cond-semantica-boolean-p
  (booleanp (cond-semantica C-1 C-2)))

(defthm propiedad-fundamental-cond-semantica
  (implies (and (es-clausula C-1)
                (es-clausula C-2)
                (modelo-clausula (asig-semantica) C-1)
                (not (modelo-clausula (asig-semantica) C-2)))
           (cond-semantica C-1 C-2)))
```

El siguiente evento realiza una instancia de la teoría genérica `*resolucion*` utilizando las funciones anteriores.

312
```
(definstancia-*resolucion*
  ((condicion   cond-semantica)
   (asignacion  asig-semantica))
  "-semantica")
```

Una vez evaluado este evento, se obtienen automáticamente las funciones que implementan los algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución semántica. También se obtienen los eventos que establecen las propiedades de corrección y completitud del algoritmo de decisión de insatisfacibilidad. Nótese que es necesario proporcionar un valor concreto para la función `asig-semantica` para poder utilizar los algoritmos, pues la asignación que proporciona es necesaria para comprobar la condición de resolución.

9.3.2.5 Resolución con conjunto soporte

Definición 9.25 *Un subconjunto T de un conjunto de cláusulas S es un conjunto soporte de S si $S-T$ es satisfacible. Dadas dos cláusulas C_1 y C_2, diremos que C se obtiene por resolución con soporte T a partir de C_1 y C_2 si C es la resolvente entre C_1 y C_2 con respecto a algún literal L y las dos cláusulas padre no pertenecen simultáneamente a S.*

En esta estrategia de resolución se parte de un conjunto de cláusulas satisfacible $(S - T)$ y la condición de resolución se establece con respecto a dicho conjunto. Consideramos en un encapsulado de ACL2, un conjunto satisfacible de cláusulas proporcionado por la función constante `S-soporte`. Las propiedades que debe cumplir esta función son que el valor que devuelve es un conjunto de cláusulas y que dicho conjunto es satisfacible. Para esto último consideramos en el mismo encapsulado, una función constante `asig-soporte` que proporciona una asignación modelo de dicho conjunto. El valor devuelto por esta función ha de ser una lista de asociación.

313

```
(defthm alistp-asig-soporte
  (alistp (asig-soporte)))

(defthm soporte-es-forma-clausal
  (es-forma-clausal (S-soporte)))

(defthm soporte-es-satisfacible
  (modelo-forma-clausal (asig-soporte) (S-soporte)))
```

De esta forma la condición de resolución con soporte comprueba que una de las dos cláusulas que recibe como argumento no pertenece al conjunto proporcionado por la función `S-soporte`. Esta función verifica las propiedades presentadas en 256.

314
```
(defun cond-soporte (C-1 C-2)
  (or (not (conjunto-miembro C-1 (S-soporte)))
      (not (conjunto-miembro C-2 (S-soporte)))))

(defcong igual-conjunto iff (cond-soporte C-1 C-2) 1)

(defcong igual-conjunto iff (cond-soporte C-1 C-2) 2)

(defthm cond-soporte-conmuta
  (iff (cond-soporte C-1 C-2)
       (cond-soporte C-2 C-1)))

(defthm cond-soporte-boolean-p
  (booleanp (cond-soporte C-1 C-2)))
```

Asociada a esta condición de resolución consideramos la asignación proporcionada por la función `asig-soporte`. De esta forma, si una cláusula es falsa en esta asignación, dicha cláusula no pertenece al conjunto proporcionado por `S-soporte` y, junto con cualquier otra cláusula, cumple la condición de resolución.

315
```
(defthm propiedad-fundamental-cond-soporte
  (implies (and (es-clausula C-1)
                (es-clausula C-2)
                (not (modelo-clausula (asig-soporte) C-2)))
           (cond-soporte C-1 C-2)))
```

El siguiente evento realiza una instancia de la teoría genérica `*resolucion*` utilizando las funciones anteriores.

316
```
(definstancia-*resolucion*
  ((condicion   cond-soporte)
   (asignacion  asig-soporte))
  "-soporte")
```

Una vez evaluado este evento, se obtienen automáticamente las funciones que implementan los algoritmos de saturación y de decisión de insatisfacibilidad basados en resolución con conjunto soporte. También se obtienen los eventos que establecen las propiedades de corrección y completitud del algoritmo de decisión de insatisfacibilidad. Nótese que es necesario proporcionar un valor concreto para la función `S-soporte` para poder utilizar los algoritmos.

9.4 Ejemplos

En esta sección explicamos cómo se utilizan los procedimientos de decisión desarrollados en este capítulo, presentando datos sobre su evaluación en algunos de los problemas propuestos por Plaisted en [64]. Para distintos valores de N se considera el conjunto de cláusulas asociado al siguiente conjunto de fórmulas:

$$\begin{array}{cc} P_1 \wedge Q_1 \rightarrow P_2 & P_1 \wedge Q_1 \rightarrow Q_2 \\ P_2 \wedge Q_2 \rightarrow P_3 & P_2 \wedge Q_2 \rightarrow Q_3 \\ & \cdots \\ P_{N-1} \wedge Q_{N-1} \rightarrow P_N & P_{N-1} \wedge Q_{N-1} \rightarrow Q_N \\ P_1 \qquad Q_1 & \neg P_N \vee \neg Q_N \end{array}$$

Esta familia está definida en el libro `plaisted.lisp` del directorio `0-ejemplos` para los valores de $N = 5, 6, 7, 8, 9, 10$.

Una vez certificados los libros, ponemos en funcionamiento el sistema, evaluando la orden `acl2` en el directorio `9-resolucion`:

```
../calculos-proposicionales/9-resolucion> acl2
...

ACL2 Version 2.6.  Level 1.  Cbd
"../calculos-proposicionales/9-resolucion".

ACL2 !>
```

A continación incluimos el libro que contiene la teoría desarrollada en este capítulo. Los libros de los que éste depende son incluidos automáticamente por el sistema:

```
ACL2 !>(include-book "resolucion-gen")
```

En la tabla 9.1 se muestra información sobre el tiempo de comprobación de la insatisfacibilidad de los conjuntos de cláusulas asociados al problema de Plaisted [64], para $N = 5, 6, 7, 8, 9, 10$. Estos tiempos incluyen la evaluación del procedimiento de decisión `INSAT-resolucion-<condicion>`, para la resolución binaria, positiva y negativa.

Tamaño del problema	5	6	7	8	9	10
Resolución binaria	8.720	250.380	—	—	—	—
Resolución positiva	0.000	0.000	0.010	0.000	0.010	0.000
Resolución negativa	0.040	0.220	0.420	0.810	2.770	3.810

Tabla 9.1: Tiempos de evaluación para las fórmulas de Plaisted

Sumario

En este capítulo:

- Se ha presentado el concepto clásico de resolvente binaria y el de resolvente condicionada. Este último depende de una condición que da lugar a distintas estrategias de resolución. Se ha definido el concepto de conjunto de cláusulas saturado por resolución condicionada. Se han formalizado estos conceptos y se han demostrado algunas de sus propiedades más importantes.

- Se ha definido un algoritmo que realiza un proceso de saturación por resolución condicionada, es decir, a partir de un conjunto de cláusulas obtiene otro que contiene al original y es saturado por resolución condicionada. Se ha demostrado que este algoritmo termina para cualquier conjunto de cláusulas de partida.

- La aparición de la cláusula vacía en el proceso de saturación indica la insatisfacibilidad del conjunto de cláusulas inicial. Utilizando esta idea se ha desarrollado un procedimiento de decisión de insatisfacibilidad basado en resolución condicionada. Se ha demostrado que este procedimiento es correcto.

- Se han desarrollado conceptos relativos a conjuntos representantes de colecciones de conjuntos. Se han formalizado estos conceptos y se han demostrado algunas de sus propiedades.

- Se ha presentado la prueba de Bezem del teorema de completitud de la resolución condicionada. Este teorema se ha utilizado para demostrar la completitud del procedimiento de decisión de insatisfacibilidad basado en resolución condicionada. La prueba ha sido formalizada en ACL2.

- Se ha definido una teoría genérica que facilita la reutilización de los resultados presentados. Utilizando la herramienta de instanciación genérica se han obtenido automáticamente procedimientos verificados de decisión de insatisfacibilidad basados en resolución binaria, positiva, negativa, semántica y con conjunto soporte.

Capítulo 10

Conclusiones

En los capítulos precedentes se ha presentado una teoría computacional, desarrollada en ACL2, sobre cálculos proposicionales. Desde nuestro punto de vista, los principales objetivos conseguidos en esta memoria son los siguientes:

- Se ha realizado un modelo formal de la lógica proposicional implementado en Common Lisp. Este modelo sirve de base para la implementación y verificación formal de algoritmos específicos para esta lógica. La formalización realizada es robusta en el sentido de que puede ser modificada para adaptarse a distintas sintaxis, sin afectar al desarrollo de los algoritmos o a la prueba de sus propiedades.

- Se ha formalizado la semántica clásica y la semántica trivalorada fuerte de Kleene para la lógica clásica, comprobando a lo largo de la formalización de los distintos cálculos lógicos, que éstos se pueden utilizar indistintamente en ambas semánticas sin que esto afecte a sus propiedades de corrección y completitud.

- Se han formalizado cálculos lógicos basados en tableros semánticos y secuentes. Esta formalización es genérica en el sentido de que, en los procedimientos de decisión de satisfacibilidad basados en estos cálculos, se ha asumido la existencia de cierta función de selección con determinadas propiedades, cuya definición no se proporciona expresamente. Esta función de selección proporciona una forma de incorporar heurísticas a estos procedimientos.

- Se ha desarrollado un marco genérico para la definición de procedimientos de decisión de satisfacibilidad, en el que tanto el cálculo basado en tableros semánticos como el basado en secuentes se pueden expresar como casos particulares. Este marco genérico es interesante por dos razones. Por un lado, el formalismo teórico abstrae propiedades de los distintos métodos, a partir de las cuales su funcionamiento es similar. Estas propiedades son la base para demostrar la corrección y completitud del procedimiento de decisión de satisfacibilidad en el marco genérico. Por otro lado, la formalización del

marco genérico y el uso de unas herramientas de instanciación desarrolladas para tal fin, nos permiten obtener de forma automática funciones verificadas que implementan procedimientos de decisión de satisfacibilidad para distintos cálculos lógicos.

- Se han formalizado procedimientos de decisión de satisfacibilidad para conjuntos de cláusulas basados en el procedimiento de Davis y Putnam. Primero se ha demostrado que la regla de bifurcación es, por si sóla, un procedimento correcto y completo de decisión de satisfacibilidad para cláusulas y como un refinamiento de dicho procedimiento, se incorporan las reglas de eliminación de cláusulas unitarias y literales puros. Al igual que en los cálculos desarrollados previamente, se asume una función de selección que proporciona un literal para aplicar la regla de bifurcación en cada paso. La definición de esta función no se proporciona explícitamente. Para una función de selección espécifica, se demuestra que el procedimiento basado en bifurcación es equivalente al que incorpora las reglas de eliminación de cláusulas unitarias y literales puros y por tanto, este último también es correcto y completo.

- Se ha desarrollado un procedimiento de saturación por resolución condicionada, basado en una propiedad que condiciona la obtención de resolventes. De esta forma, varios refinamientos del procedimiento de resolución se pueden expresar como casos particulares de este procedimiento. Se ha demostrado que, si en el proceso de saturación por resolución condicionada de un conjunto de cláusulas, no aparece la cláusula vacía, entonces el conjunto de cláusulas original es satisfacible. Para ello se ha formalizado la prueba de Bezem del teorema de completitud condicionada que, a partir de una asignación asociada a la condición de resolución, construye un modelo del conjunto de cláusulas saturado. Esta formalización es la única automatización que conocemos de dicha prueba. Basado en este procedimiento se ha desarrollado un procedimiento de decisión de insatisfacibilidad para conjuntos de cláusulas, mediante la saturación del mismo por resolución condicinada. Se han demostrado las propiedades de corrección y completitud de este procedimiento.

A lo largo del desarrollo realizado, se han construido herramientas que extienden el sistema ACL2 para facilitar la tarea del usuario. Por un lado se ha desarrollado una herramienta, `defmul`, que, de forma automática, extiende una relación bien fundamentada definida sobre un conjunto A, a multiconjuntos de elementos de A. La extensión obtenida también es una relación bien fundamentada. Este herramienta es de gran utilidad para demostrar la terminación de funciones entre cuyos argumentos aparece una lista de elementos del conjunto A de los que algunos, en las llamadas recursivas, son reemplazados por otros más pequeños con respecto a la relación bien fundamentada en A. En este sentido, ha

sido utilizada en la prueba de terminación del procedimiento de transformación a forma clausal presentado en la sección 8.1.2 y en la prueba de terminación del procedimiento genérico de decisión de satisfacibilidad presentado en 7.

Por otro lado, el intensivo uso de la regla de instanciación funcional hace necesario disponer de una herramienta que automatice el proceso de construcción de un conjunto de eventos, cuya admisión por el sistema depende de un pequeño conjunto de funciones y propiedades de las mismas. El desarrollo de esta herramienta pone de manifiesto las posibilidades que tiene el sistema de representar y utilizar resultados de orden superior. Esta herramienta ha sido utilizada para obtener instancias concretas del marco genérico para el desarrollo de procedimientos de decisión de satisfacibilidad (capítulo 7 y sección 8.3), e instancias del procedimiento de decisión de insatisfacibilidad basado en resolución condicionada (sección 9.3).

Algunos datos de interés

En la tabla de la figura 10.1, se presentan algunos datos cuantitativos sobre la teoría desarrollada. La primera columna contiene el nombre de cada uno de los libros ACL2. Las siguientes tres columnas contienen, respectivamente, el número de líneas (excluyendo comentarios y líneas en blanco), de definiciones y de teoremas en cada uno de los libros. Estos datos dan una idea del "tamaño" de la teoría desarrollada en cada uno de los libros. La última columna indica el número de teoremas que necesitan alguna sugerencia para completar con éxito su demostración. La mayoría de estas indicaciones son para usar instancias de teoremas previos o para habilitar o deshabilitar algunas reglas asociadas a eventos.

El esfuerzo humano invertido en este trabajo es difícil de estimar, tanto más cuando se combina, en sus fases iniciales, con el aprendizaje del sistema. El desarrollo de un trabajo similar en el momento actual supondría menos horas de dedicación. Los resultados en los que se ha invertido más tiempo son la prueba de terminación del procedimiento de saturación por resolución condicionada y la automatización de la prueba de Bezem de la completitud de la resolución condicionada.

Trabajo futuro

La investigación presentada en esta memoria se puede continuar en varias direcciones:

- Existen gran variedad de cálculos y procedimientos para decidir la satisfacibilidad de una fórmula proposicional. El modelo formal de la lógica proposicional que hemos desarrollado parece un entorno adecuado para la formalización de estos cálculos y procedimientos, y la verificación automática de sus propiedades. Igualmente, es interesante estudiar hasta qué

Libro	Líneas	Definiciones	Teoremas	Consejos
sintaxis.lisp	323	22	45	12
semantica.lisp	113	11	9	1
semantica-K.lisp	122	12	8	1
tablas-de-verdad.lisp	171	13	16	7
tablas-de-verdad-K.lisp	257	15	25	11
uniforme.lisp	206	10	18	5
uniforme-K.lisp	198	10	18	5
tableros.lisp	349	13	34	12
tableros-K.lisp	355	13	34	12
secuentes.lisp	507	21	48	20
secuentes-K.lisp	520	22	48	20
multiconjuntos.lisp	944	28	149	39
defmul.lisp	886	31	29	16
teorias-genericas.lisp	115	15	2	2
listas.lisp	146	15	10	1
SAT-generico.lisp	335	16	37	13
SAT-generico-K.lisp	336	16	37	13
SAT-tableros.lisp	603	16	67	34
SAT-tableros-K.lisp	611	16	67	34
SAT-secuentes.lisp	427	20	36	9
SAT-secuentes-K.lisp	418	20	33	10
clausulas.lisp	277	16	39	11
clausulas-K.lisp	286	16	40	13
forma-clausal.lisp	371	12	47	17
forma-clausal-K.lisp	391	12	49	18
davis-putnam.lisp	621	24	82	21
davis-putnam-K.lisp	631	23	83	22
SAT-davisputnam.lisp	485	23	47	9
conjuntos.lisp	284	19	41	10
resolucion-sat.lisp	1069	27	124	46
representantes.lisp	229	11	25	11
resolucion-thm.lisp	861	13	92	50
resolucion-aux.lisp	333	0	36	13
resolucion-gen.lisp	489	20	50	11

Figura 10.1: Datos cuantitativos

punto el marco genérico proporcionado por los sistemas de transformación proposicionales es aplicable a otros cálculos y procedimientos de decisión.

- En el capítulo 8 se presentan formalizaciones de procedimientos de decisión basados en el método de Davis y Putnam. Este método es la base de gran cantidad de algoritmos de probada eficiencia para decidir la satisfacibilidad de un conjunto de cláusulas. Una diferencia fundamental entre esos algoritmos y los presentados en el capítulo 8 es que evitan la duplicidad de información que genera la regla de bifurcación. Esto se puede conseguir en ACL2 utilizando objetos de hebra simple[1]. Una interesante línea de investigación consistiría en utilizar estos objetos para desarrollar algoritmos verificados eficientes para resolver el problema SAT. De hecho ya hemos construido un algoritmo basado en estos objetos que mejora notablemente a los presentados en el capítulo 8, pero aún no hemos realizado su verificación.

- En los últimos años se están desarrollando nuevos modelos de computación: molecular, membranas, cuántica. Y uno de los primeros problemas que se suele abordar en estos modelos, quizá por ser NP–completo, es el problema de la satisfacibilidad de una fórmula proposicional. El modelo formal que hemos construido también es adecuado para verificar este tipo de experimentos realizados en modelos de computación no convencionales. En este sentido, hemos formalizado y verificado una versión recursiva del experimento de R.J. Lipton, [48], para resolver el problema de la satisfacibilidad de una fórmula proposicional, basado en el modelo de computación molecular restringido de Adleman [50]. Este trabajo ha sido presentado al tercer congreso internacional de usuarios de ACL2 [53].

- Otra línea interesante de investigación consiste en construir teorías computacionales sobre otras lógicas distintas a la proposicional clásica (modales, temporales, ...) y formalizar procedimientos de decisión para estas lógicas. Se estudiarían las características comunes a los procedimientos de decisión para estas lógicas y se intentaría extraer un marco genérico en el que expresarlos como instancias.

- De igual forma estamos en disposición de construir un modelo formal de la lógica de primer orden. En esta memoria hemos realizado una formalización de la lógica proposicional y en [66] se ha formalizado un algoritmo de unificación. Con estos resultados creemos que se puede afrontar con éxito el problema de formalizar en ACL2 la lógica de primer orden.

- En los libros ACL2 desarrollados en esta investigación se hace un uso intesivo de las características de orden superior del sistema. En el capítulo 6

[1]Del inglés *single-threaded objects*.

presentamos una herramienta de instanciación genérica que facilita la utilización de resultados de orden superior. Esta herramienta se puede mejorar en varios aspectos para facilitar la reutilización de libros ACL2. Quizá la más importante de todas estas mejoras sería la integración de la herramienta con el propio sistema, tarea para la que ya se han realizado algunos contactos con los desarrolladores del sistema.

Bibliografía

[1] Ahrendt, W., Beckert, B., Hähnle, R., Menzel, W., Reif, W., Schellhorn, G. y Schmitt, P.H. Integration of automated and interactive theorem proving. En *Automated Deduction: A Basis for Applications*, W. Bibel y P. Schmitt, Eds., vol. II, cap. 4, pp. 97–116. Kluwer, 1998.

[2] Avron, A. Gentzen–type systems, resolution and tableaux. *Journal of Automated Reasoning*, vol. 10, no. 2, pp. 265–281, 1993.

[3] Avron, A. Classical Gentzen-type Methods in Propositional Many-Valued Logics. Tutorial en el 31st IEEE International Symposium on Multiple-Valued Logic, 2001.

[4] Bachmair, L. y Ganzinger, H. Resolution Theorem Proving. En *Handbook of Automated Reasoning*, A. Robinson y A. Voronkov, Eds., cap. 1, pp. 1–82. Elsevier Science Publishers, 2001.

[5] Bayardo, R.J. y Schrag, R. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. En *Proceedings of the 14th National Conference on Artificial Intelligence*, pp. 203–208. 1997.

[6] Beckert, B. y Posegga, J. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, vol. 15, pp. 339–358, 1995.

[7] Ben-Ari, M. *Mathematical Logic for Computer Science*. Prentice Hall International, 1993.

[8] Bezem, M. Completeness of resolution revisited. *Theoretical Computer Science*, vol. 74, no. 2, pp. 227–237, 1990.

[9] Bolc, L. y Borowik, P. *Many–Valued Logics 1. Theoretical Foundations*. Springer-Verlag, 1992.

[10] Boyer, R.S., Goldschlag, D.M., Kaufmann, M. y Moore, J S. Functional Instantiation in First-Order logic. En *Artificial Intelligence and Mathematical Theory of Computation. Papers in honour of John McCarthy*, V. Lifschitz, Ed. Academic Press, New York, 1991.

[11] Boyer, R.S. y Moore, J S. *A computational logic*. ACM monograph series. Academic Press, 1979.

[12] Boyer, R. y Moore, J S. A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the ACM*, vol. 31, no. 3, pp. 441–458, 1984.

[13] Boyer, R.S. y Moore, J S. Nqthm, the Boyer–Moore theorem prover, 1997. `http://www.cs.utexas.edu/users/boyer/ftp/nqthm/index.html`.

[14] Brock, B., Kaufmann, M. y Moore, J S. ACL2 Theorems about Commercial Microprocessors. En *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas y C. A., Eds., pp. 275–293. Springer Verlag, 1998.

[15] Caldwell, J.L. *Decidability extracted: Synthesizing "Correct–by–construction" Decision Procedures from Constructive Proofs*. Tesis doctoral, Faculty of the Graduate School of Cornell University, 1998.

[16] Constable, R., Allen, S., Bromely, H., Cleveland, W. y otros. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., 1986.

[17] Cook, S.A. The complexity of theorem proving procedures. En *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation*, pp. 151–158. 1971.

[18] D'Agostino, M., Gabbay, D., Hähnle, R. y Posegga, J. *Handbook of Tableau Methods*. Kluwer, 1999.

[19] Davis, M. y Putnam, H. A computing procedure for quantification theory. *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[20] Dershowitz, N. y Manna, Z. Proving Termination with Multiset Orderings. En *Annual International Colloquium on Automata, Languages and Programming*, H. Maurer, Ed., LNCS 71, pp. 188–202. Springer–Verlag, 1979.

[21] Doets, K. *From Logic to Logic Programming*. MIT Press, Massachusetts, 1994.

[22] Dubois, O., André, P., Boufkhad, Y. y Carlier, J. SAT versus UNSAT. En *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. Johnson y T. M.A., Eds., vol. 26, pp. 415–436. AMS Press, 1993.

[23] Fernández-Ferreira, M. *Termination of term rewriting.* Tesis doctoral, Universiteit Utrecht, 1995.

[24] Fitting, M. *Proof Methods for Modal and Intuitionistic Logics.* D. Reidel Publishing Co., 1983.

[25] Fitting, M. First–order modal tableaux. *Journal of Automated Reasoning*, vol. 4, pp. 191–213, 1988.

[26] Fitting, M.C. *First–Order Logic and Automated Theorem Proving.* Springer–Verlag, New York, 1990.

[27] Freeman, J.W. *Improvements to Propositional Satisfiability Search Algorithms.* Tesis doctoral, University of Pennsylvania, Philadelphia, 1995.

[28] Gamboa, R. *Mechanically verifying real-valued algorithms in ACL2.* Tesis doctoral, University of Texas at Austin, 1999.

[29] Gentzen, G. Investigations into logical deduction. En *The collected papers of Gerhard Gentzen*, M. Szabo, Ed., pp. 68–131. North–Holland, 1969. Originalmente pubicado en 1935.

[30] Goodstein, R.L. *Recursive Number Theory.* North Holland, 1964.

[31] Gordon, M., Milner, R. y Wadsworth, C. *Edinburgh LCF: A Mechanized Logic of Computation*, vol. 78 of *Lecture Notes in Computer Science.* Springer–Verlag, 1979.

[32] Gu, J., Purdom, P.W., Franco, J. y Wah, B.W. Algorithms for the Satisfiability (SAT) Problem: A survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1997.

[33] Harrison, J. Stålmarck's algorithm as a HOL derived rule. En *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, J. v. Wright, J. Grundy, y J. Harrison, Eds., vol. 1125 of *Lecture Notes in Computer Science*, pp. 221–234. Springer Verlag, 1996.

[34] Hähnle, H. Tableaux for multiple–valued logics. En *Handbook of Tableau Methods*, M. D'Agostino, D. Gabbay, R. Hähnle, y J. Posegga, Eds., pp. 529–580. Kluwer Publishing Company, 1999.

[35] Hähnle, R. Tableaux and Related Methods. En *Handbook of Automated Reasoning*, A. Robinson y A. Voronkov, Eds., cap. 3, pp. 101–178. Elsevier Science Publishers, 2001.

[36] Hrbacek, J. y Jech, T. *Introduction to set theory.* Marcel Dekker, Inc., 1978.

[37] Kaufmann, M. Modular Proof: The Fundamental Theorem of Calculus. En *Computer–Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, y J. S. Moore, Eds., cap. 6. Kluwer Academic Publishers, 2000.

[38] Kaufmann, M., Manolios, P. y Moore, J S. *Computer–Aided Reasoning: ACL2 case studies*. Kluwer Academic Publishers, 2000.

[39] Kaufmann, M., Manolios, P. y Moore, J S. *Computer–Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[40] Kaufmann, M. y Moore, J S. A precise description of the ACL2 logic. Tech. rep., Department of Computer Sciencies, University of Texas at Austin, 1998. Véase `http://www.cs.utexas.edu/users/moore/publications/-acl2-papers.html#Foundations`.

[41] Kaufmann, M. y Moore, J S. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, vol. 26, no. 2, pp. 161–203, 2000.

[42] Kaufmann, M. y Moore, J S. ACL2 Version 2.6, 2001. `http://www.cs.utexas.edu/users/moore/acl2/`.

[43] Kaufmann, M. y R.S., Boyer. The Boyer–Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications*, vol. 29, no. 2, pp. 27–62, 1995.

[44] Kaufmann, M. y S., Moore J. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[45] Kleene, S.C. *Introduction to Metamathematics*. pp. 332–340. van Nostrand, Princeton, 1952.

[46] Kowalski, R.A. *Predicate logic as a programming language,* en J. Rosenfeld, ed., *'Information processing 74'*. pp. 569–574. North–Holland, Amsterdam, 1974.

[47] Kunen, K. A Ramsey Theorem in Boyer-Moore Logic. *Journal of Automated Reasoning*, vol. 15, no. 2, pp. 217–235, 1995.

[48] Lipton, R.J. DNA Solution of Hard Computational Problems. *Science*, vol. 268, no. 28, pp. 542–545, 1995.

[49] Lis, Z. Wynikanie semantyczne a wynikanie formalne (logical consequence, semantic and formal). *Studia Logica*, vol. 10, pp. 39–60, 1960. En polaco, con *abstract* en Inglés y Ruso.

[50] L.M., Adleman. On Constructing a Molecular Computer. En *DNA Based Computers*, R. Lipton y E. Baum, Eds. American Mathematical Society, 1996.

[51] Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J. y Ruiz-Reina, J.L. Verifying an applicative ATP using multiset relations. En *Computer Aided Systems Theory - EUROCAST 2001*, LNCS 2178, pp. 612–626. Springer–Verlag, 2001.

[52] Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J. y Ruiz-Reina, J.L. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory. Third ACL2 Workshop, 2002.

[53] Martín-Mateos, F.J., Alonso, J.A., Pérez, M.J. y Sancho, F. Molecular Computation Models in ACL2: a Simulation of Lipton's Experiment Solving SAT. Third International ACL2 Workshop, 2002.

[54] McCarthy, J. A Basis for a Mathematical Theory of Computation. En *Computer Programming and Formal Systems*, P. Braffort y D. Hirschberg, Eds., pp. 33–70. North-Holland, Amsterdam, 1963.

[55] McCune, W. Solution of the Robbins Problem. *Journal of Automated Reasoning*, vol. 19, no. 3, pp. 263–276, 1997.

[56] McCune, W. Otter: An Automated Deduction System, 2001. `http://www-unix.mcs.anl.gov/AR/otter/`.

[57] Min Li, C. y Anbulagan. Heuristics based on unit propagation for satisfiability problem. En *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pp. 366–371. 1997.

[58] Moore, J S. *Piton : a mechanically verified assembly-level language*. Kluwer Academic Publisher, 1996.

[59] Moore, J S., Lynch, T. y Kaufmann, M. A Mechanically Checked Proof of the Correctness of the Kernel of the $AMD5_K86$ Floating-Point Division Algorithm. *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 913–926, 1998.

[60] Moser, M., Ibens, O., Letz, R., Steinbach, J., Goller, C., Schumann, J. y Mayr, K. SETHEO and E-SETHEO—the CADE-13 systems. *Journal of Automated Reasoning*, vol. 18, no. 2, pp. 237–246, 1997.

[61] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. y Malik, S. Chaff: Engineering an Efficient SAT Solver. En *Design Automation Conference*, pp. 530–535. 2001.

[62] Nilsson, N.J. *Artifical Inteligence: A New Synthesis*. cap. 13. Morgan Kaufmann, 1998.

[63] Pelletier, F.J. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, vol. 2, no. 2, pp. 191–216, 1986.

[64] Plaisted, D.A. A simplified problem reduction format. *Artificial Intelligence*, vol. 18, pp. 227–261, 1982.

[65] Robinson, J.A. A Machine-Oriented Logic Based on Resolution Principle. *Journal of the ACM*, vol. 12, no. 1, pp. 23–49, 1965.

[66] Ruiz-Reina, J.L. *Una teoría computacional acerca de la lógica ecuacional*. Tesis doctoral, Universidad de Sevilla, 2001.

[67] Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J. y Martín-Mateos, F.J. Multiset relations: a tool for proving termination. Second International ACL2 Workshop, 2000.

[68] Russinoff, D. A Mechanical Prof of Quadratic Reciprocity. *Journal of Automated Reasoning*, vol. 8, no. 1, pp. 3–21, 1992.

[69] Russinoff, D. 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*, vol. 1, pp. 148–200, 1998.

[70] Russinoff, D. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, vol. 14, no. 1, 1999.

[71] Shankar, N. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, vol. 35, no. 3, pp. 475–522, 1988.

[72] Shankar, N. *Metamathematics, Machines, and Godel's Proof*. Cambridge University Press, 1994.

[73] Shankar, N. Step towards mechanizing program transformations using PVS. En *MCP'95 (Mathematics of Program Construction, Third International Conference)*, LNCS 947, pp. 50–66. Springer–Verlag, 1995.

[74] Shankar, N. Little Engines of Proof. En *Proceedings of FME'02*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[75] Shoenfield, J.R. *Mathematical Logic*. Addison–Wesley, 1967.

[76] Slind, K. Wellfounded schematic definitions. En *CADE–17, 17th International Conference on Automated Deduction*, LNCS 1831, pp. 45–63. Springer–Verlag, 2000.

[77] Smullyan, R.M. Trees and ball games. *Annals of the New York Academy of Sciences*, no. 321, pp. 86–90, 1979.

[78] Smullyan, R. M. A unifying principle in quantification theory. *Proceedings of the National Academy of Sciences of U.S.A.*, vol. 49, no. 6, pp. 828–832, 1963.

[79] Steele, G.L. *Common Lisp. The Language. 2nd. Edition.* Digital Press, 1990.

[80] Sutcliffe, G. y Suttner, C. The CADE-15 ATP system competition. *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 1–23, 1999.

[81] Talcott, C. y Kohlhase, M. Database of Existing Mechanized Reasoning Systems, 1999.
`http://www-formal.stanford.edu/clt/ARS/systems.html`.

[82] Trick, M.A. Second dimacs challenge test problems. En *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. Johnson y T. M.A., Eds., vol. 26, pp. 653–657. AMS Press, 1993.

[83] Urquhart, A. Hard examples for resolution. *Journal of the ACM*, vol. 34, no. 1, pp. 209–219, 1987.

[84] Wand, M. Continuation–based program transformation strategies. *Journal of the ACM*, vol. 27, no. 1, pp. 164–180, 1980.

[85] Wos, L. What is Automated Reasoning? *Journal of Automated Reasoning*, vol. 1, pp. 6–9, 1985.

[86] Yu, Y. *Automated Proofs of Object Code for a Widely Used Microprocessor.* Tesis doctoral, University of Texas at Austin, 1992.

[87] Zhang, H. Solving Open Quasigroup Problems by Propositional Reasoning. En *Proceedings of International Computer Symposium.* 1994.

[88] Zhang, H. SATO: An Efficient Propositional Prover. En *CADE-14, 14th International Conference on Automated Deduction*, pp. 272–275. 1997.

Glosario

A
alfa-formula, 75
arg1, 38
arg2, 38
asig-binaria, 273
asig-negativa, 277
asig-positiva, 276
asignacion-inducida, 62
asignacion-inducida-K, 67
asignaciones, 61
asignaciones-formula, 61
asume-literal, 192
asume-valor, 48
asume-valor-lista, 61
añade-elemento, 85
añade-formula, 197
añade-formulas, 198

B
base, 140
beta-formula, 75
busca-clausula-unitaria, 221
busca-literal-puro, 221

C
clausula-con-literal-cierto, 267
clausula-unitaria, 212
coleccion-clausulas-falsas-asignacion, 265
coleccion-de-conjuntos-no-vacios, 262
combina, 141
complementario, 50
componente-1, 75
componente-2, 75
componente-neg-neg, 76
cond-binaria, 273
cond-negativa, 277
cond-positiva, 275
cond-semantica, 279
cond-soporte, 281
conjuncion, 38
conjunto-miembro, 247
conjunto-representante-minimal, 260
conjunto-representante-minimal-aux, 260
conjunto-representate, 259
conjunto-testigo, 260
conjunto-unitario, 260
construye-conjunto-representante-minimal, 263
contiene-clausula-vacia, 195
CONTRAMOD-secuentes, 113
contramodelo-clausula, 193

D
d, 141
d&p-asignacion-distinguida, 231
d&p-asignacion-distinguida-lista, 234
d&p-medida, 233
d&p-modelo, 233
d&p-objeto, 231
d&p-regla-computacion, 232
d&p-representacion, 233
DAT-generico, 165
DAT-secuentes, 116
DAT-tableros-semanticos, 92
DEDUC-secuentes, 119
DEDUC-tableros-semanticos, 94
disyuncion, 38

doble-negacion, 76
dominio, 151

E
elemento-sin-conjunto-testigo, 262
elimina-clausulas-con-complementarios, 207
elimina-elemento, 243
elimina-literal, 209
elimina-una, 85
equivalencia, 38
es-atomica, 40
es-binaria, 38
es-clausula, 190
es-clausula-negativa, 276
es-clausula-positiva, 275
es-conectiva-binaria, 37
es-conectiva-monaria, 37
es-conjuncion, 42
es-diyuncion, 42
es-equivalencia, 42
es-forma-clausal, 193
es-implicacion, 42
es-literal, 76
es-literal-negativo, 76
es-literal-positivo, 76
es-literal-puro, 213
es-monaria, 38
es-negacion, 42
es-proposicional, 40
es-rama-tablero, 82
es-secuente, 104
es-simbolo-proposicional, 36
es-valor-de-verdad, 46
existe-rel-mayor, 133

F
f-base, 141
f-iterativa, 142
f-iterativa-aux, 142
f-recursiva, 141
fn-bin, 141
fn1, 136
fn1-maximo, 137
forma-clausal, 196
forma-clausal-aux, 196
forma-clausal-procesa-una, 197
funcion-de-verdad-de-
  conjuncion, 48
  conjuncion-K, 56
  disyuncion, 48
  disyuncion-K, 56
  equivalencia, 48
  equivalencia-K, 57
  implicacion, 48
  implicacion-K, 57
  negacion, 48
  negacion-K, 56

G
gen-asignacion-distinguida, 158
gen-asignacion-distinguida-lista, 160
gen-medida, 158
gen-medida-lista, 162
gen-modelo, 158
gen-objeto, 158
gen-objeto-lista, 164
gen-regla-computacion, 158
gen-representacion, 158
gen-seleccion, 161
genera-asignacion-negativa-K, 123
genera-asignacion-positiva, 115
genera-asignacion-positiva-K, 123
genera-contramodelo-secuente, 115
genera-modelo-literales, 90
genera-modelo-literales-K, 97
genera-no-modelo-secuente-K, 123

I
i, 141
igual-conjunto, 245
implicacion, 38
induccion-literal-cierto, 267
induccion-mul-rel, 139
INSAT-resolucion, 250
insercion-ordenada, 136
inserta-en-orden, 152

interseccion, 260

L
lista-en-dominio, 152
lista-formulas, 104
lista-literales-no-esta-en-fc, 231
lista-ordenada, 152
literal-cierto-clausula, 267
literal-en-forma-clausal, 213
literal-resolucion-clausulas, 244
literales-positivos, 266
literales-representante-asignacion, 266

M
medida, 141
medida-forma-clausal, 217
medida-lista, 143
medida-lista-formulas, 114
medida-resolucion, 254
medida-resolucion-cota, 254
medida-saturacion, 255
medida-secuente, 114
medida-uniforme, 78
medida-uniforme-lista, 198
medida-uniforme-rama, 89
medida-uniforme-S, 200
menor, 151
MOD-bifurcacion, 219
MOD-bifurcacion-instancia, 222
MOD-generico, 162
MOD-rama, 88
MOD-tablas-de-verdad, 66
MOD-tablas-de-verdad-K, 69
modelo, 50
modelo-clausula, 191
modelo-clausula-K, 203
modelo-conjuncion, 105
modelo-conjuncion-K, 121
modelo-disyuncion, 105
modelo-forma-clausal, 194
modelo-forma-clausal-K, 204
modelo-K, 58
modelo-lista-literales, 231
modelo-literales, 233
modelo-rama, 82
modelo-rama-K, 95
modelo-secuente, 105
mp-bin, 141
mul-diferencia, 132
mul-fn, 136
mul-mp, 131
mul-rel, 133

N
negacion, 38
neutro, 142
NO-MOD-tablas-de-verdad, 64
NO-MOD-tablas-de-verdad-K, 68
no-modelo-conjuncion, 179
no-modelo-conjuncion-K, 121
no-modelo-K, 58
no-modelo-secuente-K, 121
nueva-asignacion, 266
numero-clausulas-cota, 254
numero-clausulas-variables, 254
numero-conectivas, 41

O
OBJ-SAT-generico, 161
op, 38
orden-enteros, 154
ordena-insercion, 152

P
paratodo-existe-rel-mayor, 133
primer-contramodelo, 64
primer-modelo, 66

R
rama-cerrada, 83
reduccion-conjunto-representante, 262
reduce-coleccion-literal, 267
reduce-forma-clausal-literal, 209
reduce-forma-clausal-literal-C, 209
rel-bin, 141
resolucion-cond-clausula-conjunto, 251

resolucion-cond-clausulas, 246
resolucion-cond-saturacion, 250
resolvente, 243

S
SAT-bifurcacion, 216
SAT-bifurcacion-instancia, 222
SAT-davis-putnam, 223
SAT-generico, 161
SAT-secuentes, 117
SAT-tablas-de-verdad, 65
SAT-tablas-de-verdad-K, 69
SAT-tableros-semanticos, 91
satisfacible-en-asignaciones, 65
satisfacible-en-asignaciones-K, 69
saturacion-resolucion, 250
saturado-cond-clausula-conjunto, 247
saturado-cond-conjunto, 247
saturado-cond-conjunto-aux, 247
sec-asignacion-distinguida, 179
sec-asignacion-distinguida-lista, 182
sec-medida, 181
sec-modelo, 181
sec-objeto, 179
sec-regla-computacion, 180
sec-representacion, 180
secuente-axioma, 106
secuentes-expansion-der, 109
secuentes-expansion-izq, 108
seleccion-instancia, 221
selecciona-no-literal, 197
simbolos-proposicionales, 45
suma-1-si-es-entero, 136

T
tab-asignacion-distinguida, 171
tab-asignacion-distinguida-lista, 173
tab-medida, 172
tab-modelo, 172
tab-objeto, 170
tab-regla-computacion, 171
tab-representacion, 172
TAUT-tablas-de-verdad, 63
TAUT-tablas-de-verdad-K, 68
tiene-clausulas-con-complementarios, 210
tiene-literales-complementarios, 193

U
union-coleccion, 262

V
valida-en-asignaciones, 63
valida-en-asignaciones-K, 68
valor, 49
valor-clausula, 191
valor-clausula-K, 203
valor-de-simbolo, 47
valor-de-simbolo-K, 55
valor-forma-clausal, 193
valor-forma-clausal-K, 204
valor-K, 57
variables-forma-clausal, 254