Departamento de Ciencias de la Computación
e Inteligencia Artificial

Teoría computacional (en PVS) de la programación lógica y del análisis formal de conceptos

Memoria presentada por
María José Hidalgo Doblado
para optar al grado de
Doctor en Matemáticas
por la Universidad de Sevilla

María José Hidalgo Doblado

V. B. Director

D. José Antonio Alonso Jiménez

Sevilla, 4 de mayo de 2004

A mis padres

Agradecimientos

El trabajo desarrollado en esta memoria está parcialmente financiado por el proyecto TIC2000-1368-C03-02 del Ministerio de Ciencia y Tecnología.

En este apartado de agradecimientos quiero remontarme a los profesores Luis Laita y Alejandro Fernández que, hace ya bastantes años, me enseñaron lógica y me transmitieron su honradez en el trabajo y su amor por el conocimiento. En esta retrospectiva quiero agradecer también a Agustín Riscos y a José Antonio Alonso sus valiosos consejos a lo largo de mi trayectoria académica.

En relación con esta memoria, en primer lugar quiero agradecer al director de este trabajo y amigo, José Antonio Alonso, no sólo su dirección sino sobre todo, su comprensión y apoyo constantes. Con ellas ha sabido, en cada momento, encauzar la atención y el esfuerzo hacia los aspectos verdaderamente sustanciales. Sin lugar a dudas, sin él este trabajo no se habría realizado.

Al Grupo de lógica computacional de la Universidad de Sevilla, cuyas reuniones han servido para transmitir conocimiento, desarrollar nuevas ideas, analizar y corregir posibles caminos erróneos y, fundamentalmente, para compartir ilusiones. Quiero agradecer especialmente a Francisco Jesús Martín y a José Luis Ruiz la ayuda que me han prestado, tando en cuestiones científicas o técnicas, como en el ámbito personal.

A los miembros del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla, por haber sabido mantener un espíritu común de trabajo y solidaridad, desde su creación; por haber hecho de este espacio que compartimos mucho más que un lugar de trabajo.

A mi familia, que siempre ha confiado en mí y que, durante este último período, ha entregado también su valioso tiempo para que esta memoria se haya terminado. A José Antonio, Carmen, Laura y Elena, gracias.

A todos, gracias.

Índice general

Capítulo 1

Introducción

"Quo facto, quando orientur controversiae, non magis disputatione opus erit inter duos philosophos, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedereque ad abacos, et sibi mutuo (accito si placet amico) dicere: calculemus." G. W. Leibniz (1646–1716) [1]

1.1. Antecedentes

El conocimiento matemático es el núcleo de la ciencia y la tecnología actual. Ahora bien, el incremento de este conocimiento y de sus aplicaciones en los últimos 200 años ha sido extraordinario, sobrepasando la capacidad humana para conocer todas las matemáticas actuales. No obstante, el desarrollo de la lógica matemática y el avance de las ciencias de la computación ofrecen la posibilidad de construir un sistema digital que represente todo el conocimiento matemático relevante, de forma completamente rigurosa y reutilizable.

El manifiesto QED [4] describe un futuro en el que toda la matemática esté codificada en un sistema, mediante una representación del conocimiento estrícta-mente formal, y que use métodos mecanizados para la verificación de las pruebas. En los últimos años, han empezado a desarrollarse proyectos, como Logosphere [3] o FDL[2], siguiendo la línea del manifiesto QED, con el objetivo de construir bibliotecas digitales formalizadas de conocimiento matemático. Estos proyectos, que incluyen la representación formal del conocimiento y el uso de demostradores automáticos, presentan importantes oportunidades y retos. El conocimiento así almacenado podrá ser utilizado en otras áreas, como pueden ser la verificación formal de sistemas y de programas, o el propio desarrollo del conocimiento matemático. Con más detalle, en FDL se describen situaciones en las que se usaría el

[1] "De hecho, si surgiera una controversia, no habría más necesidad de discusión entre dos filósofos que entre dos contables. Pues, sería suficiente tomar la tiza entre las manos, sentarse con sus pizarras y, de mutuo acuerdo, decir: calculemos"

conocimiento algorítmico formal almacenado en una biblioteca digital. Además, en QED se apunta una motivación no práctica, aunque no por ello menos importante, para la creación de estas bibliotecas: QED como proyecto cultural.

Por otra parte, los esfuerzos de formalización del conocimiento matemático plantearán la necesidad de hacer explícito el conocimiento implícito que se usa habitualmente, y contribuirá a acrecentar el nivel de conexión entre áreas diferentes. Además, si se consigue la reutilización de las teorías ya formalizadas, se eliminará la multiplicación del esfuerzo que supone la construcción de todo el bagaje matemático necesario para la verificación formal de un resultado en particular.

Las matemáticas suelen considerarse como la exactitud "por excelencia". Sin embargo, el lenguaje usado habitulamente por los matemáticos puede ser, en ocasiones, notablemente vago, pero aceptado mayoritariamente. En [11] Bourbaki subraya la importancia del "abuso de lenguaje", sin el cual cualquier texto matemático sería ilegible. Por otra parte, Trybulec y Swieczkowska en [95], observan que el lenguaje de textos matemáticos no es el lenguaje natural, pero que contiene una parte de dicho lenguaje, que es fuente de ambigüedades e imprecisiones.

Otra cuestión no menos importante es la corrección del razonamiento matemático. Las pruebas matemáticas son revisadas con detalle antes de su publicación. No obstante, existen abundantes casos documentados de resultados publicados con errores en las pruebas.

Así pues, la formalización de las matemáticas incide en ambas cuestiones, precisión y corrección. Por formalización se entiende la expresión de las afirmaciones y las pruebas en un lenguaje formal, con estrictas reglas gramaticales y sin ambigüedades semánticas. Es decir, un proyecto de formalización consta de dos partes:

- Formalizar los teoremas y el contexto implícito del que dependen.
- Formalizar las pruebas de dichos teoremas y hacerlas objeto de una verificación precisa.

Por otra parte, la idea de reducir el razonamiento a algún tipo de cálculo formal es un sueño que se remonta a Raimundo Lulio y a Leibniz, quien ya concibe la idea de un lenguaje universal ("characteristica universalis") y un cálculo del razonamiento ("calculus ratiocinator"). En 1848, Boole [10] desarrolló el primer sistema formal para el razonamiento lógico, limitado al razonamiento proposicional. Posteriormente, en 1879 Frege, considerado como el padre de la lógica moderna, quiso probar que toda la matemática era pura lógica y ya usó un sistema deductivo formal definido con precisión. Por otra parte, Peano desarrolló una notación formal para expresar las proposicioness matemáticas, aunque estaba más interesado en reescribir las matemáticas en un marco formal que en el desarrollo de un sistema deductivo. A principios del siglo XX, Russell, basándose en los trabajos de Peano, desarrolló su propia lógica, introduciendo la noción de tipo y,

en 1910, Whitehead y Russell empezaron un desarrollo formal de las matemáticas en su trabajo "Principia Mathematica" [99]. Por último, Hilbert propone una metodología para el desarrollo de teorías formales, basada en el estudio de las *pruebas*, centrando su atención en probar la consistencia de dichas teorías.

1.1.1. Sistemas de razonamiento automático

El desarrollo de teorías formalizadas se sustenta en el uso de los sistemas de razonamiento automáticos, en los que se puede probar que una conjetura se deduce de un conjunto de asertos previos.

El nacimiento del razonamiento automático se puede remontar al año 1954 cuando M. Davis presentó un demostrador automático para la aritmética. Dos años después, Newell, Shaw y Simon presentaron en la conferencia de Darmouth su lógico teórico (" The Logic Theory Machine") [58], un demostrador con orientación heurística en vez de la orientación algorítmica del de Davis. A estos primeros demostradores le siguieron otros como la máquina geométrica de Gelernter [34] (1959, en la línea heurística) o los demostradores de Wang[98], Gilmore[35] y Davis y Putnam[21] (1960, en la línea algorítmica). Un año señalado en la historia del razonamiento automático es 1965, por el trabajo de Robinson donde presenta el principio de resolución [71]. En la segunda mitad de los años 60, la investigación se centra en el desarrollo de variantes y estrategias de resolución en lógica de primer orden (como la preferencia unitaria y resolución unidad (Wos 1964), estrategia del conjunto soporte (Wos y otros, 1964), hiper–resolución (Robinson, 1965), subsunción (Robinson, 1965), resolución lineal (Loveland, 1968) y resolución por entradas (Chang, 1970)). Al principio de los 70 se desarrollan demostradores especializados, en las siguientes direcciones:

- Limitando el lenguaje a cláusulas de Horn, Kowalski introduce en 1972 la estrategia de resolución SLD, que es la base de la programación lógica, y que le permitió, junto a Colmerauer, construir en 1973 la primera implementación de *Prolog*.

- La automatización de la igualdad presenta una dificultad especial. Para tratarla, se introdujo la demodulación [100] (Wos y otros, 1967), la paramodulación [101] (Wos y Robinson, 1970) y, en general, todas las técnicas relativas a los sistemas de reescritura de términos [46] (Knuth y Bendix, 1970).

- Otra dificultad se presenta en la automatización del razonamiento por inducción. En este sentido, el primer demostrador que lo aborda es el de Boyer y Moore (1973) con el que se prueban teoremas como el de la factorización prima (Boyer y Moore 1979), el teorema de Wilson (Rusinoff, 1983), la completitud de Lisp (Boyer y Moore, 1984) o el teorema de incompletitud de Gödel [79] (Shankar, 1994).

También en el año 1970 se empieza el proyecto de formalización de teorías matemáticas con el proyecto Automath de Bruijin.

El demostrador Automath es uno de los primeros sistemas de razonamiento, dentro de los sistemas cuya lógica es de orden superior, basada en teoría de tipos. En la misma línea, se han desarrollado los sistemas Nuprl [20] o Coq [25]. Otros sistemas, entre los que se pueden considerar Isabelle [66, 67], HOL [36] y PVS [59], siguen el estilo "LCF" [37]. Es decir, disponen de un núcleo pequeño de reglas de inferencia, sobre el que el usuario construye tácticas para automatizar la aplicación de dichas reglas.

Los sistemas de razonamiento automático se usan en diversos campos, como lógica, ciencias de la computación, matemáticas e ingeniería. Las áreas en los que se han obtenido resultados más importantes han sido las matemáticas, la generación y verificación de software, y la verificación de hardware:

- En matemáticas, uno de los resultados de myor impacto ha sido la prueba de la conjetura de Robbins[52] usando el sistema EQP, que es una variante de Otter [53]. El sistema Otter también ha sido usado con éxito en la resolución de problemas de naturaleza algebraica (ver [59] para otras aplicaciones).

- El uso de sistemas de razonamiento automático en la generación de programas verificados es un campo interesante, que aún está en sus comienzos. Entre los logros conseguidos en él, cabe destacar el desarrollo del sistema KIDS [85] en el Instituto de Krestel, usado para obtener programas a partir de especificaciones; y el proyecto AMPHION [1], desarrollado por la NASA.

- La verificación de programas es uno de los campos en los que se concentra un mayor número de aplicaciones de los sistemas de razonamiento automático (ver [12]). Como ejemplos, podemos citar el verificador interactivo diseñado en la Universidad de Kalsruhe (KIV [70]), que ha sido usado con éxito en diversas aplicaciones, tanto académicas como industriales; y el sistema PVS, que se ha usado en aplicaciones que incluyen la verificación de un sistema de control de vuelos (ver [59] para otras aplicaciones).

- Por último, la verificación de hardware es el campo en el que se han producido mayor cantidad de aplicaciones industriales (ver [12]). Destaquemos, entre ellas, el uso del sistema ACL2 [45] para la verificación del algoritmo de de división de coma flotante del procesador AMD K5 y el uso de PVS para la verificación de un microprocesador comercial en el campo de la aeronáutica.

Los sistemas de razonamiento automático consisten en una lógica, en la que expresar el problema formalmente, y un demostrador. Hay algunos aspectos fundamentales que diferencian a unos sistemas de razonamiento de otros, entre los que destacamos la lógica subyacente, el contenido matemático ya incorporado y el

grado de automatismo o interactividad del demostrador. En este sentido, encontramos un amplio espectro que va desde sistemas totalmente automáticos, como Otter, a sistemas que son esencialmente comprobadores de prueba, como Mizar [94]. En general, los sistemas de razonamiento que poseen lógicas más expresivas son menos automáticos, mientras que aquellos cuyas lógicas son más restrictivas poseen un mayor grado de automatismo. Otro punto a tener en cuenta es la ejecutabilidad o no de los modelos formalizados. En este aspecto también hay gran diversidad. Desde sistemas como ACL2 en el que el lenguaje de programación permite la evaluación de los modelos construidos, hasta sistemas que no contemplan la posibilidad de la evaluación.

Algunos sistemas, entre los que se encuentra PVS, tienden a proporcionar un cierto equilibrio entre automatismo e interactividad, puesto que disponen de un núcleo de reglas de inferencia, a partir de las cuales se pueden construir tácticas o estrategias que automatizan el proceso de demostración. Al mismo tiempo, la lógica subyacente a PVS es una lógica de orden superior, lo que hace que dicho sistema posea gran capacidad expresiva. En cuanto a la evaluabilidad, una parte importante de PVS es evaluable, mediante su transformación automática en código Lisp, como comentamos en el capítulo 2.

1.1.2. Objetivos

El trabajo que presentamos en esta memoria se enmarca dentro del desarrollo de teorías matemáticas formalizadas. Más concretamente, dentro de la formalización de un cuerpo base de las matemáticas usadas en el estudio de procesos computacionales. Uno de los objetivos finales es disponer de algoritmos verificados formalmente. En este sentido, puede verse como una continuación de trabajos anteriores, tales como la formalización en ACL2 del razonamiento ecuacional [74, 75] y del razonamiento proposicional [51], la formalización en Coq [25] de la resolución SLD [41], o la formalización en Mizar del retículos de los conceptos [77].

Ahora bien, el desarrollo formal de la corrección de ciertos algoritmos depende fuertemente del estudio de amplias teorías matemáticas, lo que conduce a la necesidad de la formalización de éstas. Surge, entonces, la cuestión de "cómo" realizar dicha verificación. En este sentido, otro de nuestros objetivos es probar la corrección de especificaciones evaluables de algoritmos, sin renunciar a la "naturalidad" en el razonamiento que ha de hacerse en la teoría que sustenta dicha verificación. Es decir, queremos combinar un razonamiento "natural" o elegante sobre las especificaciones, con la evaluabilidad de algunas de ellas, que representarán algoritmos de cálculo.

Para ello, hemos elegido un sistema de razonamiento, PVS, y dos teorías como objetos de formalización, la programación lógica proposicional y el análisis formal de conceptos.

La elección de PVS como sistema de razonamiento ha venido motivada por-

que en él se pueden combinar los aspectos comentados previamente: la potencia expresiva y el razonamiento "natural", con la posibilidad de obtener funciones o algoritmos evaluables.

En cuanto a la elección de las teorías a formalizar, hay que hacer notar que ambas poseen una amplia e interesante base matemática, que en las dos teorías se han desarrollado algoritmos o procesos susceptibles de ser verificados formalmente; y que, además, poseen aplicaciones relevantes en computación.

La programación lógica es, como hemos comentado anteriormente, la base del lenguaje de programación *Prolog*, cuyo procedimiento de demostración para cláusulas de Horn se conoce como resolución SLD. Aunque el método es incompleto (no puede demostrar todas las fórmulas válidas) es suficientemente potente para computar las funciones recursivas. Así, creemos que tiene interés abordar la formalización de la programación lógica en un sistema de razonamiento. Hemos formalizado la programación lógica proposicional que, junto con la unificación forman el núcleo de la programación lógica.

El análisis formal de conceptos es una teoría que proviene de un campo, en principio, diferente. Desarrollada desde la década de los 80, está basada en la formalización matemática de la noción filosófica de concepto. Centra su atención en la generación de los conceptos subyacentes a un determinado contexto[2] y en la obtención de "reglas", que describan las relaciones entre los atributos o propiedades contempladas en el contexto. El análisis formal de conceptos tiene numerosas aplicaciones, como comentamos con más detalle en el capítulo 5. En ambos casos, hemos iniciado una formalización encaminada hacia la obtención de implementaciones de los algoritmos de dichas teorías, verificados formalmente.

Además, creemos que se puede establecer una relación entre las bases de reglas de un contexto y los programas lógicos proposicionales [3], y trasvasar propiedades de una teoría a otra.

Por otra parte, el deseo de realizar un razonamiento "natural" sobre las especificaciones, nos conduce al problema de la "elección" de los tipos básicos sobre los que construir las especificaciones. En este sentido, los tipos de datos elegidos para representar formalmente las nociones de las teorías han de estar lo más próximo posible a dichas nociones. En la programación lógica proposicional hemos de representar formalmente nociones como cláusulas de Horn, programas lógicos, interpretaciones, ... Matemáticamente, un programa es un conjunto de cláusulas, y una cláusula puede considerarse como un conjunto de literales. Análogamente, un contexto formal consta de un conjunto de objetos, un conjunto de atributos y una relación entre ambos.

Así pues, en ambos casos el tipo más cercano a los objetos que se desea representar en el lenguaje de PVS es el tipo de los conjuntos (finitos o infinitos).

[2]Un contexto formal consta de un conjunto de objetos O, un conjunto de atributos A, y una relación entre ambos, especificando los atributos que posee cada objeto.

[3]Una regla es de la forma $A_1 \wedge \cdots \wedge A_k \rightarrow B_1 \wedge \cdots \wedge B_m$ que, evidentemente, equivale a un conjunto de cláusulas de Horn $A_1 \wedge \cdots \wedge A_k \rightarrow B_i,\ i = 1, \ldots, m$

Como se detallará en el capítulo 2, el lenguaje de PVS posee una representación formal de la noción de conjunto, así como de conjunto finito.

Ahora bien, aunque la elección del tipo de los conjuntos de PVS como tipo base para realizar las especificaciones haga posible un razonamiento elegante (o, mejor dicho, no artificioso) sobre ellas, tendremos el problema de que dichas especificaciones no serán evaluables, cosa que era deseable si no de forma generalizada, sí de forma puntual. Es decir, no han de ser evaluables todas las nociones que se especifiquen, sino sólo algunas especificaciones de los algoritmos.

La solución que proponemos para este problema es de carácter metodológico. Para ello, basándonos en la noción de refinamiento introducida por Jones [43], hemos establecido en PVS un marco en el que relacionar distintas especificaciones, de forma que definimos formalmente cuándo dos especificaciones diferentes corresponden a un mismo concepto. Estudiamos también las propiedades de carácter general que se conservan entre especificaciones de un mismo algoritmo. Esto nos permite trabajar en dos planos. Por una parte, realizar el razonamiento en el plano en el que la distancia entre una noción y su especificación formal es mínima, aunque ésta no sea evaluable. Y, por otra, sólo para determinadas especificaciones, obtener otra evaluable correspondiente al mismo concepto, y con las mismas propiedades (en particular, la propiedad de corrección de un algoritmo).

1.1.3. Trabajos relacionados

Como hemos comentado en el punto anterior, existen algunos trabajos de formalización de teorías matemáticas en distintos sistemas de razonamiento automático. Citamos a continuación los que guardan una mayor relación con el trabajo que presentamos.

En el sistema ACL2, J.L. Ruiz [74] ha desarrollado en 2001 una teoría formal sobre la lógica ecuacional y la reescritura de términos, construyendo algoritmos ejecutables en Common Lisp, formalmente verificados. En 2002, F.J. Martín [51] ha desarrollado, también en ACL2, un modelo formal de la lógica proposicional, formalizando algunos de los cálculos proposicionales más conocidos, para los cuales construyó y verificó procedimientos de decisión de satisfacibilidad. Por último, en 2003, I. Medina [54] ha realizado una verificación formal en ACL2 del algoritmo de Buchberger.

En relación con la formalización de la programación lógica en otros sistemas de razonamiento, M. Jaume [41] realizó en 1999 una formalización en Coq de la semántica de los programas definidos, probando el teorema de completitud débil de la resolución SLD, siguiendo un desarrollo clásico basado en el libro de J. W. Lloyd [48]. El autor destaca que en la formalización realizada ha sido necesaria la inclusión de propiedades que normalmente pasan desapercibidas en las pruebas clásicas, y que ha necesitado más de 600 de los llamados lemas "técnicos".

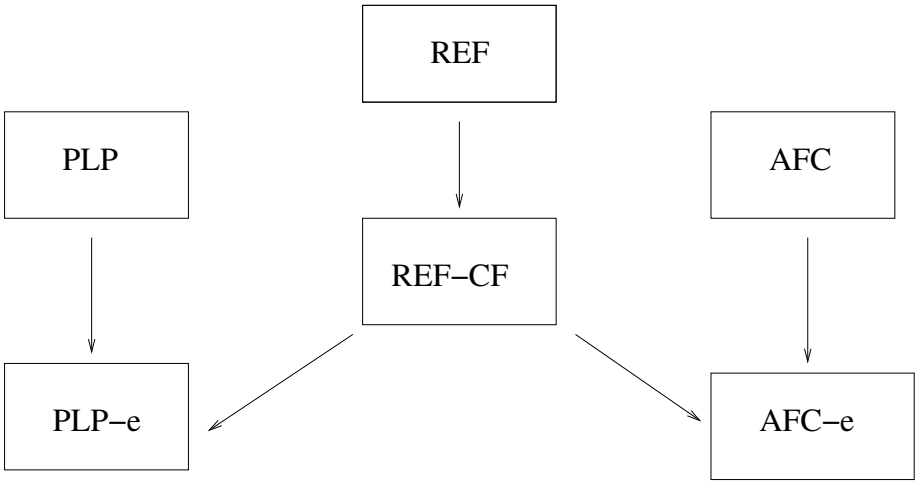
En cuanto al análisis formal de conceptos, C. Schwarzweller [77] ha codificado en Mizar el retículo de los conceptos de un contexto formal. Para ello, ha utili-

zado la formalización de la teoría de retículos incluida en la biblioteca de teorías matemáticas formalizada ya en Mizar.

Por último, en el sistema PVS, A. Dold [24] ha desarrollado un tratamiento formal en PVS de pasos de transformación en el desarrollo de programas. El trabajo ha consistido en la construcción de un marco formal unificado en el que se integran pasos y métodos de transformación usados en el desarrollo de software. Y en 2003, C. Graciani [38] ha desarrollado en PVS un marco formal en el que se describen algunos modelos de computación molecular, estableciendo la corrección de programas en dichos modelos de computación.

1.2. Estructura de la memoria

El trabajo que presentamos se puede estructurar de la siguiente forma:



donde:

- En REF se establece un marco general para el refinamiento de tipos y operaciones (incluido en el capítulo 6).
- En REF–CF se establece, como caso de estudio y para su aplicación en las teorías desarrolladas en este trabajo, un refinamiento del tipo y de las operaciones con conjuntos finitos, mediante listas (incluido en el capítulo 6).
- En PLP se formaliza la programación lógica proposicional, usando como tipo base el tipo de los conjuntos finitos de PVS (capítulos 3 y 4).
- En PLP–e se obtienen especificaciones evaluables de funciones y algoritmos de la programación lógica proposicional, a partir de las especificaciones construidas en PLP, usando para ello el refinamiento desarrollado en REF–CF (capítulo 7).

- En AFC se formalizan las nociones fundamentales y algunos algoritmos del análisis formal de conceptos, usando para ello el tipo de los conjuntos finitos de PVS (capítulo 5).

- En AFC–e se obtienen especificaciones evaluables de los algoritmos especificados en AFC (capítulo 8).

A continuación describimos cada una de las partes que componen este trabajo, señalando los aspectos más importantes de cada una de ellas.

1.2.1. El sistema PVS

En el capítulo 2 se expone una descripción de los aspectos más relevantes del sistema PVS. Con ello se pretende facilitar la comprensión de los capítulos posteriores. Hemos incluido una sección dedicada a la lógica del sistema, haciendo hincapié en el sistema de tipos de PVS, otra dedicada al demostrador, así como un ejemplo que ilustra el manejo de éste.

1.2.2. Marco para refinamientos

En el capítulo 6 se establece un marco general que permite relacionar distintas especificaciones de un mismo algoritmo. Está inspirado en la noción de refinamiento introducida por Jones en [43] y usada por Dold en [24, 23]. Esto hará posible razonar de manera natural sobre especificaciones semánticamente simples (usando tipos de datos abstractos) y relacionarlas con otras especificaciones evaluables (que usan tipos de datos concretos) de los mismos conceptos.

En primer lugar, se define el concepto de refinamiento de tipos y se establece que un determinado tipo puede ser refinado en pasos sucesivos. En segundo lugar, considerando que una especificación de un algoritmo viene dada como una operación $op : T_1 \to T_2$, donde T_1 y T_2 son los tipos que representan las posibles entradas y salidas del algoritmo, respectivamente, establecemos formalmente la condición que deben cumplir dos especificaciones para entender que representan el mismo algoritmo (diremos que una especificación refina a otra). Concretamente, otra especificación del algoritmo vendría dada por otra función $op_{ref} : R_1 \to R_2$. Si para cada i, el tipo R_i *refina* al tipo T_i, diremos que la función op_{ref} *refina* a la función op si el siguiente diagrama es commutativo:

$$\begin{array}{ccc} T_1 & \xrightarrow{op} & T_2 \\ {\scriptstyle f_1}\uparrow & \# & {\scriptstyle f_2}\uparrow \\ R_1 & \xrightarrow{op_{ref}} & R_2 \end{array}$$

A partir de esta definición, se estudian las propiedades de las especificaciones que se transmiten a las correspondientes refinadas, y se prueba que para obtener un refinamiento de una especificación es suficiente construir refinamientos de cada una de las funciones usadas en ella y combinarlas adecuadamente. Entre las propiedades, destaquemos que hemos probado que de la corrección de una especificación se deduce la corrección de cualquier otra especificación que la refine.

Por último, se establece un refinamiento del tipo de los conjuntos finitos sobre un tipo T, por el tipo de dato de las listas sobre un tipo R, a partir de un refinamiento de T por R. Además, se presentan operaciones entre listas que refinan a las operaciones entre conjuntos finitos.

1.2.3. Programación lógica proposicional

En el capítulo 3 se presenta una formalización en PVS de la lógica proposicional, basada principalmente en [48, 5, ?]. En primer lugar, se representan las fórmulas proposicionales sobre un universo como un tipo abstracto de datos en PVS, y se formaliza la semántica de la lógica proposicional, a través de la noción de interpretación.

A continuación, se describe una representación de las cláusulas de Horn, en la que el cuerpo de una cláusula es un conjunto finito de átomos. Los programas lógicos se representan también como conjuntos finitos de cláusulas. Por último, se formaliza la semántica de cláusulas a través de la semántica proposicional, se introducen los modelos de Herbrand y se prueba que son suficientes para determinar la satisfacibilidad de conjuntos de cláusulas.

En el capítulo 4 se han formalizado las distintas interpretaciones semánticas de los programas lógicos proposicionales, y se ha probado su equivalencia:

- En primer lugar, especificamos el concepto de menor modelo de Herbrand de un programa P, y probamos que dicho conjunto coincide con el conjunto de consecuencias lógicas de P.

- A continuación, formalizamos la semántica del punto fijo de los programas definidos. Para ello, utilizamos el trabajo realizado por Bartels, Dold, Pfeifer, Henke y Ruess en [8]. En él se formaliza en PVS la teoría de dominios, se prueba el teorema del punto fijo de Tarski para operadores monótonos, y se presentan dos construcciones del menor punto fijo de un operador, una para operadores monótonos y otra para operadores continuos. En el presente trabajo, formalizamos la noción de operador de consecuencia asociado a un programa, T_P; aplicamos [8] para construir el menor punto fijo de un programa, y probamos que coincide con el menor modelo de Herbrand de dicho programa.

- Por último, formalizamos el proceso de resolución SLD. Comenzamos especificando cuándo un objetivo G tiene una refutación a partir de un programa

P. Esto nos permite definir el conjunto de éxitos de P y probar que coincide con el menor modelo de Herbrand de P. Es decir, probar la adecuación y completitud del proceso.

Hay que hacer notar dos detalles de la formalización realizada:

1. La especificación de la función que obtiene la resolvente de una cláusula definida C y un objetivo G consiste en obtener el objetivo que resulta de elimina la cabeza de C del cuerpo de G y añadirle el cuerpo de C. Es decir, si C es la cláusula $A \vee \neg B_1 \vee \cdots \vee \neg B_k$ (representada por $A \leftarrow \{B_1, \ldots, B_k\}$) y G es el objetivo $\neg A_1 \vee \cdots \vee \neg A_m$ (representado por $\leftarrow \{A_1, \ldots, A_m\}$), entonces:

   $$\text{resolvente}(G, C) \equiv \leftarrow (\{A_1, \ldots, A_m\} - \{A\}) \cup \{B_1, \ldots, B_k\}$$

   Nótese que, al usar conjuntos para representar los cuerpos de las cláusulas, los átomos que los forman no están ordenados ni se contempla que haya elementos repetidos.

2. La especificación del proceso de refutación es abstracta, en el sentido de que no se maneja, como hace LLoyd [48], el objeto *derivación* ni, a partir de él, el objeto *refutación*.

   Definimos recursivamente cuándo un objetivo G tiene una refutación de longitud n a partir de un programa P y, como consecuencia, definimos que $P \cup \{G\}$ tiene una refutación si, para algún n tiene una refutación de longitud n. Demostramos, tanto la adecuación como la completitud del proceso.

Por último, formalizamos el concepto de regla de computación y especificamos cuándo un objetivo G tiene una refutación a partir de P, vía una regla de computación. Finalizamos el capítulo probando la independencia de la regla de computación y, como consecuencia, la completitud fuerte de la resolución SLD: $P \cup \{G\}$ es insatisfacible si y sólo si $P \cup \{G\}$ tiene una refutación, vía cualquier regla de computación.

Es de destacar que la formalización realizada ha permitido que las pruebas de estos resultados no se alejen de las originales. Es decir, las pruebas realizadas en PVS se corresponden esencialmente con las pruebas que se harían de forma independiente del sistema. En este sentido, podemos confirmar que se han cumplido nuestras espectativas de que la formalización, tanto de los conceptos como de las pruebas, no fueran sustancialmente diferentes (ni en tamaño ni en dificultad) de las originales.

Ahora bien, en el caso de la prueba de la independencia de la regla de computación, ésta no puede hacerse en la forma habitual puesto que, esta propiedad se suele probar basándose en el intercambio de pasos de refutación, que sólo pode-

mos hacer si estamos tratando los cuerpos de las cláusulas desde una perspectiva computacional, como listas o sucesiones de átomos [4].

En el trabajo que presentamos, hemos probado la independencia de la regla de computación en el proceso de resolución SLD abstracto (considerando los cuerpos de las cláusulas como conjuntos de átomos), sin hacer uso del intercambio de pasos de refutación. Para ello, hemos definido una función de medida h_P sobre los objetivos, de forma que si $P \cup \{G\}$ tiene una refutación y A es un átomo del cuerpo de G, exista una cláusula C en P tal que su cabeza coincida con A y $h_P(\text{resolvente}(G, C)) < h_P(G)$.

La definición de la función h_P la hemos hecho basándonos en la noción de árbol de implicación de un átomo A respecto de un programa P (introducida por Stärk en [89]), de forma que $h_P(A)$ mida el número de nodos del árbol de implicación de A con menor número de nodos. Y, para un objetivo, $h_P(G)$ sea la suma de las medidas de los átomos del cuerpo de G. Para definir la función h_P tenemos en cuenta que, si un átomo A es consecuencia lógica de P, A es un elemento del menor punto fijo de P y, por tanto, se ha introducido en algún paso de la cadena de aplicaciones del operador de consecuencia:

$$T_P^0(\emptyset) \subseteq T_P^1(\emptyset) \subseteq \ldots T_P^k(\emptyset) \subseteq T_P^{k+1}(\emptyset) \subseteq \ldots$$

Entonces, los átomos que aparecen en el árbol de implicación de A con menor número de nodos han sido introducidos en los pasos previos de la cadena. Esto nos permite dar una definición recursiva de la función h_P, susceptible de ser especificada en el lenguaje de PVS.

En el capítulo 7 presentamos especificaciones evaluables de especificaciones definidas en los capítulos 3 y 4, para lo cual usamos el refinamiento de conjuntos finitos desarrollado en el capítulo 6. En primer lugar, representamos los elementos de la programación lógica proposicional, mediante especificaciones que usan listas como tipo base. A continuación, declaramos transformaciones genéricas que relacionan ambas representaciones, describiendo mediante axiomas las propiedades que deben verificar dichas funciones. Posteriormente, estas funciones serán interpretadas mediante transformaciones concretas que verificarán los axiomas exigidos.

Concretamente, construimos especificaciones evaluables para el cálculo de la base de Herbrand, del menor modelo de Herbrand, del operador de consecuencia y del menor punto fijo de un programa definido. Probamos también que éstas refinan a las especificaciones de estos mismos conceptos, sobre las que hemos realizado todo el razonamiento. Con ello tenemos, además, las pruebas de las propiedades que verifican las especificaciones evaluables de manera inmediata.

[4] En el desarrollo presentado por Nerode en [57], se considera que los cuerpos de las cláusulas son conjuntos finitos de átomos, pero a la hora de probar la completitud del proceso subyacente a *Prolog*, se decanta por considerarlos como sucesiones y no como conjuntos.

Terminamos el capítulo presentando distintos algoritmos de refutación, según el proceso de búsqueda y la regla de computación que se utilice.

1.2.4. Análisis formal de conceptos

En el capítulo 5 presentamos una formalización en PVS de la teoría del análisis formal de conceptos, siguiendo la presentación de Ganter y Wille en [33]. Un contexto formal consta de un conjunto de objetos O, un conjunto de atributos A, y una relación entre ambos, que especifica los atributos que posee cada objeto. La noción de concepto relativo a un contexto formal sintetiza la idea de un conjunto de objetos, caracterizados por el conjunto de atributos que poseen.

En primer lugar, se representa la noción de contexto formal, se definen los operadores de derivación asociados a un contexto y se prueba que constituyen una conexión de Galois entre los conjuntos de objetos y atributos del contexto.

Usando dichos operadores se especifica la noción de concepto en un contexto formal, y se prueba que el conjunto de conceptos de un contexto, con la relación de subconcepto, tiene estructura de retículo completo. Para ello, probamos que dicha relación es un orden parcial y que todo conjunto de conceptos posee supremo e ínfimo. La prueba se hace especificando, mediante operaciones conjuntistas, funciones que obtienen el supremo y el ínfimo, respectivamente, de un conjunto de conceptos.

A continuación, se describe la especificación de un algoritmo para generar el conjunto de los conceptos de un contexto y se prueba que es correcto.

Con frecuencia, el número de objetos de un contexto es mucho mayor que el número de atributos. En estos casos, es útil disponer de reglas (o implicaciones entre atributos) que relacionan conjuntos de atributos, en el sentido siguiente: "si un objeto posee los atributos $a_1, \ldots, a_n$, también posee los atributos $b_1, \ldots, b_m$". Formalizamos la noción de implicación entre atributos y la semántica asociada, probando un resultado que relaciona las implicaciones válidas en un contexto con los conceptos de dicho contexto.

Ahora bien, el conjunto de todas las implicaciones entre atributos válidas en un contexto puede ser excesivamente grande y contener muchas implicaciones triviales. En estos casos, interesa disponer de conjuntos pequeños de implicaciones que sean suficientes para deducir las demás. Para ello, se establece la noción de *base de implicaciones* como un conjunto de implicaciones *adecuado* (cada implicación es válida), *completo* (todas las implicaciones válidas son consecuencia de la base) y *no redundante* (ninguna de sus implicaciones es consecuencia de las restantes). Terminamos el capítulo con la especificación y la prueba de su corrección de un algoritmo que, dado un contexto, construye una base de implicaciones.

Comentemos que, al igual que en la formalización de la programación lógica proposicional, el hecho de haber elegido los conjuntos finitos como tipo base para las especificaciones, ha permitido que las demostraciones realizadas en PVS hayan sido análogas a las originales. Sin embargo, estas especificaciones no son

evaluables. Ahora bien, en el capítulo 8 construimos las correspondientes especificaciones evaluables de las funciones y algoritmos definidos en el capítulo 5, usando para ello el marco para refinamientos descrito en el capítulo 6.

Concretamente, realizamos representaciones de los tipos sobre los que se sustenta la formalización de la teoría, usando listas, y construimos especificaciones evaluables de los operadores de derivación, de las nociones de concepto y subconcepto, y de las funciones que calculan el supremo y el ínfimo de una lista de conceptos, y probamos que refinan a las especificaciones previas.

Terminamos el capítulo construyendo un refinamiento evaluable del algoritmo que calcula los conceptos de un contexto formal finito, y otro del algoritmo que genera una base de implicaciones. La construcción de estas especificaciones se hace sustituyendo cada una de las funciones usadas por la correspondiente función refinada. Resaltar, por último, que la corrección de las especificaciones evaluables es inmediata a partir de la corrección de las especificaciones conjuntistas de dichos algoritmos.

1.2.5. Conclusiones y trabajo futuro

En el capítulo 9 presentamos un resumen de los objetivos conseguidos en el desarrollo de esta memoria, así como de nuestra experiencia en el trabajo con el sistema PVS. Concluimos mostrando posibles vías en las que se puede continuar el trabajo realizado.

1.2.6. Apéndices

Terminamos esta memoria con la inclusión del código íntegro desarrollado en PVS, presentado en cinco apéndices, estructurados en la forma siguiente:

- Los apéndices A y B contienen la extensión de la teoría de conjuntos y la teoría de listas en PVS, que ha sido necesaria para el desarrollo del trabajo realizado.
- El apéndice C contiene el código correspondiente a la biblioteca de teorías en las que se ha desarrollado el marco para refinamientos en PVS: REF y REF–CF.
- El apéndice D muestra el código correspondiente a la biblioteca de teorías que contienen el desarrollo de la formalización de la programación lógica proposicional en PVS: PLP y PLP–e.
- El apéndice E contiene el código correspondiente a la biblioteca de teorías con el desarrollo de la formalización del análisis formal de conceptos en PVS: AFC y AFC–e.

Capítulo 2

Preliminares

En este capítulo exponemos una visión general del sistema PVS, poniendo énfasis en las características del mismo usadas en este trabajo. El objetivo es proporcionar información suficiente acerca del sistema para la comprensión del resto de esta memoria.

2.1. PVS

El sistema PVS[1] (*prototipo de sistema de verificación*) es un entorno interactivo para el desarrollo y el análisis formal de especificaciones. PVS proporciona la mecanización necesaria para aplicar métodos formales a diversos campos, de manera rigurosa y productiva. Ha sido construido en el Laboratorio de Ciencias de la Computación del SRI sobre la experiencia adquirida con otros sistemas. Es el último dentro de una línea de sistemas de verificación, como JVS [26], HDM [72, 73], STP [83] y EHDM [55, 76]. El hecho más significativo de PVS es la integración de un lenguaje de especificación expresivo y un potente demostrador de teoremas. PVS se ha implementado en Common Lisp y usa GNU Emacs como interfaz.

PVS ha sido diseñado para servir como soporte a los métodos formales en ciencias de la computación. En particular, para la detección de errores en especificaciones, así como para la corrección de éstas. PVS ha sido usado en gran número de importantes verificaciones. Entre ellas, cabe destacar las siguientes:

- La corrección de un controlador de un cruce de railes, en tiempo real [78].
- Una inmersión del cálculo de extensiones [84].
- La corrección de transformaciones usadas en síntesis digital [69].
- La corección de protocolos entre sistemas distribuidos [17].

[1] En este trabajo usamos la versión 3.1 del sistema.

- La verificación de un microprocesador comercial en el campo de la aeronáutica, cuya implementación tiene 500.000 transistores [88].

Un resumen de las aplicaciones de PVS llevadas a cabo dentro del SRI están detalladas en [60]. Otras aplicaciones independientes del SRI pueden encontrarse en [40, 42]

En este capítulo vamos a exponer una descripción general de PVS, a fin de proporcionar el bagaje necesario para facilitar la comprensión del código y de las herramientas de PVS usados en esta memoria. Una información más amplia y detallada puede encontrarse en el manual del sistema [64], en el libro de referencia del lenguaje [63] y en la guía del demostrador [81]. Estos manuales, junto con varios tutoriales y otro material de referencia, están disponibles en la página [59].

2.1.1. La lógica de PVS

Un lenguaje de especificación es una lógica, dentro de la cual se puede formalizar el comportamiento de sistemas computacionales. La lógica de PVS es una lógica de orden superior con tipos [2], que contiene tipos funcionales, tipos producto, tipos registro y definiciones de tipos recursivos. Además, el sistema de tipos se ha extendido mediante las nociones de subtipo (análogo a subconjunto) y de tipos dependientes.

Aunque este sistema de tipos tiene algunas ventajas, pues impone una disciplina en la especificación, permite la detección precoz y rápida de errores sintácticos, y es útil en el razonamiento mecanizado, el uso de tipos en especificaciones lógicas no es ampliamente aceptado. En [47], L. Lamport y L. Paulson hacen un análisis de las ventajas e inconvenientes de los lenguajes de especificación tipados.

En las secciones siguientes nos centraremos en mostrar los aspectos fundamentales del lenguaje, así como el uso del demostrador de PVS.

La semántica de una lógica de orden superior se construye asignándole a los tipos bien formados de la lógica, conjuntos; y a los términos de la lógica, elementos del conjunto que representa a su tipo. En PVS, los tipos tienen asociada una semántica acorde con la teoría de conjuntos de Zermelo–Fraenkel, con el axioma de elección. Puesto que no detallamos aquí la semántica de PVS, para un estudio detallado de la misma, es recomendable la lectura de [61].

La teoría de pruebas de PVS está basada en el cálculo de secuentes. Un secuente es de la forma $\Sigma \vdash_\Gamma \Lambda$, donde Γ es el contexto (conjunto de definiciones y teoremas), Σ es un conjunto de fórmulas (*antecedentes*) y Λ es un conjunto de fórmulas (*consecuentes*). El significado de un secuente es que la conjunción de fórmulas de Σ implica la disyunción de fórmulas de Λ. Un exposición detallada de las reglas de inferencia de la lógica de PVS y de las reglas que determinan

[2] La idea intuitiva de una teoría tipada [27] es que existen varias clases de variables, que toman valores en dominios diferentes. Se puede ver como una generalización del concepto habitual de teoría, que dispone de un único dominio.

cuando una expresión en PVS está correctamente tipada, así como las pruebas de adecuación de las mismas puede verse en [61].

En cuanto a los lenguajes basados en la teoría de conjuntos o en lógicas de orden superior, hay un amplio espectro de ellos. El lenguaje VDM [43] está basado en una lógica de primer orden con funciones parciales, extendida con axiomas de tipos de datos. El lenguaje Z [86] está basado en una teoría de conjuntos tipada. El lenguaje de especificación OBJ [32] proporciona marcos para especificar tipos de datos y operaciones sobre tipos de datos, pero la lógica de OBJ es bastante más restrictiva que la de PVS.

El lenguaje más próximo a PVS es EHDM, que está basado en una lógica similar de orden superior con subtipos y generación de obligaciones de prueba, aunque carece de tipos dependientes y su declaración de subtipo es más restrictiva.

Otros sistemas como HOL [36] y TPS [49] también están basados en lógicas de orden superior, aunque sin las nociones de subtipos, tipos dependientes o teorías parametrizadas. Por último, sistemas como Coq [25] y Nuprl [20] están basados en lógicas de orden superior intuicionistas. Coq permite cuantificación sobre subtipos, mientra que Nuprl admite cuantificación sobre una jerarquía de tipos.

2.1.2. El lenguaje de PVS

El sistema PVS contiene: un lenguaje de especificación [3], un analizador sintáctico, un verificador de tipos, un demostrador, bibliotecas de especificaciones y herramientas de localización. Como hemos comentado, el lenguaje de PVS está construido sobre una lógica de orden superior extendida con tipos. La diferencia entre un lenguaje de especificación y un lenguaje de programación reside, fundamentalmente, en la diferencia entre una especificación (que expresa *qué* es computado) y un programa (que expresa *cómo* se computa). Aún así, el lenguaje de especificación de PVS comparte muchos aspectos con un lenguaje de programación. Por ejemplo:

- Los tipos básicos habituales: booleanos, enteros y racionales.
- Los tipos de datos usuales: vectores, registros, listas, sucesiones y tipos abstraactos de datos.
- La capacidad de definir funciones de propósito general.
- Las definiciones por recursión.
- La modularización de especificaciones grandes.

[3] Hay que hacer notar que usaremos el término especificación, tanto en un sentido estricto para hacer referencia a la expresión de una función, como en sentido amplio para referirnos al conjunto de definiciones que forman la construcción de un algoritmo o el desarrollo de una teoría.

2.1.2.1. Teorías

Las especificaciones en PVS se organizan como *teorías* y *tipos de datos*. Las teorías están enlazadas entre sí mediante una lista de importaciones y exportaciones. El preludio de PVS está constituido por un conjunto de teorías fundamentales, que están incluidas en cualquier otra sin necesidad de importarlas explícitamente.

El conjunto de ficheros y teorías que forman una especificación, junto con otros ficheros de información, constituye lo que se denomina un *contexto* de PVS. Dicho contexto conserva la información del estado de una especificación y su verificación, de una sesión de PVS a la siguiente. Las declaraciones de bibliotecas se usan para introducir un nuevo contexto en una especificación. Por ejemplo, con la expresión

```
TD: LIBRARY = ~/Tesis/auxiliares/dominios
```

se introduce el contexto formado por las teorías que se encuentran en ese directorio, y que se pueden importar mediante expresiones de la forma:

```
IMPORTING TD@continuos
```

El directorio `lib` contiene algunas bibliotecas distribuidas con PVS. Para usarlas, no es necesario declararlas previamente; basta con usar como nombre de la biblioteca el del directorio correspondiente. Por ejemplo, para importar la teoría que contiene los esquemas de inducción sobre conjuntos finitos, es suficiente la expresión siguiente:

```
IMPORTING finite_sets@finite_sets_inductions
```

En este trabajo hemos hecho un amplio uso de la biblioteca de conjuntos finitos.

Comentamos ahora cómo se construye una teoría y en una sección posterior explicamos la forma de construir un tipo abstracto de dato. Una *teoría* consta de:

- un identificador, que introduce un nombre para la teoría.
- una lista de parámetros, formada por tipos, subtipos, constantes y expresiones de importación
- cláusulas de importación y exportación: las de exportación pueden ser omitidas, mientras que las de importación son necesarias si se quiere hacer uso del contenido de una teoría.
- hipótesis, en donde se describen propiedades que se suponen en la teoría.
- cuerpo de la teoría, formado por las declaraciones de la teoría, pudiendo contener también cláusulas de importación.

Las teorías se incluyen en ficheros de texto. En cada uno de ellos pueden aparecer una o más teorías. El formato general de una teoría es el siguiente:

```
ejemplo{parámetros} :THEORY
  BEGIN
  {Parte de suposiciones: hipótesis}
  {Cuerpo: Sucesión de especificaciones}
  END ejemplo
```

Estos ficheros, generados por el usuario, tienen extensión `.pvs`. A medida que se van desarrollando pruebas, éstas quedan almacenadas de manera automática en ficheros con el mismo nombre y extensión `.prf`.

El primer paso tras incluir una serie de especificaciones en una teoría será comprobar que no hay errores sintácticos en la misma. Para ello, el analizador sintáctico de PVS analiza la especificaciones realizadas y construye representaciones internas de las mismas, que serán usadas por otras componentes del sistema. Si detecta algún error, sitúa el cursor en el lugar en el que se ha producido dicho error y emite el correspondiente mensaje. A continuación se revisará la teoría en busca de posibles errores semánticos. Es decir, el comprobador de tipos analiza la teoría para garantizar su consistencia semántica, y añade información semántica a la representación interna construida por el analizador sintáctico. Puesto que el sistema de tipos de PVS no es decidible, esto podrá dar lugar, como explicaremos a continuación, a la generación de condiciones de corrección que, en su mayoría, serán demostradas por el sistema de manera automática. Mientras alguna de ellas no haya sido demostrada, cualquier prueba que utilice, directa o indirectamente, elementos de la teoría que la ha generado, se considera incompleta.

Entre las expresiones que se pueden usar para describir las declaraciones del cuerpo de una teoría, el lenguaje de PVS ofrece una amplia variedad, que incluye:

- expresiones booleanas, usando `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `IMPLIES`, `WHEN`, `IFF`.
- expresiones condicionales, usando `IF-THEN-ELSE` y `COND`.
- expresiones numéricas
- expresiones funcionales
- expresiones de asignación, incluyendo expresiones con cuantificadores y expresiones lambda (en particular, las expresiones que denotan conjuntos, pues estos se representan mediante predicados).
- expresiones `LET` y `WHERE`, en su forma habitual.
- expresiones `CASES` para usarlas sobre tipos abstractos de datos.

2.1.2.2. Tipos

El sistema de tipos de PVS está basado en una equivalencia estructural, lo que significa que la noción de tipo está íntimamente relacionada con la idea de conjunto. En este sentido, dos tipos son iguales si tienen los mismos elementos.

Las declaraciones de tipos se usan para introducir nuevos nombres de tipos en un contexto. Pueden ser de las formas siguientes:

- Tipos no interpretados: `T: TYPE`.
- Tipos no vacíos: `T: NONEMPTY_TYPE` o `S: TYPE+`
- Subtipos no interpretados: `S: TYPE FROM T`.
- Tipos interpretados: `T: TYPE = int` (conjunto de los números enteros).
- Tipos enumerados: `T: TYPE = {a, b, c}`

Dentro de una teoría, los tipos pueden ser definidos a partir de los tipos básicos o de otros ya construidos, mediante los siguientes *constructores de tipos*:

- Tipos funcionales: $[T_1, \ldots, T_n \rightarrow T]$ representa el tipo de las funciones de $T_1 \times \cdots \times T_n \rightarrow T$.
- Tipos producto: $[T_1, \ldots, T_n]$. Los elementos de este tipo son n–tuplas cuyas componentes son elementos del tipos correspondiente. Asociadas con cada tipo producto, se dispone de las funciones de proyección `PROJ_i`, donde `PROJ_i:` $[T_1, \ldots, T_n] \rightarrow T_i$ proporciona el valor de la componente i–sima de la tupla.
- Tipos registro: $[\#\ a_1 : T_1, \ldots, a_n : T_n\ \#]$. Es un tipo similar al tipo producto, pero en éstos no importa el orden y las funciones de acceso son las funciones a_i.

Cabe destacar como hecho significativo de PVS la capacidad para definir un subtipo a partir de un predicado. Dicho subtipo estará formado por los elementos de un tipo que verifiquen el predicado dado. Por ejemplo:
`nonneg_int: NONEMPTY_TYPE =` $\{i: \texttt{int} \mid i \geq 0\}$ `CONTAINING 0`

Hay que hacer notar que los tipos que acabamos de comentar pueden ser tipos dependientes, en el sentido de que alguna de sus componentes puede depender del tipo de componentes previas. Por ejemplo:
`pila: [# tamaño: nat, elementos: [{n:nat | n`$\leq$`tamaño } → T #]`

Comentemos, por último, que dado un predicado `P`, mediante `(P)` se representa el tipo formado por los elementos que verifican `P`.

La lógica de orden superior proporciona gran riqueza expresiva, pero ha de ser estricta en relación al sistema de tipos para evitar inconsistencias. Las especificaciones en PVS son fuertemente tipadas, es decir, toda expresión tiene asociado

un tipo (no necesariamente único). La verificación de tipos, que consiste en comprobar que cada elemento de una expresión posee el tipo esperado, es pues, un factor importante, además de una forma de comprobar que una especificación tiene sentido.

La comprobación de tipos puede generar expresiones denominadas *condiciones de corrección de tipos* (TCCs), que han de ser probadas para la admisión de una especificación.

2.1.2.3. Funciones

En la lógica de orden superior, un elemento es una función, una tupla, un registro o pertenece a un tipo básico. En general, una función en el lenguaje de PVS puede representar tanto una función total como una función parcial, usando para ello la noción de predicado de subtipo. Es decir, PVS admite funciones parciales dentro del marco de una lógica de funciones totales, enriquecida con un sistema de tipos que incluye predicados de subtipo. Este uso está en consonancia con la forma de definir funciones en matemáticas, explicitando el dominio sobre el que están definidas.

En el lenguaje de PVS podemos declarar *constantes*, especificando su tipo y, a veces, un valor concreto. Estas constantes pueden ser interpretadas o no interpretadas. Por ejemplo:

```
n: int
p: int = 7
f: [int -> int] = (lambda (x:int): 2*x)
```

Las *funciones recursivas* se tratan como declaraciones de constantes, con la salvedad de que hay que proporcionarle una función de medida para garantizar su terminación, puesto que la función ha de ser total (definida en todo su dominio). Por ejemplo, la definición de la función que calcula el factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n = 0 THEN 1 ELSE n * factorial(n - 1) ENDIF
  MEASURE n
```

donde la función de medida representada por `n` es `(lambda(n:nat):n)` [4] genera las obligaciones siguientes:

```
factorial_TCC1: OBLIGATION
  FORALL (n1: nat): NOT n1 = 0 IMPLIES n1 - 1 >= 0

factorial_TCC2: OBLIGATION FORALL (n:nat): NOT n = 0 IMPLIES n-1 < n
```

[4] En el preludio de PVS se ha especificado cuándo una relación de orden está bien fundamentada y se ha establecido un axioma que asegura la buena fundamentación de la relación $<$ en los números naturales.

La primera exige que el argumento de factorial en la llamada recursiva, bajo las condiciones en las que ésta se produce, sea del tipo `nat`, y la segunda para establecer que la medida de los argumentos de las llamadas recursivas de la función es decreciente. Ambas condiciones son demostradas automáticamente por el sistema.

Por último, comentemos que las definiciones mediante recursión cruzada no están permitidas en el lenguaje de PVS.

2.1.2.4. Juicios

Como hemos comentado, una de las características más útiles de PVS es la capacidad de definir predicados de subtipo, lo cual puede dar lugar a gran cantidad de TCCs redundantes. El número de TCCs puede ser controlado mediante el uso de `juicios`. Los juicios constituyen una forma de evitar la repetida generación de la misma TCC, poniendo a disposición del demostrador propiedades acerca de los operadores, cuando actúan sobre subtipos. Hay dos tipos de juicios:

- Juicios de constante: se usan para establecer que una constante tiene un tipo más específico que su tipo declarado:

```
f(x:int):int = x*x
image_f_nat: JUDGEMENT f(x:int) HAS_TYPE nat

Se genera la siguieente TCC:
image_f_nat: OBLIGATION FORALL (x: int): f(x) >= 0;
```

  Con ello, en un contexto en el que se tenga este juicio no se generarán TCCs para asegurar que `f(x)` es un número natural.

- Juicios de subtipo: se usan para establecer que un tipo está contenido en otro:

```
nonzero_real: NONEMPTY_TYPE = {r:real| r/=0} CONTAINING 1
nonzero_rational: NONEMPTY_TYPE = {r:rational| r/=0} CONTAINING 1
nzrat_is_nzreal: JUDGEMENT nonzero_rational SUBTYPE_OF nonzero_real
```

Ante un juicio se genera la correspondiente TCC para establecer su validez y esta información es utilizada para evitar la generación del mismo en usos posteriores del tipo o la constante sobre la que se ha establecido el juicio.

2.1.2.5. Conversiones

Hay que hacer notar que, en el proceso realizado por el verificador de tipos, influyen también las funciones declaradas como *conversiones*. Las conversiones se declaran para evitar conflicto de tipos. Veamos el siguiente ejemplo: consideremos la función que transforma una lista de elementos de T en un conjunto, que se declara como conversión.

```
list2set(l) : RECURSIVE set[T] =
  CASES l OF
    null: emptyset[T],
    cons(x, y): add(x, list2set(y))
  ENDCASES
 MEASURE length

CONVERSION list2set
```

De esta forma, al ser declarada como conversión, si se le aplica a listas una función cuyo dominio está formado por conjuntos, no se produce un conflicto de tipos, sino que la función se le aplica a la imagen de las listas por la función `list2set`. Por ejemplo, si definimos

```
g(l1,l2: list[T]): set[T] = union(l1,l2)
```

Internamente, la definición de la función g que ha almacenado el sistema es

```
g(l1, l2: list[T]): set[T] = union(list2set[T](l1), list2set[T](l2))
```

Algunas de las funciones definidas en el preludio de PVS se han declarado como conversiones. Comentemos dos de las que más se hacen notar: `restrict` y `extend`. La función `restrict` permite que una función definida sobre un tipo se aplique a los elementos de un subtipo:

```
restrict [T: TYPE, S: TYPE FROM T, R: TYPE]: THEORY
 BEGIN
  f: VAR [T -> R]
  s: VAR S
  restrict(f)(s): R = f(s)
  CONVERSION restrict
  ...
 END restrict
```

La función `extend` se puede considerar inversa de la función `restrict`, pues permite extender la definición de una función a un supertipo, asignándole un valor predeterminado a los elementos del supertipo que no lo son del subtipo:

```
extend [T: TYPE, S: TYPE FROM T, R: TYPE, d: R]: THEORY
 BEGIN
  f: VAR [S -> R]
  t: VAR T
  extend(f)(t): R = IF S_pred(t) THEN f(t) ELSE d ENDIF
  restrict_extend: LEMMA restrict[T,S,R](extend(f)) = f
 END extend
```

Comentemos, también, que mediante la acción de restringir primero una función a un subtipo, y extenderla después al tipo del que procede, no se obtiene la misma función. En el preludio se proporciona una conversión para extensiones de funciones booleans, tomando `false` como valor predeterminado. Así, por ejemplo, un conjunto de números naturales "es" un conjunto de números enteros.

```
extend_bool [T: TYPE, S: TYPE FROM T]: THEORY
 BEGIN
  CONVERSION extend[T, S, bool, false]
 END extend_bool
```

Otro aspecto a tener en cuenta es el tratamiento de la igualdad entre conjuntos. Un conjunto está determinado por los elementos de un tipo que verifican un predicado, no pudiendo contener elementos de distintos tipos. Ahora bien, en ocasiones expresamos la igualdad de dos conjuntos que no están declarados sobre el mismo tipo. Por ejemplo, consideremos los conjuntos siguientes:

```
A: set[nat] = {n: nat | even?(n)}
B: set[int] = {n: int | even?(n) AND n >= 0}
```

El conjunto A está especificado como el conjunto formado por números naturales pares, y B como el conjunto formado por los números enteros positivos y pares. Evidentemente, ambas especificaciones representan el mismo conjunto y, por tanto, A y B han de ser iguales. Ahora bien, como ambos objetos están determinados por el tipo de sus elementos, dentro de la lógica de PVS, las igualdades $A = B$ y $B = A$ no tienen el mismo significado. De hecho, si expresamos dichas igualdades como lemas:

```
l1: LEMMA A = B
l2: LEMMA B = A
```

internamente se almacena:

```
l1: LEMMA A = restrict[int, nat, boolean](B)
l2: LEMMA B = extend[int, nat, bool, FALSE](A)
```

2.1.2.6. Fórmulas

Las declaraciones de *fórmulas* en PVS se usan para introducir:

- axiomas, mediante las claves `AXIOM` o `POSTULATE`.
- hipótesis, mediante la clave `ASSUMPTION` (en el apartado de hipótesis de una teoría).
- obligaciones, mediante la clave `OBLIGATION` (sólo por el sistema).
- teoremas, mediante las claves `CLAIM`,..., `LEMMA`,..., `THEOREM`.

Las declaraciones de fórmulas pueden contener variables libres, en cuyo caso el demostrador considera su clausura universal.

2.1.3. Tipos abstractos de datos

Con respecto a los *tipos abstractos de datos* (TAD), PVS proporciona un potente mecanismo para definirlos. Dicho mecanismo es más sofisticado que el usado por Boyer–Moore en [13]. Para especificar un TAD hay que proporcionar un conjunto de *constructores*, *funciones de acceso* y *reconocedores*. Veamos, como ejemplo, la representación mediante un TAD de las fórmulas proposicionales, que hemos usado en este trabajo. Consideraremos que una *fórmula proposicional* sobre un universo `T` se define, inductivamente, como sigue:

- $\bot$ es una fórmula proposicional.
- Un símbolo proposicional es una fórmula proposicional.
- Si F es una fórmula proposicional, $\neg F$ también lo es.
- Si F_1 y F_2 son fórmulas proposicionales, $F_1 \wedge F_2$ también lo es.

Para especificar el TAD que las representa, se consideran como *constructores* las funciones `falso`, `a`, `~` y `&`; como *funciones de acceso* `simb`, `fla`, `fla1`, `fla2`; y como *reconocedores* las funciones `falsa?`, `atomo?`, `negacion?` y `conjuncion?`. La especificación en PVS del conjunto de fórmulas proposicionales es la siguiente:

```
formula_prop[T: TYPE+]: DATATYPE
  BEGIN
  falso                          : falsa?
  a(simb: T)                     : atomo?
  ~(fla: formula_prop)           : negacion?
  &(fla1, fla2: formula_prop)    : conjuncion?
  END formula_prop
```

Cuando se le aplica el verificador de tipos a un TAD se crean, de forma automática, tres nuevas teorías que proporcionan los axiomas y los principios de inducción necesarios para asegurar que el TAD es el álgebra inicial definida por los constructores. Estas teorías son `formula_prop_adt`, `formula_prop_adt_map` y `formula_prop_adt_reduce`, y están contenidas en el fichero `formula_prop_adt.pvs`.

La primera de ellas, `formula_prop_adt`, contiene [5]:

- Axiomas de extensionalidad para cada constructor, por ejemplo:

```
formula_prop_and_extensionality: AXIOM
  ∀ (conjuncion?_var: (conjuncion?),
             conjuncion?_var2: (conjuncion?)):
    fla1(conjuncion?_var) = fla1(conjuncion?_var2) AND
    fla2(conjuncion?_var) = fla2(conjuncion?_var2)
    ⇒ conjuncion?_var = conjuncion?_var2;
```

[5] En toda la memoria, las expresiones que se muestran del código PVS se han modificado ligeramente para aumentar su legibilidad

- Axiomas de unicidad de la construcción:

```
formula_prop_and_eta: AXIOM
 ∀ (conjuncion?_var: (conjuncion?)):
 (fla1(conjuncion?_var) & fla2(conjuncion?_var))=conjuncion?_var
```

- Relaciones entre las funciones de acceso y los constructores, para cada par:

```
formula_prop_fla1_and: AXIOM
  ∀ (and1_var: formula_prop, and2_var: formula_prop):
    fla1(and1_var & and2_var) = and1_var;
```

- Un esquema de inducción:

```
formula_prop_induction: AXIOM
  ∀ (p: [formula_prop -> boolean]):
    (p(falso) &
      (∀ (a1_var: T): p(a(a1_var))) &
       (∀ (tilde1_var: formula_prop):
          p(tilde1_var) ⇒ p(~tilde1_var))
        &
        (∀ (and1_var: formula_prop, and2_var: formula_prop):
          p(and1_var) & p(and2_var) ⇒ p(and1_var & and2_var)))
     ⇒
     (∀ (formula_prop_var: formula_prop): p(formula_prop_var));
```

- Funciones de distribución de predicados sobre fórmulas[6]:

```
                                                                  0
 every(p: PRED[T], a1: formula_prop):  boolean =
     CASES a1
       OF falso: TRUE,
          a(a1_var): p(a1_var),
          ~(tilde1_var): every(p)(tilde1_var),
          &(and1_var, and2_var): every(p)(and1_var) AND
                                 every(p)(and2_var)
       ENDCASES;
```

[6]Se dispone también de estas funciones en forma parametrizada.

```
some(p: PRED[T], a1: formula_prop):  boolean =
    CASES a1
      OF falso: FALSE,
         a(a1_var): p(a1_var),
         ~(tilde1_var): some(p)(tilde1_var),
         &(and1_var, and2_var): some(p)(and1_var) OR
                                some(p)(and2_var)
      ENDCASES;
```

- Una relación de subtérmino, para comprobar si una fórmula ocurre como subtérmino de otra:

```
subterm(x, y: formula_prop):  boolean =
    x = y OR
     CASES y
       OF falso: FALSE,
          a(a1_var): FALSE,
          ~(tilde1_var): subterm(x, tilde1_var),
          &(and1_var, and2_var):
            subterm(x, and1_var) OR subterm(x, and2_var)
       ENDCASES;
```

- Un relación de orden:

```
<<:  (well_founded?[formula_prop]) =
    LAMBDA (x, y: formula_prop):
      CASES y
        OF falso: FALSE,
           a(a1_var): FALSE,
           ~(tilde1_var): x = tilde1_var OR x << tilde1_var,
           &(and1_var, and2_var):
             (x = and1_var OR x << and1_var) OR
              x = and2_var OR x << and2_var
        ENDCASES;
```

- Un axioma de buena fundamentación, lo que permite usar la función anterior como función de medida en las definiciones recursivas sobre el tipo de dato `formula_prop`.

```
formula_prop_well_founded: AXIOM well_founded?[formula_prop](<<);
```

- Funciones sobre naturales y ordinales, que posibilitarán la definición de medidas para definiciones recursivas sobre fórmulas proposicionales. Otras funciones más generales del mismo tipo se definen en la tercera teoría, `formula_prop_adt_reduce`.

```
reduce_nat(falsa?_fun: nat, atomo?_fun: [T -> nat],
           negacion?_fun: [nat -> nat],
           conjuncion?_fun: [[nat, nat] -> nat]):
 [formula_prop -> nat] =

  LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> nat] =
            reduce_nat(falsa?_fun, atomo?_fun, negacion?_fun,
                       conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun,
             a(a1_var): atomo?_fun(a1_var),
             ~(tilde1_var): negacion?_fun(red(tilde1_var)),
             &(and1_var, and2_var):
               conjuncion?_fun(red(and1_var), red(and2_var))
          ENDCASES;

reduce_ordinal(falsa?_fun: ordinal, atomo?_fun: [T -> ordinal],
               negacion?_fun: [ordinal -> ordinal],
               conjuncion?_fun: [[ordinal, ordinal] -> ordinal]):
 [formula_prop -> ordinal] =
    LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> ordinal] =
            reduce_ordinal(falsa?_fun, atomo?_fun, negacion?_fun,
                           conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun,
             a(a1_var): atomo?_fun(a1_var),
             ~(tilde1_var): negacion?_fun(red(tilde1_var)),
             &(and1_var, and2_var):
               conjuncion?_fun(red(and1_var), red(and2_var))
          ENDCASES;
```

La segunda teoría, `formula_prop_adt_map`, toma dos tipos `T` y `T1` como parámetros, importa la teoría `formula_prop_adt` y define las siguientes funciones:

0

```
formula_prop_adt_map[T: TYPE+, T1: TYPE+]: THEORYa
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;
  ENDASSUMING
```

```
IMPORTING formula_prop_adt

map(f: [T -> T1], a1: formula_prop[T]):  formula_prop[T1] =
    CASES a1
      OF falso: falso,
         a(a1_var): a(f(a1_var)),
         ~(tilde1_var): ~map(f, tilde1_var),
         &(and1_var, and2_var): map(f, and1_var) & map(f, and2_var)
      ENDCASES;
```

2.1.4. El demostrador de PVS

El verificador de pruebas de PVS puede considerarse como un demostrador de teoremas interactivo. Es más automático que los editores de prueba de bajo nivel y más controlable que los demostradores completamente automáticos. PVS combina el control del usuario en el desarrollo de las pruebas con la automatización de los pasos elementales. Proporciona mayor automatismo que los demostradores HOL o Nuprl, y mayor control que los demostradores Otter, Nqthm o ACL2.

En la demostración de un teorema el usuario elige los comandos a aplicar y PVS los ejecuta mostrando los resultados obtenidos. Existen comandos elementales para tratar las reglas proposicionales, el razonamiento con cuantificadores, la inducción, la reescritura, la simplificación utilizando propiedades aritméticas, procedimientos de decisión ante la igualdad, etc. Estos comandos pueden ser combinados para formar *estrategias*.

El objetivo a probar por el demostrador de PVS es un secuente, es decir, una serie de antecedentes y consecuentes, que se escribe en la forma estándard: $\mathtt{A}_1, \ldots, \mathtt{A}_n \vdash \mathtt{B}_1, \ldots, \mathtt{B}_m$. La semántica de un secuente coincide con la habitual, la conjunción de los antecedentes debe implicar la disyunción de los consecuentes; esto es, ha de ser cierta la fórmula $(\mathtt{A}_1 \wedge \cdots \wedge \mathtt{A}_n) \Rightarrow (\mathtt{B}_1 \vee \cdots \vee \mathtt{B}_m)$. Una prueba comienza con un objetivo de la forma $\vdash \mathtt{B}$, donde $\mathtt{B}$ es el teorema a demostrar.

En el desarrollo de una prueba se genera un árbol de demostración en el que todas las hojas han de ser ciertas. Los nodos del árbol son secuentes. El secuente correspondiente a la raíz del árbol es el objetivo inicial. Cuando se completa la prueba de una rama del árbol, el demostrador busca la siguiente rama sin terminar, presentando al usuario el último objetivo alcanzado para su demostración. Este proceso continúa hasta que el usuario lo interrumpa, con lo que la prueba quedaría sin terminar, o hasta que no quede ninguna rama pendiente, en cuyo caso la prueba se habrá completado.

Ante un objetivo, el usuario debe utilizar alguno de los comandos del sistema que, o bien demostrará el objetivo o bien dará lugar a nuevos subobjetivos. De esta forma se va construyendo el árbol de demostración.

Podemos clasificar los comandos implementados por el verificador de pruebas de PVS en tres grandes grupos, de los que comentamos los ejemplos más

significativos:

- Comandos "creativos": son los que se refieren a los pasos de prueba que explicitaríamos en una prueba a mano. Por ejemplo: `induct` (la prueba se hace por inducción), `inst` (instanciación de un cuantificador universal en las hipótesis, o de un cuantificador existencial en las conclusiones), `lemma` (uso de un resultado determinado) y `case` (distinción de casos en la prueba).
- Comandos "burocráticos": son aquellos que, en una prueba a mano, se hacen de forma implícita. Por ejemplo: `flatten` (simplificación disyuntiva), `skosimp*` (skolemización y simplificación), `expand` (expandir una definición), `replace` (reemplazar un término por otro equivalente) e `hide` (ocultar hipótesis o conclusiones irrelevantes).
- Comandos "potentes": son comandos que realizan acciones combinadas, mediante los cuales se pueden probar objetivos en un sólo paso. Por ejemplo: `simplify`, `prop`, `assert` y `grind`.

Entre las estrategias disponibles en la versión 3.1 de PVS, la que más hemos usado ha sido `grind-with-lemmas`, que es un potente commando que expande definiciones, instancia los lemas indicados y aplica procedimientos de decisión.

2.1.5. Un ejemplo de prueba

Veamos un pequeño ejemplo de uso del demostrador de PVS. Para ello, consideremos el problema consistente en probar que con monedas de valores 3 y 5 se puede obtener cualquier cantidad superior o igual a 8. Para ello, hay que probar lo siguiente: $\forall n \in \mathbb{N} : (\exists a, b \in \mathbb{N} : n + 8 = 3a + 5b)$.

Especificamos este enunciado en PVS como sigue:

```
monedas : THEORY
 BEGIN
  n, a, b: VAR nat
  monedas: LEMMA (FORALL n: (EXISTS a, b: n+8 = 3*a + 5*b))
 END monedas
```

La prueba "manual" de este resultado la haríamos por inducción en n:

- Caso base: $n = 0$. Basta tomar $a = b = 1$.
- Paso recursivo: suponemos que existen a y b tales que $n + 8 = 3a + 5b$, y probamos el resultado para $n + 1$.

  Para ello, consideramos dos casos:

  (1) $b = 0$. En este caso, $n+8 = 3a$, con lo que $a \geq 3$ pues $n > 0$. Entonces, $n + 1 + 8 = 3(a - 3) + 5 \times 2$. Por tanto, basta tomar $a - 3$ y 2.

(2) $b \neq 0$. Entonces: $n+1+8 = 3(a+2)+5(b-1)$. Por tanto, basta tomar $a+2$ y $b-1$.

Veamos cómo se puede interactuar con el demostrador de PVS para realizar esta prueba en el sistema. En primer lugar, con el comando `induct` le indicamos que la prueba será por inducción en n:

```
monedas :

  |-------
{1}   (FORALL n: (EXISTS a, b: n + 8 = 3 * a + 5 * b))

Rule? (induct "n")
```

Con ello, se generan dos subobjetivos. El primero de ellos se prueba proporcinándole los valores para a y b, mediante el comando `inst` y probando lo que resulta mediante `assert`.

```
Inducting on n on formula 1,
this yields  2 subgoals:
monedas.1 :

  |-------
{1}   EXISTS a, b: 0 + 8 = 3 * a + 5 * b

Rule? (inst 1 1 1)
Instantiating the top quantifier in 1 with the terms:
 1, 1,
this simplifies to:
monedas.1 :

  |-------
{1}   0 + 8 = 3 * 1 + 5 * 1

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of monedas.1.
```

El demostrador muestra ahora el segundo subobjetivo:

```
monedas.2 :

  |-------
```

```
{1}   FORALL j:
        (EXISTS a, b: j + 8 = 3 * a + 5 * b) IMPLIES
         (EXISTS a, b: j + 1 + 8 = 3 * a + 5 * b)
```

Se introducen constantes de Skolem para las variables cuantificadas universalmente en la consecuencia del secuente, y existencialmente en los antecedentes:

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
monedas.2 :

{-1}  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
{1}   EXISTS a, b: j!1 + 1 + 8 = 3 * a + 5 * b
```

Establecemos dos casos, mediante el comando `case`:

```
Rule? (case "b!1 = 0")
Case splitting on
   b!1 = 0,
this yields  2 subgoals:
monedas.2.1 :

{-1}  b!1 = 0
[-2]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
[1]   EXISTS a, b: j!1 + 1 + 8 = 3 * a + 5 * b
```

Para resolver el primero de ellos, proporcionamos la solución `a!1 - 3` y 2, lo que hace que se genere una TCC, asegurando que `a!1 - 3` es de tipo `nat`. Tanto la TCC como lo que resulta de la instanciación se prueban mediante `assert`

```
Rule? (inst + "a!1 - 3" "2")
Instantiating the top quantifier in + with the terms:
 a!1 - 3, 2,
this yields  2 subgoals:
monedas.2.1.1 :

[-1]  b!1 = 0
[-2]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
{1}   j!1 + 1 + 8 = 3 * (a!1 - 3) + 5 * 2
```

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of monedas.2.1.1.

monedas.2.1.2 (TCC):

[-1]  b!1 = 0
[-2]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
{1}   a!1 - 3 >= 0

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of monedas.2.1.2.

This completes the proof of monedas.2.1.
```

En este caso, se procede de manera análoga.

```
monedas.2.2 :

[-1]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
{1}   b!1 = 0
[2]   EXISTS a, b: j!1 + 1 + 8 = 3 * a + 5 * b

Rule? (inst + "a!1 + 2" "b!1 - 1")
Instantiating the top quantifier in + with the terms:
 a!1 + 2, b!1 - 1,
this yields  2 subgoals:
monedas.2.2.1 :

[-1]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
[1]   b!1 = 0
{2}   j!1 + 1 + 8 = 3 * (a!1 + 2) + 5 * (b!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of monedas.2.2.1.
```

```
monedas.2.2.2 (TCC):

[-1]  j!1 + 8 = 3 * a!1 + 5 * b!1
  |-------
{1}   b!1 - 1 >= 0
[2]   b!1 = 0

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of monedas.2.2.2.

This completes the proof of monedas.2.2.

This completes the proof of monedas.2.

Q.E.D.

Run time  = 0.78 secs.
Real time = 1.32 secs.
```

Comentemos que el detalle de las pruebas que mostraremos a lo largo de esta memoria se hará describiendo la prueba "matemática" que se ha realizado con el demostrador de PVS.

2.1.6. El evaluador básico

Uno de los objetivos de este trabajo es realizar especificaciones, verificar su corrección y transformarlas en otras especificaciones evaluables. Veamos qué fragmento del lenguaje de PVS es evaluable.

El lenguaje de especificación de PVS ha sido diseñado fundamentalmente para ser expresivo, no para ser evaluable. Sin embargo, un amplio fragmento de PVS resulta ser evaluable como un lenguaje funcional razonablemente eficiente, lo que se consigue generando código Common Lisp a partir de PVS. En [80] se describe el proceso de traducción de PVS a código ejecutable Common Lisp. En el proceso de generación de código, se usan vectores de hebra simple para actualizaciones destructivas seguras, lo que hace que el código generado sea competitivo en rendimiento con un código artesanal, escrito en un lenguaje de bajo nivel como C.

No todas las especificaciones en PVS son ejecutables. Ejemplos de expresiones que no son ejecutables son las siguientes:

- Expresiones con variables libres.

- Expresiones con cuantificación sobre dominios infinitos.
- Términos con símbolos de función no interpretados.
- Igualdades entre términos de orden superior.

En general, las expresiones evaluables son expresiones básicas (no contienen ninguno de los casos anteriores) construidas a partir de los tipos básicos (los que no contienen tipos de orden superior ni tipos no interpretados).

Las traducciones y optimizaciones descritas en [80] están implementadas en el *evaluador básico* de PVS, que se invoca desde la teoría, mediante el commando `M-x pvs-ground-evaluator`. Las expresiones que vayan a ser evaluadas han de escribirse en la llamada del evaluador <`GndEval`> entre comillas. Entonces, PVS traduce a Lisp el fragmento relevante de la especificación, lo compila y ejecuta el corrrespondiente código Lisp. La traducción y compilación se hace de forma perezosa (sólo lo necesario). Esto puede ocasionar que la primera evaluación sea más lenta, mientras que las siguientes, que usarán el código Lisp ya compilado, serán más rápidas. Hay que observar que las evaluaciones no serán correctas si no se han probado todas las TCCs.

Hay que tener en cuenta también que, frecuentemente, es más cómodo delarar una constante con la expresión que se desea evaluar, realizar la verificación de tipos y, después, evaluar dicha constante en el evaluador básico. Esta recomendación se vuelve imprescindible cuando la expresión que se va a evaluar contiene cadenas pues, en este caso, el doble uso de las comillas da lugar a confusión.

Como ejemplo, veamos la evaluación de la función factorial, definida previamente:

```
<GndEval> "factorial(4)"
==> 24
<GndEval> "factorial(50)"
==> 30414093201713378043612608166064768844377641568960512000000000000
```

Las evaluaciones de los algoritmos presentados en esta memoria se encuentran en los capítulos 7 y 8 de la misma.

2.1.7. Interpretaciones de teorías en PVS

En [62] se describe el mecanismo para la interpretación de teorías en PVS, que es una extensión del mecanismo de parametrización de teorías. Este mecanismo hace posible probar que una colección de teorías son interpretadas correctamente por otra colección de teorías, en las que el usuario ha proporcionado una especificación concreta para los tipos y las constantes no interpretadas.

Las interpretaciones de teorías tiene una larga historia en la lógica de primer orden [82, 27]. Su uso ha sido también relevante en la lógica de orden superior y en la teoría de tipos, dentro de lenguajes como EHDM, IMPS [29], HOL, Maude

[17], Extended ML [44] y SPECWARE [87]. En ellos, las teorías abstractas se refinan en otras más concretas, a través de interpretaciones.

Las interpretaciones de teorías en PVS proporcionan funciones que relacionan tipos y constantes no interpretados de la teoría fuente con los de la teoría interpretada. El constructor de estas funciones define la traducción básica que asocia a los elementos no interpretados de la teoría origen, expresiones de la teoría objetivo. Ahora bien, esto ha de hacerse preservando la consistencia, para lo cual todos los axiomas y teoremas de la teoría origen han de ser ciertos en la teoría objetivo. Como los teoremas se prueban a partir de los axiomas, es suficiente garantizar que dichos axiomas se tienen en la teoría objetivo. Para ello, los axiomas relacionados con la interpretación se convierten en obligaciones de prueba.

Veamos el siguiente ejemplo:

```
int1[T: TYPE, e: T]: THEORY
 BEGIN
  R: TYPE+
  c: R
  f: [R -> T]
  ax: AXIOM EXISTS (x,y: R): f(x) /= f(y)
  lema1: LEMMA EXISTS (x:T): x /= e
 END int1
```

La teoría `int1` tiene parámetros, tipos y constantes no interpretados, así como un axioma y un lema que se puede probar a partir de éste. Queremos hacer las siguientes substituciones:

```
T  ←  int
e  ←  0
R  ←  bool
c  ←  TRUE
f  ←  LAMBDA (x:bool): IF x THEN 1 ELSE 0 ENDIF
```

Para ello, se construye la teoría `int2`, como sigue:

```
int2: THEORY
 BEGIN
  IMPORTING int1[int,0]
              {{R := bool,
                c := true,
                f(x:bool) := IF x THEN 1 ELSE 0 ENDIF }}

  lema2: LEMMA EXISTS (x:int): x /= 0
 END int2
```

El verificador de tipos genera la siguientte TCC:

```
IMP_int1_ax_TCC1: OBLIGATION
  EXISTS (x, y: bool): IF x THEN 1 ELSE 0 ENDIF /=
                       IF y THEN 1 ELSE 0 ENDIF;
```

Una vez probada ésta, el lema `lema2` se prueba directamente, usando `lema1`.

Una ventaja de usar funciones en vez de parámetros es que no es necesario interpretar todos los elementos de la teoría fuente. Con este mecanismo, es fácil especificar una teoría general y disponer de instancias suyas. Las teorías como grupos, anillos o cuerpos son estructuras clásicas que se pueden axiomatizar fácilmente en términos de tipos y constantes no interpretadas. Y , a partir de ellas, obtener distintos ejemplos de dichas estructuras.

```
grupo: THEORY
 BEGIN
 G: TYPE+
 e: G
 +: [G,G -> G]
 -: [G -> G]
 x,y,z: VAR G
 ax_asociativa: AXIOM x + (y + z) = (x + y) + z
 ax_identidad: AXIOM x + e = x
 ax_inverso: AXIOM x + (- x) = e AND (- x) + x = e
 END grupo

Ejemplo:

G1: THEORY
 BEGIN
  IMPORTING grupo{{ G:= int, e := 0, + := +, - := -}}
 END G1
```

Como hemos visto, la teoría de interpretaciones de PVS se puede usar para refinar una especificación abstracta en otra concreta, que puede ser evaluable. Con ello, operaciones no ejecutables (como las de la teoría `grupo`) pueden transformarse en evaluables (las de la teoría `G1`) como resultado de la interpretación.

En este trabajo, hemos usado la interpretación de teorías para, una vez establecida una relación formal (mediante funciones no interpretadas) entre distintas representaciones de los elementos de la lógica proposicional, realizar una interpretación de éstas mediante funciones concretas, garantizando así la consistencia de dicha relación.

Ahora bien, no hemos usado la teoría de interpretación de PVS para obtener refinamientos de las especificaciones que usan la teoría de conjuntos finitos desarrollada en PVS. La razón es que esta teoría no está desarrollada de forma axiomática sobre un tipo no interpretado, por lo que el mecanismo de interpretación de teorías no es aplicable.

Otros sistemas también disponen de mecanismos similares. Por ejemplo, EHDM e IMPS permiten interpretaciones de teorías de forma similar, usando la noción de función entre módulos. El lenguaje Standard ML [56] tiene un sistema modular de estructuras, con funtores que actúan entre estructuras. El mecanismo de PVS es similar a éste, pero para un lenguaje de especificación en vez de para un lenguaje de programación. Nqthm y ACL2 disponen de la utilidad de instanciación funcional. El lenguaje SPECWARE usa la interpretación de teorías como un mecanismo para el refinamiento de especificaciones en código ejecutable.

2.2. Bibliotecas auxiliares

La filosofía seguida en este trabajo en el diseño de especificaciones ha sido usar el tipo de los conjuntos, con objeto de facilitar el razonamiento. En particular, hemos usado conjuntos finitos, con lo que ha sido posible transformar las especificaciones basadas en el tipo de los conjuntos finitos en otras especificaciones evaluables, mediante refinamiento de tipos y operaciones.

2.2.1. Teoría de conjuntos finitos

Como hemos comentado, un conjunto en PVS es, en realidad, un predicado. En el preludio se construye la teoría `sets` como:

```
sets [T: TYPE]: THEORY
 BEGIN
  set: TYPE = setof[T]
  x, y: VAR T
  a, b, c: VAR set

   ...
```

donde `setof: TYPE = [t ->bool]`.

En ella se definen las operaciones sobre conjuntos y se establecen sus propiedades. En particular, se define una función de elección para conjuntos no vacíos, `choose`, usando la función de elección `epsilon` para tipos no vacíos:

```
epsilons [T: NONEMPTY_TYPE]: THEORY
 BEGIN
  p: VAR pred[T]
  x: VAR T

  epsilon(p): T
  epsilon_ax: AXIOM (EXISTS x: p(x)) => p(epsilon(p))
 END epsilons

choose(p: (nonempty?)): (p) = epsilon(p)
```

Se definen los conjuntos finitos como el subtipo de `set[T]` que satisface el predicado `is_finite`. Dicho predicado establece la existencia de una función inyectiva en `below[N]` para algún número natural `N`, donde `below[N]` representa el conjunto de los números naturales menores que `N`.

```
below(i):  TYPE = {s: nat | s < i}
is_finite(s): bool = (EXISTS N, (f: [(s) -> below[N]]): injective?(f))
finite_set: TYPE = (is_finite) CONTAINING emptyset[T]
```

Algunas de las especificaciones realizadas sobre conjuntos finitos han sido definidas recursivamente. Para ello, hemos usado la función `choose` que escoge un elemento del conjunto y la función `rest`, que elimina de un conjunto el elemento elegido por `choose`.

```
rest(a): set = IF empty?(a) THEN a ELSE remove(choose(a), a) ENDIF
```

Como función de medida se ha usado la función `card` que especifica el cardinal de un conjunto finito. Veamos, como ejemplo, la definición de la disyunción de un conjunto finito de fórmulas proposicionales:

```
FS: VAR finite_set[formula_prop]

disyuncion(FS): RECURSIVE formula_prop =
  IF empty?(FS)
     THEN falso
     ELSE choose(FS) OR disyuncion(rest(FS))
  ENDIF
  MEASURE card(FS)
```

En la biblioteca de conjuntos finitos incluida en la distribución de PVS se completa la teoría iniciada en el preludio. En ella se incluyen distintos esquemas de inducción sobre conjuntos finitos. Mostramos aquí los esquemas que hemos usado en las pruebas desarrolladas en este trabajo:

```
S, S1: VAR finite_set
SS:    VAR non_empty_finite_set
e:     VAR T
P:     VAR pred[finite_set]
```

- `finite_set_induction`:

```
P(emptyset[T]) AND (FORALL e,S: P(S) IMPLIES P(add(e,S)))
IMPLIES (FORALL S: P(S))                                          1
```

- `finite_set_induction_rest`:

```
P(emptyset[T]) AND (FORALL SS : P(rest(SS)) IMPLIES P(SS))
IMPLIES (FORALL S : P(S))
```
2

- finite_set_induction_gen:

```
(FORALL S: (FORALL S1: card(S1) < card(S) IMPLIES P(S1))
           IMPLIES P(S))
IMPLIES (FORALL S: P(S))
```
3

Además, se incluyen también condiciones suficientes para probar que un conjunto es finito. Entre ellas, hemos usado las siguientes:

```
E: VAR finite_set[T1]
F: VAR finite_set[T2]
S: VAR set[T1]
U: VAR set[T2]
```

- injection_to_finite_set:

```
(EXISTS (f: [(S)->(F)]): injective?(f)) IMPLIES is_finite(S)
```
4

- surjection_from_finite_set:

```
(EXISTS (f: [(E)->(U)]): surjective?(f)) IMPLIES is_finite(U)
```
5

2.2.2. Otras teorías

En este trabajo hemos usado las teorías que constituyen la formalización en PVS de la teoría de dominios, realizada por F. Bartels, Dold, Pfeifer, Henke y Ruess en [8], para lo cual hemos realizado la adaptación de la misma a la versión 3.1 de PVS.

Hemos extendido la teoría de conjuntos incluida en el preludio de PVS, con definiciones y propiedades, que mostramos en el apéndice A de esta memoria. Asimismo, hemos extendido también la teoría de listas de PVS y la hemos incluido en el apéndice B.

Capítulo 3

Un modelo formal de la lógica proposicional en PVS

En este capítulo se presenta una formalización en PVS de la lógica proposicional. Está basada principalmente en los desarrollos de Lloyd y Apt [48, 5]. Otros desarrollos computacionales de la lógica proposicional pueden encontrarse en [16] que usa el sistema Nuprl y en [**?**] que usa ACL2.

La primera sección trata de la representación de las fórmulas proposicionales. Para ello, se define la noción de fórmula proposicional construida sobre un universo de símbolos, se formaliza el concepto de lenguaje proposicional y de fórmula relativa a un lenguaje.

A continuación, se describe la semántica proposicional a través de la noción de interpretación. En esta sección se formalizan los conceptos de modelo, satisfacibilidad, validez y consecuencia lógica y se establece la relación entre consecuencia lógica e insatisfacibilidad.

En la tercera sección se describe una representación de las cláusulas de Horn, independiente de las fórmulas proposicionales, su relación con éstas y se representan los programas lógicos como conjuntos finitos de cláusulas.

Por último, a partir de la semántica proposicional y de la relación entre cláusulas y fórmulas, se definen los conceptos de modelo de cláusulas, consecuencia lógica y satisfacibilidad; y se demuestra la reducción de la consecuencia lógica a la insatisfacibilidad. Finalmente, se introducen los modelos de Herbrand y se prueba que son suficientes para determinar la satisfacibilidad de conjuntos de cláusulas.

3.1. Sintaxis

En esta sección introducimos la sintaxis de las fórmulas proposicionales, así como la noción de lenguaje proposicional y de fórmula relativa a un lenguaje. En el lenguaje natural, una proposición no es más que una afirmación. La lógica proposicional, en su aspecto sintáctico, describe qué se entiende por proposición y cómo combinar proposiciones para formar otras proposiciones. El conjunto de las

fórmulas proposicionales se describe como el menor conjunto construido a partir de unas fórmulas básicas (átomos), combinándolas mediante conectivas lógicas. K. Doets [22], S. Burris [14] y M. Fitting [30] muestran distintas formas de describir el conjunto de fórmulas proposicionales.

La formalización en PVS que presentamos en esta sección se encuentra desarrollada en las teorías `formula_prop`, `atomos` y `conectivas` (páginas 289–294).

Las fórmulas proposicionales se construyen a partir de un conjunto de símbolos, Σ, utilizando las conectivas lógicas habituales: negación ($\neg$), conjunción ($\wedge$), disyunción ($\vee$), implicación ($\rightarrow$) y equivalencia ($\leftrightarrow$). Consideraremos también los símbolos $\top$ (verdad) y $\bot$ (falso).

Definición 3.1 *Una* **fórmula proposicional** *sobre Σ se define, inductivamente, como sigue:*

- *$\bot$ es una fórmula proposicional.*
- *Un símbolo proposicional es una fórmula proposicional, que llamaremos* **fórmula atómica** *o* **átomo**.
- *Si F es una fórmula proposicional, $\neg F$ también lo es.*
- *Si F_1 y F_2 son fórmulas proposicionales, $F_1 \wedge F_2$ también lo es.*

Notación 3.2 El conjunto de las fórmulas proposicionales sobre Σ lo notaremos por $\mathbb{P}(\Sigma)$.

Para describir en PVS el conjunto de las fórmulas proposicionales, consideramos un tipo no vacío `T`, que representa un determinado universo. Este tipo no constituye directamente el conjunto de símbolos proposicionales o átomos. Construimos el conjunto de las fórmulas proposicionales sobre `T` como un tipo de dato recursivo, usando una utilidad importante del lenguaje de PVS: la capacidad para especificar tipos abstractos de datos. Este tipo de dato está formado por una constante que representa a $\bot$, un constructor de átomos a partir de los elementos del universo `T`, y dos constructores de fórmulas proposicionales, que representan la negación y la conjunción de fórmulas, respectivamente. Así, la especificación en PVS del conjunto de fórmulas proposicionales es la siguiente:

6

```
formula_prop[T: TYPE+]: DATATYPE
  BEGIN
  falso                          : falsa?
  a(simb: T)                     : atomo?
  ~(fla: formula_prop)           : negacion?
  &(fla1, fla2: formula_prop)    : conjuncion?
  END formula_prop

  F1, F2: VAR formula_prop[T]
```

En ella, `falso`, `a`, `~` y `&` son *constructores*, `simb`, `fla`, `fla1`, `fla2` son *funciones de acceso* y `falsa?`, `atomo?`, `negacion?` y `conjuncion?` son *reconocedores* del tipo de dato `formula_prop`.

Como hemos comentado en 2.1.3, cuando en PVS se realiza la comprobación de tipos en un tipo de dato, se generan de manera automática, las teorías `formula_prop_adt`, `formula_prop_adt_map` y `formula_prop_adt_reduce` (en el fichero `formula_prop_adt.pvs`, páginas 289–**??**), en las que se describen los axiomas y las funciones, que determinan el uso del tipo de dato especificado.

Los símbolos proposicionales o átomos (Σ) son, pues, los construidos mediante `a` sobre los elementos del tipo `T`. Para representarlos, declaramos el tipo específico `atomo`:

7
```
atomo: TYPE = (atomo?)
```

A partir del conjunto de fórmulas proposicionales especificado por el tipo de dato `formula_prop`, definimos las fórmulas que se obtienen mediante las conectivas correspondientes a la disyunción, implicación y equivalencia. Incluimos también la fórmula obtenida como la negación de la fórmula $\bot$ (`falso`), que notamos por $\top$ (`verdad`).

Definición 3.3 *Sean $F_1, F_2 \in \mathbb{P}(\Sigma)$. Definimos:*

- *La fórmula $\top$ como $\neg \bot$.*
- *La fórmula $F_1 \vee F_2$ como $\neg(\neg F_1 \wedge \neg F_2)$*
- *La fórmula $F_1 \rightarrow F_2$ como $\neg F_1 \vee F_2$*
- *La fórmula $F_1 \leftrightarrow F_2$ como $(F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$*

8
```
verdad: formula_prop = ~falso
; ∨ (F1, F2): formula_prop  = ~(~F1 & ~F2)
; =>(F1, F2): formula_prop  = ~F1 ∨ F2
; <=>(F1, F2): formula_prop = (F1 => F2) & (F2 => F1)
```

Como hemos comentado en el capítulo 2, el lenguaje de especificación de PVS tiene la capacidad de expresar el concepto de conjunto de elementos de un tipo determinado. Así, si queremos refererirnos formalmente a un conjunto de fórmulas, no es necesario considerar una representación del mismo, sino que podemos directamente, manejar conjuntos (finitos o infinitos) de fórmulas proposicionales. Como operaciones generalizadas, definimos la conjunción y disyunción de un conjunto finito de fórmulas.

Definición 3.4 *Sea FS un conjunto finito de fórmulas proposicionales. La* **conjunción** *de FS se define como*

$$\bigwedge(FS) = \begin{cases} \top & \text{si } FS = \emptyset \\ F \wedge \bigwedge(FS - \{F\}) & \text{si } FS \neq \emptyset \text{ y } F \text{ es un elemento elegido de } FS \end{cases}$$

Análogamente, la **disyunción** *de FS se define como*

$$\bigvee(FS) = \begin{cases} \bot & \text{si } FS = \emptyset \\ F \vee \bigvee(FS - \{F\}) & \text{si } FS \neq \emptyset \text{ y } F \text{ es un elemento elegido de } FS \end{cases}$$

Para especificarlas en PVS, hemos de realizar una definición recursiva sobre un conjunto finito. Ahora bien, el tipo de los conjuntos finitos de PVS no está construido como un tipo de dato recursivo, sobre los que las definiciones se realizan, de manera natural, de forma recursiva. Aún así, podemos realizar definiciones recursivas sobre conjuntos finitos usando la función de elección, `choose`, la función `rest` y proporcionando como medida para garantizar la terminación, el cardinal del conjunto mediante la función `card`, como hemos comentado en 2.2.1

9

```
FS: VAR finite_set[formula_prop]

conjuncion(FS): RECURSIVE formula_prop =
  IF empty?(FS)
     THEN verdad
     ELSE choose(FS) & conjuncion(rest(FS))
  ENDIF
  MEASURE card(FS)

disyuncion(FS): RECURSIVE formula_prop =
  IF empty?(FS)
     THEN falso
     ELSE choose(FS) ∨ disyuncion(rest(FS))
  ENDIF
  MEASURE card(FS)
```

Terminamos esta sección definiendo el conjunto de símbolos proposicionales de una fórmula, sus propiedades, y la noción de *base de Herbrand* de un lenguaje proposicional.

Definición 3.5 *Sea F una fórmula proposicional sobre Σ. El* **conjunto de símbolos proposicionales** *de F, $SP(F)$, se define como sigue:*

$$SP(F) = \begin{cases} \emptyset & \text{si } F = \bot \\ \{F\} & \text{si } F \text{ es un átomo} \\ SP(G) & \text{si } F = \neg G \\ SP(F_1) \cup SP(F_2) & \text{si } F = F_1 \wedge F_2 \end{cases}$$

La especificación de la función `simb_prop` en PVS se realiza por recursión en la estructura de la fórmula. En la definición, la medida usada para garantizar la terminación es la inducida por la relación de subfórmula ($\ll$) de la teoría `formula_prop_adt`.

10
```
simb_prop(F): RECURSIVE finite_set[atomo] =
  CASES F OF
   falso:      ∅,
   a(x):       singleton[atomo[T]](a(x)),
   ~(F1):      simb_prop(F1),
   &(F1,F2):   simb_prop(F1) ∪ simb_prop(F2)
  ENDCASES
  MEASURE F BY <<
```

En los lemas siguientes, se prueba que el conjunto de símbolos proposicionales de una combinación de fórmulas es la unión de los conjuntos de símbolos proposicionales de las fórmulas que la forman.

Lema 3.6 *Sean F_1 y F_2 dos fórmulas proposicionales. Se verifica:*

- $SP(F_1 \vee F_2) = SP(F_1) \cup SP(F_2)$
- $SP(F_1 \to F_2) = SP(F_1) \cup SP(F_2)$

11
```
simb_prop_disyuncion_bin: LEMMA
   simb_prop(F1 ∨ F2) = simb_prop(F1) ∪ simb_prop(F2)

simb_prop_implicacion: LEMMA
   simb_prop(F1 => F2) = simb_prop(F1) ∪ simb_prop(F2)
```

Lema 3.7 *Sea $\mathcal{FS}$ un conjunto finito de fórmulas proposicionales. Entonces,*

- $SP(\bigwedge \mathcal{FS}) = \bigcup\{SP(F) : F \in \mathcal{FS}\}$
- $SP(\bigvee \mathcal{FS}) = \bigcup\{SP(F) : F \in \mathcal{FS}\}$

La formalización de este lema en PVS usa la función del preludio `image`[1]:

12
```
simb_prop_conjuncion_s_l1: LEMMA
  simb_prop(conjuncion(FS)) = Union(image(simb_prop)(FS))
```

[1]La función `image` proporciona la imagen de un conjunto por una función.

Demostración:

La prueba se realiza por inducción en $\mathcal{FS}$, según el esquema proporcionado por `finite_set_induction_rest` [2] de la biblioteca de conjuntos finitos, usando la definición de SP sobre la conjunción de dos fórmulas.

□

Finalizamos esta sección introduciendo las nociones de lenguaje proposicional y base de Herbrand, relativa a un lenguaje. En general, un lenguaje proposicional está constituido por un alfabeto (símbolos proposicionales, conectivas y signos de puntuación) y por las fórmulas proposicionales cuyos símbolos proposicionales pertenecen a dicho alfabeto. Puesto que las fórmulas se construyen mediante las conectivas lógicas a partir de los símbolos proposicionales, identificamos por simplicidad, el lenguaje con el conjunto de sus símbolos proposicionales. Es decir, identificamos un lenguaje proposicional con el tipo `atomo` declarado en [7]

Nos interesa considerar también lenguajes restringidos a un subconjunto de símbolos. Para ello, declaramos el tipo `lenguaje`, mediante el que representamos los lenguajes proposicionales restringidos, y definimos el concepto de fórmula proposicional relativa a un lenguaje como aquella cuyos símbolos pertenecen a dicho lenguaje.

13
```
lenguaje: TYPE = set[atomo]
```

En lo que sigue, LE representará un lenguaje.

14
```
LE: VAR lenguaje
```

Declaramos el tipo de las fórmulas proposicionales de un lenguaje LE, como un subtipo de las fórmulas proposicionales, de la forma siguiente:

15
```
formula_prop_l(LE): TYPE =
  {F: formula_prop | simb_prop(F) ⊆ LE} CONTAINING falso

JUDGEMENT formula_prop_l(LE) SUBTYPE_OF formula_prop
```

En lo que sigue, estableceremos las definciones y propiedades sobre fórmulas proposicionales en general. En particular, se tendrán para fórmulas proposicionales de un lenguaje LE.

Dado un lenguaje LE, la **base de Herbrand** asociada a LE es el conjunto de símbolos proposicionales de LE; debido a la identificación que hemos hecho de un lenguaje con sus símbolos proposicionales, la base de Herbrand asociada a LE es el propio lenguaje.

16
```
base_herbrand(LE): set[atomo] = LE
```

3.2. Semántica proposicional

En esta sección presentamos la formalización de la semántica clásica de la lógica proposicional, realizada en la teoría `semantica_proposicional` (páginas 297–303). Tratamos la representación de la noción de interpretación y formalizamos los conceptos de modelo, satisfacibilidad, validez y consecuencia lógica.

Una **interpretación** I de un lenguaje proposicional consiste en una aplicación del lenguaje en el conjunto con dos valores de verdad, verdadero y falso. En la formalización en PVS, los valores de verdad serán los propios del lenguaje de PVS, `TRUE` y `FALSE`.

En principio, trabajaremos con una noción de interpretación general, asociada al lenguaje que comprende todos los símbolos proposicionales, y no a un lenguaje restringido. Representaremos una interpretación I como un conjunto de átomos, considerando que si $A \in I$, se aplica en `TRUE`, y si $A \notin I$, se aplica en `FALSE`. Y consideraremos una **interpretación de un lenguaje** LE, o **interpretación de Herbrand del lenguaje**, como una interpretación I contenida en el conjunto de símbolos de LE.

En PVS declaramos los tipos correspondientes, incluyendo un tipo específico para interpretaciones finitas.

17
```
interpretacion: TYPE = set[atomo]
I, I1, I2: VAR interpretacion

interpretacion_finita: TYPE = finite_set[atomo]

es_interpretacion_herbrand(LE)(I): bool = subset?(I,LE)
interpretacion_l(LE): TYPE = (es_interpretacion_herbrand(LE))
```

Notación 3.8 Denotamos por $\mathcal{I}$ al conjunto de interpretaciones sobre Σ, y por $\mathcal{IF}$ al conjunto de interpretaciones finitas.

Obviamente, si de una interpretación nos quedamos con los átomos de un lenguaje, obtenemos una interpretación de Herbrand de dicho lenguaje.

Lema 3.9 *La intersección de una interpretación con un lenguaje es una interpretación de Herbrand de dicho lenguaje.*

18
```
interpretacion_restringida: LEMMA
  es_interpretacion_herbrand(LE)(intersection(I,LE))
```

En adelante, estableceremos las definiciones y propiedades sobre las interpretaciones en general; en particular, se tendrán para las interpretaciones de un lenguaje LE.

3.2.1. Modelos

Para entender el significado de un programa lógico o de una fórmula proposicional en general, se proporcionan las nociones semánticas correspondientes.

Nos proponemos, pues, especificar en PVS cuándo una fórmula es verdad en una interpretación. Las definiciones se van a hacer sobre interpretaciones y fórmulas proposicionales, en general, independientes de un lenguaje restringido.

Entenderemos que la fórmula $\bot$ es falsa en cualquier interpretación, y que un átomo es cierto en una interpretación si es un elemento de ella. A partir de esto, definiremos cuándo una fórmula será cierta en una interpretación, en función de las conectivas básicas ($\neg$ y $\wedge$) que intervienen en su estructura.

Definición 3.10 *Decimos que una interpretación I* **es modelo** *de una fórmula proposicional F, y lo representamos por $I \models F$, si se cumple alguna de las siguientes condiciones:*

- *F es atómica y $F \in I$*
- *$F = \neg G$ e I no es modelo de G*
- *$F = F_1 \wedge F_2$ e I es modelo de F_1 y de F_2.*

La especificación en PVS se realiza por inducción en la estructura de la fórmula:

19

```
es_modelo(I,F) : RECURSIVE bool =
  CASES F OF
   falso:      FALSE,
   a(x):       F ∈ I,
   ~(F1):      NOT es_modelo(I,F1),
   &(F1,F2):   es_modelo(I,F1) ∧ es_modelo(I,F2)
  ENDCASES
  MEASURE F BY <<
```

En particular, especificamos la noción de modelo de Herbrand de una fórmula relativa a un lenguaje.

Definición 3.11 *Sea LE un lenguaje y F una fórmula de LE. Decimos que una interpretación I es* **modelo de Herbrand** *de F si es interpretación de Herbrand de LE y modelo de F.*

20

```
es_modelo_herbrand(LE)(I:interpretacion,
                       F:formula_prop_l(LE)): bool =
  es_interpretacion_herbrand(LE)(I) ∧ es_modelo(I,F)
```

A partir de la definición de `es_modelo`, establecemos las reglas semánticas que caracterizan el concepto de modelo, atendiendo a la estructura de la fórmula.

Lema 3.12 *Sean I una interpretación y F_1, F_2 fórmulas proposicionales. Se verifica:*

- $I \models \top$
- $I \models (F_1 \vee F_2)$ *si y sólo si* $I \models F_1$ *ó* $I \models F_2$
- $I \models (F_1 \rightarrow F_2)$ *si y sólo si* $I \not\models F_1$ *ó* $I \models F_2$
- $I \models (F_1 \leftrightarrow F_2)$ *si y sólo si* $I \models F_1 \Leftrightarrow I \models F_2$

21

```
es_modelo_verdad: LEMMA
  es_modelo(I,verdad)

es_modelo_disyuncion: LEMMA
  es_modelo(I,F1 ∨ F2) ⇔ (es_modelo(I,F1) OR es_modelo(I,F2))

es_modelo_implicacion: LEMMA
  es_modelo(I,F1 => F2) ⇔ ((NOT es_modelo(I,F1)) OR es_modelo(I,F2))

es_modelo_equivalencia: LEMMA
  es_modelo(I,F1 <=> F2) ⇔ (es_modelo(I,F1) ⇔ es_modelo(I,F2))
```

Dada un interpretación I y una fórmula proposicional F, la valoración de los símbolos proposicionales de F a través de I será determinante para decidir si I es o no modelo de F. Por tanto, para comprobar si una interpretación I es modelo de una fórmula F, bastará considerar sólo los elementos de I que son símbolos proposicionales de F.

Lema 3.13 *Si I_1 e I_2 son modelos de los mismos símbolos proposicionales de F, entonces $I_1 \models F$ si y sólo si $I_2 \models F$.*

Como la lógica de PVS es tipada, tanto el dominio como el rango de cualquier función corresponde a un determinado tipo. Así, `es_modelo(I,A)` es de tipo booleano. Y el uso del predicado `=` sobre elementos booleanos tiene el mismo significado que la equivalencia lógica. Por ello, la especificación en PVS de este lema puede hacerse como sigue:

22

```
es_modelo_simb_prop_interpretacion: LEMMA
  (∀ A: member(A,simb_prop(F)) ⇒ (es_modelo(I1,A) = es_modelo(I2,A)))
  ⇒
  es_modelo(I1,F) = es_modelo(I2,F)
```

Demostración:
Por inducción en la estructura de la fórmula, aplicando las propiedades anteriores.

□

Como aplicación directa de esta propiedad probamos el resultado siguiente, que permite, para determinar si un interpretación es modelo de una fórmula F, reducirla a los símbolos proposicionales de dicha fórmula.

Lema 3.14 *Una interpretación I es modelo de F si y sólo si la intersección de I con los símbolos proposicionales de F es modelo de F.*

23
```
modelo_restringido: LEMMA
  es_modelo(I,F) = es_modelo(I ∩ simb_prop(F),F)
```

El concepto de modelo de una fórmula nos permite establecer la noción de fórmulas semánticamente equivalentes, como aquellas que tienen el "mismo significado", lo que precisamos en la definición siguiente.

Definición 3.15 *Si dos fórmulas proposicionales tienen exactamente los mismos modelos, diremos que dichas fórmulas son* **lógicamente equivalentes** *y lo representaremos por $F1 \equiv F2$.*

24
```
equivalentes(F1,F2): bool =
  ∀ I: es_modelo(I,F1) ⇔ es_modelo(I,F2)
```

Lema 3.16 *La relación $\equiv$ verifica las siguientes propiedades:*

1. *La fórmula $\top$ es un elemento neutro para la conjunción.*

   25
   ```
   verdad_neutro_conjuncion: CLAIM
     equivalentes(F & verdad, F)
   ```

2. *La fórmula $\bot$ es un elemento neutro para la disyunción.*

   26
   ```
   equivalentes_falso: LEMMA
     equivalentes(F, F ∨ ⊥)
   ```

3. *Estabilidad por negación:*

   27
   ```
   equivalentes_neg: LEMMA
     equivalentes(F1, F2) ⇒ equivalentes(~F1, ~F2)
   ```

4. *Estabilidad por disyunción:*

```
28
equivalentes_disy: LEMMA
  equivalentes(F1, F2) ⇒ equivalentes(F1 ∨ F, F2 ∨ F)
```

5. *Es una relación de equivalencia.*

```
29
equivalentes_equivalencia: LEMMA equivalence?(equivalentes)
```

Terminamos esta sección extendiendo la definición de modelo a un conjunto de fórmulas y probando alguna de sus propiedades más útiles.

Definición 3.17 *Decimos que una interpretación I es* **modelo de un conjunto** *de fórmulas proposicionales $\mathcal{S}$ si es modelo de todas las fórmulas de $\mathcal{S}$.*

```
30
S, S1, S2: VAR set[formula_prop]

es_modelo(I,S): bool = ∀ F: F ∈  S ⇒ es_modelo(I,F)
```

Lema 3.18 *Sean I una interpretación y $\mathcal{S}_1$, $\mathcal{S}_2$ dos conjuntos de fórmulas proposicionales. Si $I \models \mathcal{S}_1$ y $\mathcal{S}_2 \subset \mathcal{S}_1$, entonces $I \models \mathcal{S}_2$.*

```
31
modelo_de_subconjuntos: LEMMA
  es_modelo(I,S1) & S2 ⊆ S1 ⇒ es_modelo(I,S2)
```

Lema 3.19 *Sean I una interpretación y $\mathcal{S}$ un conjunto de fórmulas proposicionales. Entonces:*

- *$I \models \mathcal{S} \cup \{F\}$ si y sólo si $I \models \mathcal{S}$ e $I \models F$*
- *$I \models S \cup \{\neg F\}$ si y sólo si $I \models S$ e $I \not\models F$*

```
32
modelo_add: LEMMA
  es_modelo(I,add(F,S)) ⇔ (es_modelo(I,S) &  es_modelo(I,F))

modelo_add_neg: LEMMA
  es_modelo(I,add(~F,S)) ⇔ (es_modelo(I,S) & NOT es_modelo(I,F))
```

3.2.2. Satisfacibilidad y validez

A la hora de formalizar los conceptos de satisfacibilidad y validez de fórmulas, lo haremos sobre fórmulas proposicionales en general, independientes de un lenguaje, y considerando también como posibles modelos la noción más general de interpretación. Después, especificaremos los conceptos relativos a un lenguaje y estableceremos la relación entre ambos.

Definición 3.20 *Sea F una fórmula proposicional. Decimos que:*

- *F es* **satisfacible** *si alguna interpretación es modelo de F.*
- *F es* **válida** *(o* **tautología***) si toda interpretación es modelo de F.*

33
```
es_satisfacible(F): bool = ∃ I: es_modelo(I,F)
es_valida(F): bool = ∀ I: es_modelo(I,F)
```

Definición 3.21 *Una fórmula F de un lenguaje LE es* **satisfacible en el lenguaje** *si alguna interpretación del lenguaje es modelo de F.*

34
```
es_satisfacible(LE)(F:formula_prop_l(LE)): bool =
  ∃ (I: interpretacion_l(LE)): es_modelo(I, F)
```

Lema 3.22 *Una fórmula F de un lenguaje LE es satisfacible en el lenguaje si y sólo si es satisfacible.*

35
```
es_satisfacible_le: LEMMA
  ∀(F:formula_prop_l(LE)): es_satisfacible(LE)(F)⇔ es_satisfacible(F)
```

Demostración:

Sea F una fórmula del lenguaje LE.

($\Rightarrow$) Por la definición de satisfacibilidad, teniendo en cuenta que toda interpretación del lenguaje LE es una interpretación.

($\Leftarrow$) Si F es satisfacible, existe una interpretación I que es modelo de F. Consideremos $I_1 = I \cap SP(F)$. Entonces, por *17* I_1 es una interpretación de Herbrand de LE, y por *23* se tiene que $I_1 \models F$. Luego, F es satisfacible en LE.

□

Definición 3.23 *Una fórmula F de un lenguaje LE es* **válida en el lenguaje** *si toda interpretación del lenguaje es modelo de F.*

```
es_valida(LE)(F:formula_prop_l(LE)): bool =
   ∀ (I:interpretacion_l(LE)): es_modelo(I,F)
```
36

Lema 3.24 *Una fórmula F de un lenguaje LE es válida en el lenguaje si y sólo si es válida.*

```
es_valida_le: LEMMA
  ∀ (F:formula_prop_l(LE)): es_valida(LE)(F) ⇔ es_valida(F)
```
37

Demostración:

Sea F una fórmula del lenguaje LE.

($\Rightarrow$) Si F es válida en LE, toda interpretación de Herbrand de LE es modelo de F. Sea I una interpretación y consideremos $I_1 = I \cap SP(F)$, que es una interpretación de Herbrand de LE. Por ser F válida en LE, $I_1 \models F$. Y, por 23, $I \models F$. Por tanto, F es válida.

($\Leftarrow$) Por la definición de validez, considerando que toda interpretación del lenguaje LE es una interpretación.

□

Extendemos las definiciones de satisfacibilidad y validez a conjuntos de fórmulas.

Definición 3.25 *Sea $\mathcal{S}$ un conjunto de fórmulas proposicionales. Decimos que:*

- *$\mathcal{S}$ es* **satisfacible** *si existe una interpretación I que es modelo de $\mathcal{S}$.*
- *$\mathcal{S}$ es* **insatisfacible** *si para toda interpretación I, I no es modelo de $\mathcal{S}$.*
- *$\mathcal{S}$ es* **válido** *si toda interpretación I es modelo de $\mathcal{S}$.*
- *$\mathcal{S}$ es* **no válido** *si existe una interpretación I que no es modelo de $\mathcal{S}$.*

```
es_satisfacible(S): bool = ∃ I: es_modelo(I, S)

es_insatisfacible(S): bool = ∀ I: NOT es_modelo(I, S)

es_valido(S): bool = ∀ I: es_modelo(I, S)

es_no_valido(S): bool = ∃ I: NOT es_modelo(I, S)
```
38

Lema 3.26 *Si $F_1 \equiv F_2$, entonces $\mathcal{S} \cup \{F_1\}$ es insatisfacible si y sólo si $\mathcal{S} \cup \{F_2\}$ es insatisfacible.*

```
39
equivalentes_insatisfacible: LEMMA
   equivalentes(F1, F2) ⇒
   ∀ S: es_insatisfacible(add(F1,S)) ⇔ es_insatisfacible(add(F2, S))
```

La prueba es inmediata usando 32

3.2.3. Consecuencia lógica

Seguiremos el mismo esquema que en el apartado anterior. Es decir, en primer lugar, formalizaremos el concepto de consecuencia lógica independiente de un lenguaje. En segundo lugar, especificaremos los conceptos relativos a un lenguaje y estableceremos la relación entre ambos.

Definición 3.27 *Una fórmula proposicional F es* **consecuencia lógica** *de un conjunto de fórmulas $\mathcal{S}$, y se representa por $\mathcal{S} \models F$, si todo modelo de $\mathcal{S}$ también es modelo de F.*

```
40
es_cons_logica(F,S): bool =
 ∀ I: es_modelo(I,S) ⇒ es_modelo(I,F)
```

Definición 3.28 *Una fórmula F de un lenguaje LE es* **consecuencia lógica en** *LE de un conjunto de fórmulas de LE, si toda interpretación del lenguaje LE que es modelo de $\mathcal{S}$, es modelo de F.*

```
41
es_cons_logica(LE)(F:formula_prop_l(LE),
                   S:set[formula_prop_l(LE)]): bool =
  ∀ (I:interpretacion_l(LE)): es_modelo(I,S) ⇒ es_modelo(I,F)
```

Establecemos la caracterización del concepto de consecuencia lógica relativa a un lenguaje, lema 3.31, usando los lemas que enunciamos a continuación:

Lema 3.29 *Sea F una fórmula de un lenguaje LE e I una interpretación. Entonces, $I \models F$ si y sólo si $I \cap LE \models F$.*

```
42
interpretacion_restringida_es_modelo_formula: LEMMA
  ∀ (F:formula_prop_l(LE)): es_modelo(I,F) ⇔ es_modelo(I∩LE,F)
```

Lema 3.30 *Sea $\mathcal{S}$ un conjunto de fórmulas de un lenguaje LE e I una interpretación. Entonces, $I \models \mathcal{S}$ si y sólo si $I \cap LE \models \mathcal{S}$.*

```
43
interpretacion_restringida_es_modelo_conjunto: LEMMA
  ∀ (S:set[formula_prop_l(LE)]): es_modelo(I,S) ⇔ es_modelo(I∩LE,S)
```

Lema 3.31 *Sean F una fórmula de un lenguaje LE y $\mathcal{S}$ un conjunto de fórmulas de LE. Entonces, F es consecuencia lógica de $\mathcal{S}$ en LE si y sólo si F es consecuencia lógica de $\mathcal{S}$.*

44
```
es_cons_logica_le: LEMMA
  ∀ (F:formula_prop_l(LE), S:set[formula_prop_l(LE)]):
  es_cons_logica(LE)(F,S) ⇔ es_cons_logica(F,S)
```

Demostración:
Sea F una fórmula de LE y $\mathcal{S}$ un conjunto de fórmulas del lenguaje LE.

($\Rightarrow$) En efecto, sea I una interpretación que es modelo de $\mathcal{S}$. Consideremos $I_1 = I \cap LE$ que, por 18, es interpretación del lenguaje LE. Y, por 43, $I_1 \models \mathcal{S}$. Luego, por hipótesis $I_1 \models F$ y, por 42, $I \models F$. Por tanto, $\mathcal{S} \models F$.

($\Leftarrow$) Por la definición 40, teniendo en cuenta que toda interpretación del lenguaje LE es una interpretación.

□

Establecemos ahora las propiedades de la relación de consecuencia lógica, tanto para fórmulas lógicamente equivalentes, como atendiendo a la estructura de la fórmula. En particular, establecemos una condición necesaria y suficiente para que una conjunción de fórmulas sea consecuencia lógica de un conjunto de fórmulas proposicionales.

Lema 3.32 *Sean $\mathcal{S}$ un conjunto de fórmulas proposicionales y F_1 y F_2 dos fórmulas proposicionales lógicamente equivalentes. Entonces, $\mathcal{S} \models F_1$ si y sólo si $\mathcal{S} \models F_2$.*

45
```
equivalentes_cons_log: LEMMA
  equivalentes(F1, F2) ⇒
  ∀ S: es_cons_logica(F1,S) ⇔ es_cons_logica(F2, S)
```

Lema 3.33 *Sean $\mathcal{S}$ un conjunto de fórmulas proposicionales, F_1 y F_2 dos fórmulas proposicionales. Entonces, $\mathcal{S} \models F_1 \wedge F_2$ si y sólo si $\mathcal{S} \models F_1$ y $\mathcal{S} \models F_2$*

46
```
cons_log_conj_binaria: LEMMA
  es_cons_logica(F1&F2,S) ⇔
  (es_cons_logica(F1,S) & es_cons_logica(F2,S))
```

Lema 3.34 *Sea $\mathcal{FS}$ un conjunto finito de fórmulas proposicionales y $\mathcal{S}$ un conjunto de fórmulas proposicionales. Entonces:*

$\mathcal{S} \models \bigwedge \mathcal{FS}$ *si y sólo si para toda* $F \in \mathcal{FS}$, $\mathcal{S} \models F$

47
```
cons_log_conjuncion_s: LEMMA
 es_cons_logica(conjuncion(FS),S) ⇔ (∀F:F∈FS ⇒ es_cons_logica(F,S))
```

Demostración:
Se prueba por inducción en $\mathcal{FS}$, según el esquema `finite_set_induction_rest` *2*, usando la propiedad para la conjunción de dos fórmulas *46*.

□

Terminamos la sección estableciendo el resultado que permite reducir la consecuencia lógica a la insatisfacibilidad.

Teorema 3.35 *Sea* $\mathcal{S}$ *un conjunto de fórmulas y* F *una fórmula proposicional. Entonces,* F *es consecuencia lógica de* $\mathcal{S}$ *si y sólo si* $\mathcal{S} \cup \{\neg F\}$ *es insatisfacible.*

48
```
cons_logica_insatisfacible: LEMMA
  es_cons_logica(F,S) ⇔ es_insatisfacible(add(~F,S))
```

Demostración:
La prueba se realiza a partir de las definiciones, usando el lema *32*.

□

3.3. Cláusulas de Horn

En esta sección, introducimos una clase especial de fórmulas proposicionales: las cláusulas de Horn. El desarrollo formal en PVS se encuentra en la teoría `clausulas` (páginas 294–297). En general, una **cláusula proposicional** es una fórmula proposicional de la forma $L_1 \vee \cdots \vee L_n$, donde cada L_i es un literal, es decir, un átomo o la negación de un átomo. Si agrupamos los átomos positivos y los negativos, podemos escribir la cláusula en la forma $A_1 \vee \cdots \vee A_k \vee \neg B_1 \vee \cdots \vee \neg B_m$. O, de forma lógicamente equivalente, $B_1 \wedge \cdots \wedge B_m \rightarrow A_1 \vee \cdots \vee A_k$. En programación lógica, se emplea la notación siguiente: $A_1, \ldots, A_k \leftarrow B_1, \ldots, B_m$, donde $A_1, \ldots, A_k$ se denominan *conclusiones* y $B_1, \ldots, B_m$, *premisas*. Una **cláusula de Horn** es una cláusula en la que $k \leq 1$. En este caso, la conclusión recibe el nombre de *cabeza* y las premisas, *cuerpo*.

En PVS, construiremos el tipo que representa a las cláusulas de Horn de forma independiente de las fórmulas proposicionales. A continuación, le asignaremos a cada cláusula de Horn una fórmula proposicional semánticamente equivalente.

Para ello, declaramos en primer lugar el tipo formado por los átomos junto con la fórmula $\bot$. Mediante este tipo, denominado `cabeza_cl`, representaremos el tipo de los elementos que pueden ser cabezas de las cláusulas de Horn.

Si vemos una cláusula como una fórmula proposicional, su cuerpo es una disyunción de átomos, en la que puede haber átomos repetidos. Ahora bien, desde el punto de vista semántico, el que algún átomo esté repetido o el "orden" en el que escribamos dichos átomos, no proporciona cláusulas esencialmente diferentes. Por ello, a la hora de decidir cómo representar el cuerpo de una cláusula de Horn, hemos decidido hacerlo mediante un conjunto finito de átomos. De esta forma, el razonamiento formal se hará sobre conjuntos.

Así pues, representamos una cláusula de Horn como una estructura con dos campos, `cabeza` y `cuerpo`, como sigue:

49
```
cabeza_cl: TYPE = {F: formula_prop |falsa?(F) OR atomo?(F)}

clausula_horn: NONEMTY_TYPE = [# cabeza: cabeza_cl,
                                 cuerpo: finite_set[atomo] #]

CH: VAR clausula_horn
```

Notación 3.36 El conjunto de las cláusulas de Horn, construidas a partir del conjunto de átomos o símbolos proposicionales Σ, lo denotaremos por $\mathbb{C}(\Sigma)$.

Definición 3.37 *Sea CH una cláusula de Horn:*

- *Si CH es de la forma $A \leftarrow B_1, \ldots, B_m$, se denomina* **clásusula definida** *o* **cláusula de programa definido**. *Si el cuerpo es vacío, decimos que es una* **cláusula unitaria** *o un* **hecho**. *Si no lo es, decimos que es una* **regla**.

- *Si CH es de la forma $\leftarrow B_1, \ldots, B_m$, se denomina* **objetivo definido**.

- *Un objetivo definido cuyo cuerpo es vacío, se denomina* **cláusula vacía**, *y se representa por $\square$.*

La especificación en PVS de estas definiciones es la siguiente:

49
```
es_clausula_def(CH): bool = atomo?(cabeza(CH))
es_objetivo_def(CH): bool = falsa?(cabeza(CH))

clausula_def: TYPE = (es_clausula_def)
objetivo_def: TYPE = (es_objetivo_def)

JUDGEMENT clausula_def SUBTYPE_OF clausula_horn
JUDGEMENT objetivo_def SUBTYPE_OF clausula_horn
```

```
C, C1, C2: VAR clausula_def
G, G1, G2: VAR objetivo_def

regla?(C): bool = nonempty?(cuerpo(C))
hecho?(C): bool = empty?(cuerpo(C))
vacia?(G): bool = empty?(cuerpo(G))
```

Por un **programa definido** se entiende un conjunto finito de cláusulas definidas. A la hora de elegir cómo representar formalmente un programa, tendremos en cuenta que, esencialmente, el programa es el mismo aunque cambie el orden en el que se muestren las cláusulas que lo forman, o tenga cláusulas repetidas. Por esta razón, elegimos representar un programa definido como un conjunto finito de fórmulas.

Ahora bien, el hecho de que el programa sea esencialmente el mismo, no influirá en la *interpretación declarativa* de los programas lógicos, puesto que la cuestión será *qué* es lo que puede ser deducido del programa. Pero sí puede tener repercusión en la *interpretación procedimental* de los mismos, pues se tratará, entonces, de *cómo* se lleva a cabo la deducción.

La especificación en PVS es, pues, la siguiente:

50
```
programa: TYPE = finite_set[clausula_def]
P, P1, P2: VAR programa
```

Como hemos comentado, vamos a definir una aplicación que asocie a cada cláusula de Horn una fórmula proposicional y, a partir de ella, estableceremos la semántica relativa a las cláusulas.

Definición 3.38 *Sea Σ un conjunto de átomos, $\mathbb{P}(\Sigma)$ el conjunto de fórmulas proposicionales sobre Σ y $\mathbb{C}(\Sigma)$ el conjunto de cláusulas de Horn sobre Σ. Definimos la aplicación*

$$fla : \mathbb{C}(\Sigma) \longrightarrow \mathbb{P}(\Sigma)$$

como $fla(A \leftarrow B_1, \ldots, B_m) = \bigwedge\{B_1, \ldots, B_n\} \rightarrow A$

51
```
fla(CH): formula_prop[T] =
  conjuncion(cuerpo(CH)) => cabeza(CH)
```

Definimos también la imagen de un conjunto de cláusulas de Horn, mediante la aplicación anterior:

52
```
SC: VAR set[clausula_horn]
conj_fla(SC): set[formula_prop[T]] = image(fla)(SC)
```

Como consecuencia de las propiedades de la función `image`, se prueba fácilmente que los conjuntos finitos de cláusulas de Horn se transforman en conjuntos finitos de fórmulas.

53
```
FSC:VAR finite_set[clausula_horn]
JUDGEMENT conj_fla(FSC) HAS_TYPE finite_set[formula_prop[T]]
```

3.4. Semántica de cláusulas

En esta sección, describimos la semántica de las cláusulas de Horn, a partir de la semántica proposicional y de la relación establecida entre cláusulas y fórmulas. Se introducirán también los modelos de Herbrand y se usarán para determinar la satisfacibilidad de un conjunto de cláusulas. Su formalización en PVS se encuentra en la teoría `semantica_clausulas` (páginas 303–309).

3.4.1. Modelos

Definición 3.39 *Decimos que una interpretación I* **es modelo de una cláusula de Horn** *CH, y lo notamos por $I \models CH$, si I es modelo de la fórmula proposicional asociada.*

54
```
es_modelo(I, CH): bool = es_modelo(I, fla(CH))
```

A partir de la definición, caracterizamos las interpretaciones que son modelos de cláusuas definidas y de objetivos.

Lema 3.40 *Una interpretación I es modelo de la cláusula definida C si y sólo si la cabeza de C pertenece a I o algún átomo del cuerpo de C no pertenece a I.*

55
```
es_modelo_clausula_def_caract: LEMMA
  es_modelo(I,C) ⇔ cabeza(C) ∈ I ∨ ∃ A: (A ∈ cuerpo(C) &  A ∉ I)
```

Lema 3.41 *Una interpretación I es* **modelo** *del objetivo definido G si y sólo si algún elemento del cuerpo de G no pertenece a I.*

56
```
es_modelo_objetivo_def_caract: LEMMA
  es_modelo(I,G) ⇔ ∃ A: A ∈ cuerpo(G) & A∉ I
```

La prueba se hace considerando la fórmula correspondiente y las propiedades de `es_modelo` en función de las conectivas.

Una vez que la definición de modelo de una cláusula de Horn la hemos reducido a la de modelo de la fórmula asociada, las demás definiciones semánticas

las haremos directamente sobre cláusulas, usando la noción de modelo previa. Además, para cada definición, estableceremos la equivalencia con la noción correspondiente de fórmulas proposicionales.

En primer lugar, extendemos la definición de modelo a conjuntos de cláusulas.

Definición 3.42 *Decimos que una interpretación I* **es modelo de un conjunto de cláusulas de Horn** *$\mathcal{SC}$ si I es modelo de todas las cláusulas de $\mathcal{SC}$. Análogamente, para un programa definido P.*

En PVS, especificaremos las nociones de modelo y consecuencia lógica para un conjunto cualquiera de cláusulas de Horn. En particular, estas definiciones se tendrán para programas, que son conjuntos finitos de cláusulas definidas. Ahora bien, a la hora de usarlos sobre programas, el mecanismo de tipos de PVS extiende el tipo de los elementos del programa a cláusulas de Horn. Esto puede hacer que las demostraciones sean tediosas y dificultar su legibilidad. Para evitarlo, las definiciones de modelo y consecuencia lógica las hacemos tanto para conjuntos de cláusulas de Horn como para programas, haciendo uso de la posibilidad de sobrecargar un símbolo.

57
```
es_modelo(I, SC): bool = ∀ CH: CH ∈ SC ⇒ es_modelo(I, CH)

es_modelo(I, P): bool = ∀ C: C ∈ P ⇒ es_modelo(I, C)
```

Probamos que esta definición es equivalente a que I sea modelo del conjunto de fórmulas proposicionales asociado.

Lema 3.43 *Sea I una interpretación y $\mathcal{SC}$ un conjunto de cláusulas. Entonces, I es modelo de $\mathcal{SC}$ si y sólo si I es modelo del conjunto de fórmulas asociado.*

58
```
es_modelo_cl_fla: LEMMA
  es_modelo(I,SC) ⇔ es_modelo(I,conj_fla(SC))
```

3.4.2. Consecuencia lógica y satisfacibilidad

Definimos la noción de consecuencia lógica de un conjunto de cláusulas, tanto para átomos como para cláusulas de Horn. Y, en cada caso, establecemos la correspondiente equivalencia con fórmulas proposicionales.

Definición 3.44 *Decimos que un átomo A es* **consecuencia lógica** *de un conjunto de cláusulas $\mathcal{SC}$, y lo notamos por $\mathcal{SC} \models A$, si todo modelo de $\mathcal{SC}$ es modelo de A. Análogamente, para programas definidos.*

```
es_cons_logica(A,SC): bool = ∀ I: es_modelo(I,SC) ⇒ es_modelo(I,A)

es_cons_logica(A,P): bool = ∀ I: es_modelo(I,P) ⇒ es_modelo(I,A)
```
59

Lema 3.45 *Un átomo A es consecuencia lógica de un conjunto de cláusulas $\mathcal{SC}$ si y sólo si A es consecuencia lógica del conjunto de fórmulas asociado a $\mathcal{SC}$. Análogamente para programas definidos.*

```
es_cons_logica_atomo_cl_fla: LEMMA
  es_cons_logica(A,SC) ⇔ es_cons_logica(A,conj_fla(SC))

es_cons_logica_atomo_cl_fla_programa: LEMMA
  es_cons_logica(A,P) ⇔ es_cons_logica(A,conj_fla(P))
```
60

Definición 3.46 *Decimos que una cláusula de Horn CH es* **consecuencia lógica** *de un conjunto de cláusulas $\mathcal{SC}$, y lo notamos por $\mathcal{S} \models CH$, si todo modelo de $\mathcal{SC}$ es modelo de CH.*

```
es_cons_logica(CH,SC): bool =
  ∀ I: es_modelo(I,SC) ⇒ es_modelo(I,CH)
```
61

Se tiene la correspondiente equivalencia con fórmulas proposicionales:

```
es_cons_logica_cl_fla: LEMMA
  es_cons_logica(CH,SC) ⇔ es_cons_logica(fla(CH),conj_fla(SC))
```
62

Como el hecho $A \leftarrow$ y el átomo A son lógicamente equivalentes, se tiene el siguiente resultado:

Lema 3.47 *Sean P un programa definido y A un átomo. Entonces, $P \models A$ si y sólo si $P \models (A \leftarrow)$.*

```
cons_log_l_hecho: LEMMA
  es_cons_logica(A,P) ⇔ es_cons_logica(hecho(A), P)
```
63

donde la función `hecho`[2] construye el hecho cuya cabeza es A.

Definición 3.48 *Sea $\mathcal{SC}$ un conjunto de cláusulas de Horn. Decimos que:*

- *$\mathcal{SC}$ es* **satisfacible** *si alguna interpretación es modelo de $\mathcal{SC}$.*

[2]La función `hecho` está definida como `hecho(A) = clausula_horn(A, ∅)`, donde `clausula_horn(F, FSA) = (# cabeza:= F, cuerpo:= FSA #)`.

- $\mathcal{SC}$ *es* **insatisfacible** *si ninguna interpretación es modelo de* $\mathcal{SC}$.

64
```
es_satisfacible(SC): bool = ∃ I: es_modelo(I,SC)
es_insatisfacible(SC): bool = ∀ I: NOT es_modelo(I,SC)
```

La correspondiente equivalencia con fórmulas proposicionales se establece mediante el siguiente lema.

Lema 3.49 *El conjunto de cláusulas de Horn* $\mathcal{SC}$ *es satisfacible si y sólo si el conjunto de fórmula proposicionales asociado es satisfacible.*

65
```
es_satisfacible_cl_fla: LEMMA
  es_satisfacible(SC) ⇔ es_satisfacible(conj_fla(SC))

es_insatisfacible_cl_fla: LEMMA
  es_insatisfacible(SC) ⇔ es_insatisfacible(conj_fla(SC))
```

Por último, establecemos la relación entre la consecuencia lógica para átomos y la insatisfacibilidad.

Lema 3.50 *Un átomo* A *es consecuencia lógica de un conjunto de cláusulas de Horn* $\mathcal{SC}$ *si y sólo si* $\mathcal{SC} \cup \{\leftarrow A\}$ *es insatisfacible.*

66
```
cons_log_insatisfacilidad: THEOREM
  es_cons_logica(A,SC) ⇔
  es_insatisfacible(add(objetivo(singleton(A)),SC))
```

donde `objetivo(FA)`[3] construye el objetivo definido cuyo cuerpo es el conjunto finito de átomos FA.

Terminamos la sección con el siguiente resultado, que usaremos en el desarrollo de la semántica procedimental de los programas lógicos.

Teorema 3.51 *Sea* $\mathcal{SC}$ *un conjunto de cláusulas y* G *un objetivo. Entonces:* $SC \cup \{G\}$ *es insatisfacible si y sólo si todos los átomos del cuerpo de* G *son consecuencia lógica de* $\mathcal{SC}$.

67
```
cons_log_objetivo: THEOREM
 ∀ (G:objetivo_def):
    es_insatisfacible(add(G,SC)) ⇔
    (∀ A: A ∈ cuerpo(G) ⇒ es_cons_logica(A,SC))
```

[3]La función `objetivo` está definida como `objetivo(FA) = clausula_horn(falso, FA)`

Demostración:
Sean G un objetivo y $\mathcal{SC}$ un conjunto de cláusulas. Entonces

$\mathcal{SC} \cup \{G\}$ es insatisfacible
$\Leftrightarrow$ $conj\_fla(\mathcal{SC}) \cup \{fla(G)\}$ es insatisfacible [64] y [65]
$\Leftrightarrow$ $conj\_fla(\mathcal{SC}) \cup \{\neg \bigwedge\{A : A \in cuerpo(G)\}\}$ es insatisfacible [39]
$\Leftrightarrow$ $conj\_fla(\mathcal{SC}) \models \bigwedge\{A : A \in cuerpo(G)\}$ [48]
$\Leftrightarrow$ $conj\_fla(\mathcal{SC}) \models A, \forall A \in cuerpo(G)$ [47]
$\Leftrightarrow$ $\mathcal{SC} \models A, \forall A \in cuerpo(G)$ [60]

□

3.4.3. Modelos de Herbrand

En esta sección construimos la base de Herbrand de un conjunto de cláusulas $\mathcal{SC}$, definimos qué se entiende por interpretaciones y modelos de Herbrand y probamos que un conjunto de cláusulas $\mathcal{SC}$ es insatisfacible si y sólo si no tiene modelos de Herbrand.

Para ello, en primer lugar, definimos una función que obtiene los símbolos proposicionales de una cláusula como el conjunto de símbolos de la fórmula asociada.

Definición 3.52 *El* **conjunto de símbolos proposicionales de una cláusula** *CH es el conjunto de símbolos de la fórmula asociada a CH.*

[68]
```
simb_prop(CH): finite_set[atomo] = simb_prop(fla(CH))
```

En particular, probamos que la cabeza de una cláusula es un símbolo proposicional de ella y caracterizamos el conjunto de símbolos de una cláusula en función de su cabeza y su cuerpo.

Lema 3.53 *La cabeza de una cláusula definida C pertenece a los símbolos proposicionales de C.*

[69]
```
simb_prop_contiene_cabeza: LEMMA
  cabeza(C) ∈ simb_prop(C)
```

Lema 3.54 *El conjunto de los símbolos proposicionales de CH es la unión de los símbolos proposicionales de la cabeza de CH con el cuerpo de CH.*

[70]
```
CNS_simb_prop: LEMMA
  simb_prop(CH) = simb_prop(cabeza(CH)) ∪ cuerpo(CH)
```

Dado un lenguaje proposicional LE, declaramos el tipo que representa al conjunto de las cláusulas de Horn relativas a LE:

```
clausula_horn_l(LE): TYPE =
 {CH: clausula_horn | simb_prop(CH) ⊆ LE }
```
71

Dado un conjunto de cláusulas $\mathcal{SC}$, consideramos el lenguaje asociado a $\mathcal{SC}$, $LE(\mathcal{SC})$, como el conjunto de átomos de $\mathcal{SC}$ y definimos la **base de Herbrand** asociada a $\mathcal{SC}$ como la base de Herbrand del lenguaje $LE(\mathcal{SC})$.

```
LE(SC): set[atomo] = Union(image(simb_prop)(SC))
base_herbrand(SC): set[atomo] = base_herbrand(LE(SC))
```
72

Lema 3.55 *La base de Herbrand asociada a un conjunto finito de cláusulas es finita.*

```
JUDGEMENT base_herbrand(FSC:finite_set[clausula_horn[T]]) HAS_TYPE
          finite_set[atomo]
```
73

Definición 3.56 *Decimos que I es una* **interpretación de Herbrand de un conjunto de cláusulas** $\mathcal{SC}$ *si es una interpretación del lenguaje asociado a $\mathcal{SC}$. Y decimos que I es un* **modelo de Herbrand** *de $\mathcal{SC}$ si es una interpretación de Herbrand y es modelo de $\mathcal{SC}$.*

```
es_interpretacion_herbrand(I,SC): bool =
  es_interpretacion_herbrand(LE(SC))(I)

es_modelo_herbrand(I,SC): bool =
  es_modelo(I,SC) & es_interpretacion_herbrand(I,SC)
```
74

Según la definición 64, determinar la insatisfacibilidad de un conjunto de cláusulas requiere comprobar que ninguna interpretación es modelo de dicho conjunto. Ahora bien, el teorema 3.62 permitirá reducir dicha comprobación sólo a los modelos de Herbrand correspondientes al conjunto de cláusulas. Los siguientes lemas son auxiliares para dicho teorema.

Lema 3.57 *Sean I_1, I_2 interpretaciones tales que son modelos de los mismos símbolos de la cláusula CH. Entonces, $I_1 \models CH$ si y sólo si $I_2 \models CH$.*

```
es_modelo_simb_prop_modelos_cl: LEMMA
  (∀ A: A ∈ simb_prop(CH) ⇒ (es_modelo(I1,A) = es_modelo(I2,A)))
  ⇒ (es_modelo(I1,CH) ⇔ es_modelo(I2,CH))
```
75

Lema 3.58 *Sea I un modelo de la cláusula de Horn CH. Entonces, $I \cap SP(CH)$ también lo es.*

```
76
modelo_c_simb: LEMMA
  es_modelo(I,CH) ⇒ es_modelo(I ∩ simb_prop(CH),CH)
```

Lema 3.59 *Sean I una interpretación, $\mathcal{SC}$ un conjunto de cláusula de Horn y CH una cláusula de $\mathcal{SC}$. Si $I \models CH$, entonces $I \cap BH(\mathcal{SC}) \models CH$.*

```
77
modelo_c_base_h: LEMMA
  es_modelo(I,CH) & CH ∈ SC ⇒ es_modelo(I ∩ base_herbrand(SC), CH)
```

Teorema 3.60 *Sea I un modelo del conjunto de cláusula de Horn $\mathcal{SC}$. Entonces, $I \cap BH(\mathcal{SC})$ es modelo de Herbrand de $\mathcal{SC}$.*

```
78
modelo_imp_modelo_herbrand: THEOREM
  es_modelo(I,SC) ⇒ es_modelo_herbrand(I ∩ base_herbrand(SC), SC)
```

Demostración:

Sea I un modelo del conjunto de cláusulas $\mathcal{SC}$. Entonces, $I \cap BH(\mathcal{SC})$ es una interpretación de Herbrand de $\mathcal{SC}$, y por 77, se tiene que $I \cap BH(\mathcal{SC}) \models \mathcal{SC}$. Luego, $I \cap BH(\mathcal{SC})$ es modelo de Herbrand de $\mathcal{SC}$

□

Como consecuencia, se tiene el mismo resultado para programas.

Corolario 3.61 *Sea I un modelo del programa definido P. Entonces, $I \cap BH(P)$ es modelo de Herbrand de P.*

```
79
modelo_imp_modelo_herbrand_p: COROLLARY
  es_modelo(I,P) ⇒ es_modelo_herbrand(I ∩ base_herbrand(P), P)
```

Terminamos esta capítulo con el teorema que reduce la insatisfacibilidad de $\mathcal{SC}$ a la comprobación de que $\mathcal{SC}$ no tiene modelos de Herbrand.

Teorema 3.62 *Sea $\mathcal{SC}$ un conjunto de cláusulas. Entonces, $\mathcal{SC}$ es insatisfacible si y sólo si $\mathcal{SC}$ no tiene modelos de Herbrand.*

```
80
insatisfacible_mod_herbrand: THEOREM
  es_insatisfacible(SC) ⇔ ∀ I: NOT es_modelo_herbrand(I,SC)
```

Demostración:

($\Leftarrow$) Es suficiente considerar que toda interpretación de Herbrand de $\mathcal{SC}$ es una interpretación.

($\Rightarrow$) Basta tener en cuenta que, si $I \models \mathcal{SC}$, entonces por $\boxed{78}$, $I \cap BH(\mathcal{SC})$ es un modelo de Herbrand de $\mathcal{SC}$.

□

Capítulo 4

Semántica de los programas lógicos

En este capítulo, se trata la semántica declarativa, procedimental y del punto fijo de los programas lógicos. Seguiremos, esencialmente, el desarrollo de J. Lloyd [48].

En la primera sección, se construye el menor modelo de Herbrand de un programa P, y se prueba que está formado por los átomos de la base de Herbrand de P que son consecuencias lógicas de P.

A continuación, se define el operador de consecuencia inmediata asociado a un programa, y se caracteriza la semántica de los programas definidos mediante el punto fijo de dicho operador.

Por último, en la tercera sección, se introduce la semántica procedimental de los programas definidos y se establece la adecuación y la completitud (débil) del método de resolución SLD. Finalmente, se introduce el concepto de regla de computación, se prueba la independencia de la regla de computación y se concluye con la completitud fuerte del método de resolución SLD.

4.1. Semántica declarativa

En esta sección construimos el menor modelo de Herbrand de un programa definido, $MMH(P)$, como la intersección de los modelos de Herbrand de P y probamos que el menor modelo de Herbrand de P coincide con el conjunto de átomos de la base de Herbrand de P, que son consecuencias lógicas de P. La formalización en PVS se encuentra desarrollada en la teoría `semantica_declarativa` (páginas 309–313).

En primer lugar, vemos que la base de Herbrand de un programa definido P es modelo de Herbrand de P, usando para probarlo la caracterización de modelo de una cláusula definida 55.

Lema 4.1 *La base de Herbrand de un programa definido P es modelo de Her-*

brand de P.

```
BH_es_modelo_de_Herbrand: THEOREM                                    81
  es_modelo_herbrand(base_herbrand(P),P)
```

Demostración:
Basta considerar que $BH(P)$ es modelo de cada cláusula C de P, pues contiene a la cabeza de C.

□

4.1.1. Menor modelo de Herbrand

El menor modelo de Herbrand de un programa P se define como la intersección de los modelos de Herbrand de P. Para realizar la especificación en PVS, en primer lugar, se caracterizan los modelos de Herbrand de P como los elementos del conjunto potencia de $LE(P)$ que son modelos de P[1].

Lema 4.2 *Una interpretación I es modelo de Herbrand de un programa P si y sólo si pertenece al conjunto potencia del lenguaje de P y es modelo de P.*

```
CNS_modelo_herbrand: LEMMA                                           82
   es_modelo_herbrand(I,P) ⇔ powerset(LE(P))(I) & es_modelo(I,P)
```

Definición 4.3 *El conjunto de los modelos de Herbrand de P es*

$$CMH(P) = \{I : I \in \mathcal{P}(LE(P)) \ \wedge \ I \models P\}$$

```
conj_modelos_herbrand(P): finite_set[set[atomo]] =                   83
  {I: set[atomo[T]] | powerset(LE(P))(I) & es_modelo(I,P)}
```

Además, usando la definición, se prueba que todos los conjuntos de $CMH(P)$ son finitos.

```
conj_modelos_herbrand_finitos: LEMMA                                 84
  every(is_finite)(conj_modelos_herbrand(P))
```

Definición 4.4 *El* **menor modelo de Herbrand** *de P, $MMH(P)$, es la intersección del conjunto $CMH(P)$.*

[1] Aunque podríamos definir $MMH(P)$ como el conjunto $\{I : I \subseteq LE(P) \wedge I \models P\}$, lo expresamos usando el conjunto potencia $\mathcal{P}(LE)$ pues será útil a la hora de especificar la correspondiente función evaluable, que describimos en el capítulo 7.

```
menor_modelo_herbrand(P): finite_set[atomo] =
  Intersection(conj_modelos_herbrand(P))
```
85

Para establecer que $MMH(P) \models P$, probamos que la intersección de una familia de modelos de un programa definido P es modelo de P. Como lema previo, demostramos que la intersección de una familia de modelos de una cláusula definida C es modelo de C.

Definición 4.5 *Decimos que un conjunto de interpretaciones $\mathcal{CI}$ es un* **conjunto de modelos** *de C si todo elemento de $\mathcal{CI}$ es modelo de C.*

```
es_conjunto_de_modelos?(CI:set[set[atomo]],C): bool =
  ∀ I: I ∈ CI ⇒ es_modelo(I, C)
```

Lema 4.6 *La intersección de una familia de modelos de una cláusula definida C es modelo de C.*

```
inters_familia_modelo_c_d: LEMMA
  ∀ (CI:set[set[atomo]]):
  es_conjunto_de_modelos?(CI,C) ⇒ es_modelo(Intersection(CI), C)
```
86

Demostración:

Sea C la cláusula definida $A \leftarrow B_1, \ldots, B_m$ y $\mathcal{CI}$ un conjunto de modelos de C. Veamos que $\bigcap \mathcal{CI} \models C$. En efecto, si $A \in \bigcap \mathcal{CI}$, por 55, $\bigcap \mathcal{CI} \models C$. En caso contrario, $\exists I \in \mathcal{CI} : A \notin I$. Por ser $I \models C$, $\exists B_j : B_j \notin I$. Luego, $B_j \notin \bigcap \mathcal{CI}$ y, por tanto, por 55, $\bigcap \mathcal{CI} \models C$.

□

Como consecuencia se tiene el resultado para programas.

Lema 4.7 *Sea P un programa definido y $\mathcal{CI}$ un conjunto de modelos de P. Entonces, $\bigcap \mathcal{CI} \models P$.*

```
es_conjunto_de_modelos?(CI:set[set[atomo]],P): bool =
  ∀ I: I ∈ CI ⇒ es_modelo(I, P)

inters_familia_modelo_conj_c_d: LEMMA
  es_conjunto_de_modelos?(CI,P) ⇒ es_modelo(Intersection(CI),P)
```
87

A partir de este lema, se obtiene el teorema siguiente:

Teorema 4.8 *Sea P un programa definido. Entonces, $MMH(P)$ es un modelo de Herbrand de P contenido en todo modelo de Herbrand de P.*

88
```
menor_modelo_es_modelo_herbrand: THEOREM
  es_modelo_herbrand(menor_modelo_herbrand(P),P)

menor_modelo_es_el_menor: THEOREM
  es_modelo_herbrand(I,P) ⇒ menor_modelo_herbrand(P) ⊆ I
```

Corolario 4.9 *El menor modelo de Herbrand de P está contenido en la base de Herbrand de P.*

89
```
menor_modelo_es_el_menor_corol: COROLLARY
   menor_modelo_herbrand(P) ⊆ base_herbrand(P)
```

Demostración:
Se prueba como consecuencia directa de *88* y *81*.

□

Corolario 4.10 *El menor modelo de Herbrand de P, $MMH(P)$, está contenido en todos los modelos de P.*

90
```
mmh_menor_todos_los_modelos: COROLLARY
  es_modelo(I,P) ⇒ menor_modelo_herbrand(P) ⊆ I
```

Demostración:
Si $I \models P$, por *79*, se tiene que $I \cap BH(P)$ es modelo de Herbrand de P. Entonces, por *88*, $MMH(P) \subseteq I \cap BH(P)$. Luego, $MMH(P) \subseteq I$.

□

4.1.2. Consecuencias lógicas de un programa

En esta subsección, formalizamos en PVS el resultado, debido a van Emden y Kowalski [96], que establece que $MMH(P)$ está formado por los átomos de la base de Herbrand de P, que son consecuencia lógica de P.

Teorema 4.11 *Sea P un programa definido. Entonces,*

$$MMH(P) = \{A \in BH(P) : P \models A\}$$

91
```
con_log_equiv_menor_modelo: THEOREM
  menor_modelo_herbrand(P) =
  {A: atomo | A ∈ base_herbrand(P) & es_cons_logica(A,P)}
```

Demostración:
Se prueba por doble inclusión. En primer lugar, establecemos que

$$MMH(P) \subseteq \{A \in BH(P) : P \models A\}$$

```
menor_modelo_con_log: LEMMA
  A ∈ menor_modelo_herbrand(P)) ⇒
  A ∈ base_herbrand(P) & es_cons_logica(A,P)
```

En efecto, si $A \in MMH(P)$, entonces $A \in I$ para todo modelo de Herbrand I de P. Luego, si I es modelo de Herbrand de P, $I \not\models \leftarrow A$. Por tanto $P \cup \{\leftarrow A\}$ no tiene modelos de Herbrand. Luego, por *80*, $P \cup \{\leftarrow A\}$ es insatisfacible y, por *66*, $P \models A$. Además, por *89*, $A \in BH(P)$.
En segundo lugar, probamos

$$\{A \in BH(P) : P \models A\} \subseteq MMH(P)$$

92
```
con_log_menor_modelo: LEMMA
   A ∈ base_herbrand(P) & es_cons_logica(A,P) ⇒
   A ∈ menor_modelo_herbrand(P)
```

Sea A un átomo de la base de Herbrand de P tal que $P \models A$. Basta probar que $A \in I$, para todo I modelo de Herbrand de P. En efecto, sea I un modelo de Herbrand de P. Entonces, por ser $P \models A$, $I \models A$ y, por tanto, $A \in I$.

□

Si denotamos por $CL(P)$ el conjunto de átomos de la base de Herbrand de P, que son consecuencias lógicas de P, obtenemos como corolario, que $MMH(P) = CL(P)$.

93
```
CL(P): finite_set[atomo[T]] =
  {A| A ∈ base_herbrand(P) & es_cons_logica(A,P)}

con_log_equiv_menor_modelo_c: COROLLARY menor_modelo_herbrand(P)=CL(P)
```

4.2. Semántica del punto fijo

En esta sección se caracteriza la semántica de los programas definidos mediante el punto fijo del operador de consecuencia inmediata. En [8] se ha formalizado la teoría de dominios en PVS y se ha probado el teorema del punto fijo de Tarski [93]. Además, se ha descrito una construcción del menor punto fijo para funciones monótonas y para funciones continuas. Nosotros usaremos que el operador

de consecuencia es monótono y la construcción del menor punto fijo para funciones monótonas que se describe en la teoría de dominios [8], para definir el menor punto fijo de un programa P y demostrar que coincide con el menor modelo de Herbrand de P. Además, se prueba que dicho operador es continuo y se le aplica también la construcción del menor punto fijo para funciones continuas desarrollada en la teoría de dominios.

Por último, se demuestra que el menor punto fijo de un programa P se alcanza en una potencia del operador consecuencia inmediata menor o igual que el cardinal de la base de Herbrand de P. La formalización de la sección se encuentra desarrollada en las teorías `consecuencia_i`, `inclusion_opc` y `semantica_p_f` (páginas 313–320).

4.2.1. Operador de consecuencia inmediata

En esta subsección se define el operador de consecuencia inmediata asociado a un programa. Se prueba que es monótono y se caracterizan los modelos del programa como los puntos prefijos de dicho operador.

En primer lugar, definimos el operador de consecuencia inmediata asociado a P sobre una interpretación general, no sólo sobre las interpretaciones de Herbrand de P. Probamos que es monótono y que su rango son las interpretaciones de Herbrand de P.

Definición 4.12 *Sea P un programa e $\mathcal{I}$ el conjunto de interpretaciones. El* **operador de consecuencia inmediata** *de P,*

$$T_P : \mathcal{I} \longrightarrow \mathcal{I}$$

se define como:

$$T_P(I) = \{A |\ \exists C \in P : C = A \leftarrow B_1, \ldots, B_n \ \wedge\ \{B_1, \ldots, B_n\} \subseteq I\}$$

94
```
c_i(P)(I):set[atomo] =
 {A:atomo |∃ C: C ∈ P & A=cabeza(C) & cuerpo(C) ⊆ I }
```

Veamos las propiedades de este operador.

Lema 4.13 *Sea P un programa definido e I una interpretación. Entonces, el conjunto de consecuencias inmediatas de I mediante T_P está contenido en la base de Herbrand de P. Es decir, $T_P(I) \subseteq BH(P)$.*

95
```
c_i_en_base_herbrand: LEMMA
   c_i(P)(I) ⊆ base_herbrand(P)
```

La prueba se basa en la propia definición del operador y en que las cabezas de las cláusulas de P pertenecen a la base de Herbrand de P.

Como consecuencia, se obtiene que para toda interpretación I, $T_P(I)$ es una interpretación de Herbrand de P y, por tanto, una interpretación finita. Es decir, el rango del operador T_P es el conjunto de interpretaciones finitas. Esto nos permite definir un operador de consecuencia inmediata restringido, como sigue[2]:

Definición 4.14 *Sea P un programa y sea $\mathcal{IF}$ el tipo de las interpretaciones finitas. El* **operador de consecuencia inmediata restringido** *de P,*

$$T_{Pf} : \mathcal{IF} \longrightarrow \mathcal{IF}$$

se define como: $T_{Pf}(I) = T_P(I)$, para $I \in \mathcal{IF}$.

96
```
FI:   VAR interpretacion_finita

c_i_f(P)(FI): interpretacion_finita = c_i(P)(FI)
```

Lema 4.15 *Para todo programa definido P, T_P es una función monótona.*

97
```
c_i_monotona: LEMMA
   I ⊆ J => c_i(P)(I) ⊆ c_i(P)(J)
```

La prueba se obtiene directamente a partir de la definición y de las propiedades de las operacionnes sobre conjuntos incluidas en el preludio de PVS.

En el proceso de clausura que conlleva el cálculo del menor punto fijo del operador T_P, se trabaja con las potencias de dicho operador sobre el conjunto vacío. Dichas potencias se definen recursivamente como:

- $T_P^0 = T_P^0(\emptyset) = \emptyset$
- $T_P^{n+1} = T_P^{n+1}(\emptyset) = T_P(T_P^n(\emptyset))$

En PVS, expresamos las potencias del operador de consecuencia inmediata, mediante la función `iterate`[3]

98
```
potencia_c_i(P,n): interpretacion = iterate(c_i(P),n)(emptyset)
```

Lema 4.16 *La base de Herbrand de un programa definido P contiene a todas las potencias del operador consecuencia inmediata de P.*

[2]En el capítulo 7 construiremos una especificación evaluable del operador restringido T_{Pf}.

[3]La función `iterate` está definida en el preludio de PVS como
`iterate(f, n)(x) = IF n = 0 THEN x ELSE f(iterate(f, n-1)(x)) ENDIF`

```
base_herbrand_contiene_potencias: LEMMA                                    99
  iterate(c_i(P),n)(emptyset) ⊆ base_herbrand(P)
```

Demostración:
Directamente, por la definición de `iterate` y 95.

□

Lema 4.17 *Sea P un programa definido. Las sucesivas potencias del operador consecuencia inmediata de P forman una cadena ascendente.*

```
c_i_iterate_vacio: LEMMA                                                   100
  iterate(c_i(P),n)(emptyset) ⊆ iterate(c_i(P),1+n)(emptyset)
```

Demostración:
Por inducción en n, usando que T_P es monótono 97.

□

Corolario 4.18 *Sea P un programa definido y m, n números naturales. Entonces,*

$$m \leq n \Rightarrow T_P^m(\emptyset) \subseteq T_P^n(\emptyset)$$

```
c_i_iterate_vacio_gen: COROLLARY                                           101
   m <= n ⇒ iterate(c_i(P),m)(emptyset) ⊆ iterate(c_i(P),n)(emptyset)
```

Demostración:
Por inducción en n, usando que T_P es monótono y 100.

□

Terminamos esta sección con el resultado que caracteriza los modelos de un programa P como los puntos prefijos del operador T_P. Recordemos que si $(D, \leq)$ es un orden parcial y $f : D \to D$, se dice que $x \in D$ es un punto prefijo de f si $f(x) \leq x$. (Nótese que en la expresión del teorema no usamos explícitamente la noción de punto prefijo).

Teorema 4.19 *Sea P un programa definido e I una interpretación. Entonces, $I \models P$ si y sólo si $T_P(I) \subseteq I$.*

```
c_i_modelos_h: THEOREM                                                     102
   es_modelo(I,P) ⇔ c_i(P)(I) ⊆ I
```

Demostración:

Sea I un modelo de P. Veamos que $T_P(I) \subseteq I$. En efecto, sea $A \in T_P(I)$. Por la definición de T_P, A es la cabeza de una cláusula C de P, tal que I contiene al cuerpo de C. Como $I \models P$, también $I \models C$, por lo que $A \in I$, por 55.

Recíprocamente, sea I una interpretación tal que $T_P(I) \subseteq I$ y C la cláusula de P, $A \leftarrow B_1, \ldots, B_k$. Veamos que $I \models C$. En efecto, si $\{B_1, \ldots, B_k\} \subseteq I$, entonces $A \in T_P(I)$ y, por tanto, $A \in I$. Luego, $I \models C$. En caso contrario, por 55, $I \models C$. □

4.2.2. Menor punto fijo de un programa

En esta subsección formalizamos el resultado, debido a van Emden y Kowalski [96], que proporciona una caracterización del menor modelo de Herbrand de un programa definido P, en función de los puntos fijos del operador de consecuencia inmediata de P. Para ello, usaremos la formalización de la teoría de dominios en PVS, realizada en [8], que hemos adaptado a la versión 3.1 de PVS. La biblioteca en la que se encuentran las teorías formalizadas correspondientes la denotamos por `PF` y nos referiremos a cada teoría con el prefijo `PF@`.

El uso de estas teorías, mediante el mecanismo de importación de PVS, requiere probar que la inclusión de conjuntos es un orden parcial precompleto, con el vacío como su menor elemento:

```
subset?_es_precop: LEMMA precpo?(subset?)

vacio_es_bottom: LEMMA bottom?(subset?)(emptyset)

subset?_es_cpo: THEOREM cpo?(subset?, emptyset)
```

En la teoría `PF@monotonic` se formaliza la noción de operador monótono, mediante el predicado `monotonic?` y el tipo de operadores monótonos como `Monotonic`. Usando la propiedad 97, se tiene que el operador de consecuencia inmediata asociado a un programa es de tipo `Monotonic`.

Lema 4.20 *El operador de consecuencia inmediata T_P es monótono.*

```
c_i_monotonic: LEMMA monotonic?(c_i(P))
```

Nota 4.21 En la teoría `PF@fixpoints` se formaliza el concepto de menor punto fijo como sigue: Sea $\leq$ un orden parcial en D, $x, y \in D$ y $f : D \to D$.

```
% Def (x es punto fijo de f)
fixpoint?(f)(x): bool = (f(x) = x)

% Def (x es menor punto fijo de f)
least_fixpoint?(f)(x) : bool =
  fixpoint?(f)(x) & (∀ y: fixpoint?(f)(y) ⇒ x <= y)

% Def (f tiene menor punto fijo)
mu_exists?(f): bool = nonempty?(least_fixpoint?(f))

% Tipo (de los menores puntos fijos de f)
LFP(f): TYPE = (least_fixpoint?(f))

% Tipo (de las funciones con menor punto fijo)
Mu_Exists: TYPE = (mu_exists?)

% Lema (unicidad del menor punto fijo)
least_fix_unique: LEMMA unique?(least_fixpoint?(f))

% Corolario (unicidad del menor punto fijo)
lfp_singleton: COROLLARY
  ∀ (f: Mu_Exists): singleton?(least_fixpoint?(f))

% Def (menor punto fijo de las funciones con punto fijo)
mu(f: Mu_Exists): LFP(f) = choose(least_fixpoint?(f))
```

Por otra parte, en la teoría `PF@admisible` se define la noción de predicado admisible de la forma siguiente: Sea $\leq$ un orden parcial precompleto en D. Un predicado ϕ sobre D es admisible si para toda cadena Cad, si sus elementos verifican ϕ, entonces el supremo de Cad también verifica ϕ.

```
admissible?(φ: pred[D]): bool =
  ∀ (Cad: Chain): every(φ)(Cad) ⇒ φ(lub(Cad))
```

Por último, en la teoría `PF@fixpoints_mono` se construye el menor punto fijo de los operadores monótonos como sigue: Sea $\leq$ un orden parcial completo en D con `bottom` como menor elemento, $x, y \in D$, $f : D \to D$ una función monótona y S un un conjunto de elementos de D.

```
% Def. (S es cerrado por pasos respecto del operador f)
step_closed?(f)(S): bool = (FORALL (y: (S)): S(f(y)))

% Def. (S es cerrado respecto del operador f)
closed?(f)(S): bool =
  contains?(bottom)(S) & step_closed?(f)(S) & admissible?(S)
```

```
% Def. El menor cerrado respecto de f es la intersección de todos
% los cerrados respecto de f.
X(f): set[D] = ⋀(closed?(f))

% Lemas que justifican la definición anterior
X_is_closed      : LEMMA closed?(f)(X(f))
X_is_least_closed: LEMMA closed?(f)(S) IMPLIES subset?(X(f), S)

% Def. (elementos maximales)
Max(A): set[D] = {x: D | max?(x, A)}

% Lema: El menor cerrado respecto de f tiene elemento maximal.
X_has_max: LEMMA nonempty?[D](Max(X(f)))

% Def. (menor punto fijo de f)
u(f): D = choose[D](Max(X(f)))

% Lema que justifica la definición anterior
mu_char: LEMMA mu(f) = u(f)

% Algunas propiedades del menor punto fijo demostrada en la teoría
% son las siguientes:
% Teorema de Knaster-Tarski:
KnasterTarski: THEOREM mu_exists?(f)

% Lema (de Park): el menor punto fijo de f es menor o igual que
% los puntos prefijos de f.
park: LEMMA f(x) <= x IMPLIES mu(f) <= x
```

Como hemos comentado, en la teoría auxiliar `inclusion_opc` (páginas 315–316), se establece que la relación de subconjunto, junto con el conjunto vacío, dotan a los conjuntos sobre un tipo `T` de la estructura de orden parcial completo. Así, se dispone del teorema del punto fijo para funciones monótonas y funciones continuas sobre conjuntos de átomos. Por otra parte, puesto que las interpretaciones son conjuntos de átomos y el operador de consecuencia inmediata es monótono, se puede usar la construcción anterior y definir el menor punto fijo de un programa como el menor punto fijo del operador de consecuencia inmediata.

Definición 4.22 *Sea P un programa. Definimos el* **menor punto fijo** *de P, y lo notamos por $mpf(P)$, como el menor punto fijo del operador T_P.*

```
mpf(P): interpretacion = u(c_i(P))
```
103

Teorema 4.23 *Sea P un programa definido. Entonces, el menor modelo de Herbrand de P, $MMH(P)$, coincide con el menor punto fijo del operador T_P, $mpf(P)$.*

```
mmh_es_mpf: THEOREM                                              104
   menor_modelo_herbrand(P) = u(c_i(P))
```

Demostración:

- $mpf(P) \subseteq MMH(P)$

  Por el lema de Park es suficiente probar que $T_P(MMH(P)) \subseteq MMH(P)$, lo que se tiene por la caracterización de los modelos de P mediante el operador T_P [102], y por ser $MMH(P)$ modelo de P [88].

- $MMH(P) \subseteq mpf(P)$

  Por ser $MMH(P)$ el menor modelo de Herbrand de P, basta probar que $mpf(P)$ es un modelo de Herbrand de P. En efecto, por ser punto fijo, $mpf(P) = T_P(mpf(P))$ y, por [102], se tiene que $mpf(P)$ es modelo de P. Además, por [95], es interpretación de Herbrand. Por tanto, $mpf(P)$ es modelo de Herbrand de P. Luego, $MMH(P) \subseteq mpf(P)$.

□

Como consecuencia, se obtienen los siguientes resultados:

Corolario 4.24 *Si I es un punto fijo del operador T_P, entonces existe una interpretación finita contenida en I, que es punto fijo de T_P.*

```
existencia_punto_fijo_finito: COROLLARY
  fixpoint?(c_i(P))(I)
  ⇒ ∃ J: is_finite(J) & fixpoint?(c_i(P))(J) & J ⊆ I
```

Demostración:

Basta tener en cuenta que $MMH(P)$ es finito, que es punto fijo de P, pues $MMH(P) = mpf(P)$, y que cualquier punto fijo I de P es punto prefijo. Luego, si I es punto fijo de P, por [102], $I \models P$. Por tanto $MMH(P) \subseteq I$.

□

Corolario 4.25 *El menor punto fijo de un programa P, $mpf(P)$, coincide con el conjunto de las consecuencias lógicas de P, $CL(P)$.*

```
mpf_es_cl: COROLLARY mpf(P) = CL(P)                              105
```

4.2.3. Construcción del menor punto fijo de un programa

Nota 4.26 En la teoría `PF@continuous` se estudian las funciones continuas de la forma siguiente: Sean D y R dos conjuntos parcialmente ordenados precompletos y sea $f : D \to R$.

```
% Def. Se dice que f es continua si para toda cadena Cad de D,
% la imagen de Cad por f tiene supremo y la imagen por f del
% supremo de Cad es el supremo de la imagen de Cad por f.
continuous?(f) : bool =
  FORALL Cad: lub_exists?(set_image(f)(Cad)) &
              (f(lub(Cad)) = lub(set_image(f)(Cad)))

% Juicio: Las funciones continuas son monótonas
JUDGEMENT Continuous SUBTYPE_OF Monotonic
```

Además, en la teoría `PF@monotonic` se demuestra que las funciones monótonas conservan las cadenas

```
image_preserves_chains: LEMMA
   FORALL (K: poD.Chain, f: Monotonic): chain?(set_image(f)(K))
```

Por otra parte, en la teoría `PF@fixpoints_cont` se construye el menor punto fijo de los operadores continuos de la siguiente forma: Sean $\leq$ un orden parcial completo en D con `bottom` como menor elemento, $f : D \to D$ una función monótona y $g : D \to D$ una función continua.

```
% Def. Conjunto de potencias de f a partir del menor elemento; es
% decir, {bottom, f(bottom), f(f(bottom)), ...}
bottom_iterations(f): Chain[D,<=] =
  seq_to_set(LAMBDA n: iterate(f, n)(bottom))

% Teorema: Si g es continua, entonces el supremo del conjunto de
% potencias de g a partir del menor elemento es el menor punto fijo
% de g.
fixpoint_theorem: THEOREM
  mu(g) = lub(bottom_iterations(g))
```

Así, para poder construir el menor punto fijo de un programa P como el supremo del conjunto de potencias de P, hemos de probar que el operador de consecuencia inmediata asociado, T_P, es continuo.

Lema 4.27 *Sean C una cadena formada por conjuntos de átomos, e I un conjunto finito de átomos contenido en el supremo de C. Entonces existe un elemento de la cadena que contiene a I.*

```
c_i_es_continua_l1: LEMMA                                          106
  ∀ (I: finite_set[atomo]):
    ∀ C: I ⊆ lub(C) & is_finite(I) ⇒ ∃ (A:(C)): I ⊆ A
```

La prueba de este lema se realiza por inducción en I, siguiendo el esquema de inducción para conjuntos finitos [1].

Teorema 4.28 *Sea P un programa definido. Entonces, el operador T_P es continuo.*

```
c_i_es_continua: THEOREM continuous?(c_i(P))                       107
```

Demostración:

Sea $\mathcal{I}$ el conjunto de las interpretaciones y Cad una cadena de elementos de $\mathcal{I}$. Según la definición de continuidad, hay que probar que $T_P(Cad)$ tiene supremo, y que $supremo(T_P(Cad)) = T_P(supremo(Cad))$.

1. En efecto, por ser T_P monótona, se tiene que $T_P(Cad)$ es una cadena y, por ser $(\mathcal{I}, \subseteq)$ un orden parcial precompleto, toda cadena tiene supremo. Por tanto, $T_P(Cad)$ tiene supremo.

2. Veamos que $supremo(T_P(Cad)) = T_P(supremo(Cad))$:

   - $supremo(T_P(Cad)) \subseteq T_P(supremo(Cad))$

     Es suficiente probar que $T_P(supremo(Cad))$ es una cota superior del conjunto imagen[4] $T_P(Cad)$.

     ```
     c_i_es_continua_l3_1: LEMMA
       chain?(C) ⇒ ub?(c_i(P)(lub(C)), set_image(c_i(P))(C))
     ```

   - $T_P(supremo(Cad)) \subseteq supremo(T_P(Cad))$

     ```
     c_i_es_continua_l2: LEMMA
      c_i(P)(lub(C)) ⊆
      po[set[atomo],sets[atomo].subset?].lub(set_image(c_i(P))(C))
     ```

     Sea $A \in T_P(supremo(Cad))$. Entonces, existe una cláusula $C \in P$, de la forma $A \leftarrow B_1, \ldots, B_k$ tal que $\{B_1, \ldots, B_k\} \subseteq supremo(Cad)$. Entonces, por la propiedad [106], $\{B_1, \ldots, B_k\} \subseteq I$, para algún I de la cadena Cad. Por tanto, $A \in T_P(I)$ que, a su vez, es un elemento de la cadena imagen $T_P(Cad)$. Por tanto, $T_P(I) \subseteq supremo(T_P(Cad))$. Luego, $A \in supremo(T_P(Cad))$.

[4] La función `set_image` del preludio obtiene el conjunto imagen de un conjunto S mediante una función f.

□

Así, una vez que se tiene que el operador T_P es continuo, se puede obtener el menor punto fijo de T_P de la forma siguiente:

Teorema 4.29 *Sea P un programa definido. Entonces, el menor punto fijo P es el supremo del conjunto de potencias del operador de consecuencia inmediata a partir del conjunto vacío. Es decir,*

$$mpf(P) = supremo(\{T_P^n(\emptyset) : n \in \mathbb{N}\})$$

108
```
IMPORTING PF@fixpoints_cont[set[atomo], subset?, emptyset]

punto_fijo_c_i: THEOREM
  mu(c_i(P)) = lub(bottom_iterations(c_i(P)))

calculo_mpf: COROLLARY
  mpf(P) = lub(bottom_iterations(c_i(P)))
```

4.2.4. Cálculo del menor punto fijo de un programa

En esta sección probamos que, para cualquier programa P, el supremo de la cadena

$$T_P^0 \subseteq T_P^1 \subseteq \cdots \subseteq T_P^k \ldots$$

se alcanza para m, con $m \leq |BH(P)|$.

El esquema de la prueba es el siguiente: Por una parte, se tiene que $BH(P)$ es una interpretación finita y cota superior de la cadena. Por tanto, los conjuntos que forman la cadena no pueden crecer indefinidamente. Más aún, no pueden crecer más que $BH(P)$, por lo que la cadena es, necesariamente, finita. Y por otra, en el momento en el que, en un paso, la cadena no aumente estrictamente, se ha alcanzado el punto fijo y, por tanto, la cadena se estabiliza. Ahora bien, este paso tiene que ser para una potencia m, con $m \leq |BH(P)|$.

En PVS, establecemos los siguientes resultados.

Lema 4.30 *La base de Herbrand de P es cota superior del conjunto de potencias del operador consecuencia inmediata.*

109
```
base_herbrand_cota_superior_c_i: COROLLARY
  ub?(base_herbrand(P), bottom_iterations(c_i(P)))
```

Demostración:

Por aplicación directa de *99*.

□

Lema 4.31 *Si en el paso $n+1$, la cadena formada por las potencias del operador consecuencia inmediata a partir del vacío no ha colapsado, $|T_P^{n+1)}(\emptyset)| \geq n+1$.*

110
```
cardinal_potencias_c_i_b: LEMMA
  strict_subset?(potencia_c_i(P,n), potencia_c_i(P,n+1))
  ⇒ card(potencia_c_i(P,n+1)) >= n+1
```

Demostración:

Por inducción en n, usando las propiedades de conjuntos incluidas en el preludio de PVS.

□

Lema 4.32 *Sea P un programa definido y $n = |BH(P)|$. Entonces, $T_P^n(\emptyset)$ es un punto fijo del programa.*

111
```
CS_potencia_punto_fijo: LEMMA
  n = card(base_herbrand(P))
  ⇒ fixpoint?(c_i(P))(iterate(c_i(P),n)(emptyset))
```

Demostración:

Por *100*, se tiene que $T_P^n(\emptyset) \subseteq T_P(T_P^n(\emptyset))$. Si no se tuviera la igualdad, por *110*, $|T_P(T_P^n(\emptyset))| \geq n+1$. Por otra parte, por *99*, $T_P(T_P^n(\emptyset)) \subseteq BH(P)$. Luego, $|T_P(T_P^n(\emptyset))| \leq n$. Por tanto, ha de ser $T_P^n(\emptyset) = T_P(T_P^n(\emptyset))$. Es decir, $T_P^n(\emptyset)$ es punto fijo de T_P.

□

Lema 4.33 *Sea P un programa definido. Si $T_P^m(\emptyset)$ es punto fijo de T_P, entonces $T_P^n(\emptyset) = T_P^m(\emptyset)$, $\forall n \geq m$.*

112
```
CN_potencia_punto_fijo: LEMMA
  fixpoint?(c_i(P))(iterate(c_i(P),m)(emptyset))
  ⇒ ∀ n: m <= n ⇒
          iterate(c_i(P),n)(emptyset) = iterate(c_i(P),m)(emptyset)
```

La prueba se realiza, directamente, por inducción y simplificación en n.

Corolario 4.34 *Sea I una interpretación. Si $I \in \{T_P^n(\emptyset) : n \in \mathbb{N}\}$, entonces $I = T_P^n(\emptyset)$, para algún $n \leq |BH(P)|$.*

```
113
CN_pertenencia_potencias_c_i: COROLLARY
  I ∈ bottom_iterations(c_i(P)) ⇒
  ∃ n: n <= card(base_herbrand(P)) & I = iterate(c_i(P),n)(emptyset)
```

Como consecuencia, se obtiene que el conjunto de potencias del operador de consecuencia inmediata de un programa, a partir del vacío, es finito.

Lema 4.35 *Para todo programa definido P, el conjunto $\{T_P^n(\emptyset) : n \in \mathbb{N}\}$ es finito.*

```
114
potencias_c_i_finito: LEMMA
  is_finite(bottom_iterations(c_i(P)))
```

Demostración:

Sea P un programa definido y $n = |BH(P)|$. Por 5, es suficiente considerar la función $f : \{0, 1, \ldots, n+1\} \to \{T_P^n(\emptyset) : n \in \mathbb{N}\}$, definida como $f(k) = T_P^k(\emptyset)$ y probar que es sobreyectiva.

□

Teorema 4.36 *El menor punto fijo de un programa P se alcanza en una potencia del operador de consecuencia inmediata menor o igual que el cardinal de la base de Herbrand de P.*

```
115
mpf_potencia_c_i: THEOREM
  ∃ n: n <= card(base_herbrand(P)) &
           mpf(P) = iterate(c_i(P),n)(emptyset)
```

Demostración:

Sea P un programa definido. Por 108 hay que probar que $\exists n \leq |BH(P)|$ tal que $supremo(\{T_P^n(\emptyset) : n \in \mathbb{N}\}) = T_P^n(\emptyset)$ y, por 113, es suficiente probar que $supremo(\{T_P^n(\emptyset) : n \in \mathbb{N}\}) \in \{T_P^n(\emptyset) : n \in \mathbb{N}\})$.

Ahora bien, en la teoría `PF@propiedades_cadena` se ha probado que el supremo de toda cadena finita en un conjunto dotado de un orden parcial precompleto pertenece a la cadena. Luego, es suficiente probar que la cadena formada por las potencias del operador de consecuencia inmediata a partir del vacío es finita, lo que se tiene por 114.

□

Corolario 4.37 *Sea P un programa definido. Entonces, $mpf(P) = T_P^n(\emptyset)$, siendo $n = |BH(P)|$.*

```
116
mpf_potencia_c_i_corol: COROLLARY
   mpf(P) = iterate(c_i(P),card(base_herbrand(P)))(emptyset)
```

Como consecuencia, se obtiene también el mismo resultado considerando el operador de consecuencia inmediata, restringido a interpretaciones finitas[5].

Corolario 4.38 *El menor punto fijo de un programa P es la n–sima potencia del operador consecuencia inmediata restringido, siendo $n = |BH(P)|$.*

```
117
mpf_potencia_c_i_corol_f: COROLLARY
  mpf(P) = iterate(c_i_f(P),card(base_herbrand(P)))(emptyset)
```

4.3. Semántica procedimental

El método de prueba subyacente a *Prolog* es una variante del *método de resolución*, debido originalmente a Robinson [71]. Dicho método se usa para determinar si una fórmula F es consecuencia de un conjunto de fórmulas $\mathcal{S}$, lo que equivale a demostrar que el conjunto de cláusulas correspondientes a $\mathcal{S} \cup \{\neg F\}$ es insatisfacible.

En general, una cláusula proposicional es un conjunto finito de literales (átomos o negaciones de átomos). El proceso de resolución general se aplica a un conjunto de cláusulas, y se basa en la noción de *resolvente*: dadas las cláusulas C_1 y C_2, y el átomo A, con $A \in C_1$ y $\neg A \in C_2$, la cláusula $C = (C_1 \backslash \{A\}) \cup (C_2 \backslash \{\neg A\})$ se denomina resolvente de C_1 y C_2.

El método de resolución para determinar la insatisfacibilidad de un conjunto $\mathcal{S}$ consiste en calcular la clausura de $\mathcal{S}$ bajo resolución, $\mathcal{R}(\mathcal{S})$, y comprobar si la cláusula vacía $\square \in \mathcal{R}(\mathcal{S})$, pues se verifica el siguiente resultado:

$$\mathcal{S} \text{ es insatisfacible } \Leftrightarrow \square \in \mathcal{R}(\mathcal{S})$$

Una *refutación* de $\mathcal{S}$ es una sucesión finita de cláusulas $C_1, C_2, \ldots, C_n = \square$, tales que cada C_i pertenece a $\mathcal{S}$, o es la resolvente de las cláusulas C_j, C_k, con $j, k < i$.

Ahora bien, cuando se busca sistemáticamente una refutación para un conjunto dado $\mathcal{S}$, se puede generar un espacio de búsqueda muy grande. Por ello, se han desarrollado distintos refinamientos del método de resolución. Consideremos uno de ellos: la resolución *lineal por entradas*.

- Una *deducción* por resolución lineal de C a partir de $\mathcal{S}$ es una sucesión de pares $< C_0, B_0 >, \ldots, < C_n, B_n >$, tales que $C = C_{n+1}$ y

[5] Este resultado lo usaremos en el capítulo 7 para construir una especificación evaluable que calcule el menor punto fijo de un programa P.

- $C_0, B_i \in \mathcal{S}$
- Cada $C_{i+1}, i \leq n$ es una resolvente de C_i y B_i.

- Si $C = \square$, se dice que es una *refutación lineal* de $\mathcal{S}$.

En el contexto de la resolución lineal, los elementos de $\mathcal{S}$ usados en la deducción suelen denominarse cláusulas de entrada y C_0 cláusula de partida.

Si denotamos por $\mathcal{L}(\mathcal{S})$ al conjunto de todas las cláusulas linealmente deducibles de $\mathcal{S}$, se tiene que si $\mathcal{S}$ es un conjunto de cláusulas de Horn, entonces

$$\mathcal{S} \text{ es insatisfacible } \Leftrightarrow \square \in \mathcal{L}(\mathcal{S})$$

La visión general de un programa es que tenemos un conjunto de reglas y hechos, de los cuales queremos deducir consecuencias. En general, queremos saber si una conjunción de hechos $B_1 \wedge \cdots \wedge B_n$ se puede deducir de un programa P. La idea es añadirle a P el objetivo $G :\leftarrow \{B_1, \ldots, B_n\}$ (es decir, la fórmula equivalente $\neg B_1 \vee \cdots \vee \neg B_n$) y determinar si el conjunto de cláusulas de Horn $P \cup \{G\}$ es insatisfacible. En caso afirmativo, por 67, tendremos que $\forall i, B_i$ es consecuencia lógica de P.

La representación elegida de cláusula de Horn usa conjuntos finitos de átomos, por lo que entre ellos no hay un orden determinado ni se tienen elementos repetidos. Sin embargo, un procedimiento que tenga que ser evaluado en un ordenador tendrá que usar secuencias o listas de átomos, en vez de conjuntos[6].

La especificación del método de resolución que vamos a hacer en esta sección se encuentra desarrollada en la teoría `sld_resolucion` (páginas 320–324). Tiene las siguientes características:

- Las cláusulas están representadas usando conjuntos finitos de átomos.
- Especificamos el método de resolución lineal con cláusula de partida, siendo el programa un conjunto finito de cláusulas definidas.
- En cada paso, el átomo del objetivo G elegido para resolver no dependerá del uso de una regla de selección determinada.

En primer lugar, establecemos la noción de resolvente de una cláusula definida y un objetivo definido.

Definición 4.39 *Sean $G :\leftarrow \{A_1, \ldots, A_m\}$ un objetivo definido, y $C : A \leftarrow \{B_1, \ldots, B_k\}$ una cláusula definida. Decimos que G' es una resolvente de G y C si existe A_j tal que $A = A_j$ y $G' :\leftarrow (\{A_1, \ldots, A_m\} \setminus \{A\}) \cup \{B_1, \ldots, B_k\}$*

Para especificar en PVS la noción de resolvente de un objetivo y una cláusula, establecemos previamente el tipo de objetos sobre los que tiene sentido definirla:

[6] En el capítulo 7 especificaremos procedimientos evaluables del método de resolución.

```
tiene_resolvente(G,C): bool = NOT vacia?(G) & cabeza(C)∈cuerpo(G)                    118

PD: TYPE = {par: [objetivo_def[T], clausula_def[T]] |
                 LET (G, C) = par IN tiene_resolvente(G,C)}

resolvente(par:PD): objetivo_def[T] =
  LET (G,C) = par IN objetivo(remove(cabeza(C),cuerpo(G)) ∪ cuerpo(C))
```

Desde el punto de vista algorítmico, la función `resolvente` especificada no representa el cálculo de la resolvente en un proceso concreto de resolución. Ahora bien, la idea del proceso general de resolución es la siguiente: dado un conjunto de reglas y hechos (representados por el programa P), y una pregunta (representada por el objetivo G), el proceso consiste en ir eliminando los subobjetivos de G, formados por cada átomo de su cuerpo que, pueden deducirse de P. En este sentido, en cada paso del proceso de resolución que estamos especificando se elimina (o se resuelve) un átomo diferente. En cambio, si el proceso maneja listas o sucesiones, un mismo átomo puede estar repetido y resolverse varias veces, pues en cada paso se elimina sólo el átomo que ocupa una posición.

A partir de la definición de resolvente, especificamos la noción de *refutación*. Para ello, en primer lugar, definimos el concepto de *refutación de longitud n*, por inducción en n, como sigue.

Definición 4.40 *Sea P un programa definido y G un objetivo definido. Decimos que $P \cup \{G\}$* **tiene una refutación de longitud** *n si se verifica alguna de las condiciones siguientes:*

- *$n = 0$ y $G = \square$*
- *$n > 0$ y P contiene una cláusula C tal que cabeza(C) $\in$ cuerpo(G) y $P \cup \{$resolvente(G,C)$\}$ tiene una refutación de longitud $n - 1$.*

```
tiene_refutacion_long(P,G,n): RECURSIVE bool =                                       119
  IF n = 0
     THEN vacia?(G)
     ELSE ∃ C: C ∈ P & cabeza(C) ∈ cuerpo(G) &
               tiene_refutacion_long(P,resolvente(G,C), n-1)
  ENDIF
  MEASURE n
```

Definición 4.41 *Sea P un programa definido y G un objetivo definido. Decimos que $P \cup \{G\}$* **tiene una refutación** *si $P \cup \{G\}$ tiene una refutación de longitud n, para algún n.*

```
tiene_refutacion(P,G): bool = ∃ n: tiene_refutacion_long(P,G,n)                      120
```

A partir de la noción de refutación, construimos el conjunto de *éxitos de P* como el conjunto de átomos A de la base de Herbrand de P tales que $P \cup \{\leftarrow \{A\}\}$ tiene una refutación.

Definición 4.42 *Sea P un programa definido. El* **conjunto de éxitos** *de P es*

$$Exitos(P) = \{A | A \in BH(P) \ \wedge \ P \cup \{\leftarrow \{A\}\} \textit{ tiene una refutación}\}$$

121
```
Exitos(P): finite_set[atomo[T]] =
  {A | A ∈ base_herbrand(P)) &
       tiene_refutacion(P,objetivo(singleton(A))) }
```

4.3.1. Adecuación del método de resolución SLD

El objetivo de esta subsección es demostrar el siguiente teorema de adecuación.

Teorema 4.43 *Sea P un programa definido y G un objetivo definido. Si $P \cup \{G\}$ tiene una refutación, entonces $P \cup \{G\}$ es insatisfacible.*

122
```
sld_adecuacion: THEOREM
  tiene_refutacion(P,G) ⇒ es_insatisfacible(add(G, P))
```

La demostración se basa en el siguiente lema, que establece que para todo n, si $P \cup \{G\}$ tiene una refutación de longitud n, $P \cup \{G\}$ es insatisfacible.

123
```
tiene_refutacion_long_insatisfacible: LEMMA
  tiene_refutacion_long(P,G,n) ⇒ es_insatisfacible(add(G, P))
```

Demostración:
Por inducción en n:

- $n = 0$. Entonces, $G = \square$ y, por tanto, $P \cup \{\square\}$ es insatisfacible.
- Supongamos que $P \cup \{G\}$ tiene una refutación de longitud $n + 1$. Según la definición, P tiene una cláusula C tal que cabeza(C) $\in$ cuerpo(G) y $P \cup \{\text{resolvente(G,C)}\}$ tiene una refutación de longitud n. Por hipótesis de inducción $P \cup \{\text{resolvente(G,C)}\}$ es insatisfacible.

  Veamos que $P \cup \{G\}$ es insatisfacible. En efecto, sea I una interpretación. Probamos que $I \not\models P \cup \{G\}$.

  - Si $I \not\models P$, entonces $I \not\models P \cup \{G\}$.

- Si $I \models P$, entonces $I \not\models$ resolvente(G,C), por ser $P \cup \{\text{resolvente(G,C)}\}$ insatisfacible. Entonces, por 56,

$$\text{cuerpo(resolvente(G,C))} = (\text{cuerpo(G)} \setminus \{A\}) \cup \text{cuerpo(C)} \subseteq I$$

donde A = cabeza(C). Luego, cuerpo(C) $\subseteq I$. Además $I \models C$, pues $C \in P$. Por tanto, por 55, $A \in I$. Como también cuerpo(G)$\setminus\{A\} \subseteq I$, se tiene cuerpo(G) $\subseteq I$ y, por tanto, $I \not\models G$.

□

Como consecuencia de 122 y 67, se obtienen los siguientes resultados:

Corolario 4.44 *Sea P un programa definido y G un objetivo definido. Si $P \cup \{G\}$ tiene una refutación, entonces $P \models A$, para todo átomo A del cuerpo de G.*

124
```
tiene_refutacion_cons_logica: COROLLARY
  tiene_refutacion(P,G) ⇒ (∀ A: A ∈ cuerpo(G) ⇒ es_cons_logica(A,P))
```

Corolario 4.45 *Sea P un programa definido. Entonces, Exitos(P) $\subseteq CP(P)$*

Y, como consecuencia de 124 y 91, se obtiene:

Corolario 4.46 *Dado un programa definido P, se tiene Exitos(P) $\subseteq MMH(P)$*

125
```
exitos_contenido_menor_modelo_herbrand: COROLLARY
  Exitos(P) ⊆ menor_modelo_herbrand(P)
```

Como $MMH(P) = mpf(P)$, tenemos también que Exitos(P) $\subseteq mpf(P)$. Veamos una propiedad que precisa más, en función de la longitud de la refutación.

Proposición 4.47 *Sean P un programa definido y G un objetivo definido. Si $P \cup \{G\}$ tiene una refutación de longitud n, entonces cuerpo(G) $\subseteq T_P^n(\emptyset)$*

126
```
tiene_refutacion_long_consecuencia_i: LEMMA
  tiene_refutacion_long(P,G,n) ⇒ cuerpo(G) ⊆ iterate(c_i(P),n)(∅)
```

Demostración:

Por inducción en n:

- Si $n = 0$, entonces $G = \Box$. Luego, cuerpo(G) = $\emptyset \subseteq T_P^0(\emptyset)$.

- Supongamos que $P \cup \{G\}$ tiene una refutación de longitud $n+1$. Por definición, existe una cláusula $C \in P$, tal que $P \cup \{\text{resolvente(G,C)}\}$ tiene una refutación de longitud n. Por hipótesis de inducción,

$$\text{cuerpo(resolvente(G,C))} = (\text{cuerpo(G)} \setminus \{A\}) \cup \text{cuerpo(C)}) \subseteq T_P^n(\emptyset)$$

donde $A = \text{cabeza(C)}$. Entonces, $\text{cuerpo(resolvente(G,C))} \subseteq T_P^{n+1}(\emptyset)$ por *100* y, por tanto, $\text{cuerpo(G)} \setminus \{A\} \subseteq T_P^{n+1}(\emptyset)$. Por otra parte, por definición de T_P, $A \in T_P^{n+1}(\emptyset)$. Luego, $\text{cuerpo(G)} \subseteq T_P^{n+1}(\emptyset)$.

□

4.3.2. Completitud del método de resolución SLD

El objetivo de esta subsección es demostrar el siguiente teorema de completitud.

Teorema 4.48 *Sea P un programa definido y G un objetivo definido. Si $P \cup \{G\}$ es insatisfacible, entonces $P \cup \{G\}$ tiene una refutación.*

La prueba de este teorema necesita algunos resultados previos.

En particular, probamos una propiedad que nos garantiza que si todos los átomos A del cuerpo de un objetivo G verifican que $P \cup \{\leftarrow \{A\}\}$ tiene una refutación, entonces $P \cup \{G\}$ tiene una refutación. Para probarlo, necesitamos el siguiente lema.

Lema 4.49 *Sean P un programa definido, G un objetivo definido y A un átomo. Si $P \cup \{G\}$ tiene una refutación de longitud m y $P \cup \{\leftarrow \{A\}\}$ tiene una refutación de longitud k, entonces $P \cup \{\leftarrow$ cuerpo(G)$\cup \{A\}\}$ tiene una refutación de longitud $n \leq m + k$.*

127

```
composicion_refutaciones_1: LEMMA
  tiene_refutacion_long(P,G,m) &
  tiene_refutacion_long(P, objetivo(singleton(A)), k) ⇒
  ∃n:n <= m+k & tiene_refutacion_long(P,objetivo(add(A,cuerpo(G))),n)
```

Demostración:

Por inducción en m:

- Si $m = 0$, entonces $G = \square$. Luego, podemos tomar $n = k$.
- Sea $m \neq 0$ y supongamos que $P \cup \{G\}$ tiene una refutación de longitud m.
  - Si $A \in \text{cuerpo}(G)$, entonces $\leftarrow \text{cuerpo}(G) \cup \{A\} = G$, por lo que es suficiente tomar $n = m$.

- Si $A \notin$ cuerpo(G), entonces existe $C \in P$ tal que tiene resolvente con G, y $P \cup \{\text{resolvente}(G, C)\}$ tiene una refutación de longitud $m-1$. Por hipótesis de inducción, existe $n_1 \leq m-1+k$ tal que $P \cup \{\leftarrow \text{cuerpo}(\text{resolvente}(G, C)) \cup \{A\}\}$ tiene una refutación de longitud n_1. Entonces, basta tomar $n = n_1 + 1$ y tener en cuenta que, debido a que $A \notin$ cuerpo(G),

  $$\leftarrow \text{cuerpo}(\text{resolvente}(G, C)) \cup \{A\} = \text{resolvente}(\leftarrow \text{cuerpo}(G) \cup \{A\}, C)$$

□

Lema 4.50 *Sea P un programa definido y F un conjunto finito de átomos. Si $P \cup \{\leftarrow \{A\}\}$ tiene una refutación, para todo átomo A de F, entonces $P \cup \{\leftarrow F\}$ tiene una refutación.*

128
```
composicion_refutaciones_2: LEMMA
 (∀ A: A ∈ F ⇒ tiene_refutacion(P, objetivo(singleton(A))))
 ⇒ tiene_refutacion(P, objetivo(F))

composicion_refutaciones_2_corol: COROLLARY
 (∀ A: A ∈ cuerpo(G) ⇒ tiene_refutacion(P,objetivo(singleton(A))))
 ⇒ tiene_refutacion(P, G)
```

Demostración:

Por inducción en F, siguiendo el esquema *2*, descrito en el capítulo 2:

- Si $F = \emptyset$, entonces $P \cup \{\square\}$ tiene una refutación de longitud 0.
- Sea $F \neq \emptyset$ y supongamos que $P \cup \{\leftarrow \{A\}\}$ tiene una refutación, para todo átomo $A \in \texttt{rest}(F)$. Por hipótesis de inducción, $P \cup \{\leftarrow \texttt{rest}(F)\}$ tiene una refutación. Además, como $\texttt{choose}(F) \in F$, $P \cup \{\leftarrow \{\texttt{choose}(F)\}\}$ tiene una refutación. Entonces, por *127*, $P \cup \{\leftarrow F\}$ tiene una refutación, pues $F = \texttt{rest}(F) \cup \{\texttt{choose}(F)\}$

□

La siguiente proposición puede considerarse, en cierto sentido, recíproca de *126*.

Proposición 4.51 *Sean P un programa definido y G un objetivo definido. Si cuerpo$(G) \subseteq T_P^n(\emptyset)$, entonces $P \cup \{G\}$ tiene una refutación.*

129
```
consecuencia_i_tiene_refutacion: LEMMA
  A∈iterate(c_i(P),n)(∅) ⇒ tiene_refutacion(P,objetivo(singleton(A)))
```

Demostración:
Por inducción en n:

- $n = 0$, trivial.
- Sea $n \neq 0$ y supongamos que $A \in T_P^n(\emptyset)$. Por definición, existe $C \in P$ de la forma $A \leftarrow \{B_1, \ldots, B_k\}$ con $\{B_1, \ldots, B_k\} \subseteq T_P^{n-1}(\emptyset)$. Por hipótesis de inducción, para todo i, $P \cup \{\leftarrow \{B_i\}\}$ tiene una refutación. Entonces, por [128], $P \cup \{\leftarrow \{B_1, \ldots, B_k\}\}$tiene una refutación. Por tanto, $P \cup \{\leftarrow \{A\}\}$ tiene una refutación.

□

Corolario 4.52 *Sea P un programa definido. Entonces, $MMH(P) \subseteq$ Exitos(P)*

[130]
```
menor_modelo_herbrand_contenido_exitos: COROLLARY
   menor_modelo_herbrand(P) ⊆ Exitos(P)
```

Demostración:

$$
\begin{array}{lll}
 & A \in MMH(P) & \\
\Rightarrow & A \in mpf(T_P) & (\text{por } \boxed{104}) \\
\Rightarrow & A \in supremo(\{T_P^n(\emptyset) : n \in \mathbb{N}\} & (\text{por } \boxed{107}) \\
\Rightarrow & \exists n : A \in T_P^n(\emptyset) & \\
\Rightarrow & P \cup \{\leftarrow \{A\}\} \text{ tiene refutación} & (\text{por } \boxed{129}) \\
\Rightarrow & A \in \text{Exitos}(P) & (\text{por } \boxed{121})
\end{array}
$$

□

Como consecuencia se obtiene el resultado siguiente:

Teorema 4.53 *Sea P un programa definido. Entonces, $MMH(P) =$ Exitos(P)*

[131]
```
menor_modelo_herbrand_exitos: THEOREM
  menor_modelo_herbrand(P) = Exitos(P)
```

Terminamos la subsección con la prueba del teorema de completitud del método de resolución SLD.

[132]
```
sld_completitud: THEOREM
   es_insatisfacible(add(G,P)) ⇒ tiene_refutacion(P,G)
```

Demostración:

Si $P \cup \{G\}$ es insatisfacible, $P \cup \{G\}$ no tiene modelos. En particular, $MMH(P)$ no es modelo de $P \cup \{G\}$. Como $MMH(P) \models P$, se tiene que $MMH(P) \not\models G$. Luego, cuerpo$(G) \subseteq MMH(P)$. Por *130*, cuerpo$(G) \subseteq$ Exitos(P), por lo que para todo átomo A del cuerpo(G), $P \cup \{\leftarrow \{A\}\}$ tiene una refutación. Luego, por *128*, $P \cup \{G\}$ tiene una refutación.

□

4.3.3. Completitud fuerte de la resolución SLD

El resultado de completitud previo, también llamado de completitud débil, establece que si $P \cup \{G\}$ es insatisfacible, existe alguna refutación de $P \cup \{G\}$. Ahora bien, si queremos disponer de un procedimiento concreto para encontrar dicha refutación, es necesario considerar dos cuestiones: el orden en el que se van a ir resolviendo los átomos del objetivo y el proceso de búsqueda.

En esta subsección introducimos el concepto de regla de computación, que se usará para seleccionar los átomos en un proceso de refutación. El objetivo es es establecer la completitud fuerte del método de resolución SLD: " si $P \cup \{G\}$ es insatisfacible, entonces $P \cup \{G\}$ tiene una refutación mediante cualquier regla de computación". Para ello, probaremos la independencia de la regla de computación: "si $P \cup \{G\}$ tiene una refutación, entonces $P \cup \{G\}$ tiene una refutación vía cualquier regla de computación". La formalización en PVS se encuentra desarrollada en la teoría `sld_resolucion_via` (páginas 324–337).

En primer lugar, establecemos la noción de regla de computación y declaramos el tipo que las representa:

133
```
es_regla_computacion(f:[objetivo_def[T] -> atomo[T]]): bool =
  ∀ G: f(G) ∈ cuerpo(G)

regla_computacion: TYPE = (es_regla_computacion)

f: VAR regla_computacion
```

Definimos el concepto de refutación vía una regla de computación, de manera análoga al concepto general de refutación *120*.

Definición 4.54 *Sea P un programa definido, G un objetivo definido y f una regla de computación. Decimos que $P \cup \{G\}$* **tiene una refutación vía f de longitud** *n si se verifica alguna de las condiciones siguientes:*

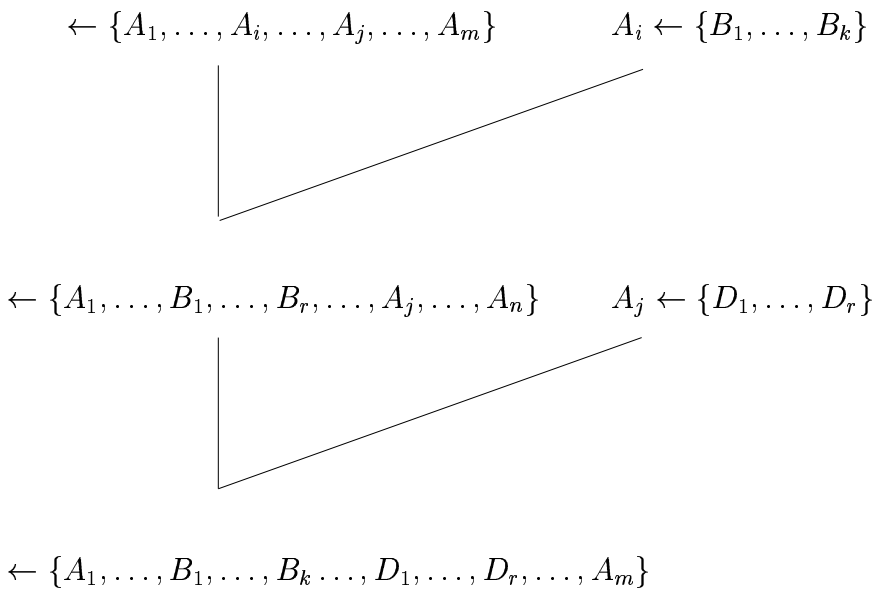
- *$n = 0$ y $G = \square$.*
- *$n > 0$ y P contiene una cláusula C tal que cabeza(C) $= f($cuerpo(G)$)$ y $P \cup \{$resolvente(G,C)$\}$ tiene una refutación vía f de longitud $n - 1$.*

```
tiene_refutacion_long_via(P,G,n,f) : RECURSIVE bool =
  IF n = 0
     THEN vacia?(G)
     ELSE ∃ C: C ∈ P & cabeza(C) = f(G)  &
                tiene_refutacion_long_via(P,resolvente(G,C),n-1,f)
  ENDIF
  MEASURE n
```
134

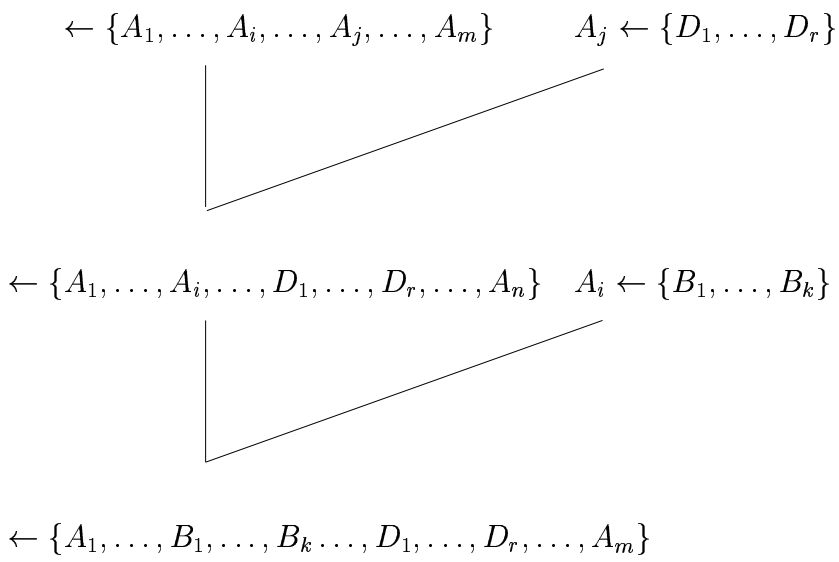
Definición 4.55 *Sea P un programa definido, G un objetivo definido y f una regla de computación. Decimos que $P \cup \{G\}$* **tiene una refutación vía** *f si $P \cup \{G\}$ tiene una refutación vía f de longitud n, para algún n.*

```
tiene_refutacion_via(P,G,f): bool =
  ∃ n: tiene_refutacion_long_via(P,G,n,f)
```
135

En [48], Lloyds prueba la independencia de la regla de computación basándose en la aplicación reiterada, un número finito de veces, de un lema técnico: el lema del intercambio. Dicho lema asegura que, dado un objetivo $G :\leftarrow \{A_1, \ldots, A_m\}$, si en dos pasos consecutivos de resolución se resuelve seleccionando primero A_i y después A_j (con $i \neq j$), se obtiene el mismo objetivo que si primero se selecciona A_j y después A_i. La demostración se basa en la consideración del cuerpo de una cláusula (y, en particular, de un objetivo) como una sucesión o una lista ordenada de átomos. Entonces, si en dos pasos sucesivos del proceso de refutación de $P \cup \{G\}$ se resuelve eligiendo primero A_i y después A_j, se obtiene:



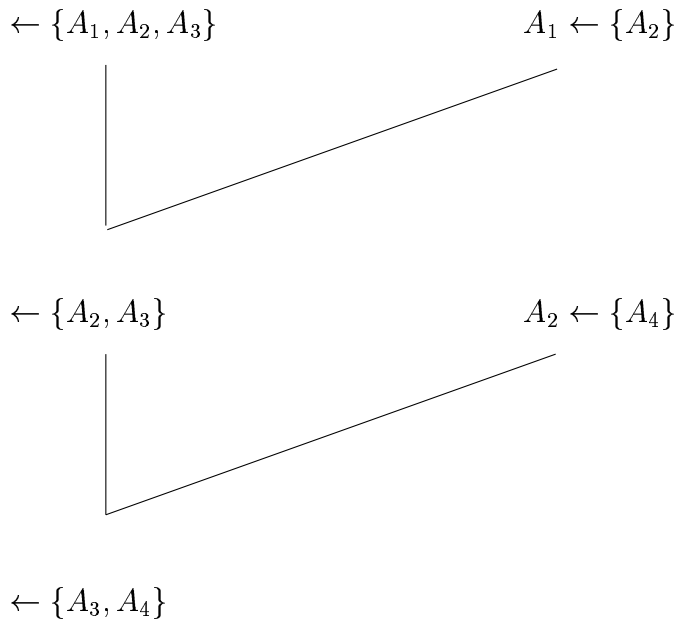
De la misma forma, si se resuelve eligiendo primero A_j y después A_i, se obtiene:



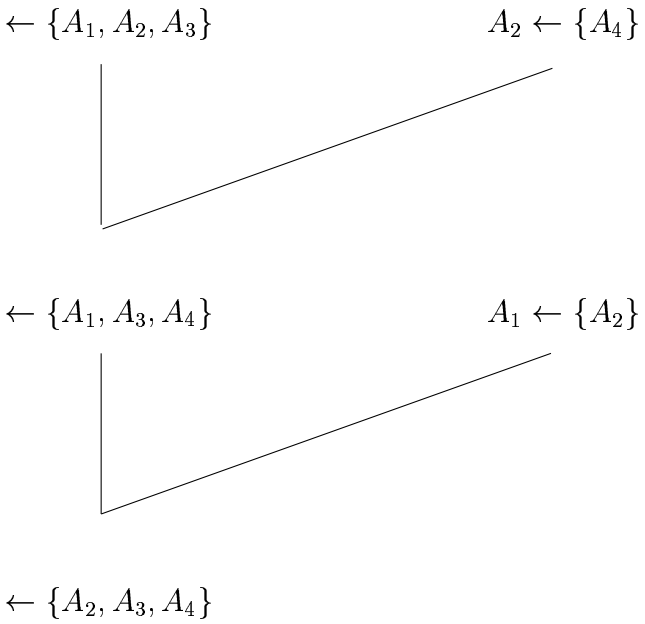
Es evidente que ambos objetivos son el mismo, puesto que lo que se hace es sustituir el átomo que ocupa una **posición** determinada, sin reordenarlos y sin que influya la posible repetición de los átomos.

Ahora bien, esta propiedad no se tiene si representamos las cláusulas mediante conjuntos finitos, y el proceso de sustitución de un átomo del cuerpo del objetivo consiste en eliminar dicho elemento y unir el conjunto resultante con el conjunto que representa al cuerpo de la cláusula. Veamos, para ello, el siguiente ejemplo.

Ejemplo 4.56 Consideremos el objetivo $G :\leftarrow \{A_1, A_2, A_3\}$ y las cláusulas $C_1 : A_1 \leftarrow \{A_2\}$ y $C_2 : A_2 \leftarrow \{A_4\}$. Si resolvemos primero con C_1 y después con C_2 se tiene:



Mientras que si resolvemos primero con C_2 y después con C_1 tenemos:



Es decir, en general:

$$\text{resolvente}(\text{resolvente}(G, C_i), C_j) \neq \text{resolvente}(\text{resolvente}(G, C_j), C_i)$$

puesto que si denotamos por C^- el cuerpo de una cláusula definida C se tiene que:

$$((\text{cuerpo}(G) \setminus \{A_i\}) \cup C_i^-) \setminus \{A_j\}) \cup C_j^- \neq ((\text{cuerpo}(G) \setminus \{A_j\}) \cup C_j^- \setminus \{A_i\}) \cup C_i^-$$

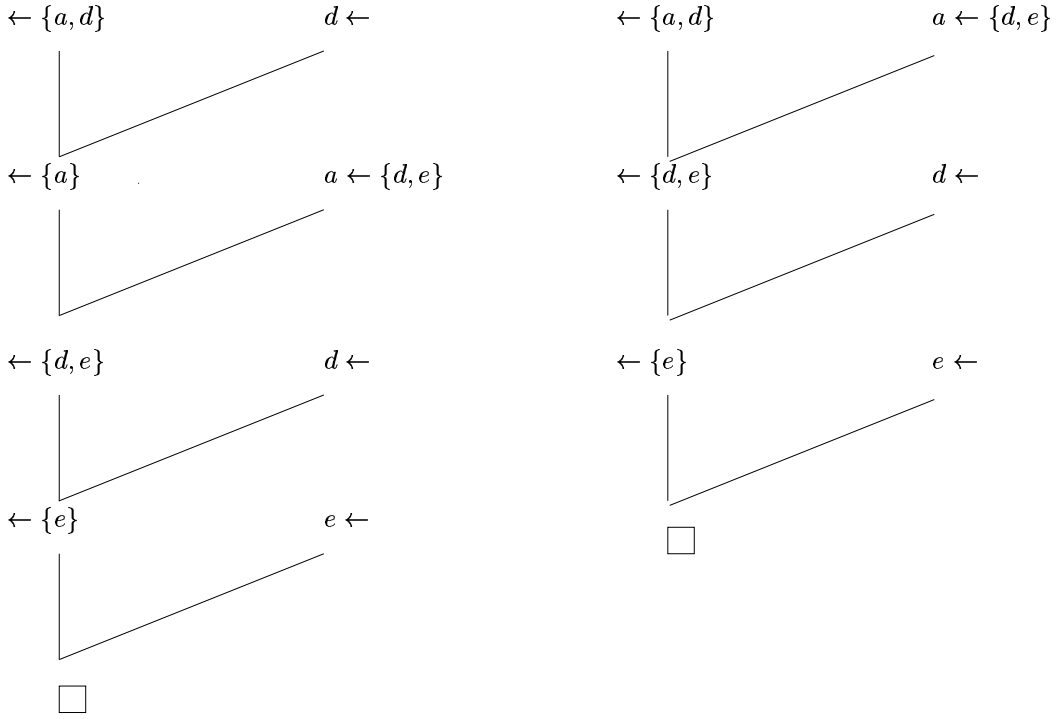
Por otra parte, A. Nerode y R. Shore [57] prueban la completitud fuerte de la resolución SLD por inducción en la longitud de la prueba, usando implícitamente que los pasos de una refutación se pueden cambiar de orden. Ahora bien, aunque en el desarrollo previo de la resolución SLD en [57], se considera que los cuerpos de las cláusulas son conjuntos finitos de átomos, para probar la completitud del proceso subyacente a *Prolog* se decanta por considerarlos como sucesiones y no como conjuntos.

Pues bien, aunque no es cierto el lema del intercambio con esta representación, sí lo es la independencia de la regla de computación, pues la idea subyacente es que no importa el orden en que se resuelvan todos los elementos de un objetivo. Puesto que la definición de la existencia de una refutación se ha hecho de manera recursiva, la idea es probar, por inducción, que si $P \cup \{G\}$ tiene una refutación y f es una regla de computación, entonces $P \cup \{G\}$ tiene una refutación vía f. La cuestión es que no se puede hacer inducción en la longitud de la refutación, pues ésta no es la misma en cualquier refutación; depende del orden en el que se resuelven los átomos.

Ejemplo 4.57 Consideremos el programa

$$P = \{e \leftarrow \{c\}, c \leftarrow, d \leftarrow, e \leftarrow, a \leftarrow \{d, e\}, a \leftarrow \{f\}\}$$

y el objetivo $G :\leftarrow \{a, d\}$. Obsérvese que la longitud de las refutaciones de G dependen del orden en el que se resuelvan los átomos.



Entonces, la cuestión es: dados un programa P, un objetivo G tal que $P \cup \{G\}$ tiene una refutación, y un átomo cualquiera A del cuerpo de G, probar que existe alguna cláusula C en P con la que "resolver" A, de forma que $P \cup \{\text{resolvente}(G, C)\}$ sea "menor" que $P \cup \{G\}$ en algún sentido. Es decir, hemos de definir una función de medida h_P sobre G de forma que si $P \cup \{G\}$ tiene una refutación y $A \in \text{cuerpo}(G)$, exista C en P tal que $\text{cabeza}(C) = A$ y

$$h_P(\text{resolvente}(G, C)) < h_P(G)$$

En [6], Apt y van Emden introducen la noción de árbol SLD para $P \cup \{G\}$, y la de objetivo k–refutable, donde k acota la longitud de las posibles refutaciones de G en todos los árboles SLD cuya raiz es G. Usando estos conceptos, se prueba que si cada átomo A_i del cuerpo de un objetivo G es k_i–refutable, entonces G es $k_1 + \cdots + k_m$–refutable, por inducción en $k_1 + \cdots + k_m$. A partir de este resultado

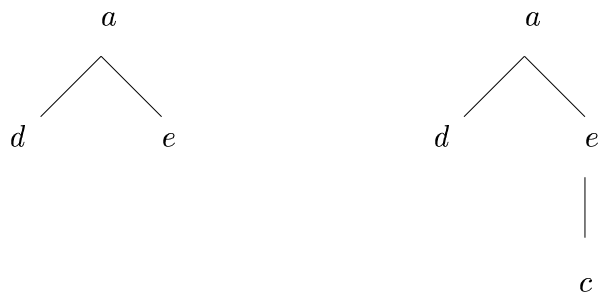
se construye la demostración de la completitud fuerte de la resolución SLD. Hay que hacer notar que, en este caso, los cuerpos de las cláusulas se consideran como sucesiones finitas, por lo que en el razonamiento se puede suponer que, en cada paso de refutación, su longitud disminuye.

Por otra parte, en [89], Stärk introduce la noción de árbol de implicación respecto de un programa y prueba la completidud fuerte de la resolución SLD, usando inducción en el número total de nodos de los árboles de implicación de los átomos del objetivo.

Para nuestra prueba, nos apoyaremos en la noción de árbol de implicación introducida en [89]. Sea P un programa definido y A un átomo, un **árbol de implicación de A respecto de P** es un árbol cuyos nodos son átomos tal que

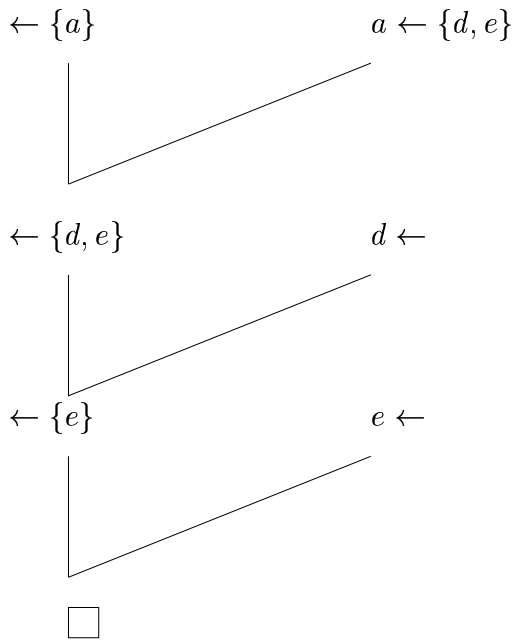
- Su raiz es A.
- Para cada nodo N del árbol, si $B_1, \ldots, B_k$ son sus hijos, existe una cláusula en P de la forma $N \leftarrow \{B_1, \ldots, B_k\}$ (si un nodo N no tiene hijos, entonces el hecho $N \leftarrow$ es una cláusula de P).

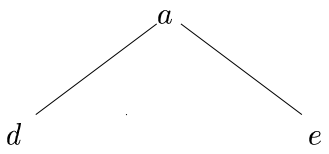
Ejemplo 4.58 Los dos árboles de implicación del átomo a respecto del programa del ejemplo 4.57:



Obsérvese que si A no es consecuencia lógica de P, entonces no tiene árbol de implicación, mientras que si A es consecuencia lógica de P puede tener varios árboles de implicación, con distinto número de nodos. Además, cada uno de ellos está relacionado con una refutación de A. Más aún, la longitud de la refutación depende del número de nodos del árbol de implicación correspondiente, de forma que la refutación de menor longitud se corresponde con el árbol de implicación con menor número de nodos.

Nota 4.59 Existe un paralelismo entre los árboles de implicación del átomo a y sus refutaciones, como se puede apreciar en el siguiente gráfico:





La idea es, pues, definir una función de medida de forma que, si $P \models A$, refleje lo siguiente:

- $h_P(A)$ = número de nodos del árbol de implicación de A con menor número de nodos.
- $h_P(G)$ = suma de las medidas de los átomos del cuerpo de G.

Para definirla, notemos que si $P \cup \{G\}$ es insatisfacible, todos los átomos del cuerpo de G son consecuencias lógicas de P y, por tanto, son elementos del menor punto fijo de P. Es decir, han sido generados en algún paso k de la cadena

$$T_P^0(\emptyset) \subseteq T_P^1(\emptyset) \subseteq \ldots T_P^k(\emptyset) \subseteq T_P^{k+1}(\emptyset) \subseteq \ldots$$

Además, tenemos en cuenta que el número de nodos del menor árbol de implicación está íntimamente relacionado con el menor k en el que el átomo A se introduce en algún conjunto de la cadena. En efecto, si A es consecuencia lógica de P, entonces el árbol de implicación de A con menor número de nodos tiene **un** nodo más que el total de nodos de los menores árboles de implicación de los átomos que se han introducido al eliminar A. Es decir, $h_P(A)$ ha de ser el siguiente del mínimo de las medidas sobre cuerpo(C), donde $C \in P$, la cabeza de C coincide con A, y los elementos del cuerpo de C se han introducido en la cadena en los pasos anteriores a k.

Si A no es consecuencia lógica de P, como no se puede hablar del árbol de implicación de A, le asignaremos un valor alto, del orden de n^n, siendo n el cardinal de la base de Herbrand de P. De esta forma, su medida será mayor que la medida de cualquier átomo que sea consecuencia lógica de P.

Definición 4.60 *Sea P un programa definido, $n = |BH(P)|$ y A un átomo. Definimos el* **menor exponente** *de A con respecto a P como el menor k tal que A pertenece a la potencia k–sima del operador T_P, si A es consecuencia lógica de P, o 0 en caso contrario. Es decir,*

$$m\_exponente_P(A) = \begin{cases} \mu k_{<n}(A \in T_P^k(\emptyset)) & si \quad P \models A \\ 0 & si \quad P \not\models A \end{cases}$$

Nótese que podemos acotar el valor de k pues sabemos que la cadena $T_P^k(\emptyset)$ colapsa en algún paso $m \leq |BH(P)|$.

Entonces, si A es consecuencia lógica de P, su menor exponente con respecto a P es estríctamente positivo y, además, existe alguna cláusula en P de forma que los átomos de su cuerpo han sido introducidos en pasos anteriores; es decir, sus menores exponentes con respecto a P son menores que el de A.

Esto nos permite establecer la siguiente definición recursiva de la función de medida h_P.

Definición 4.61 *Sean P un programa definido, A un átomo y n el cardinal de la base de Herbrand de P. Definimos la función de medida h_P como sigue:*

$$h_P(A) = \begin{cases} n^n & si \quad P \not\models A \\ 1 + min\left\{ \sum_{B \in cuerpo(C)} h_P(B) : C \in \mathcal{CR}_P(A) \right\} & si \quad P \models A \end{cases}$$

donde $\mathcal{CR}_P(A) = \{C : C \in P \wedge cabeza(C) = A \wedge cuerpo(C) \subseteq T_P^{k-1}(\emptyset)\}$, siendo $k = m\_exponente_P(A)$.

Si G es un objetivo definido, se define $h_P(G) = \sum_{B \in cuerpo(G)} h_P(B)$

Veamos cómo definir esta función en PVS. En primer lugar, para especificar en PVS la función que determina el menor exponente de un átomo respecto de un programa, usamos la función del preludio `min`[7]. Previamente, probamos el siguiente lema que justifica que el conjunto sobre el que se aplica la función `min` es no vacío.

[7] La función `min` está definida sobre conjuntos no vacíos de números naturales como `min(S): {a | S(a) AND (FORALL x: S(x) IMPLIES a <= x)}`

Lema 4.62 *Sean P un programa definido, A un átomo y n el cardinal de la base de Herbrand de P. Si $P \models A$, entonces A pertenece a una potencia del operador T_P menor o igual que n.*

136
```
cons_logica_exp: LEMMA
   A ∈ CL(P)) ⇒ ∃ (k: nat): k ≤ card(base_herbrand(P))) &
                            A ∈ iterate(c_i(P),k)(emptyset)
```

Demostración:
La prueba se hace utilizando las propiedades *91*, *104* y *115*.

□

La especificación en PVS de la función menor exponente queda como sigue:

137
```
m_exponente(A,P): nat =
   IF A ∈ CL(P)
   THEN min({k: nat | k ≤ card(base_herbrand(P))) &
                      A ∈ iterate(c_i(P),k)(emptyset))})
    ELSE 0
    ENDIF
```

En el siguiente lema, se establecen y se prueban las propiedades fundamentales de la función `m_exponente`:

Lema 4.63 *Sean P un programa definido, A un átomo y k el menor exponente de A con respecto a P. Entonces:*

- $k \leq |BH(P)|$

  138
  ```
  cota_superior_m_exponente: LEMMA
    m_exponente(A,P) <= card(base_herbrand(P))
  ```

- *Si $P \models A$, entonces $k \geq 1$*

  139
  ```
  cons_logica_m_exponente: LEMMA
    A ∈ CL(P) IMPLIES m_exponente(A,P) >= 1
  ```

- *Si $A \in T_P^m(\emptyset)$, entonces $k \leq m$*

  140
  ```
  cota_m_exponente_c_i: LEMMA
    A ∈ iterate(c_i(P),m)(∅) ⇒ m_exponente(A,P) <= m
  ```

- *Si* $P \models A$, *entonces* $\exists C : C \in P \wedge \texttt{cabeza}(C) = A \wedge \texttt{cuerpo}(C) \subseteq T_P^{k-1}(\emptyset)$

141
```
cons_logica_exp_clausula: LEMMA
   A ∈ CL(P) & m_exponente(A,P) = k
   ⇒
   ∃ C: C ∈ P & cabeza(C) = A &
         cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅)
```

- *Si* $k = 1$, *entonces* $(A \leftarrow) \in P$

142
```
cs_m_exponente_1: LEMMA
    m_exponente(A, P) = 1 ⇒ hecho(A) ∈ P
```

En las pruebas de las propiedades previas se hace uso, fundamentalmente, de las propiedades de la función `min`, además de *105* y *108*.

La cuestión ahora es cómo especificar la función h_P en el lenguaje de PVS, teniendo en cuenta que no es posible realizar una definición mediante recursión cruzada. Para ello, hacemos lo siguiente:

1. Definir una función `medida_a`, respecto de un programa P, sobre conjuntos finitos de átomos, particularizando en dicha definición el caso de un conjunto unitario.

2. A partir de ella, definir una función `medida_g` sobre objetivos definidos.

La definición recursiva de la función `medida_a` es la siguiente:

Definición 4.64 *Sea* P *un programa definido,* F *un conjunto finito de átomos y* n *el cardinal de la base de Herbrand de* P. *Definimos* $\texttt{medida\_a}(F, P)$ *como sigue:*

- 0, *si* $F = \emptyset$
- n^n, *si* $F = \{A\}$ *y* $P \not\models A$
- 1, *si* $F = \{A\}$, $P \models A$ *y* $m\_exponente_P(A) = 1$
- $1 + min\{\texttt{medida\_a}(C^-, P) : C \in \mathcal{CR}_P(A)\}$, *si* $F = \{A\}$, $P \models A$ *y* $m\_exponente_P(A) = k > 1$
- $\texttt{medida\_a}(\{A_1\}, P) + \texttt{medida\_a}(\{A_2, \ldots, A_m\}, P)$, *si* $F = \{A_1, A_2, \ldots, A_m\}$

donde C^- *denota el cuerpo de la cláusula definida* C.

Definición 4.65 *Sea P un programa definido y G un objetivo definido. Definimos la* medida de G respecto de P *como*

$$\texttt{medida\_g}(G, P) = \texttt{medida\_a}(cuerpo(G), P)$$

Nótese que la función `medida_a` es recursiva. Por tanto, para que la correspondiente especificación en PVS sea admitida es necesario probar su terminación, para lo cual también hemos de proporcionarle una función de medida que decrezca en cada llamada recursiva. Si observamos la definición, vemos que hay dos tipos de llamadas recursivas. En las del primer tipo,

$$\texttt{medida\_a}(\{A\}) = 1 + min\{\texttt{medida\_a}(\text{cuerpo}(C), P) : C \in P \wedge cabeza(C) = A \wedge \text{cuerpo}(C) \subseteq T_P^{k-1}(\emptyset)\}$$

con $k = m\_exponente_P(A)$, el factor que decrece estríctamente es el menor exponente de la potencia del operador de consecuencia inmediata en la que se encuentran todos los átomos del conjunto. Y en la del segundo tipo,

$$\texttt{medida\_a}(\{A_1, A_2, \ldots, A_m\}, P) = \texttt{medida\_a}(\{A_1\}, P) + \texttt{medida\_a}(\{A_2, \ldots, A_m\}, P)$$

decrece el cardinal del conjunto.

En consecuencia, hemos de definir una función que mida el menor exponente de un conjunto de átomos, con respecto a un programa. Es decir, de forma que si todos los átomos del conjunto son consecuencias lógicas de P, la función refleje el menor exponente, k, de las potencias del operador de consecuencia, $T_P^k(\emptyset)$, que contienen a dicho conjunto.

Definición 4.66 *Sea P un programa definido y F un conjunto finito de átomos. Definimos la función $med\_exp(F, P)$ como el máximo de los menores exponentes con respecto a P de los elementos de F, si F no es vacío, ó 0 en otro caso. Es decir,*

$$med\_exp(F, P) = \begin{cases} max\{m\_exponente_P(A) : A \in P\}, & si \quad F \neq \emptyset \\ 0, & si \quad F = \emptyset \end{cases}$$

Su especificación en PVS queda de la siguiente forma[8]:

143

```
med_exp(F,P): nat =
    IF empty?(F)
    THEN 0
    ELSE max(image((LAMBDA A: m_exponente(A,P)),F))
    ENDIF
```

[8]En la biblioteca de conjuntos finitos incluida en PVS, se define la función `max` como: `max(SS): {a: T | SS(a) AND (FORALL (x: T): SS(x) IMPLIES x <= a)}` donde `<=` es un orden total sobre `T` y `SS` es un conjunto finito no vacío de elementos de `T`.

En los siguientes lemas, probamos sus propiedades fundamentales, que serán necesarias para probar la terminación de la definición de la función `medida_a`.

Lema 4.67 *(monotonía de la función med_exp): Sean P un programa definido y F, H conjuntos finitos de átomos tales que $F \subseteq H$. Entonces,*

$$med\_exp(F, P) \leq med\_exp(H, P)$$

144
```
med_exp_subconjunto: LEMMA
   F ⊆ H ⇒ med_exp(F,P) ≤ med_exp(H,P)
```

Lema 4.68 *Sea P un programa definido y A un átomo. Entonces,*

$$med\_exp(\{A\}, P) = m\_exponente_P(A)$$

145
```
med_exp_singleton: LEMMA
    med_exp(singleton(A),P) = m_exponente(A,P)
```

Lema 4.69 *Sea P un programa, A un átomo que es consecuencia lógica de P y k su menor exponente con respecto a P. Si C es una cláusula de P, cuya cabeza coincide con A y cuyo cuerpo está contenido en la potencia $(k-1)$–sima de T_P, entonces $med\_exp(cuerpo(C), P) < med\_exp(\{A\}, P)$.*

146
```
med_exp_l1: LEMMA
   A ∈ CL(P) & m_exponente(A,P) = k & C ∈ P & cabeza(C) = A &
   cuerpo(C) ⊆ iterate(c_i(P),k-1)(emptyset)
   ⇒ med_exp(cuerpo(C),P) < med_exp(singleton(A),P)
```

Lema 4.70 *Sean P un programa, F un conjunto finito de átomos, A un átomo de F que es consecuencia lógica de P y k su menor exponente con respecto a P. Si C es una cláusula de P, cuya cabeza coincide con A y cuyo cuerpo está contenido en la potencia $(k-1)$–sima de T_P, entonces*

$$med\_exp(cuerpo(C), P) < med\_exp(F, P)$$

.

147
```
med_exp_l2: LEMMA
   A ∈ CL(P) & m_exponente(A,P) = k & A ∈ F & C ∈ P & cabeza(C) = A &
   cuerpo(C) ⊆ iterate(c_i(P),k-1)(emptyset)
   ⇒ med_exp(cuerpo(C),P) < med_exp(F,P)
```

Estos lemas nos permiten probar las condiciones de terminación generadas por la especificación de la función `medida_a`, usando como función de medida el orden lexicográfico inducido por *med_exp*(F, P) y `card`(F).

148
```
medida_a(F,P): RECURSIVE nat =
    IF empty?(F) THEN 0
    ELSIF singleton?(F)
    THEN LET n=card(base_herbrand(P)), k=m_exponente(choose(F),P) IN
         IF NOT member(choose(F), CL(P))
         THEN  expt(n,n)
         ELSIF k = 1 THEN 1
         ELSE 1 + min({j:nat | ∃ C: C ∈ P & cabeza(C) = choose(F)
                               & cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅)
                               & j = medida_a(cuerpo(C),P)})
         ENDIF
   ELSE medida_a({choose(F)},P) + medida_a(rest(F),P)
   ENDIF
   MEASURE lex2(med_exp(F,P),card(F))
```

Las condiciones de terminación generadas por esta definición se prueban usando los lemas anteriores *146* y *147*.

Ahora, estamos en condiciones de definir la medida de un objetivo respecto de un programa definido como la función `medida_a` aplicada a su cuerpo.

149
```
medida_g(G,P): nat = medida_a(cuerpo(G), P)
```

Una vez definida la función `medida_g`, retomamos el problema de cómo probar la independencia de la regla de computación y, como consecuencia, la completitud fuerte de la resolución SLD. Veamos un esquema de la prueba que vamos a realizar:

(1) En primer lugar, probamos que si $P \cup \{G\}$ es insatisfacible, entonces para todo átomo A del cuerpo de G, existe una cláusula $C \in P$, tal que

- cabeza$(C) = A$
- `medida_g`(resolvente$(G, C), P) <$ `medida_g`(G, P)
- $P \cup \{\text{resolvente}(G, C)\}$ es insatisfacible

(2) Usando el resultado anterior y los teoremas de completitud débil y de adecuación de la resolución SLD, probamos que si $P \cup \{G\}$ tiene una refutación, entonces para todo átomo A del cuerpo de G, existe una cláusula $C \in P$, tal que

- cabeza$(C) = A$
- `medida_g`(resolvente$(G, C), P) <$ `medida_g`(G, P)
- $P \cup \{\text{resolvente}(G, C)\}$ tiene una refutación

(3) A continuación, probamos la independencia de la regla de computación, por inducción en la medida inducida por `medida_g`, usando el resultado (2).

(4) Por último, probamos la completitud fuerte, usando el teorema de completitud débil y la independencia de la regla de computación.

La demostración del primer resultado la hacemos eligiendo C de la siguiente forma: dado A tal que $P \models A$, consideramos las cláusulas de P cuyas cabezas coinciden con A y tales que los átomos que constituyen sus cuerpos han sido introducidos en una potencia del operador de consecuencia inmediata anterior a la menor potencia en la que se ha introducido A; de entre éstas, elegimos aquella que proporciona el menor valor de la función de medida `medida_a`(cuerpo$(C), P)$.

Para poder especificar esta elección en PVS definimos el conjunto que hemos denotado por $\mathcal{CR}_P(A)$, es decir, el formado por aquellas cláusulas de P que, en un proceso de resolución pueden servir para resolver A, decreciendo el menor exponente de los elementos que se introducen (los del cuerpo de la cláusula).

150
```
conj_clausulas_resolv(A,P): finite_set[clausula_def[T]] =
  LET k = m_exponente(A,P) IN
      IF k >= 1
         THEN {C | C ∈ P & cabeza(C) = A &
                   cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅))}
         ELSE emptyset
      ENDIF
```

Establecemos las propiedades de esta especificación, que nos van a permitir formalizar dicha elección.

Lema 4.71 *Si A es consecuencia lógica de P, el conjunto $\mathcal{CR}_P(A)$ no es vacío.*

151
```
conj_clausulas_resolv_cl: LEMMA
   A ∈ CL(P)) ⇒ nonempty?(conj_clausulas_resolv(A,P))
```

Corolario 4.72 *Si A es consecuencia lógica de P, el conjunto de los números naturales formado por las medidas de los cuerpos de las cláusulas de $\mathcal{CR}_P(A)$ es no vacío.*

152
```
conj_clausulas_resolv_med: COROLLARY
  A ∈ CL(P)
  ⇒ nonempty?(image((lambda(C):medida_a(cuerpo(C),P)),
                    conj_clausulas_resolv(A,P)))
```

Lema 4.73 *Sea A un átomo tal que $P \models A$ y sea k el menor exponente de A con respecto a P. Entonces, existe $C \in P$ de forma que cabeza$(C) = A$, cuerpo$(C) \subseteq T_P^{k-1}(\emptyset)$ y proporciona el menor valor para* medida_a$(cuerpo(C), P)$.

153
```
medida_a_unitario_cl: LEMMA
  A ∈ CL(P)) & m_exponente(A,P) = k ⇒
  ∃ C:C ∈ P & cabeza(C) = A & cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅) &
      medida_a(cuerpo(C),P) =
      min({j:nat| ∃ C: C ∈ P & cabeza(C) = A &
                       cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅) &
                       j = medida_a(cuerpo(C),P)})
```

Corolario 4.74 *Sea A un átomo tal que $P \models A$ y sea $k > 1$ el menor exponente de A con respecto a P. Entonces, existe $C \in P$ tal que cabeza$(C) = A$, cuerpo$(C) \subseteq T_P^{k-1}(\emptyset)$ y* medida_a$(\{A\}, P) >$ medida_a$(cuerpo(C), P)$.

154
```
medida_a_unitario_cl_mayor: COROLLARY
  A ∈ CL(P) & m_exponente(A,P) = k & k > 1 ⇒
  ∃ C: C ∈ P & cabeza(C) = A & cuerpo(C) ⊆ iterate(c_i(P),k-1)(∅) &
       medida_a({A},P) > medida_a(cuerpo(C),P)
```

Tengamos en cuenta también que en la prueba del resultado (1) hemos de utilizar algunas propiedades de la función medida_a. En primer lugar, observemos que la idea era que medida_a sobre un conjunto finito de átomos fuera la suma de medida_a sobre cada uno de sus elementos. En este sentido, la especificación en PVS contempla que

$$\texttt{medida\_a}(F) = \texttt{medida\_a}(\{\texttt{choose}(F)\}, P) + \texttt{medida\_a}(\texttt{rest}(F), P)$$

que parece, en principio, una característica específica de un elemento, choose(F). Veamos que, en realidad, la igualdad se tiene para cualquier elemento de F.

Lema 4.75 *Sea P un programa y F un conjunto finito de consecuencias lógicas de P. Entonces, para todo $A \in F$,*

$$\texttt{medida\_a}(F, P) = \texttt{medida\_a}(\{A\}, P) + \texttt{medida\_a}(F \setminus \{A\}, P)$$

155
```
medida_a_descomp: LEMMA
  F ⊆ CL(P) & A ∈ F
  ⇒ medida_a({A},P) + medida_a(remove(A,F),P) = medida_a(F,P)
```

Demostración:

Por inducción en F, siguiendo el esquema de inducción fuerte en el cardinal, `finite_set_induction_gen`, descrito en [3]:
Se consideran los siguientes casos:

1. $F = \emptyset$. Trivial.
2. F es unitario. Trivial, pues $F \setminus \{A\} = \emptyset$ y $\texttt{medida\_a}(\emptyset) = 0$.
3. F no es unitario.

   Por definición de `medida_a`,

   $$\texttt{medida\_a}(F, P) = \texttt{medida\_a}(\{\texttt{choose}(F)\}, P) + \texttt{medida\_a}(\texttt{rest}(F), P)$$

   Entonces:

   - Si $A = \texttt{choose}(F)$, ya está probado.
   - Si $A \neq \texttt{choose}(F)$, entonces $A \in \texttt{rest}(F)$. Luego, por hipótesis de inducción aplicada a $\texttt{rest}(F)$ y A, tenemos (*)

     $$\texttt{medida\_a}(\texttt{rest}(F), P) = \texttt{medida\_a}(\{A\}, P) + \texttt{medida\_a}(\texttt{rest}(F) \setminus \{A\})$$

     Ahora bien, por la definición de `rest` y el lema `remove_commutativo`[9] se tiene

     $$\texttt{rest}(F) \setminus \{A\} = (F \setminus \{\texttt{choose}(F)\}) \setminus \{A\} = (F \setminus \{A\}) \setminus \{choose(F)\}$$

     Aplicándo de nuevo la hipótesis de inducción a $F \setminus \{A\}$ y $\texttt{choose}(F)$ se tiene (**)

     $$\begin{aligned}\texttt{medida\_a}(F \setminus \{A\}, P) = {} & \texttt{medida\_a}(\{\texttt{choose}(F)\}, P) + \\ & \texttt{medida\_a}((F \setminus \{A\}) \setminus \{\texttt{choose}(F)\})\end{aligned}$$

     Entonces, usando (*) y (**), se tiene:

     $$\texttt{medida\_a}(\{A\}, P) + \texttt{medida\_a}(F \setminus \{A\}, P) = \texttt{medida\_a}(F, P)$$

□

En segundo lugar, probamos la siguente propiedad relativa a la medida de la unión de dos conjuntos.

[9] Esta propiedad se encuentra entre las propiedades de la teoría de conjunto que se han incluido en el apéndice A (páginas 238–241).

Lema 4.76 *Sea P un programa definido y sean F y H dos conjuntos finitos de consecuencias lógicas de P. Entonces,*

$$\texttt{medida\_a}(F \cup H, P) \leq \texttt{medida\_a}(F, P) + \texttt{medida\_a}(H, P)$$

.

156
```
medida_a_union: LEMMA
  F ⊆ CL(P)) & H ⊆ CL(P)
  ⇒ medida_a(union(F,H),P) <= medida_a(F,P) + medida_a(H,P)
```

Demostración:
La prueba se hace por inducción en $\texttt{card}(F) + \texttt{card}(H)$, usando *155*.

□

Estamos ya en condiciones de enunciar y probar los resultados de independencia de la regla de computación y de completitud fuerte.

Teorema 4.77 *Sean P un programa y G un objetivo definido. Si $P \cup \{G\}$ es insatisfacible, para todo átomo A del cuerpo de G, entonces existe una cláusula $C \in P$, tal que su cabeza coincide con A, $P \cup \{resolvente(G, C)\}$ es insatisfacible y $\texttt{medida\_g}(resolvente(G, C), P) < \texttt{medida\_g}(G, P)$.*

157
```
insatisfacible_medida_g: THEOREM
  es_insatisfacible(add(G,P)) ⇒
  ∀ A: A ∈ cuerpo(G) ⇒
       ∃ C: C ∈ P & cabeza(C) = A &
            medida_g(resolvente(G,C),P) < medida_g(G,P) &
            es_insatisfacible(add(resolvente(G,C),P))
```

Demostración:
Si $P \cup \{G\}$ es insatisfacible, todos los átomos del cuerpo de G son consecuencias lógicas de P. Sea A un átomo del cuerpo de G y sea $r = m\_exponente(A, P)$. Por *139*, $r \geq 1$. Consideramos dos casos:

Caso 1: $r > 1$

Por *154*, sea C una cláusula de P tal que $\text{cabeza}(C) = A$, $\text{cuerpo}(C) \subseteq T_P^{r-1}(\emptyset)$ y $\texttt{medida\_a}(\text{cuerpo}(C), P) < \texttt{medida\_a}(\{A\}, P)$. Si tomamos esta cláusula, tenemos:

Por una parte:

$$\begin{array}{rll}
 & \texttt{medida_g}(\text{resolvente}(G, G)) & \\
= & \texttt{medida_a}((\text{cuerpo}(G) \setminus \{A\}) \cup \text{cuerpo}(C)) & \boxed{118} \\
\leq & \texttt{medida_a}(\text{cuerpo}(G) \setminus \{A\}) + \texttt{medida_a}(\text{cuerpo}(C)) & \boxed{156} \\
< & \texttt{medida_a}(\text{cuerpo}(G) \setminus \{A\}) + \texttt{medida_a}(\{A\}, P) & \\
= & \texttt{medida_g}(G, P) & \boxed{155}
\end{array}$$

Y por otra, veamos que $P \cup \{\text{resolvente}(G, C)\}$ es insatisfacible. Para ello, por 67, es suficiente probar que todo átomo del cuerpo de G es consecuencia lógica de P. En efecto, basta considerar que cuerpo(resolvente$(G, C)) =$ (cuerpo$(G) \setminus \{A\}) \cup$cuerpo$(C)$ y que, tanto los elementos de cuerpo$(G) \setminus \{A\}$ como los de cuerpo(C) son consecuencias lógicas de P.

Caso 2: $r = 1$

En este caso, por 142, se tiene que el hecho $A \leftarrow$ es una cláusula de P. Tomando esta cláusula, por un razonamiento análogo al caso anterior, se verifica que medida_g(resolvente$(G, A \leftarrow)) <$ medida_g(G, P) y que $P \cup \{\text{resolvente}(G, C)\}$ es insatisfacible.

□

Usando 157, 122 y 132 se obtiene el siguiente resultado.

Teorema 4.78 *Sean P un programa y G un objetivo definido. Si $P \cup \{G\}$ tiene una refutación, para todo átomo A del cuerpo de G, existe una cláusula $C \in P$, tal que su cabeza coincide con A, $P \cup \{resolvente(G, C)\}$ tiene una refutación y* medida_g$(resolvente(G, C), P) <$ medida_g(G, P).

158
```
tiene_refutacion_medida_g: THEOREM
  tiene_refutacion(P,G) ⇒
  ∀ A: A ∈ cuerpo(G) ⇒
       ∃ C: C ∈ P & cabeza(C) = A &
            medida_g(resolvente(G,C),P) < medida_g(G,P) &
            tiene_refutacion(P, resolvente(G,C))
```

Teorema 4.79 *(Independencia de la regla de computación).*
Sean P un programa definido, G un objetivo definido y f una regla de computación. Si $P \cup \{G\}$ tiene una refutación, entonces $P \cup \{G\}$ tiene una refutación vía f.

159
```
sld_independiente_regla_comp: THEOREM
   tiene_refutacion(P,G) ⇒ tiene_refutacion_via(P,G,f)
```

Demostración:
Sea f una regla de computación. Probamos el resultado, por inducción en medida_g(G, P). Es decir, suponemos que se tiene el resultado para todo P', G', tales que medida_g$(G', P') <$ medida_g(G, P) y lo probamos para P, G.

En efecto, supongamos que $P \cup \{G\}$ tiene una refutación. Por 158, para todo $A \in$ cuerpo(G) existe $C \in P$ tal que

- cabeza$(C) = A$

- `medida_g`(resolvente$(G, C), P) <$ `medida_g`(G, P)
- $P \cup \{\text{resolvente}(G, C)\}$ tiene una refutacion.

En particular se tiene para $f(G)$, que pertenece al cuerpo de G por ser f regla de computación. Así, sea C una tal cláusula. Aplicando la hipótesis de inducción a P y a resolvente(G, C), se tiene que $P \cup \{\text{resolvente}(G, C)\}$ tiene una refutación vía f. Luego, $P \cup \{G\}$ tiene una refutación vía f.

□

Teorema 4.80 *(Completitud fuerte de la resolución SLD).*
Sean P un programa, G un objetivo y f una regla de computación. Si $P \cup \{G\}$ es insatisfacible, entonces $P \cup \{G\}$ tiene una refutación vía f.

160
```
sld_completitud_via: THEOREM
    es_insatisfacible(add(G,P)) ⇒ tiene_refutacion_via(P,G,f)
```

Demostración:
Por aplicación directa de *132* y *159*.

□

Capítulo 5

Un modelo del análisis formal de conceptos en PVS

En este capítulo se presenta una formalización en PVS de la teoría del análisis formal de conceptos. En la primera sección se introducen las nociones de contexto formal y de concepto en un contexto, probándose que el conjunto de conceptos de un contexto formal tiene estructura de retículo completo.

En la segunda sección se especifica un algoritmo para obtener el conjunto de los conceptos de un contexto formal finito, y se prueba su corrección.

En la tercera, se introduce la noción de implicación entre atributos y su semántica. El principal resultado de la sección establece la igualdad entre los modelos de las implicaciones válidas en un contexto y las intenciones de los conceptos de dicho contexto.

Por último, se establece la noción de base de implicaciones entre atributos como un conjunto de implicaciones adecuado (cada implicación es válida), completo (todas las implicaciones válidas son consencuencia de la base) y no redundante (ninguna de sus implicaciones es consecuencia de las restantes). Se define la base de Duquenne-Guigues, se demuestra que es una base de implicaciones y se especifica un algoritmo para calcularla, probándose su correción.

5.1. Introducción

En las últimas décadas, el *descubrimiento de conocimiento en bases de datos* (KDD[1]) se ha convertido en un tópico importante, tanto en el campo de la investigación como en el de las aplicaciones industriales. Su objetivo principal es la extracción no trivial de conocimiento válido, potencialmente útil y comprensible, a partir de grandes bases de datos.

El Análisis Formal de Conceptos [2][33], introducido en la década de los 80 como

[1]Del inglés *Knowledge Discovery in Databases.*

[2]En inglés FCA: *Formal Concept Analysis*

una abstracción matemática de la noción de "concepto", se ha convertido en los últimos años en una teoría potente para el análisis de datos y el descubrimiento de conocimiento [92, 90].

Parte de la noción de contexto formal, que consiste en un conjunto de objetos O, un conjunto de atributos A, y una relación entre ambos, especificando los atributos que posee cada objeto. Y centra su atención en la generación de los conceptos subyacentes al contexto, así como a la obtención de "reglas", que describan las relaciones entre los atributos de dicho contexto.

La minería de datos se usa en la parte computacional del KDD, especialmente en lo concerniente al descubrimiento de reglas asociación y de patrones frecuentes en un conjunto de datos. Los métodos del FCA se han usado en la minería de datos, lo que permite mejorar el tiempo en la extracción de reglas de asociación minimales y no redundantes, en datos con gran correlacción. Además, mediante el retículo de los conceptos, se mejora la visualización de las estructuras conceptuales, con objeto de encontrar patrones, regularidades, excepciones, ect.

Entre los algoritmos de obtención de reglas de asociación que hacen uso del FCA se encuentran A–Close [65], PASCAL [9], Closet [68] y TITANIC [91]. El sistema TOSCANA [97], de uso comercial, permite analizar sistemas conceptuales de datos mediante aplicación de técnicas del FCA. El sistema CONEXP, que proporciona un entorno para el análisis y la exploración de los conceptos de un contexto, lo hemos usado para explorar algunos ejemplos y obtener gráficamente el retículo de los conceptos de un contexto, que mostramos en este capítulo.

La variedad de dominios a los que se ha aplicado el FCA va desde medicina, psicología y ciencias sociales hasta ciencias de la información o ingeniería civil (véase `www.upris.org.uk/fca/fca.html`). Como ejemplos, enumeramos algunas de sus aplicaciones:

- Clasificación y procesamiento de correos electrónicos [19].
- Descubrimiento de conocimiento en textos médicos [18].
- Técnicas de aprendizaje [28].
- Análisis de herencias en una jerarquía de clases [7].

En este capítulo formalizamos en PVS los fundamentos y algunos algoritmos de la teoría del FCA, con la idea de que sirva de base para la obtención de implementaciones de los algoritmos de esta teoría, verificados formalmente. Para ello, seguiremos el desarrollo del libro de Ganter y Wille "Formal Concept Analysis" [33]. Como puede observarse en dicho texto, las nociones básicas del FCA están basadas en el uso de conjuntos. Por ejemplo, el entorno en el que se desarrolla la teoría está formado por un *conjunto* de objetos y un *conjunto* de atributos, junto con información acerca de los atributos que posee cada objeto. Entonces, a la hora de especificar estas nociones en PVS, podemos usar el tipo `set[T]`, que representa a los conjuntos con elementos en `T`, o el tipo `finite_set[T]` si

sólo queremos considerar conjuntos finitos. De esta forma, tanto las especificaciones en PVS como el razonamiento con ellas, estarán muy próximas a la teoría matemática original.

Además, también queremos construir especificaciones de algunos algoritmos (por ejemplo, uno para obtener los conceptos de un contexto formal, o bien otro para obtener una base de reglas del contexto) y que dichas especificaciones sean evaluables. Pero si el tipo que usamos para construirlas es `set[T]` o `finite_set[T]`, las especificaciones no serán evaluables. Ahora bien, en el capítulo 6 vamos a desarrollar un marco que posibilita la construcción de especificaciones evaluables, usando listas, a partir de especificaciones que usan conjuntos finitos. Por ello, en este capítulo, describiremos una formalización de la teoría del FCA, basada en el uso conjuntos finitos. Y, en el capítulo 8, construiremos especificaciones evaluables de los algoritmos descritos en éste.

5.2. El retículo de los conceptos

En esta sección se introducen los conceptos de contexto formal y concepto formal, probándose que el conjunto de conceptos de un contexto, junto con la relación de subconcepto tiene estructura de retículo completo. La formalización en PVS se encuentra desarrollada en la teoría `contextos_formales` (páginas 371–378).

El marco en el que se establecen los conceptos se conoce como **contexto formal**: una terna (O, A, I), donde O es un conjunto de **objetos**, A es un conjunto de **atributos** e I es una **relación** entre los elementos de O y los de A, es decir, un subconjunto de $O \times A$. Si $(d, a) \in I$, decimos que "el objeto d **tiene** el atributo a".

Ejemplo 5.1 Ilustramos esta definición con un ejemplo relativo a relaciones binarias. Los objetos son seis conjuntos de pares, y los atributos cinco propiedades: reflexiva, simétrica, transitiva, total y de equivalencia. Representamos el contexto mediante una **tabla**, en la que se marcan los objetos que poseen un determinado atributo.

	Reflexiva	**S**imétrica	**Tr**ansitiva	**To**tal	**E**quivalencia
S_1	**x**	**x**	**x**		**x**
S_2		**x**		**x**	
S_3		**x**	**x**	**x**	
S_4			**x**	**x**	
S_5	**x**		**x**	**x**	
S_6	**x**	**x**		**x**	

Para la formalización en PVS, consideramos dos tipos no interpretados, `T1 T2`, que denotan preobjetos y preatributos, respectivamente, y usamos una estructura

con tres campos para representar un tipo generalizado sobre el cual definiremos el tipo que representa a los contextos formales finitos [3]:

```
FFCT: TYPE =  [# obj:      finite_set[T1],
                 atrib:    non_empty_finite_set[T2],
                 relacion: finite_set[[T1, T2]] #]
```
161

Expresamos el tipo de los contextos formales finitos como un subtipo de FFCT:

```
FFC:TYPE = {C:FFCT | LET Ob = obj(C), A = atrib(C), I = relacion(C)
                     IN ∀ par∈I: LET (ob, a) = par IN ob∈Ob ∧ a∈A}
```
162

En todo el capítulo, $C = (O, A, I)$ denotará un contexto formal finito. Usaremos las variable d, $d1$, $d2$ para representar preobjetos y a, $a1$, $a2$ para preatributos.

```
C:       VAR FFC
d,d1,d2: VAR T1
a,a1,a2: VAR T2
```

5.2.1. Operaciones básicas en contextos formales: derivación

Dado un contexto formal finito $C = (O, A, I)$, se definen los operadores de derivación asociados entre los conjuntos potencia $\mathcal{P}(O)$ y $\mathcal{P}(A)$, y se prueba que constituyen una conexión de Galois entre O y A. En PVS definiremos los operadores de derivación sobre los tipos más generales, `finite_set[T1]` y `finite_set[T2]` y, posteriormemte, los restringiremos a los tipos determinados por los respectivos conjuntos potencia.

Definición 5.2 *Dado un conjunto de preobjetos X, la* **intención** *de X con respecto a C, denotada por X', se define como:*

$$X' = \begin{cases} A, & si \quad X = \emptyset \\ \{a \in T_2 : (\forall d \in X) \quad dIa\}, & si \quad X \neq \emptyset \end{cases}$$

```
intencion(C)(X): finite_set[T2] =
   IF empty?(X)
      THEN atrib(C)
      ELSE {a: T2 | ∀ (d:(X)): relacion(C)(d, a)}
   ENDIF
```
163

[3]Para que el contexto no sea vacío, supondremos que, al menos, el conjunto de atributos es no vacío.

Es decir, si X es un conjunto de objetos de C, la intención de X respecto de C es el conjunto de atributos de C que poseen todos los objetos de X.

Definición 5.3 *Dado un conjunto de preatributos Y, la* **extensión** *de Y con respecto a C, denotada por Y', se define como*

$$Y' = \begin{cases} O, & si \quad Y = \emptyset \\ \{d \in T_1 : (\forall a \in Y) \quad dIa\}, & si \quad Y \neq \emptyset \end{cases}$$

```
164
extension(C)(Y: finite_set[T2]): finite_set[T1] =
   IF empty?(Y)
      THEN obj(C)
      ELSE {d: T1 | ∀ (a:(Y)): relacion(C)(d, a)}
   ENDIF
```

Dualmente, si Y es un conjunto de atributos de C, la extensión de Y es el conjunto de objetos de C con todos loss atributos de Y.

En el contexto del ejemplo 5.1, se tiene

- $\{S_1, S_2\}' = \{S\}$, $\{S_6\}' = \{R, S, To\}$
- $\{S\}' = \{S_1, S_2, S_3, S_6\}$, $\{Tr, To, E\}' = \emptyset$

Se prueba que el rango del operador `intencion(C)` está contenido en $\mathcal{P}(A)$.

```
165
intencion_subset_atrib: LEMMA  intencion(C)(X) ⊆ atrib(C)
```

Por tanto, podemos definir el operador restringido a los correspondientes conjuntos potencia, como sigue:

```
intencion_r(C)(X: (powerset(obj(C)))): (powerset(atrib(C))) =
   intencion(C)(X)
```

Análogamente, se prueba que el rango del operador `extension(C)` está contenido en $\mathcal{P}(O)$, lo que permite restringirlo a los conjuntos potencia:

```
166
extension_subset_obj: LEMMA  extension(C)(Y) ⊆ obj(C)

extension_r(C)(Y: (powerset(atrib(C)))): (powerset(obj(C))) =
  extension(C)(Y)
```

A partir de los operadores previos, definimos los operadores de clausura, de forma que si X es un conjunto de objetos de C, la clausura de X es el conjunto de objetos de C con todos los atributos que poseen los elementos de X. Análogamente, si Y es un conjunto de atributos de C, la clausura de Y es el conjunto de atributos de C que poseen los objetos con los atributos de Y.

La especificación de estos operadores en PVS la hacemos sobre conjuntos de preobjetos y de preatributos, como sigue:

```
clausura_o(C)(X): finite_set[T1] = extension(C)(intencion(C)(X))
clausura_a(C)(Y): finite_set[T2] = intencion(C)(extension(C)(Y))
```
167

Y, a partir de ellos, los operadores de clausura restringidos a conjuntos de objetos y de atributos de C.

Definición 5.4 *Dado un conjunto de objetos X, la* **clausura** *de X con respecto a C, denotada por X'', es el conjunto de objetos de C que poseen todos los atributos de X'. Análogamente, dado un conjunto de atributos Y, la* **clausura** *de Y con respecto a C, denotada por Y'', es el conjunto de atributos de C que poseen todos los objetos de X'.*

```
clausura_o_r(C)(X: (powerset(obj(C)))): (powerset(obj(C))) =
  clausura_o(C)(X)

clausura_a_r(C)(Y: (powerset(atrib(C)))): (powerset(atrib(C))) =
  clausura_a(C)(Y)
```
168

En el ejemplo 5.1, $\{Tr, To, E\}'' = \{R, S, Tr, To, E\}$ y $\{S_1, S_2\}'' = \{S_1, S_2, S_3, S_6\}$.

5.2.2. Propiedades de los operadores de derivación

En esta subsección establecemos las propiedades de los operadores definidos, que garantizan que `intencion_r(C)` y `extension_r(C)` constituyen una conexión de Galois entre O y A.

Lema 5.5 *Dados X_1, X_2, conjuntos finitos de preobjetos, se verifica*

$$X_1 \subseteq X_2 \Longrightarrow X_2' \subseteq X_1'$$

```
intencion_subset: LEMMA
  X1 ⊆ X2 ⇒ intencion(C)(X2) ⊆ intencion(C)(X1)
```
169

Lema 5.6 *Dados Y_1, Y_2, conjuntos finitos de preatributos, se verifica*

$$Y_1 \subseteq Y_2 \Longrightarrow Y_2' \subseteq Y_1'$$

```
extension_subset: LEMMA
  Y1 ⊆ Y2 ⇒ extension(C)(Y2) ⊆ extension(C)(Y1)
```
170

Lema 5.7 *Sea C un contexto formal y X un conjunto de objetos de C. Entonces,*

$$X \subseteq X'' \quad y \quad X''' = X'$$

171
```
subset_clausura_o: LEMMA
  X ⊆ obj(C) ⇒ X ⊆ clausura_o(C)(X)

int_clausura_o: LEMMA
  X ⊆ obj(C) ⇒ intencion(C)(clausura_o(C)(X)) = intencion(C)(X)
```

Demostración:

Por doble inclusión, usando esencialmente las propiedades de la teoría de conjuntos, incluida en el preludio de PVS.

□

Lema 5.8 *Sea C un contexto formal e Y un conjunto de atributos de C. Entonces,*

$$Y \subseteq Y'' \quad e \quad Y''' = Y'$$

172
```
subset_clausura_a: LEMMA
  Y ⊆ atrib(C) ⇒ Y ⊆ clausura_a(C)(Y)

ext_clausura_a: LEMMA
  Y ⊆ atrib(C) ⇒ extension(C)(clausura_a(C)(Y)) = extension(C)(Y)
```

Lema 5.9 *Sean X un conjunto de objetos de C e Y un conjunto de atributos de C. Son equivalentes:*

1. $X \subseteq Y'$
2. $Y \subseteq X'$
3. $X \times Y \subseteq I$

173
```
equivalente_subset_1: LEMMA
  X ⊆ obj(C) & Y ⊆ atrib(C) ⇒
  (X ⊆ extension(C)(Y) ⟺ Y ⊆ intencion(C)(X))

equivalente_subset_2: LEMMA
  X ⊆ obj(C) & Y ⊆ atrib(C)
  ⇒ (X ⊆ extension(C)(Y) ⟺ ∀ (d:(X), a:(Y)): relacion(C)(d,a))
```

Demostración:

Probamos (1) $\Longleftrightarrow$ (2) y (1) $\Longleftrightarrow$ (3), a partir de las definiciones.

□

5.2.3. Conceptos en un contexto formal

La noción de concepto relativo a un contexto formal sintetiza la idea de un conjunto de objetos, caracterizados por el conjunto de atributos que poseen. Dado un contexto formal C, se considera que un concepto que se puede "extraer" de C está formado por un conjunto de objetos de C, junto con el conjunto de atributos que los caracterizan. La definición formal de esta noción de concepto es la siguiente.

Definición 5.10 *Un* **concepto formal** *en un contexto formal $C = (O, A, I)$ es un par (X, Y) donde $X \subseteq O$, $Y \subseteq A$, $X' = Y$ e $Y' = X$. Decimos que X e Y son la extensión y la intención, respectivamente, del concepto (X, Y).*

174
```
es_concepto_c?(C)((X,Y)): bool =
  X⊆obj(C)) & Y⊆atrib(C)) & intencion(C)(X)=Y & extension(C)(Y)=X

extension_concepto(C)(par:concepto_c(C)): finite_set[T1] = PROJ_1(par)
intencion_concepto(C)(par:concepto_c(C)): finite_set[T2] = PROJ_2(par)
```

En todo contexto, siempre podemos extraer al menos un concepto, el formado por los objetos que poseen todos los atributos de C, como muestra el siguiente resultado.

Lema 5.11 *El conjunto de los conceptos de un contexto $C = (O, A, I)$ es no vacío, pues (A', A'') es un concepto.*

175
```
concepto_c_no_vacio_2: LEMMA
  es_concepto_c?(C)(extension(C)(atrib(C)), clausura_a(C)(atrib(C)))

concepto_c_no_vacio_corol: COROLLARY ∃ (x: concepto_c(C)): TRUE
```

Una vez probado que en todo contexto C existe algún concepto, establecemos el tipo de los conceptos sobre un contexto C como:

176
```
concepto_c(C): NONEMPTY_TYPE = (es_concepto_c?(C))
```

Entre los conceptos de un contexto formal C se puede establecer una relación natural, entendiendo que un concepto contiene o es "mayor" que otro cuando contiene a los objetos de éste.

Definición 5.12 *Sea C un contexto. Decimos que el concepto (X_1, Y_1) es* **subconcepto** *de (X_2, Y_2) si $X_1 \subseteq X_2$ (o, lo que es equivalente, $Y_2 \subseteq Y_1$).*

177
```
subconcepto_c?(C)(par1, par2: concepto_c(C)): bool =
  PROJ_1(par1) ⊆ PROJ_1(par2)

cond_equiv_subconcepto_c: LEMMA
  ∀ (par1, par2: concepto_c(C)):
  subconcepto_c?(C)(par1, par2) ⟺ PROJ_2(par2) ⊆ PROJ_2(par1)
```

Demostración:

La prueba de la condición equivalente de subconcepto se realiza a partir de las definiciones, usando los lemas *169* y *170*.

□

Notación 5.13 Denotamos por $\mathcal{B}(O, A, I)$ el conjunto de los conceptos de un contexto formal $C = (O, A, I)$ y por $\leq$ la relación de subconcepto definida en $\mathcal{B}(O, A, I)$, siempre que no se preste a confusión.

Una característica importante del conjunto de conceptos de un contexto es que forman un retículo completo. Probemos que, en efecto, $\mathcal{B}(O, A, I)$, junto con la relación $\leq$ tiene estructura de retículo completo. Para ello, probamos en primer lugar, que $\leq$ define un orden parcial en $\mathcal{B}(O, A, I)$.

Lema 5.14 *La relación $\leq$ es reflexiva y transitiva. Es decir, $(\mathcal{B}(O, A, I), \leq)$ es un preorden.*

178
```
subconcepto_c_es_reflexiva: LEMMA
  reflexive?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_transitiva: LEMMA
  transitive?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_preorden: LEMMA
  preorder?[concepto_c(C)](subconcepto_c?(C))
```

Demostración:

Las pruebas se realizan directamente, usando las propiedades correspondientes de la teoría de conjuntos, incluidas en el preludio de PVS.

□

Lema 5.15 *La relación $\leq$ es antisimétrica. Es decir, $(\mathcal{B}(O, A, I), \leq)$ es un orden parcial.*

```
179
subconcepto_c_es_antisimetrica: LEMMA
  antisymmetric?[concepto_c(C)](subconcepto_c?(C))
```

Demostración:

La demostración se realiza aplicando extensionalidad, y usando tanto la definición de `subconcepto_c?` como la condición equivalente.

□

Por tanto, se tiene que $\leq$ es un orden parcial en $\mathcal{B}(O, A, I)$.

```
180
subconcepto_c_es_orden_parcial: LEMMA
  partial_order?[concepto_c(C)](subconcepto_c?(C))

JUDGEMENT subconcepto_c?(C) HAS_TYPE (partial_order?[concepto_c(C)])
```

Así, estamos en condiciones de enunciar el resultado que garantiza que todo conjunto de conceptos de un contexto formal C, $\mathcal{S}$, posee supremo e ínfimo. La prueba se va a hacer construyendo, efectivamente, el supremo y el ínfimo de $\mathcal{S}$. Como estamos tratando con contextos formales finitos, el conjunto de posibles conceptos es también finito. Así pues, sólo consideraremos conjuntos finitos de conceptos. Además, en el capítulo 8 construiremos especificaciones evaluables de las funciones que calculan el supremo y el ínfimo de un conjunto finito de conceptos. Por ello, a la hora de especificar la unión o la intersección de conjuntos finitos usaremos las funciones `union_f` e `intersection_f`, que definen la unión y la intersección, respectivamente, de un conjunto finito de conjuntos finitos [4].

En primer lugar, consideramos el conjunto de intenciones (respec. extensiones) de un conjunto de conjuntos de preobjetos (respec. preatributos). Las especificaciones en PVS de las funciones que obtienen dichos conjuntos, son las siguientes:

```
181
conj_intenciones_c(C)(F): finite_set[finite_set[T2]] =
   image(intencion(C))(F)

conj_extensiones_c(C)(G): finite_set[finite_set[T1]] =
   image(extension(C))(G)
```

La propiedad fundamental es que la intersección arbitraria de extensiones (resp. intenciones) es una intención (resp. extensión). Aunque estas propiedades son ciertas para conjuntos cualesquiera, las enunciamos en PVS para conjuntos finitos, pues estamos considerando contextos formales finitos. En cualquier caso, en la demostración no se hace uso de esta característica.

[4] Ver 282 y 284

Lema 5.16 *Sea $\mathcal{F}$ una familia no vacía de conjuntos de preobjetos de un contexto formal C. Se tiene que*

$$(\bigcup \mathcal{F})' = \bigcap \{X' : X \in \mathcal{F}\}$$

182
```
intencion_union_gen: LEMMA
  nonempty?(F) ⇒
  intencion(C)(union_f(F)) = intersection_f(conj_intenciones_c(C)(F))
```

Demostración:
Por doble inclusión, usando las definiciones y propiedades de la teoría de conjuntos.

□

De igual forma, se tiene la propiedad análoga para extensiones.

Lema 5.17 *Sea $\mathcal{G}$ una familia no vacía de conjuntos de preatributos de un contexto formal C. Se tiene que*

$$(\bigcup \mathcal{G})' = \bigcap \{Y' : Y \in \mathcal{G}\}$$

183
```
extension_union_gen: LEMMA
  nonempty?(G) ⇒
  extension(C)(union_f(G)) = intersection_f(conj_extensiones_c(C)(G))
```

Para probar que el conjunto de conceptos de un contexto formal C es un retículo completo, vamos a construir, explícitamente, el supremo y el ínfimo de un conjunto de conceptos $\mathcal{S}$. Para ello, necesitamos acceder tanto a las intenciones como a las extensiones de los elementos de $\mathcal{S}$, para lo cual definimos las siguientes funciones:

184
```
intenciones_conj_conceptos(C)(S: finite_set[concepto_c(C)]):
                              finite_set[finite_set[T2]] =
    image(intencion_concepto(C))(S)

extensiones_conj_conceptos(C)(S: finite_set[concepto_c(C)]):
                              finite_set[finite_set[T1]]=
    image(extension_concepto(C))(S)
```

Usándolas, especificamos las funciones que calculan el supremo y el ínfimo como sigue:

$$\sup(\{(X_k, Y_k) : k \in K\}) = \Big((\bigcup_{k\in K} X_k)'', \bigcap_{k\in K} Y_k\Big)$$

$$\inf(\{(X_k, Y_k) : k \in K\}) = \Big(\bigcap_{k \in K} X_k, (\bigcup_{k \in K} Y_k)'' \Big)$$

185
```
supremo(C)(S:non_empty_finite_set[concepto_c(C)]): concepto_c(C) =
  (clausura_o(C)(union_f(extensiones_conj_conceptos(C)(S))),
   intersection_f(intenciones_conj_conceptos(C)(S)))

infimo(C)(S:non_empty_finite_set[concepto_c(C)]): concepto_c(C) =
  (intersection_f(extensiones_conj_conceptos(C)(S)),
   clausura_a(C)(union_f(intenciones_conj_conceptos(C)(S))))
```

Obsérvese que, en las definiciones de ínfimo y supremo se ha declarado que el rango de ambas funciones es `concepto_c(C)`. Por tanto, para su admisión, se generan las siguientes condiciones de subtipo que tienen que ser probadas.

186
```
supremo_TCC1: OBLIGATION
  ∀ (C: contexto_formal, S: set[concepto_c(C)]):
   es_concepto_c?(C)
     (clausura_o(C)(union_f(extensiones_conj_conceptos(C)(S)))),
      intersection_f(intenciones_conj_conceptos(C)(S))

infimo_TCC1: OBLIGATION
  ∀ (C: contexto_formal, S: set[concepto_c(C)]):
   es_concepto_c?(C)
     (intersection_f(extensiones_conj_conceptos(C)(S)),
      clausura_a(C)(union_f(intenciones_conj_conceptos(C)(S))))
```

Por ello, antes de definir las funciones `infimo` y `supremo`, se establece el siguiente lema.

Lema 5.18 *Dado el conjunto de conceptos* $\mathcal{S} = \{(X_k, Y_k) : k \in K\}$, *se tiene que*

$$\{X_k : k \in K\} = \texttt{conj\_extensiones\_c}(\{Y_k : k \in K\})$$

$$\{Y_k : k \in K\} = \texttt{conj\_intenciones\_c}(\{X_k : k \in K\})$$

187
```
conj_ext_conceptos_es_conj_extensiones: LEMMA
  ∀ (S:finite_set[concepto_c(C)]):
  extensiones_conj_conceptos(C)(S) =
  conj_extensiones_c(C)(intenciones_conj_conceptos(C)(S))

conj_int_conceptos_es_conj_intenciones: LEMMA
  ∀ (S:finite_set[concepto_c(C)]):
  intenciones_conj_conceptos(C)(S) =
  conj_intenciones_c(C)(extensiones_conj_conceptos(C)(S))
```

Demostración:

Para demostrar los lemas previos es suficiente probar las siguientes igualdades entre funciones:

```
intencion(C) o extension_concepto(C) = intencion_concepto(C)
extension(C) o intencion_concepto(C) = extension_concepto(C)
```

y usar la propiedad que garantiza que dadas dos funciones f y g y un conjunto $\mathcal{S}$, $(g \circ f)(\mathcal{S}) = g(f(\mathcal{S}))$.

□

Una vez admitidas las definiciones probamos que, en efecto, obtienen el ínfimo y el supremo, respectivamente, de $\mathcal{S}$. Para especificar estos resultados en PVS usamos las funciones del preludio `greatest_lower_bound?` y `least_upper_bound?`. [5]

Lema 5.19 *Para todo conjunto finito no vacío de conceptos de C, $\mathcal{S}$,* `infimo`$(\mathcal{S})$ *es la mayor de las cotas inferiores de $\mathcal{S}$.*

188
```
infimo_es_infimo_p: THEOREM
  ∀ (S:non_empty_finite_set[concepto_c(C)]):
     greatest_lower_bound?(subconcepto_c?(C))(infimo(C)(S), S)
```

Este lema genera la siguiente condición, que se prueba usando el lema *175*

189
```
infimo_es_infimo_c_TCC1: OBLIGATION
  ∀ (C: contexto_formal): ∃ (x: concepto_c(C)): TRUE
```

Lema 5.20 *Para todo conjunto finito no vacío de conceptos de C, $\mathcal{S}$,* `supremo`$(\mathcal{S})$ *es la menor de las cotas superiores de $\mathcal{S}$.*

190
```
supremo_es_supremo_p: THEOREM
  ∀ (S:non_empty_finite_set[concepto_c(C)]):
  least_upper_bound?(subconcepto_c?(C))(supremo(C)(S), S)
```

Demostración:

Las demostraciones se realizan a partir de las definiciones, usando la condición equivalente de `subconcepto_c?`.

□

Así, hemos probado en PVS el siguiente teorema:

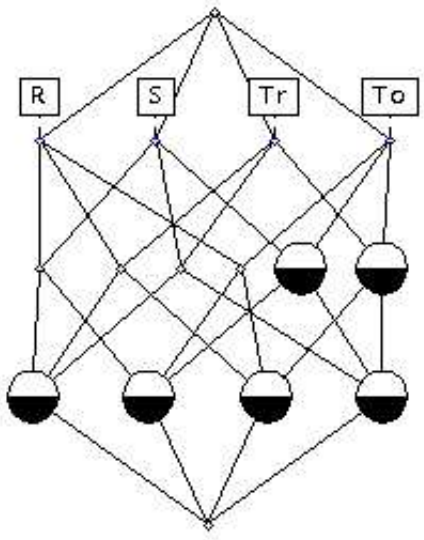
[5] La función `greatest_lower_bound?(<)(x,S)` comprueba que `x` es la mayor de las cotas inferiores de `S`. Análogamente, `greatest_lower_bound?(<)(x,S)` comprueba que `x` es la menor de las cotas superiores de `S`.

Teorema 5.21 $(\mathcal{B}(O, A, I), \leq)$ *es un retículo completo.*

En el ejemplo 5.1, el conjunto de los conceptos del contexto es:

$\{(\{S_1, S_2, S_3, S_4, S_5, S_6\}, \emptyset)$, $(\{S_1, S_5, S_6\}, \{R\})$, $(\{S_1, S_2, S_3, S_6\}, \{S\})$,
$(\{S_1, S_6\}, \{R, S\})$, $(\{S_1, S_3, S_4, S_5\}, \{Tr\})$, $(\{S_1, S_5\}, \{R, Tr\})$,
$(\{S_1, S_3\}, \{S, Tr\})$, $(\{S_2, S_3, S_4, S_5, S_6\}, \{To\})$, $(\{S_5, S_6\}, \{R, To\})$,
$(\{S_2, S_3, S_6\}, \{S, To\})$, $(\{S_6\}, \{R, S, To\})$, $(\{S_3, S_4, S_5\}, \{Tr, To\})$,
$(\{S_5\}, \{R, Tr, To\})$, $(\{S_3\}, \{S, Tr, To\})$, $(\{S_1\}, \{R, S, Tr, E\})$,
$(\emptyset, \{R, S, Tr, To, E\})\}$

Y el retículo correspondiente es el siguiente:



donde están marcados con una etiqueta aquellos nodos que represeentan conceptos cuya intención está formada por un único atributo.

5.3. Algoritmo de generación de los conceptos

En esta sección mostramos una especificación de un algoritmo para generar el conjunto de los conceptos de un contexto formal finito, y probamos su corrección. En primer lugar, establecemos las siguientes propiedades, a partir de las cuales se pueden especificar algoritmos duales para generar los conceptos de un contexto formal:

(1) Todo concepto del contexto $C = (O, A, I)$ es de la forma (X'', X') para algún

conjunto de objetos X. Y, recíprocamente, todo par de la forma (X'', X') donde X es un conjunto de objetos de C, es un concepto de C.

(2) Todo concepto del contexto $C = (O, A, I)$ es de la forma (Y', Y'') para algún conjunto de atributos Y. Y, recíprocamente, todo par de la forma (Y', Y'') donde Y es un conjunto de atributos de C, es un concepto de C.

(3) Para todo conjunto de objetos X, $X' = \bigcap_{d \in X} \{d\}'$

(4) Para todo conjunto de atributos Y, $Y' = \bigcap_{a \in Y} \{a\}'$

Como puede obsevarse, estas propiedades dan lugar a dos posibles algoritmos, uno basado en las propiedades (1) y (3), y otro dual usando las propiedades (2) y (4). Como es más usual que en un contexto el número de atributos sea considerablemente menor que el número de objetos, vamos a describir el algoritmo que usa las propiedades (2) y (4), que consiste en:

1. Generar el conjunto de todas las posibles extensiones de los conceptos, usando (4).

2. Para cada elemento del conjunto generado en el paso anterior, obtener su intención y formar el concepto correspondiente. De esta forma, se tendría el conjunto de todos los conceptos del contexto.

La formalización en PVS de la sección se encuentra desarrollada en la teoría `conceptos_contexto` (páginas 379–383).

Lema 5.22 *Para todo conjunto de objetos X, (X'', X') es un concepto de C.*

191
```
forma_de_concepto_f_obj_1_cl: LEMMA
  ∀ (X: finite_set[T1]):
  X ⊆ obj(C) ⇒ es_concepto_c?(C)(clausura_o(C)(X), intencion(C)(X))
```

Demostración:
Es suficiente aplicar la definición de subconcepto y el lema *171*.

□

Lema 5.23 *Todo concepto del contexto C es de la forma (X'', X') para algún conjunto de objetos X.*

192
```
forma_de_concepto_f_obj_2_cl: LEMMA
  ∀ (par: concepto_c(C)): ∃ (X: finite_set[T1]):
  X ⊆ obj(C) & par = ((clausura_o(C)(X)), intencion(C)(X))
```

Demostración:
Dado un concepto de C, basta tomar X como la extensión de dicho concepto.
□

Entonces, si denotamos por `CONCEPTOS(C)` al conjunto de conceptos del contexto formal C:

193
```
CONCEPTOS(C): set[[finite_set[T1], finite_set[T2]]] =
  {par: [finite_set[T1], finite_set[T2]]| es_concepto_c?(C)(par)}
```

se tiene que `CONCEPTOS(C)` $= \{(X'', X') : X \subseteq O\}$.

```
forma_de_conceptos_obj: LEMMA
  CONCEPTOS(C) =
  {par: [finite_set[T1], finite_set[T2]]| ∃ (X: finite_set[T1]):
        X ⊆ obj(C) & par=((clausura_o(C)(X)),intencion(C)(X))}
```

Lema 5.24 *Para todo conjunto de atributos Y, (Y', Y'') es un concepto de C.*

194
```
forma_de_concepto_f_atrib_1_cl: LEMMA
  ∀ (Y: finite_set[T2]):
  Y⊆atrib(C) ⇒ es_concepto_c?(C)((extension(C)(Y),clausura_a(C)(Y)))
```

Demostración:
Basta aplicar el lema *172*.
□

Lema 5.25 *Todo concepto del contexto C es de la forma (Y', Y'') para algún conjunto de atributos Y.*

195
```
forma_de_concepto_f_atrib_2_cl: LEMMA
  ∀ (par: concepto_c(C)): ∃ (Y: finite_set[T2]):
  Y ⊆ atrib(C) & par = (extension(C)(Y), clausura_a(C)(Y))
```

Como consecuencia, se tiene que `CONCEPTOS(C)` $= \{(Y', Y'') : Y \subseteq A\}$.

```
forma_de_conceptos_atrib: LEMMA
  CONCEPTOS(C) =
  {par: [finite_set[T1], finite_set[T2]]| ∃ (Y: finite_set[T2]):
        Y ⊆ atrib(C) & par=(extension(C)(Y), clausura_a(C)(Y))}
```

Finalmente, nótese que las propiedades (3) y (4) a las que hemos hecho referencia en la introducción de la sección son casos particulares de los resultados *182* y *183*.

Así pues, el algoritmo para generar los conceptos de un contexto formal finito $C = (O, A, R)$ que especificamos es el siguiente:

1. Generar el conjunto $\mathcal{S}$ de todas las extensiones del conjunto de atributos de C, $A = \{a_1, ..., a_n\}$, de forma recursiva. Es decir, $\mathcal{S} = \{Y' : Y \subseteq A\}$.

2. Para cada elemento X del conjunto generado en el apartado anterior, obtener su intención y formar el concepto correspondiente, obteniéndose el conjunto $\{(X, X') : X \in \mathcal{S}\} = \{(Y', Y'') : Y \subseteq A\} =$ `CONCEPTOS(C)`.

   La construcción de $\mathcal{S}$ la hacemos en pasos sucesivos, como sigue:

   - Paso 0: generar el conjunto de extensiones de $\emptyset$, $\mathcal{S}_0 = \{O\}$ pues $\emptyset' = O$.
   - Paso $k+1$: generar el conjunto de extensiones de $\{a_1, \ldots, a_{k+1}\}$, denotado por $\mathcal{S}_{k+1}$, a partir del conjunto de extensiones de $\{a_1, \ldots, a_k\}$, denotado por $\mathcal{S}_k$, usando (4):

$$\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{D \cap \{a_{k+1}\}' : D \in \mathcal{S}_k\}$$

La especificación de este algoritmo en PVS queda como sigue:

196
```
genera_conceptos(C): set[[finite_set[T1], finite_set[T2]]] =
  LET fun = LAMBDA(X: finite_set[T1]): (X, intencion(C)(X)) IN
      image(fun)(genera_extensiones(C))
```

donde la función `genera_extensiones` construye el conjunto formado por las extensiones de los conceptos de C, de la forma siguiente:

197
```
genera_extensiones(C): finite_set[finite_set[T1]] =
  genera_extensiones_aux(C)(atrib(C), {obj(C)})

genera_extensiones_aux(C)(Y: finite_set[T2],
                          S: finite_set[finite_set[T1]]):
                          RECURSIVE finite_set[finite_set[T1]] =
  IF empty?(Y)
  THEN S
  ELSE genera_extensiones_aux(C)(rest(Y),
                                 agrega_extension_elto(C)(choose(Y),S))
  ENDIF
  MEASURE card(Y)
```

Como auxiliar, hemos usado la función `agrega_extension_elto` que, dados a y $\mathcal{S}$, construye el conjunto $\mathcal{S} \cup \{D \cap \{a\}' : D \in \mathcal{S}\}$, cuya especificación en PVS es:

198
```
agrega_extension_elto(C)(a: T2, S: finite_set[finite_set[T1]]):
                         finite_set[finite_set[T1]] =
  LET fun = LAMBDA(D: finite_set[T1]): D ∩ extension(C)({a})) IN
      union(S, image(fun)(S))
```

Vamos a probar que el algoritmo de generación de los conceptos de C es correcto. Es decir, que los elementos del conjunto `genera_conceptos(C)` son, justamente, los conceptos de C. Como hemos denotado por `CONCEPTOS(C)` al conjunto de conceptos del contexto formal C, el teorema de corrección del algoritmo `genera_conceptos` en PVS es el siguiente:

199
```
correccion_genera_conceptos_c: THEOREM
  genera_conceptos(C) = CONCEPTOS(C)
```

La demostración de este teorema requiere algunos lemas previos que prueben, esencialmente, la corrección de la función `genera_extensiones`. Es decir, hemos de probar que, en efecto, `genera_extensiones(C)` es el conjunto $\{Y' : Y \subseteq A\}$.

En primer lugar, definimos un predicado que comprueba que todos los elementos de un conjunto determinado son extensiones de conjuntos de atributos de C, como sigue:

200
```
es_conj_extensiones_relativas_f?(C)(S: finite_set[finite_set[T1]])
                                  : bool =
  ∀ (X:(S)): ∃ (Y:finite_set[T2]): Y ⊆ atrib(C) & X = extension(C)(Y)
```

Las propiedades destacables de cada una de las funciones que intervienen en la especificación de la función `genera_conceptos` y que se usan para probar su corrección, son las siguientes:

Lema 5.26 *Sea a un atributo de C y $\mathcal{S}$ un conjunto finito de extensiones de atributos de C. Entonces,* `agrega_extension_elto(C)(a,S)` *también es un conjunto de extensiones de C.*

201
```
agrega_extension_elto_correcto_relativo: LEMMA
  ∀ (a: T2, S: finite_set[finite_set[T1]]):
  es_conj_extensiones_relativas_f?(C)(S) & atrib(C)(a) ⇒
  es_conj_extensiones_relativas_f?(C)(agrega_extension_elto(C)(a, S))
```

Lema 5.27 *Sea Y un conjunto finito de atributos de C y $\mathcal{S}$ un conjunto finito cuyos elementos son extensiones de conjuntos de atributos de C. Entonces, el conjunto* `genera_extensiones_aux(C)(Y,S)` *es un conjunto de extensiones de conjuntos de atributos de C.*

202
```
genera_extensiones_aux_correcto_relativo: LEMMA
  ∀ (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
   es_conj_extensiones_relativas_f?(C)(S) & Y ⊆ atrib(C) ⇒
   es_conj_extensiones_relativas_f?(C)(genera_extensiones_aux(C)(Y,S))
```

Demostración:

Por inducción en Y, según el esquema *2*, descrito en el capítulo 2.

□

Con este resultado, se prueba de forma inmediata el siguiente lema.

Lema 5.28 *El conjunto* `genera_extensiones(C)` *es un conjunto de extensiones de conjuntos de atributos de C.*

203
```
genera_extensiones_correcto_relativo: LEMMA
  es_conj_extensiones_relativas_f?(C)(genera_extensiones(C))
```

De esta forma, estamos en condiciones de probar una de las inclusiones del teorema de corrección de la función `genera_conceptos`.

Teorema 5.29 *Todo elemento del conjunto* `genera_conceptos(C)` *es un concepto de C.*

204
```
genera_conceptos_correcto_b: LEMMA
  ∀ (par:[finite_set[T1], finite_set[T2]]):
    par ∈ genera_conceptos(C) ⇒ es_concepto_c?(C)(par)
```

Demostración:

Sea *par* un elemento de `genera_conceptos(C)`. Por la defincón de la función, $par = (X, X')$ siendo X un elemento del conjunto `genera_extensiones(C)`. Por *203*, X es la extensión de un conjunto de atributos de C, es decir, $\exists Y \subseteq atrib(C)$ tal que $X = Y'$. Luego, $par = (Y', Y'')$, que es un concepto de C, por *194*.

□

Para probar la otra inclusión, establecemos previamente los resultados necesarios relativos a las funciones auxiliares.

Lema 5.30 *Sean Y un conjunto finito de preatributos y $\mathcal{S}$ un conjunto finito cuyos elementos son conjuntos finitos de preatributos. Entonces, S es subconjunto de* `genera_extensiones_aux(C)(Y,S)`.

```
genera_extensiones_aux_completo_l1:LEMMA                                  205
  ∀ (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
  S ⊆ genera_extensiones_aux(C)(Y,S))
```

Lema 5.31 *Sea Y un conjunto finito de preatributos y $\mathcal{S}$ un conjunto finito de conjuntos finitos de preatributos. Si $D \in S$ es un conjunto finito de objetos de C y $Z \subseteq Y$, entonces $D \cap Z' \in$*`genera_extensiones_aux(C)(Y,S)`.

```
genera_extensiones_aux_completo_l2:LEMMA                                  206
  ∀ (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
   ∀ (Z: finite_set[T2], D: finite_set[T1]):
   Z ⊆ Y & D ⊆ obj(C) & D ∈ S
   ⇒ D ∩ extension(C)(Z) ∈ genera_extensiones_aux(C)(Y, S)
```

Demostración:

Por inducción en Y, según el esquema [2], descrito en el capítulo 2:

- $Y = \emptyset$, trivial
- $Y \neq \emptyset$

  Por definición,

  $\texttt{genera\_extensiones\_aux}(C)(Y, \mathcal{S}) =$
  $\texttt{genera\_extensiones\_aux}(C)(\texttt{rest}(Y),$
  $\texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}))$

  Sabemos que

  $$\forall D \in \mathcal{S},\ D \cap \{\texttt{choose}(Y)\}' \in \texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S})$$

  Por hipótesis de inducción

  $$\forall D_1 \in \texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}),\ \forall Z_1 \subseteq \texttt{rest}(Y)$$

  $D_1 \cap Z_1'$ es un elemento de

  $\texttt{genera\_extensiones\_aux}(C)(\texttt{rest}(Y),$
  $\texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}))$

  Entonces, dados $D \in \mathcal{S}$ y $Z \subseteq Y$, consideramos dos casos

Caso 1: $\texttt{choose}(Y) \notin Z$

Entonces, $Z \subseteq \texttt{rest}(Y)$. Por otra parte, si $D \in \mathcal{S}$,
$D \in \texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}))$
Luego, por h. i., $D \cap Z' \in \texttt{genera\_extensiones\_aux}(C)(Y, \mathcal{S})$

Caso 2: $\texttt{choose}(Y) \in Z$

Sabemos que existe $Z_1 \subseteq \texttt{rest}(Y)$, tal que $Z = Z_1 \cup \{\texttt{choose}(Y)\}$.
Luego, $Z' = Z_1' \cap \{\texttt{choose}(Y)\}'$
Sea $D \in \mathcal{S}$. Entonces,

$$D \cap Z' = D \cap (Z_1' \cap \{\texttt{choose}(Y)\}') = (D \cap \{\texttt{choose}(Y)\}') \cap Z_1'$$

Por tanto,

$$D \cap \{\texttt{choose}(Y)\}' \in \texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}))$$

Luego, por h.i. $(D \cap \{\texttt{choose}(Y)\}') \cap Z_1'$ es un elemento de

$$\texttt{genera\_extensiones\_aux}(C)(\texttt{rest}(Y), \texttt{agrega\_extension\_elto}(\texttt{choose}(Y), \mathcal{S}))$$

Es decir $D \cap Z' \in \texttt{genera\_extensiones\_aux}(C)(Y, \mathcal{S})$.

□

Con este lema, se prueban fácilmente los siguientes resultados, con lo que se concluye la prueba de la corrección de la función genera_conceptos.

Lema 5.32 *Si Y es un conjunto finito de atributos de C, entonces Y' es un elemento del conjunto* genera_extensiones(C).

207
```
genera_extensiones_completo: LEMMA
  ∀(Y: finite_set[T2]):
   Y ⊆ atrib(C) ⇒ extension(C)(Y) ∈ genera_extensiones(C)
```

Teorema 5.33 *El conjunto* genera_conceptos(C) *contiene a todos los conceptos de C.*

208
```
genera_conceptos_completo: THEOREM
  ∀(par: concepto_c(C)): par ∈ genera_conceptos(C)
```

5.4. Implicaciones entre atributos

A veces, hay que clasificar una gran cantidad de objetos con respecto a un número, relativamente pequeño, de atributos. En estos casos, interesa disponer de reglas que expresen las relaciones entre los atributos del contexto, en el sentido siguiente: "si un objeto posee los atributos $a_1, \ldots, a_n$, también posee los atributos $b_1, \ldots, b_m$". Esta reglas se denominan *implicaciones entre atributos*. En esta

sección se introduce la noción de implicación entre atributos y su semántica. El principal teorema establece la igualdad entre los modelos de las implicaciones válidas en un contexto y las intenciones de los conceptos de dicho contexto. Con ello, se pone de manifiesto la relación entre los conceptos de un contexto y las implicaciones entre atributos de dicho contexto. El contenido de la sección se encuentra desarrollado en la teoría `implicaciones_atrib` (páginas 383–386).

Definición 5.34 *Una* **implicación entre atributos** *en un contexto formal finito C es un par de conjuntos de atributos (Y_1, Y_2). Notación: $Y_1 \to Y_2$.*

En PVS, declaramos en primer lugar, el tipo `implicacion_gen` para representar implicaciones entre preatributos y, para cada contexto C, establecemos el subtipo de las implicaciones entre atributos de C, como sigue:

209
```
implicacion_gen: TYPE = [# antecedente: finite_set[T2],
                           consecuente: finite_set[T2] #]

implicacion(C): TYPE =
  {imp: implicacion_gen | antecedente(imp) ⊆ atrib(C) &
                          consecuente(imp) ⊆ atrib(C) }
```

Describimos a continuación las nociones semánticas sobre las implicaciones de un contexto formal C.

Definición 5.35 *Sea $Y_1 \to Y_2$ una implicación entre atributos del contexto C y Z un conjunto finito de preatributos. Decimos que Z* **respeta** *o* **es modelo** *de $Y_1 \to Y_2$ si $Y_1 \not\subseteq Z$ ó $Y_2 \subseteq Z$.*

En el contexto del ejemplo 5.1, el conjunto $\{R, S\}$ es modelo de la implicación $\{R\} \to \{S\}$.

210
```
respeta_imp?(C)(Z: finite_set[T2], imp: implicacion(C)): bool =
  antecedente(imp) ⊈ Z ∨ consecuente(imp) ⊆ Z
```

Lema 5.36 *Sea $Y_1 \to Y_2$ una implicación entre atributos del contexto C y Z un conjunto finito de preatributos. Entonces, Z respeta $Y_1 \to Y_2$ si y sólo si $Y_1 \subseteq Z \Rightarrow Y_2 \subseteq Z$.*

211
```
respeta_imp_equiv: LEMMA
 ∀ (Z: finite_set[T2], imp: implicacion(C)):
  respeta_imp?(C)(Z, imp) ⇔
  antecedente(imp) ⊆ Z ⇒ consecuente(imp) ⊆ Z)
```

Definición 5.37 *Decimos que una implicación entre atributos de un contexto* $C = (O, A, I)$, $Y_1 \to Y_2$, *es* **válida en** C *si para todo objeto* $d \in O$, $\{d\}'$ *respeta la implicación. Notación:* $C \models Y_1 \to Y_2$

212
```
es_valida?(C)(imp: implicacion(C)): bool =
  ∀ (d: T1): d ∈ obj(C) ⇒ respeta_imp?(C)(intencion(C)({d}), imp)
```

En el ejemplo 5.1, la implicación $\{R, S, Tr\} \to \{E\}$ es válida en el contexto, pues $\{E\} \subseteq \{R, S, Tr\}' = \{R, S, Tr, E\}$.

Las implicaciones válidas en un contexto formal C se pueden caracterizar en función del operador de clausura, mediante el siguiente resultado.

Lema 5.38 $Y_1 \to Y_2$ *es válida en* C *si y sólo si* $Y_2 \subseteq Y_1''$

213
```
valida_prop_1_cl: LEMMA
  ∀ (imp: implicacion(C)):
   es_valida?(C)(imp) ⇔
   consecuente(imp) ⊆ clausura_a(C)(antecedente(imp))
```

Otras caracterizaciones son las siguientes:

Lema 5.39 *Una implicación del contexto* C, $Y_1 \to Y_2$, *es válida en* C *si y sólo si* $Y_1 \to \{a\}$ *es válida en* C, *para todo* $a \in Y_2$.

214
```
valida_prop_2: LEMMA
  ∀ (imp: implicacion(C)):
  es_valida?(C)(imp)
  ⇔
  (∀(a:T2): a ∈ consecuente(imp) ⇒
                es_valida?(C)((# antecedente:= antecedente(imp),
                                consecuente:= singleton(a) #)))
```

Lema 5.40 *(Comprobación en el retículo de conceptos) Una implicación de* C, $Y_1 \to Y_2$ *es válida en* C *si y sólo si* $(Y_1', Y_1'') \leq (\{a\}', \{a\}'')$, *para todo* $a \in Y_2$.

215
```
valid_prop_3_cl: LEMMA
  ∀ (imp: implicacion(C)):
   es_valida?(C)(imp) ⇔
    (∀(a:T2): a ∈ consecuente(imp) ⇒
     subconcepto_c?(C)((extension(C)(antecedente(imp)),
                         clausura_a(C)(antecedente(imp))),
                        (extension(C)(singleton(a)),
                         clausura_a(C)(singleton(a)))))
```

Definición 5.41 *Sea $\mathcal{L}$ un conjunto de implicaciones del contexto C*

- *Un conjunto finito de preatributos Z* **respeta o es modelo** *de $\mathcal{L}$ si Z respeta todas las implicaciones de $\mathcal{L}$.*

```
respeta_imp?(C)(Z: finite_set[T2], L:set[implicacion(C)]): bool =
  ∀ (imp: (L)): respeta_imp?(C)(Z, imp)
```
216

- *Una implicación de C, $Y_1 \to Y_2$,* **es consecuencia** *de $\mathcal{L}$ si todo conjunto de atributos Z que respete $\mathcal{L}$, también respeta $Y_1 \to Y_2$. Notación: $\mathcal{L} \models Y_1 \to Y_2$*

```
es_consecuencia(C)(imp:implicacion(C), L:set[implicacion(C)])
                  :bool =
  ∀ (Y: finite_set[T2]):
  Y ⊆ atrib(C) & respeta_imp?(C)(Y,L) ⇒ respeta_imp?(C)(Y,imp)
```
217

En el contexto del ejemplo 5.1, la implicación $\{R\} \to \{R, S\}$ es consecuencia del conjunto $\{\{R\} \to \{S\}\}$.

Finalizamos la sección con el resultado que relaciona los modelos de las implicaciones válidas en un contexto C con las intenciones de los conceptos de dicho contexto. Para ello, denotamos por `VALIDAS(C)` el conjunto de implicaciones de un contexto C, válidas en dicho contexto.

```
VALIDAS(C): set[implicacion(C)] =
  {imp:implicacion(C)| es_valida?(C)(imp)}
```
218

Y por `MODELOS(C)(L)` el conjunto de los modelos de las implicaciones de $\mathcal{L}$.

```
MODELOS(C)(L: set[implicacion(C)]): set[finite_set[T2]] =
  {Z: finite_set[T2] | Z ⊆ atrib(C) & respeta_imp?(C)(Z,L)}
```
219

Teorema 5.42 *El conjunto de los modelos de las implicaciones válidas en un contexto formal C es el conjunto de las intenciones de los conceptos de C.*

```
modelos_validas: THEOREM
  MODELOS(C)(VALIDAS(C)) = intenciones_conj_conceptos(C)(CONCEPTOS(C))
```
220

Demostración:

La prueba se hace por doble inclusión:

- Si Z es un modelo de las implicaciones válidas en C, entonces Z es modelo de $Z \to Z''$, que es válida por 213. Por tanto, $Z = Z''$. Luego, si consideramos el concepto de C, (Z', Z), se tiene que Z es su intención.

- Por otra parte, si Z es la intención de un concepto de C, dicho concepto es de la forma (Z', Z). Veamos que, en este caso, Z es modelo de todas las implicaciones válidas en C. En efecto, sea $Y_1 \to Y_2$ una implicación válida en C (por 213, $Y_2 \subseteq Y_1''$) y supongamos que $Y_1 \subseteq Z$. Entonces, $Y_2 \subseteq Y_1'' \subseteq Z'' = Z$. Por tanto, Z es modelo de $Y_1 \to Y_2$.

□

5.5. Base de Duquenne–Guigues

5.5.1. Base de implicaciones

El conjunto de todas las implicaciones entre atributos válidas en un contexto puede ser excesivamente grande y contener muchas implicaciones triviales. Por ello, interesa disponer de conjuntos pequeños de implicaciones que sean suficientes para deducir las demás. Para ello, establecemos la noción de base de implicaciones.

Una **base de implicaciones** entre atributos es un conjunto de implicaciones adecuado (cada implicación es válida), completo (todas las implicaciones válidas son consecuencia de la base) y no redundante (ninguna de sus implicaciones es consecuencia de las restantes). En esta sección, se define la base de Duquenne–Guigues, se demuestra que es una base de implicaciones y se especifica un algoritmo para calcularla, demostrándose su corrección. La formalización en PVS está desarrollada en la teoría `base_DG` (páginas 386–392).

Definición 5.43 *Sea $\mathcal{L}$ un conjunto de implicaciones de un contexto formal C.*

- *Decimos que $\mathcal{L}$ es* **adecuado respecto de** *C si todos sus elementos son implicaciones válidas en C.*

221
```
es_adecuado(C)(L:set[implicacion(C)]): bool =
  ∀(imp: (L)): es_valida?(C)(imp)
```

- *Decimos que $\mathcal{L}$ es* **completo respecto de** *C si toda implicación $Y_1 \to Y_2$ válida en C, es consecuencia de $\mathcal{L}$.*

$$C \models Y_1 \to Y_2 \Rightarrow \mathcal{L} \models Y_1 \to Y_2$$

```
es_completo(C)(L:set[implicacion(C)]): bool =
  ∀(imp: implicacion(C)):
  es_valida?(C)(imp) ⇒  es_consecuencia(C)(imp,L)
```
222

- *Decimos que $\mathcal{L}$ es* **no redundante respecto de** *C si ninguna implicación de $\mathcal{L}$ es consecuencia del resto de las implicaciones de $\mathcal{L}$.*

$$Y_1 \to Y_2 \in \mathcal{L} \Rightarrow \mathcal{L} \setminus \{Y_1 \to Y_2\} \not\models Y_1 \to Y_2$$

```
es_no_redundante(C)(L:set[implicacion(C)]): bool =
  ∀(imp: (L)): NOT es_consecuencia(C)(imp, remove(imp, L))
```
223

- *Decimos que $\mathcal{L}$ es* **cerrado respecto de** *C si contiene a todas sus consecuencias.*

```
es_cerrado(C)(L:set[implicacion(C)]): bool =
  ∀(imp: implicacion(C)): es_consecuencia(C)(imp,L) ⇒ imp ∈ L
```
224

Así, dado un contexto formal C, se busca un conjunto de implicaciones de C, $\mathcal{L}(\mathcal{C})$, que sea adecuado, no redundante y completo. Guigues y Duquenne [39] han probado que, para todo contexto con un número finito de atributos, existe un conjunto de implicaciones con estas características, llamado **stem base** o **base de Duquenne–Guigues**.

Para ello, se define la noción de pseudointención en un contexto C como un conjunto de atributos, que no coincide con su clausura, pero que contiene las clausuras de todos sus subconjuntos propios, que son pseudointenciones.

Definición 5.44 *Un conjunto de atributos de C, P, es una* **pseudointención** *de un contexto formal C si:*

- *$P \neq P''$ (o, lo que es equivalente, $P'' \not\subseteq P$)*
- *Para toda pseudointención R con $R \subset P$, $R'' \subseteq P$.*

En PVS, especificamos la noción de pseudointención, respecto de un contexto C, sobre conjuntos finitos de preatributos:

```
es_pseudo_intencion?(C)(Y): RECURSIVE bool =
  Y ≠ clausura_a(C)(Y) &
  ∀ (R: finite_set[T2]): R ⊂ Y & es_pseudo_intencion?(C)(R)
  ⇒ clausura_a(C)(R) ⊆ Y
  MEASURE card(Y)
```
225

A partir de este predicado establecemos el tipo de las pseudointenciones relativas a un contexto C como

```
pseudo_int(C): TYPE = (es_pseudo_intencion?(C))
```
226

y definimos el conjunto de las pseudointenciones de un contexto C que son conjuntos de atributos de C:

```
PSEUDOINT(C): set[finite_set[T2]] =
  {Y | Y ⊆ atrib(C) & es_pseudo_intencion?(C)(Y)}
```
227

En el contexto del ejemplo 5.1, el conjunto $\{E\}$ es una pseudointención, ya que $\{E\}'' = \{R, S, Tr, E\} \not\subseteq \{E\}$ y contiene la clausura de todos sus subconjuntos propios, pues $\emptyset'' = \emptyset \subseteq \{E\}$. En cambio, $\{Tr, To, E\}$ no es pseudointención, pues aunque $\{Tr, To, E\}'' = \{R, S, Tr, To, E\} \not\subseteq \{Tr, To, E\}$, no contiene a $\{E\}''$.

Definición 5.45 *Sea C un contexto formal finito. La* **base de Duquenne–Guigues** *de C es el conjunto*

$$\mathcal{L}(\mathcal{C}) = \{Y \to Y'' : Y \in \texttt{PSEUDOINT(C)}\}$$

En el ejemplo 5.1, la base de Duquenne–Guigues es:

$$\mathcal{L}(\mathcal{C}) = \{\{E\} \to \{R, S, Tr, E\}, \{R, S, Tr, E\} \to \{E\}\}$$

La especificación en PVS del conjunto $\mathcal{L}(\mathcal{C})$ es la siguiente:

```
IMPS(C): set[implicacion(C)] =
  image(LAMBDA (Y): (# antecedente := Y,
                       consecuente := clausura_a(C)(Y) #))
      (PSEUDOINT(C))
```
228

Veamos que, en efecto, la base de Duquenne-Guigues de C es una base de implicaciones de C. Para ello, se prueban los siguientes resultados:

Lema 5.46 *Si el conjunto de atributos Y es una pseudointención, entonces Y no es modelo de la implicación $Y \to Y''$.*

```
cn_pseudo_intencion_cl: LEMMA
  es_pseudo_intencion?(C)(Y) & Y ⊆ atrib(C)
  ⇒ NOT respeta_imp?(C)(Y, (# antecedente:= Y,
                              consecuente:= clausura_a(C)(Y) #))
```
229

Lema 5.47 *La base de Duquenne–Guigues de un contexto C es un conjunto de implicaciones adecuado.*

```
IMPS_es_adecuado: LEMMA es_adecuado(C)(IMPS(C))
```
230

Demostración:
Es consecuencia directa de *213*.

□

Lema 5.48 *La base de Duquenne–Guigues de un contexto formal C es un conjunto de implicaciones no redundante.*

```
IMPS_es_no_redundante: LEMMA es_no_redundante(C)(IMPS(C))
```
231

Demostración:
Sea $Y \to Y''$ una implicación de $\mathcal{L}(\mathcal{C})$. Por construcción de la base, Y es un conjunto de atributos de C, que es pseudointención. Veamos que $Y \to Y''$ no es consecuencia de $\mathcal{L}(\mathcal{C}) \setminus \{Y \to Y''\}$. En efecto, si consideramos el propio Y verifica:

- Por *229*, Y no es modelo de $Y \to Y''$.
- Veamos que Y es modelo de todas las implicaciones de $\mathcal{L}(\mathcal{C}) \setminus \{Y \to Y''\}$. En efecto, tomemos una implicación $Z \to Z''$ de dicho conjunto. Evidentemente, $Y \neq Z$. Entonces, si $Z \subseteq Y$, necesariamente $Z \subset Y$ y, por ser Z pseudointención, se tiene que $Z'' \subseteq Y$. Por tanto, Y es modelo de $Z \to Z''$.

□

Lema 5.49 *La base de Duquenne–Guigues de un contexto C es un conjunto completo de implicaciones.*

```
IMPS_es_completo: LEMMA es_completo(C)(IMPS(C))
```
232

Demostración:
Hay que probar lo siguiente: toda implicación $Y_1 \to Y_2$ válida en $C = (O, A, I)$, es consecuencia de $\mathcal{L}(C)$. Es decir, que si $R \subseteq A$ y R respeta $\mathcal{L}(C)$, entonces R también respeta $Y_1 \to Y_2$. Lo probamos, distinguiendo dos casos:

- Caso 1: $R = R''$
  Supongamos $Y_1 \subseteq R$, entonces usando los lemas *169* y *170*, se tiene que $Y_1'' \subseteq R''$. Por otra parte, por ser válida, $Y_2 \subseteq Y_1''$. Luego, $Y_2 \subseteq R'' = R$.

- Caso 2: $R \neq R''$
  En primer lugar, comprobamos que si R respeta $\mathcal{L}(\mathcal{C})$ y $R \neq R''$, entonces R es una pseudointención. En efecto, es suficiente probar que se verifica la segunda condición: sea Q una pseudointención tal que $Q \subset R$. Por ser pseudointención, $Q \to Q'' \in \mathcal{L}(\mathcal{C})$ y, por tanto, R respeta $Q \to Q''$. Luego, $Q'' \subseteq R$.

  Entonces, $R \to R'' \in \mathcal{L}(\mathcal{C})$. Como R respeta $\mathcal{L}(\mathcal{C})$, R respeta $R \to R''$ y, por tanto, $R'' \subseteq R$. Como siempre se verifica $R \subseteq R''$, tendríamos $R = R''$, en contra de la hipótesis.

□

5.5.2. Generación de la base de Duquenne–Guigues

La base de Duquenne–Guigues no es la única base de implicaciones de un contexto formal C. Pero juega un papel especial, por dos razones: tiene un número mínimo de elementos (i.e., no existe otra base con menos elementos) y puede ser obtenida de forma algorítmica. En esta subsección se presenta un algoritmo (`genera_imps`) para calcularla y se demuestra su corrección.

La definición de una función que genere la base de Duquenne–Guigues es inmediata si se dispone del conjunto de las pseudointenciones. Es decir, si `gen_s(C)` es una función tal que `gen_s(C) = PSEUDOINT(C)`, entonces `genera_imps(C) = image(f)(gen_s(C))`, donde $f(Y) = Y \to Y''$. Por tanto, será suficiente especificar un algoritmo (`gen_s`) que calcule dicho conjunto y demostrar su corrección.

La definición recursiva de pseudointención nos proporciona un algoritmo para generarlas, de forma escalonada según el cardinal, como sigue:

- Paso 0: obtener las pseudointenciones de C de cardinal 0: $\mathcal{S}_0 = \{\emptyset\}$ si $\emptyset$ es pseudointención y $\mathcal{S}_0 = \emptyset$, en otro caso.

- Paso $k+1$: obtener las pseudointenciones de C de cardinal menor o igual que $k+1$, $\mathcal{S}_{k+1}$, a partir de las pseudointenciones de C de cardinal menor o igual que k, $\mathcal{S}_k$, para $k \leq card(A)$. Para ello, introducimos una noción restringida de pseudointención como sigue: Sea $\mathcal{S}$ un conjunto de conjuntos de preatributos y P un conjunto de preatributos. Decimos que P es una **pseudointención restringida** a $\mathcal{S}$ si verifica:

  - $P \neq P''$
  - Para todo $X \in \mathcal{S}$, si $X \subset P$ entonces $X'' \subseteq P$

  Entonces, el conjunto $\mathcal{S}_{k+1}$ puede escribirse como:

$$\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{P \subseteq A : |P| = k+1 \wedge$$
$$P \text{ es una pseudointención restringida a } \mathcal{S}_k\}$$

Finalmente, el conjunto $\mathcal{S}_{|A|}$ está formado por todas las pseudointenciones con cardinal menor o igual que $|A|$. Es decir, $\mathcal{S}_{|A|}$ = `PSEUDOINT(C)`.

Veamos cómo especificar en PVS el cálculo por pasos del conjunto $\mathcal{S}_{|A|}$. En primer lugar, especificamos la noción de pseudointención, respecto de un contexto C, restringida a un conjunto $\mathcal{S}$, como sigue:

233
```
es_pseudo_int_rest?(C)(Y: finite_set[T2],
                       S: finite_set[finite_set[T2]]): bool =
  clausura_a(C)(Y) ⊈ Y &
  ∀ (Y1:(S)): Y1 ⊂ Y ⇒ clausura_a(C)(Y1) ⊆ Y
```

Notación 5.50 Sean $\mathcal{A}$ y $\mathcal{S}$ conjuntos finitos de conjuntos finitos de preatributos. Representamos por `pseudo_restringidas(C)(A,S)` el conjunto de los elementos de $\mathcal{A}$ que son pseudointenciones restringidas a $\mathcal{S}$.

234
```
pseudo_restringidas(C)(A: finite_set[finite_set[T2]],
                       S: finite_set[finite_set[T2]]):
                      finite_set[finite_set[T2]] =
  {Y: finite_set[T2] | Y ∈ A & es_pseudo_int_rest?(C)(Y,S)}
```

También especificamos en PVS el conjunto $\mathcal{S}_k$ como

```
S(C:FFC, (k:nat)): finite_set[finite_set[T2]] =
  {X: finite_set[T2] | X ⊆ atrib(C) &
                       es_pseudo_intencion?(C)(X) & card[T2](X) <= k}
```

y probamos que

```
  union(S(C,k),
        pseudo_restringidas(C)(subconjuntos(atrib(C),1+k), S(C,k)))
   = S(C, 1 + k)
```

donde la función `subconjuntos(B,n)`, incluida en el apéndice A, obtiene los subconjuntos de B de cardinal n.

Para especificar la función `gen_s` definimos la función auxiliar `gen_s_pasos`, que toma como argumentos un conjunto finito de atributos Y, un número natural $k \leq card(Y)$ y un conjunto finito de conjuntos finitos de preatributos $\mathcal{S}$. La función `gen_s_pasos` es una función recursiva que, en cada paso, le añade a $\mathcal{S}$ los subconjuntos de Y de cardinal j, que son pseudointenciones restringidas a $\mathcal{S}$, empezando por $j = k$ hasta $j = |Y|$.

```
gen_s_pasos(C)(Y: finite_set[T2],
               k: upto(card(Y)),
               S: finite_set[finite_set[T2]]):
               RECURSIVE set[finite_set[T2]] =
  LET NS = pseudo_restringidas(C)(subconjuntos(Y,k), S) IN
   IF k = card(Y) THEN  union(S, NS)
   ELSE  gen_s_pasos(C)(Y,k+1,union(S,NS)) ENDIF
 MEASURE card(Y)-k
```
235

Si los argumentos iniciales son el conjunto de atributos A, 0 y $\emptyset$, en cada paso k el conjunto $\mathcal{S}$ que se va construyendo es, precísamente, $\mathcal{S}_k$. Luego, la salida del algoritmo será $\mathcal{S}_{|A|}$, que coincide con el conjunto `PSEUDOINT(C)`.

Así la especificación del algoritmo que obtiene todas las pseudointenciones de un contexto finito C, es:

```
gen_s(C):set[finite_set[T2]] = gen_s_pasos(C)(atrib(C),0,emptyset)
```
236

Probamos en PVS que el algoritmo representado por la función `gen_s` es correcto.

Teorema 5.51 *Sea C un contexto formal finito. Entonces,* `gen_s(C)` *es el conjunto de las pseudointenciones de C, que son conjuntos de atributos de C.*

```
correccion_gen_s: THEOREM gen_s(C) = PSEUDOINT(C)
```
237

Demostración:

La prueba se realiza probando las dos inclusiones.

1. En primer lugar, probamos que si Z es un elemento de `gen_s(C)`, entonces Z es un conjunto de atributos de C, que es pseudointención de C:

   ```
   gen_s_correcto: LEMMA
     ∀(Z:finite_set[T2]): Z∈gen_s(C) ⇒ es_pseudo_intencion?(C)(Z)

   gen_s_correcto_b: LEMMA
     ∀(Z:finite_set[T2]): Z ∈ gen_s(C) ⇒ Z ⊆ atrib(C)
   ```

   Para ello, introducimos la notación $\mathcal{SC}(C, Y, k)$ para representar el conjunto formado por los conjuntos de atributos de C, contenidos en Y, que son pseudointenciones de C, de cardinal menor que k:

```
pseudo_card(C,Y, (k:nat) ): finite_set[finite_set[T2]] =
   {X: finite_set[T2] | X⊆atrib(C) & es_pseudo_intencion?(C)(X)
                        & X ⊆ Y & card[T2](X) < k}
```

Así, hemos de probar que si $\mathcal{S} = \mathcal{SC}(C, Y, k)$ y Z es un elemento del conjunto `gen_s_pasos(C)(Y,k,S)`, entonces $Z \subseteq Y$ y Z es una pseudointención de C.

La expresión en PVS de este resultado es:

```
gen_s_pasos_correcto_l0: LEMMA
  FORALL (Y: finite_set[T2],
          k: upto(card(Y)),
          S: finite_set[finite_set[T2]],
          Z: finite_set[T2]):
  Y ⊆ atrib(C) & S=pseudo_card(C,Y,k) & Z∈ gen_s_pasos(C)(Y,k,S)
  ⇒ Z ⊆ Y & es_pseudo_intencion?(C)(Z)
```

La prueba se realiza por inducción en `card(Y)-k`, usando que

$\mathcal{SC}(C, Y, k+1) =$
$\mathcal{SC}(C, Y, k) \cup \texttt{pseudo\_restringidas}(C)(\texttt{subconjuntos}(Y, k), \mathcal{SC}(C, Y, k))$

2. En segundo lugar, probamos que si Z es un conjunto de atributos de C, que es pseudointención de C, entonces Z es un elemento de `gen_s(C)`.

   Para ello, se prueba por inducción en `card(Y)-k` que, si Z es un conjunto de atributos de cardinal menor o igual que k, y **no** ha sido generado mediante `gen_s_pasos(C)(Y,k,S)`, con $\mathcal{S} = \mathcal{SC}(C, Y, k)$, entonces Z **no** es una pseudointención de C.

```
gen_s_pasos_completo_l4: LEMMA
  FORALL (Y: finite_set[T2],
          k: upto(card(Y)),
          S: finite_set[finite_set[T2]],
          Z: finite_set[T2]):
  Y ⊆ atrib(C) & Z ⊆ atrib(C) & S = pseudo_card(C,Y,k) &
  Z ⊆ Y & card(Z) <= k & Z ∉ gen_s_pasos(C)(Y,k,S)
  ⇒ NOT es_pseudo_intencion?(C)(Z)
```

Como lema auxiliar, probamos también que la función `gen_s_pasos` es decreciente en k

```
gen_s_pasos_subset_k: LEMMA
  ∀ (Y: finite_set[T2], k: upto(card(Y))):
  Y ⊆ atrib(C) & k ≠ card(Y) ⇒
  gen_s_pasos(C)(Y,k+1,pseudo_card(C,Y,k+1)) ⊆
  gen_s_pasos(C)(Y,k,pseudo_card(C,Y,k))
```

con lo que se verifica, directamente, que

```
gen_s_pasos_subset: LEMMA
  ∀ (Y: finite_set[T2], k: upto(card(Y))):
  Y ⊆ atrib(C) & card(Y) ≠ 0 ⇒
  gen_s_pasos(C)(Y,k,pseudo_card(C,Y,k)) ⊆ gen_s_pasos(C)(Y,0,∅)
```

□

Obsérvese que el algoritmo para generar la base de Duquenne–Guigues recibe como dato de entrada un contexto formal finito C y devuelve el conjunto formado por las implicaciones que forman dicha base. A la hora de especificar el tipo del rango de dicho algoritmo podemos establecer que sea un conjunto de implicaciones genéricas (`set[implicacion_gen]`), o bien conjuntos de implicaciones específicas de C (`set[implicacion(C)]`). Este segundo caso es más restrictivo pero, con respecto a la prueba de su corrección es más adecuado, pues ésta consiste en probar que el conjunto generado coincide con el conjunto `IMPS(C)`, cuyo tipo es `set[implicacion(C)]`. Ahora bien, en el capítulo 8 realizaremos una construcción evaluable de este algoritmo y, en cierto sentido "equivalente" a él. Para ello, es más conveniente que el tipo del rango sea `set[implicacion_gen]`. Por esta razón, realizamos dos especificaciones, una con cada uno de los posibles tipos para el rango, y probamos que coinciden sobre el rango menor.

238

```
genera_imps_g(C): set[implicacion_gen] =
  LET fun = LAMBDA (Y: finite_set[T2]):
               (# antecedente:= Y, consecuente:= clausura_a(C)(Y) #)
      IN image(fun)(gen_s(C))

genera_imps(C): set[implicacion(C)] =
  LET fun = LAMBDA (Y: finite_set[T2]):
               (# antecedente:= Y, consecuente:= clausura_a(C)(Y) #)
      IN image(fun)(gen_s(C))

genera_imps_igual_genera_imps_g: LEMMA
  genera_imps(C) = genera_imps_g(C)
```

Una vez probada la corrección del algoritmo que genera el conjunto de pseudointenciones de un contexto formal finito C, finalizamos la sección estableciendo la corrección del algoritmo que genera la base de Duquenne–Guigues.

Teorema 5.52 *Dado un contexto formal finito* C, `genera_imps(C)` $= \mathcal{L}(C)$.

```
correccion_genera_imp: THEOREM genera_imps(C) = IMPS(C)
```
239

Demostración:

Por definición, `genera_imps(C) = image(f)(gen_s(C))`, donde la función f es $f(Y) = Y \to Y''$. Además, por *237*, `gen_s(C) = PSEUDOINT(C)`. Luego, `genera_imps(C) = image(f)(PSEUDOINT(C)) = IMPS(C)`.

□

Capítulo 6

Marco genérico para refinamientos en PVS

En el marco de un sistema formal, en nuestro caso PVS, se desea realizar una especificación de un proceso o algoritmo, probar sus propiedades (por ejemplo, su corrección) y que dicho algoritmo sea evaluable (incluso eficiente). El problema que surge, con frecuencia, es que si la especificación es evaluable no suele ser fácil o natural razonar sobre ella; y si se razona fácilmente, no se puede evaluar o es muy ineficiente. Las cuestiones que nos plantea este problema son, pues, cómo conciliar los dos aspectos, por cuál decantarse, o qué relación hay entre distintas especificaciones de un algoritmo.

Hay que tener en cuenta que, a la hora de construir una especificación de un algoritmo, lo primero que se hace es elegir los tipos de datos que se van a usar. Esta elección determinará tanto la forma de razonar sobre ella como su posible evaluabilidad o eficiencia.

En nuestro caso, la especificación del cálculo del menor punto fijo de un programa definido realizada en el capítulo 4 *108* no es evaluable, pues está basada en el uso del tipo de los conjuntos finitos. ¿Qué hacer, entonces, si queremos calcular efectivamente el menor punto fijo de un programa? Una opción sería realizar otra especificación basada en tipos de datos evaluables y probar de nuevo sus propiedades. O bien, realizar otra especificación y relacionarla con la anterior, de forma que las propiedades que verifica la primera se "trasladen" a la segunda.

Esto último es justamente lo que queremos hacer. Pero no sólo en este caso. Queremos establecer un marco general, que permita relacionar distintas especificaciones de un algoritmo. El punto clave será representar un tipo abstracto TA_1 por otro tipo TA_2, sobre el que las operaciones sean evaluables (esta representación puede hacerse en pasos sucesivos). Y las operaciones sobre el tipo más general habrán de ser "refinadas" por otras, de forma que presenten el mismo comportamiento. Dicho marco está basado en la noción de refinamiento introducida por Jones en [43], usada por Dold en [24, 23]. En [43] se define la noción de *data reification*, que trata sobre la transición entre tipos de datos abstractos y

concretos, y de la corrección de la misma.

La principal ventaja será poder realizar especificaciones semánticamente simples (usando tipos de datos abstractos) sobre las que se pueda razonar bien, y relacionarlas con otras especificaciones evaluables (que usan tipos de datos concretos) de los mismos conceptos y, en cierto sentido, equivalentes a la anterior.

Como hemos comentado en capítulo dedicado a introducir el sistema PVS, el lenguaje de especificación de PVS está diseñado para servir como una lógica expresiva y no como un lenguaje de programación. Por ello, no todas las especificaciones en PVS son evaluables. En particular, no lo son las que hacen uso del tipo de los conjuntos finitos. Aún así, se dispone de un interesante fragmento del lenguaje que sí es evaluable como se describe en [80], y hemos comentado en 2.1.6.

En este capítulo establecemos las nociones de refinamiento de tipos y de operaciones, y estudiamos las propiedades de las operaciones que se conservan a través de los refinamientos. Como caso particular, estudiamos el refinamiento del tipo de los conjuntos finitos de PVS mediante el tipo de dato listas. De esta forma, como las especificaciones realizadas en la formalización de las teorías en los capítulos anteriores usan conjuntos finitos, podremos refinarlas por otras basadas en listas, que serán evaluables.

6.1. Refinamiento de tipos

En esta sección se define el concepto de refinamiento de tipos, se establece la relación de igualdad inducida en un tipo por un refinamiento y se prueban algunas de sus propiedades. La formalización en PVS se encuentra desarrollada en las teorías `refinamiento`, `refinamiento_equiv` y `refinamiento_prod_cart` (páginas 260–262).

Definición 6.1 *Sean R y T dos tipos no interpretados. Decimos que R es un* **refinamiento** *del tipo T mediante la función $f : R \longrightarrow T$, si f es sobreyectiva (i.e. si todo elemento de T puede ser representado por un elemento de R).*

240
```
f: VAR [R->T]
es_refinamiento?(f): bool = surjective?(f)
```

Evidentemente, $id : T \longrightarrow T$ es un refinamiento.

241
```
identidad_ref: LEMMA es_refinamiento?[T, T](id)
```

Por abuso de lenguaje, diremos indistintamente que R es un refinamiento de T mediante $f : R \longrightarrow T$, que $f : R \longrightarrow T$ es un refinamiento, que f es un refinamiento de T por R, o que R refina a T mediante f. En lo que sigue, g

representará un refinamiento de T por R. Es decir, los elementos de R pueden verse como representaciones o concreciones de los elementos de T (que se ven como abstracciones).

```
g: VAR (es_refinamiento?[T, R])
```
242

La función g induce, de manera natural, una relación de equivalencia en R: dos elementos de R son equivalentes si representan el mismo elemento de T. Es decir, dados $r_1, r_2 \in R$, $r_1 \equiv_g r_2 \Leftrightarrow g(r_1) = g(r_2)$.

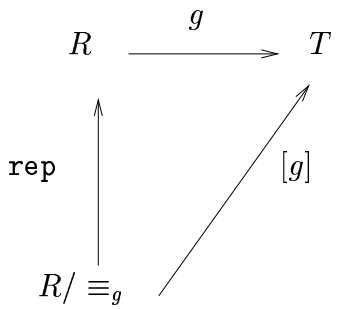
```
igual?(g)(r1,r2:R): bool = g(r1) = g(r2)

igual?_es_de_equivalencia: LEMMA equivalence?[R](igual?(g))
```
243

En caso de que g sea inyectiva, la relación $\equiv_g$ es la igualdad entre los elementos del tipo R.

```
igual?_injective: LEMMA
  injective?(g) ⇒ igual?(g) = =
```
244

Nótese que el lenguaje de PVS permite hacer un amplio uso de la sobrecarga de los símbolos. En este caso, hemos usado el símbolo `=` con un doble significado: como la igualdad entre los elementos de R y como la igualdad entre dos relaciones.

En principio, si R refina a T mediante g, la relación entre los elementos no tiene por qué ser biyectiva. Ahora bien, si consideramos el tipo formado por el conjunto cociente $R/\equiv_g$ y definimos la aplicación correspondiente $[g] : R/\equiv_g \longrightarrow T$ como $[g]([r]) = g(r)$, entonces $R/\equiv_g$ es un refinamiento de T, mediante la función inyectiva $[g]$.



En la teoría `equivalence_class`[1] se ha formalizado el conjunto cociente generado por una relación de equivalencia == sobre un tipo T mediante `E[T,==]`, la clase de equivalencia de un elemento x de T por `equiv_class(x)`, y la función que elige un representante de una clase de equivalencia por `rep`. Entonces, representando en PVS la función $[g]$ por f_equiv(g), se tiene:

[1]Esta teoría, descrita en [62] se ha incluido en el apéndice C (páginas 259–259).

```
f_equiv(g)(A:E[R, igual?(g)]):T = g(rep(A))

clases_equiv_refinamiento: LEMMA
  es_refinamiento?[T, E[R, igual?(g)]](f_equiv(g))

f_equiv_inyectiva: LEMMA injective?(f_equiv(g))
```
245

Por último, establecemos que un determinado tipo puede ser refinado en pasos sucesivos. Y que el producto cartesiano de dos refinamientos de tipos es un refinamiento del producto cartesiano de los tipos.

Lema 6.2 *Si R es un refinamiento de T mediante f y Q es un refinamiento de R mediante g, entonces Q es un refinamiento de T mediante $f \circ g$.*

```
comp_es_refinamiento: LEMMA
   es_refinamiento?[T, R](f) & es_refinamiento?[R, Q](g) =>
   es_refinamiento?[T, Q](f o g)
```
246

Definición 6.3 *Dadas las funciones $f_1 : R_1 \longrightarrow T_1$ y $f_2 : R_2 \longrightarrow T_2$, definimos la función $f_1 \times f_2 : R_1 \times R_2 \longrightarrow T_1 \times T_2$ como: $(f_1 \times f_2)(r_1, r_2) = (f_1(r_1), f_2(r_2))$*

```
prod_cart(f1,f2)(par:[R1, R2]): [T1, T2] =
   (f1(PROJ_1(par)), f2(PROJ_2(par)))
```
247

Lema 6.4 *Si R_1 es un refinamiento de T_1 mediante f_1 y R_2 es un refinamiento de T_2 mediante f_2, entonces el producto cartesiano $R_1 \times R_2$ es un refinamiento del producto cartesiano $T_1 \times T_2$ mediante $f_1 \times f_2$.*

```
prod_cart_es_refinamiento: LEMMA
  es_refinamiento?[T1, R1](f1) &  es_refinamiento?[T2, R2](f2)
  ⇒ es_refinamiento?[[T1,T2], [R1, R2]](prod_cart(f1, f2))
```
248

6.2. Refinamiento de operaciones

En el marco que estamos desarrollando, queremos relacionar distintas especificaciones de un mismo algoritmo. Esencialmente, podemos considerar una especificación de un algoritmo como una operación o función $op_1 : T_1 \to T_2$, donde T_1 y T_2 son los tipos que representan las posibles entradas y salidas del algoritmo, respectivamente. En este sentido, otra especifación del mismo algoritmo vendría dada por otra función $op_2 : R_1 \to R_2$, donde R_1 se usa para representar las entradas y R_2 para representar las salidas del algoritmo. De manera natural, se entiende que op_1 y op_2 son especificaciones del mismo algoritmo si tienen el mismo

"comportamiento", lo que expresaremos mediante el concepto de *refinamiento de operaciones*.

En esta sección se presenta la formalización en PVS de la noción refinamiento de operaciones, realizada en la teoría `refinamiento_oper` (páginas 263–263). En ella, se establece cuándo una operación es un refinamiento de otra, suponiendo que entre los tipos que constituyen los dominios y los rangos de las operaciones se tienen los refinamientos correspondientes.

Consideremos una función $op : T_1 \to T_2$ y supongamos que tenemos los refinamientos de tipos dados por las funciones $f_1 : R_1 \to T_1$ y $f_2 : R_2 \to T_2$. Entonces, decimos que la función $op_{ref} : R_1 \to R_2$ es un **refinamiento** de op si el siguiente diagrama es commutativo:

$$\begin{array}{ccc} T_1 & \xrightarrow{op} & T_2 \\ {\scriptstyle f_1}\uparrow & \# & \uparrow{\scriptstyle f_2} \\ R_1 & \xrightarrow{op_{ref}} & R_2 \end{array}$$

Figura 6.1: Diagrama relativo al refinamiento de una operación

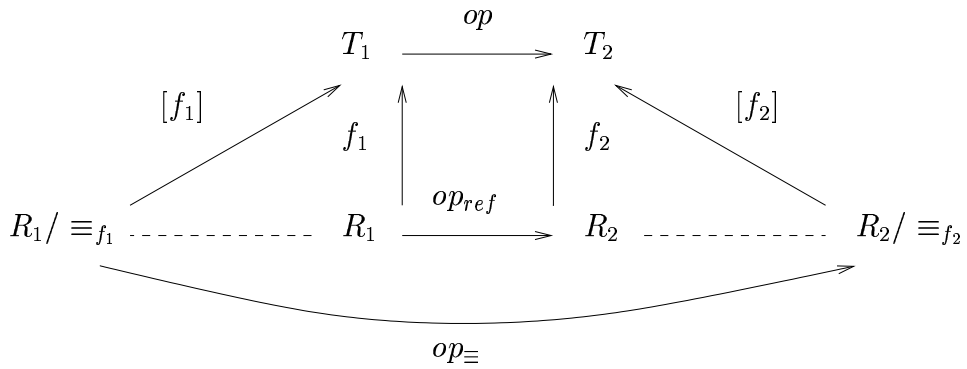
249

```
refinamiento_oper[T1, T2, R1, R2: TYPE,
                  (IMPORTING refinamiento)
                  f1: (es_refinamiento?[T1, R1]),
                  f2: (es_refinamiento?[T2, R2])]: THEORY

op: VAR [T1 -> T2],  op_ref: VAR [R1 -> R2]

es_refinamiento_op?(op, op_ref): bool =
   ∀ (r1:R1): op(f1(r1)) = f2(op_ref(r1))
```

Obsérvese que, entre los parámetros de la teoría `refinamiento_oper`, están incluidas las funciones f_1 y f_2, que constituyen los refinamientos entre los dominios y los rangos, respectivamente.

Por otra parte, dada una operación op y un refinamiento suyo, op_{ref}, se define la correspondiente operación entre los conjuntos cocientes de los tipos refinados, y se prueba que es también un refinamiento de op.

Lema 6.5 *Sean $f_1 : R_1 \longrightarrow T_1$ y $f_2 : R_2 \longrightarrow T_2$ refinamientos de tipos. Sea $op : T_1 \longrightarrow T_2$ una función y $op_{ref} : R_1 \longrightarrow R_2$ un refinamiento de op. La función $op_{\equiv} : R_1/\equiv_{f_1} \longrightarrow R_2/\equiv_{f_2}$, definida por $op_{\equiv}([r_1]) = [op_{ref}(r_1)]$ es un refinamiento de op.*




```
op_ref_equiv(op,op_ref)(A: E[R1, igual?(f1)]): E[R2, igual?(f2)] =
   equiv_class[R2, igual?(f2)](op_ref(rep(A)))

op_ref_equiv_es_refinamiento: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) =>
    es_refinamiento_op?[T1, T2, E[R1, igual?(f1)], E[R2, igual?(f2)],
                        f_equiv[T1, R1](f1), f_equiv[T2, R2](f2)]
       (op, op_ref_equiv(op, op_ref))
```

250

En esencia, la definición de refinamiento de una función requiere que la función op_{ref} tenga el mismo comportamiento que op. Además, es deseable que propiedades de la función op se "trasladen" a la función op_{ref}. En particular, si op y op_{ref} son especificaciones de un algoritmo, queremos que de la corrección de op pueda deducirse la corrección de op_{ref}. Siendo más precisos, supongamos que hemos probado la corrección de la función op en términos de precondición y postcondición. Es decir, si p es el predicado que describe la condición que han de verificar las entradas del algoritmo especificado por op, y q describe la relación entre el dato de entrada y el de salida, suponemos que hemos probado un teorema de la forma:

$$(\forall x \in T_1)[p(x) \Rightarrow q(x, op(x))]$$

Para expresar que la función op_{ref} tiene el mismo comportamiento que op, consideramos los predicados p_{ref} y q_{ref}, refinamientos de p y q, respectivamente, y probamos que se verifica

$$(\forall z \in R_1)[p_{ref}(z) \Rightarrow q_{ref}(z, op_{ref}(z))]$$

que es, justamente, el teorema de corrección correspondiente a op_{ref}.

```
p: VAR pred[T1],       p_ref: VAR pred[R1]
q: VAR pred[[T1, T2]], q_ref: VAR pred[[R1, R2]]

ref_preserva_correccion: LEMMA
  LET f = prod_cart[T1, T2, R1, R2](f1,f2) IN
  es_refinamiento_op?[T1,T2,R1,R2,f1,f2](op,op_ref) &
  es_refinamiento_op?[T1,bool,R1,bool,f1,id[bool]](p,p_ref) &
  es_refinamiento_op?[[T1,T2],bool,[R1,R2],bool,f,id[bool]](q,q_ref) &
  (∀ (x: T1): p(x) ⇒ q(x, op(x))) ⇒
  ∀ (z: R1): p_ref(z) ⇒ q_ref(z, op_ref(z))
```
251

La prueba se realiza directamente, a partir de las definiciones.

Es decir, si hemos probado la corrección de una especificación de un algoritmo, tenemos garantizada la corrección de cualquier otra especificación de dicho algoritmo, siempre que probemos que es un refinamiento de la primera.

Otro aspecto importante a tener en cuenta es que, normalmente, en la especificación de un algoritmo intervienen varias funciones. Veamos, entonces, que los refinamientos de operaciones se mantienen mediante la composición de funciones. Esto nos permitirá refinar cada una de las funciones que se usen en una especificación, y combinarlas para obtener así un refinamiento de dicha especificación.

Lema 6.6 *Sean $f_i : R_i \longrightarrow T_i$, con $i = 1, 2, 3$ refinamientos de tipos. Si $op_{1_{ref}} : R_1 \longrightarrow R_2$ refina a la operación $op_1 : T_1 \longrightarrow T_2$ y $op_{2_{ref}} : R_2 \longrightarrow R_3$ refina a $op_2 : T_2 \longrightarrow T_3$, entonces la composición $op_{2_{ref}} \circ op_{1_{ref}}$ refina a $op_2 \circ op_1$.*

La expresión de este resultado en PVS es:

```
comp_oper_es_refinamiento: LEMMA
  es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op1, op1_ref) &
  es_refinamiento_op?[T2, T3, R2, R3, f2, f3](op2, op2_ref) ⇒
  es_refinamiento_op?[T1,T3,R1,R3,f1,f3](op2 o op1,op2_ref o op1_ref)
```
252

$$\begin{array}{ccccc} T_1 & \xrightarrow{op_1} & T_2 & \xrightarrow{op_2} & T_3 \\ f_1\uparrow & \# & f_2\uparrow & \# & f_3\uparrow \\ R_1 & \xrightarrow{op_{1_{ref}}} & R_2 & \xrightarrow{op_{2_{ref}}} & R_3 \end{array}$$

Figura 6.2: Refinamiento de la composición de operaciones

Corolario 6.7 *Si op_{ref} refina a op, entonces la iteración n–sima de op_{ref} refina a la iteración n–sima de op.*

```
iteracion_es_refinamiento: COROLLARY                                    253
  es_refinamiento_op?(op,op_ref)
  ⇒ es_refinamiento_op?(iterate(op,n),iterate(op_ref,n))
```

Demostración:

Por inducción en n, usando que la iteración $(j+1)$-sima de una función es la composición de la función con su iteración j-sima y la propiedad 252.

□

Por último, estudiamos algunas propiedades sobre relaciones que se conservan a través de un refinamiento.

Lema 6.8 *Sean $f : R \longrightarrow T$ un refinamiento de tipos, ρ una relación en T y ϕ una relación en R, que es un refinamiento de ρ. Entonces, si ρ es una relación de equivalencia, ϕ también lo es.*

$$\begin{array}{ccc} T \times T & \xrightarrow{\rho} & \texttt{bool} \\ f \times f \uparrow & \# & id \uparrow \\ R \times R & \xrightarrow{\phi} & \texttt{bool} \end{array}$$

Figura 6.3: Las relaciones de equivalencia se preservan mediante refinamientos.

```
refinamiento_preserva_rel_equivalencia: LEMMA                           254
   es_refinamiento_op?[[T,T],bool,[R,R],bool,prod_cart(f,f),id[bool]]
                      (ρ, φ)
   & equivalence?(ρ)
   ⇒ equivalence?(φ)
```

Hay que hacer notar que la propiedad de antisimetría de una relación no se conserva a través de un refinamiento, si éste no es inyectivo. Es decir, si $\rho : T \to T$ es una relación antisimétrica y $\phi : R \to R$ es un refinamiento suyo, entonces se tiene: $\rho(r_1, r_2) \wedge \rho(r_2, r_1) \Rightarrow \phi(f(r_1), f(r_2)) \wedge \phi(f(r_2), f(r_1))$. Y, por ser ϕ antisimétrica, $f(r_1) = f(r_2)$, de donde no se deduce, en general, que $r_1 = r_2$. Como consecuencia, tenemos que la propiedad de ser relación de orden parcial no se conserva a través de un refinamiento. Ahora bien, si consideramos la relación inducida en el conjunto cociente $R/\equiv_f$, sí podemos asegurar que se conserva dicha propiedad.

6.3. Caso de estudio: un refinamiento de conjuntos finitos

En esta sección se establece un refinamiento del tipo de los conjuntos finitos sobre un tipo T, `finite_set[T]`, por el tipo de dato de las listas sobre un tipo R, `list[R]`, a partir de un refinamiento de T por R. Además, se presentan operaciones entre listas que refinan a las operaciones entre conjuntos finitos. La formalización en PVS se encuentra desarrollada en las teorías `refinamiento_conj_finitos`, `refinamiento_conj_finitos_gen`, `refinamiento_conj_finitos_image`, `refinamiento_powerset` y `refinamiento_subconjuntos_card` (páginas 267–288).

6.3.1. Refinamiento de tipo

Como hemos comentado al principio del capítulo, las especificaciones que usan conjuntos finitos no son evaluables. En principio, parece natural representar un conjunto finito por una lista con los mismos elementos, aunque no de manera única, pues puede variar el orden de los elementos o haber elementos repetidos. Por otra parte, es posible que se manejen conjuntos finitos cuyos elementos sean de un tipo que también sea necesario refinar. Así, si consideramos que el tipo R refina al tipo T, representaremos un conjunto finito de elementos de T mediante una lista de elementos de R.

Para ello, dado un refinamiento $f : R \longrightarrow T$, definimos la función recursiva `c(f)` que transforma un elemento de `list[R]` en un elemento de `finite_set[T]` y probamos que es sobreyectiva.

255

```
c(f)(l: list[R]): RECURSIVE finite_set[T] =
  CASES l OF
   null: ∅,
   cons(x, l1): add(f(x), c(f)(l1))
  ENDCASES
  MEASURE length(l)
```

En primer lugar, caracterizamos la función `c(f)`, probando que [2]

$$\mathtt{c(f)}(l) = \{f(x) : x \in l\}$$

Lema 6.9 *Sea l una lista de elementos de R e y un elemento de T. Entonces,*

$$y \in c(f)(l) \Leftrightarrow \exists x \in l : \ y = f(x)$$

[2] Obsérvese que también podríamos haber definido la función `c(f)` de esta forma.

```
imagen_c_caract: LEMMA                                              256
  member(y, c(f)(l)) ⇔ ∃ x: member(x, l) & y = f(x)
```

Demostración:
Por inducción y simplificación en l.

□

Lema 6.10 *El tipo* `list[R]` *es un refinamiento del tipo* `finite_set[T]` *mediante la función* $c(f)$.

```
c_es_refinamiento: LEMMA                                            257
  es_refinamiento?[finite_set[T], list[R]](c(f))
```

Demostración:
Por la definición refinamiento de tipos, hay que probar

$$(\forall y \in T : \exists x \in R : f(x) = y) \Rightarrow (\forall Y \in \texttt{finite\_set}[\mathrm{T}] : \exists l \in \texttt{list}[\mathrm{R}] : c(f)(l) = Y)$$

Lo probamos por inducción en Y, según el esquema de inducción [1], descrito en el capítulo 2.

□

6.3.2. Refinamiento de operaciones

Una vez que tenemos un refinamiento de `finite_set[T]` mediante `list[R]`, para cada operación con conjuntos finitos presentamos una operación con listas, que la refina. En algunos casos, probaremos que una de las operaciones sobre listas definidas en el preludio refina a una operación sobre conjuntos finitos y, en otros, tendremos que definir la operación correspondiente. La siguiente tabla muestra una lista con las operaciones sobre conjuntos finitos y sus correspondientes refinamientos.

conjuntos finitos	listas	figura
empty?	null?	6.4
add	cons	6.5
member	member, member(igual?(f))	6.7
union	append	6.8
intersection	inter, inter(igual?(f))	6.9
remove	elimina, elimina(igual?(f))	6.10
choose	car	6.11
rest	rest_l, rest_l(igual?(f))	6.12
subset?	sublista?, sublista?(igual?(f))	6.13

card	cardinal_|, cardinal_|(igual?(f))	6.15
Union	Append	6.16
Intersection	Inter, Inter(igual?(f))	6.17
powerset	powerset_ref	6.19

6.3.2.1. Refinamiento de `empty?`

Lema 6.11 *El predicado* `null?` *es refinamiento de* `empty?`.

$$\begin{array}{ccc} \texttt{finite\_set}[\texttt{T}] & \xrightarrow{\texttt{empty?}} & \texttt{bool} \\ \texttt{c}(f)\uparrow & \# & id\uparrow \\ \texttt{list}[\texttt{R}] & \xrightarrow{\texttt{null?}} & \texttt{bool} \end{array}$$

Figura 6.4: Refinamiento del predicado `empty?`

258
```
null?_es_refinamiento_empty?: LEMMA
  es_refinamiento_op?[finite_set[T],bool,list[R],bool,c(f),id[bool]]
                     (empty?, null?)
```

6.3.2.2. Refinamiento de `add`

Lema 6.12 *La operación* `cons` *es refinamiento de* `add`.

$$\begin{array}{ccc} T \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{add}} & \texttt{finite\_set}[T] \\ f \times \texttt{c}(f)\uparrow & \# & \texttt{c}(f)\uparrow \\ R \times \texttt{list}[R] & \xrightarrow{\texttt{cons}} & \texttt{list}[R] \end{array}$$

Figura 6.5: Refinamiento de `add`

259
```
cons_es_refinamiento_add: THEOREM
  es_refinamiento_op?[[T, finite_set[T]], finite_set[T],
                      [R, list[R]], list[R],
                      prod_cart(f, c(f)), c(f)]
                     (add, cons)
```

6.3.2.3. Refinamiento de `member`

El predicado de pertenencia a una lista (`member`) no es, en general, un refinamiento del predicado de pertenencia a conjuntos finitos (`member`), puesto que si el refinamiento subyacente $f : R \longrightarrow T$ no es inyectivo, puede haber elementos $x \notin l$ tales que $f(x) \in c(f)(l)$. Por tanto, el diagrama correspondiente no sería commutativo.

Lema 6.13 *Si f es inyectivo, el predicado de pertenencia a una lista es un refinamiento del predicado de pertenencia a conjuntos finitos.*

$$\begin{array}{ccc} T \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{member}} & \texttt{bool} \\ f \times \texttt{c}(f) \uparrow & \# & id \uparrow \\ R \times \texttt{list}[R] & \xrightarrow{\texttt{member}} & \texttt{bool} \end{array}$$

Figura 6.6: Refinamiento de `member`

260

```
member_es_refinamiento_member: THEOREM
   injective?(f) ⇒
    es_refinamiento_op?[[T, finite_set[T]], bool,
                        [R,  list[R]], bool,
                        prod_cart(f, c(f)), id[bool]]
                       (member, member)
```

En los casos en los que f sea inyectivo tenemos, pues, que el predicado `member` refina a `member`. Esto comprende uno de los casos más frecuentes. Concretamente, el caso de representar los conjuntos finitos con elementos en T mediante listas de elementos de T. Ahora bien, no debemos olvidar que el objetivo que perseguimos es construir especificaciones evaluables de un algoritmo a partir de una especificcación abstracta del mismo. Y, puede ocurrir que para ello, sea necesario refinar también el tipo T, mediante una función no inyectiva [3].

En ese caso, hemos de disponer de un predicado sobre listas que refine a `member`. Para ello, se define un predicado de pertenencia a listas, módulo una relación de equivalencia, se caracterizan los elementos del conjunto $c(f)$ en función de dicho predicado, y se prueba que el predicado de pertenencia a listas, módulo la relación $\equiv_f$, refina a `member`.

[3]Por ejemplo, en el caso de que se necesite refinar un conjunto cuyos elementos son, a su vez, conjuntos.

261
```
rel: VAR (equivalence?[R])

member(rel)(x,l): RECURSIVE bool =
  CASES l OF
   null: FALSE,
   cons(a, l2): rel(a, x) ∨ member(rel)(x, l2)
  ENDCASES
  MEASURE length(l)
```

Lema 6.14 *Sea l una lista de elementos de R e y un elemento de T. Entonces,*

$$y \in c(f)(l) \Leftrightarrow \exists x \in_{\equiv_f} l : \ y = f(x)$$

262
```
imagen_c_caract_rel: LEMMA
   y ∈ c(f)(l) ⇔ ∃ x: member(igual?(f))(x,l) & y=f(x)
```

$$\begin{array}{ccc} T \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{member}} & \texttt{bool} \\ f \times \texttt{c}(f) \uparrow & \# & id \uparrow \\ R \times \texttt{list}[R] & \xrightarrow{\texttt{member}(\equiv_f)} & \texttt{bool} \end{array}$$

Figura 6.7: Refinamiento de `member`

Lema 6.15 *Si $f : R \longrightarrow T$ es un refinamiento de tipos, entonces el predicado* `member(igual?(f))` *refina al predicado* `member`.

263
```
member_rel_es_refinamiento_member: THEOREM
   es_refinamiento_op?[[T, finite_set[T]], bool,
                       [R, list[R]], bool,
                       prod_cart(f, c(f)), id[bool]]
                      (member, member(igual?(f)))
```

Evidentemente, en el caso en que se refine el tipo de los conjuntos finitos con elementos en T por listas de elementos de T de manera natural, es decir $f = id_T$, se tendrá que `member`$(\equiv_{id_T})$ es el predicado `member`. Más aún, se tiene que si f es inyectivo, entonces `member`$(\equiv_f)$ es `member`. Así, el resultado *260* se puede obtener como consecuencia de *263*.

```
member_member_rel: LEMMA
  injective?(f) => member(igual?(f)) = member
```

En lo que sigue, `rel` es una relación de equivalencia en R.

6.3.2.4. Refinamiento de `union`

Lema 6.16 *La operación* `append` *es un refinamiento de la operación* `union`.

$$\begin{array}{ccc} \texttt{finite\_set}[T] \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{union}} & \texttt{finite\_set}[T] \\ \texttt{c}(f) \times \texttt{c}(f) \uparrow & \# & \texttt{c}(f) \uparrow \\ \texttt{list}[R] \times \texttt{list}[R] & \xrightarrow{\texttt{append}} & \texttt{list}[R] \end{array}$$

Figura 6.8: Refinamiento de `union`

264

```
append_es_refinamiento_union: LEMMA
  es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                      [list[R], list[R]], list[R],
                      prod_cart(c(f),c(f)), c(f)]
                     (union, append)
```

6.3.2.5. Refinamiento de `interseccion`

A la hora de refinar la intersección de conjuntos finitos, hay que tener en cuenta que si definimos la operación `inter` sobre listas de manera natural usando el predicado `member`, sólo tendremos que esta operación refina a la intersección en el caso de que f sea injectivo. Como nos interesa tener un refinamiento de la intersección en todos los casos, seguimos el siguiente proceso.

En primer lugar, definimos la intersección de dos listas, tanto módulo una relación de equivalencia como directamente, y probamos que esta última es un caso particular de la primera.

264

```
inter(rel)(l1,l2): RECURSIVE list[T] =
    CASES l1 OF
     null: null,
     cons(x,ls): IF member(rel)(x, l2)
                    THEN cons(x, inter(rel)(ls, l2))
                    ELSE inter(rel)(ls, l2) ENDIF
    ENDCASES
    MEASURE length(l1)
```

```
inter(l1,l2): RECURSIVE list[T] =
    CASES l1 OF
     null: null,
     cons(x,ls): IF member(x, l2)
                   THEN cons(x, inter(ls, l2))
                   ELSE inter(ls, l2) ENDIF
    ENDCASES
    MEASURE length(l1)

inter_rel_id: LEMMA
    inter(=) = inter
```

En segundo lugar, probamos que la operación intersección, módulo la igualdad definida por f, ($\texttt{inter}_{\equiv_f}$), es un refinamiento de la intersección de conjuntos finitos. Como caso particular, se tiene que si f es inyectivo, entonces `inter` también refina la intersección de conjuntos finitos, puesto que en ese caso, $\texttt{inter}_{\equiv_f} = \texttt{inter}$.

$$\begin{array}{ccc} \texttt{finite\_set}[T] \times \texttt{finite\_set}[T] & \xrightarrow{\text{interseccion}} & \texttt{finite\_set}[T] \\ \texttt{c}(f) \times \texttt{c}(f) \uparrow & \# & \texttt{c}(f) \uparrow \\ \texttt{list}[R] \times \texttt{list}[R] & \xrightarrow{\text{inter}_{\equiv_f}} & \texttt{list}[R] \end{array}$$

Figura 6.9: Refinamiento de `intersection`

```
                                                                    265
inter_rel_es_refinamiento_intersection: THEOREM
    es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                        [list[R], list[R]], list[R],
                        prod_cart(c(f),c(f)), c(f)]
                       (intersection, inter(igual?(f)))

inter_inter_rel: LEMMA
    injective?(f) => inter(igual?(f)) = inter

inter_es_refinamiento_intersection: COROLLARY
    injective?(f) ⇒
    es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                        [list[R], list[R]], list[R],
                        prod_cart(c(f),c(f)), c(f)]
                       (intersection, inter)
```

El proceso que hemos seguido para refinar la operación intersección de conjuntos finitos será el que seguiremos para la mayoría de las operaciones con conjuntos finitos.

6.3.2.6. Refinamiento de `remove`

Para refinar la operación `remove` definimos, en primer lugar, las operaciones $\texttt{elimina}_{\equiv}$ y `elimina` y probamos que esta última es un caso particular de la primera.

266

```
elimina(rel)(x,l): RECURSIVE list[R] =
    CASES l OF
     null: null,
     cons(y,l2): IF rel(x, y)
                    THEN elimina(rel)(x, l2)
                    ELSE cons(y, elimina(rel)(x, l2)) ENDIF
    ENDCASES
    MEASURE length(l)

elimina(x,l): RECURSIVE list[R] =
    CASES l OF
     null: null,
     cons(y,l2): IF x=y
                    THEN elimina(x, l2)
                    ELSE cons(y, elimina(x, l2)) ENDIF
    ENDCASES
    MEASURE length(l)

elimina_rel_id: LEMMA
    elimina(=) = elimina
```

Probamos que la operación $\texttt{elimina}_{\equiv_f}$ es un refinamiento de la operación `remove` de conjuntos y, como caso particular, se tiene que si f es inyectivo, entonces `elimina` también lo es, puesto que en ese caso, $\texttt{elimina}_{\equiv_f} = \texttt{elimina}$.

$$\begin{array}{ccc} T \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{remove}} & \texttt{finite\_set}[T] \\ f \times \texttt{c}(f) \uparrow & \# & \texttt{c}(f) \uparrow \\ R \times \texttt{list}[R] & \xrightarrow{\texttt{elimina}_{\equiv_f}} & \texttt{list}[R] \end{array}$$

Figura 6.10: Refinamiento de `remove`

267
```
elimina_es_refinamiento_remove: THEOREM
    injective?(f) ⇒
    es_refinamiento_op?[[T, finite_set[T]], finite_set[T],
                        [R, list[R]], list[R],
                        prod_cart(f, c(f)), c(f)]
                       (remove, elimina)

elimina_elimina_rel: LEMMA
    injective?(f) ⇒ elimina(igual?(f)) = elimina

elimina_rel_es_refinamiento_remove: THEOREM
    es_refinamiento_op?[[T, finite_set[T]], finite_set[T],
                        [R, list[R]], list[R],
                        prod_cart(f, c(f)), c(f)]
                       (remove, elimina(igual?(f)))
```

6.3.2.7. Refinamiento de `choose`

En las especificaciones de algunos algoritmos usaremos la función `choose` de PVS, que elige un elemento de un conjunto no vacío. Por tanto, si luego queremos construir un refinamiento evaluable de ella, hemos de disponer de una función que refine a `choose`. Consideraremos, como función para elegir un elemento de una lista no vacía, la función `car` (i.e., elegimos el primer elemento de la lista). Ahora bien, como la función de elección de conjuntos no está determinada, hemos de introducir el siguiente axioma: "la función de elección `epsilon` sobre un conjunto de la forma $c(f)(l)$, siendo l una lista no vacía, elige el elemento $f(\texttt{car}(l))$".

268
```
car_epsilon: POSTULATE
    cons?(l) ⇒ epsilon(c(f)(l)) = f(car(l))
```

Hay que hacer notar que la función de elección de conjuntos sólo tiene sentido sobre conjuntos no vacíos. Por tanto, la operación sobre listas que la refine sólo estará definida sobre listas no vacías. Por ello, para tener el diagrama con los refinamientos de tipos correspondientes es necesario disponer de un refinamiento del tipo de los conjuntos finitos no vacíos por el de listas no vacías. Así, dado el refinamiento de tipos $f : R \to T$, definimos la función `cnv(f)`, que transforma una lista no vacía de elementos de R en un conjunto finito no vacío de elementos de T, y probamos que es refinamiento de tipos.

269
```
cnv(f)(l: (cons?[R])): non_empty_finite_set[T] = c(f)(l)

cnv_es_refinamiento_no_vacio: THEOREM
    es_refinamiento?[non_empty_finite_set[T], (cons?[R])](cnv(f))
```

De esta forma, estamos en disposición de probar que la función `car` refina a la función `choose`, restringida a los conjuntos finitos no vacíos.

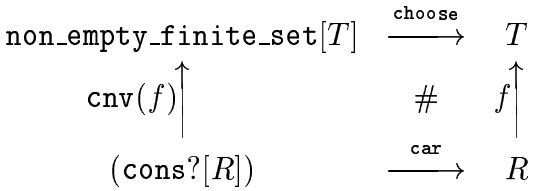


Figura 6.11: Refinamiento de `choose`

270

```
car_es_refinamiento_choose_2: THEOREM
    es_refinamiento_op?[non_empty_finite_set[T], T,
                        (cons?[R]), R, cnv(f), f]
                       (choose, car)
```

6.3.2.8. Refinamiento de `rest`

Asociada a la función de elección, se tiene la función `rest`, que elimina de un conjunto no vacío el elemento elegido por `choose`. Para refinarla, definimos las operaciones $\texttt{rest\_l}_{\equiv}$ y `rest_l` y probamos que esta última es un caso particular de la primera.

271

```
rest_l(rel)(l): list[R] =
    IF null?(l) THEN null ELSE elimina(rel)(car(l), l) ENDIF

rest_l(l): list[R] =
    IF null?(l) THEN null ELSE elimina(car(l), l) ENDIF

rest_l_rel_id: LEMMA
    rest_l(=) = rest_l
```

Probamos que la operación $\texttt{rest\_l}_{\equiv_f}$ es un refinamiento de la operación `rest` de conjuntos y, como caso particular, se tiene que si f es inyectivo, entonces `rest_l` también lo es, puesto que en ese caso, $\texttt{rest\_l}_{\equiv_f} = \texttt{rest\_l}$.

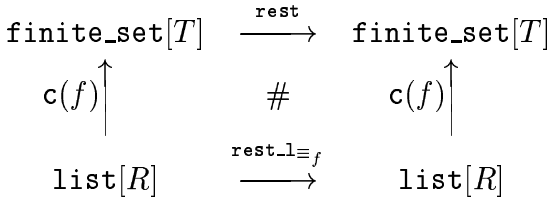


Figura 6.12: Refinamiento de `rest`

272

```
rest_l_rel_es_refinamiento_rest: THEOREM
    es_refinamiento_op?[finite_set[T], finite_set[T],
                        list[R], list[R], c(f), c(f)]
                       (rest, rest_l(igual?(f)))

rest_l_rest_l_rel: LEMMA
    injective?(f) => rest_l(igual?(f)) = rest_l

rest_l_es_refinamiento_rest: COROLLARY
    injective?(f) ⇒
    es_refinamiento_op?[finite_set[T], finite_set[T],
                        list[R], list[R], c(f), c(f)]
                       (rest, rest_l)
```

6.3.2.9. Refinamiento de `subset?`

Para refinar la operación `subset?` definimos, en primer lugar, las operaciones $\texttt{sublista?}_{\equiv}$ y `sublista?` y probamos que esta última es un caso particular de la primera.

272

```
sublista?(rel)(l1,l2): RECURSIVE bool =
    CASES l1 OF
     null: TRUE,
     cons(x, l): member(rel)(x, l2) & sublista?(rel)(l, l2)
    ENDCASES
    MEASURE length(l1)

sublista?(l1,l2): RECURSIVE bool =
    CASES l1 OF
     null: TRUE,
     cons(x, l): member(x, l2) & sublista?(l, l2)
    ENDCASES
    MEASURE length(l1)
```

```
sublista_rel_id: LEMMA
    sublista?(=) = sublista?
```

Probamos que la operación $\texttt{sublista?}_{\equiv_f}$ es un refinamiento de la operación `subset?` de conjuntos y, como caso particular, se tiene que si f es inyectivo, entonces `sublista?` también lo es, puesto que en ese caso, $\texttt{sublista?}_{\equiv_f} = \texttt{sublista?}$.

$$\begin{array}{ccc} \texttt{finite\_set}[T] \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{subset?}} & \texttt{bool} \\ \texttt{c}(f) \times \texttt{c}(f) \uparrow & \# & id \uparrow \\ \texttt{list}[R] \times \texttt{list}[R] & \xrightarrow{\texttt{sublista?}_{\equiv_f}} & \texttt{bool} \end{array}$$

Figura 6.13: Refinamiento de `subset?`

273

```
sublista?_rel__es_refinamiento_subset?: THEOREM
    es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                        [list[R], list[R]], bool,
                        prod_cart(c(f),c(f)), id[bool]]
                       (subset?, sublista?(igual?(f)))

sublista_sublista_rel: LEMMA
    injective?(f) ⇒ sublista?(igual?(f)) = sublista?

sublista?_es_refinamiento_subset?: COROLLARY
    injective?(f) ⇒
    es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                        [list[R], list[R]], bool,
                        prod_cart(c(f),c(f)), id[bool]]
                       (subset?, sublista?)
```

Además, a partir del predicado `sublista?` se define `sublista_estricto?` y se prueba que refina a `strict_subset?`.

6.3.2.10. Refinamiento de la igualdad de conjuntos finitos

En primer lugar, definimos un predicado de igualdad entre conjuntos finitos, susceptible de ser refinado por un predicado sobre listas, y probamos que es una relación de equivalencia.

```
igual_c?(A,B:finite_set[T]):bool = A = B

igual_c?_es_de_equivalencia: LEMMA
    equivalence?[finite_set[T]](igual_c?)
```
274

Para refinarlo definimos, en primer lugar, los predicados igual_l?$_{\equiv}$ e igual_l? y probamos que este último es un caso particular del primero.

```
igual_l?(rel)(l1,l2): bool =
   sublista?(rel)(l1, l2) & sublista?(rel)(l2, l1)

igual_l?(l1,l2): bool = sublista?(l1, l2) & sublista?(l2, l1)

igual_l_rel_id: LEMMA  igual_l?(=) = igual_l?
```
275

Después, probamos que el predicado igual_l?$_{\equiv_f}$ es un refinamiento del predicado igual_c? de conjuntos y, como caso particular, se tiene que si f es injectivo, entonces igual_l? también lo es, puesto que en ese caso, igual_l?$_{\equiv_f}$ = igual_l?.

$$\begin{array}{ccc} \texttt{finite\_set}[T] \times \texttt{finite\_set}[T] & \xrightarrow{\texttt{igual\_c?}} & \texttt{bool} \\ \uparrow \texttt{c}(f) \times \texttt{c}(f) & \# & \uparrow id \\ \texttt{list}[R] \times \texttt{list}[R] & \xrightarrow{\texttt{igual\_l?}_{\equiv_f}} & \texttt{bool} \end{array}$$

Figura 6.14: Refinamiento de igual_c?

```
igual_l?_rel_es_refinamiento_igual_c: THEOREM
    es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                        [list[R], list[R]], bool,
                        prod_cart(c(f),c(f)), id[bool]]
                       (igual_c?, igual_l?(igual?(f)))

igual_l_igual_l_rel: LEMMA
    injective?(f) ⇒ igual_l?(igual?(f)) = igual_l?

igual_l?_es_refinamiento_igual_c: COROLLARY
    injective?(f) ⇒
    es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                        [list[R], list[R]], bool,
                        prod_cart(c(f),c(f)), id[bool]]
                       (igual_c?, igual_l?)
```
276

6.3.2.11. Refinamiento del cardinal

Definimos el cardinal de una lista, módulo una relación de equivalencia, como el número de elementos no equivalentes de la lista. Para ello, eliminamos de la lista los elementos equivalentes según la relación, y calculamos su longitud.

277
```
elimina_duplicados(rel)(l): RECURSIVE list[T] =
    CASES l OF
     null: null,
     cons(x, l1): IF member(rel)(x, l1)
                     THEN elimina_duplicados(rel)(l1)
                     ELSE cons(x, elimina_duplicados(rel)(l1)) ENDIF
    ENDCASES
    MEASURE length(l)

cardinal_l(rel)(l): nat =
    length(elimina_duplicados(rel)(l))
```

Análogamente, se define la operación `cardinal_l` y se prueba que es un caso particular de la anterior.

278
```
cardinal_l_rel_id: LEMMA
    cardinal_l(=) = cardinal_l
```

Probamos que la operación $\texttt{cardinal\_l}_{\equiv_f}$ refina a la función cardinal de un conjunto finito:

$$\begin{array}{ccc} \texttt{finite\_set}[T] & \xrightarrow{\texttt{card}} & \mathbb{N} \\ c(f)\uparrow & \# & id\uparrow \\ \texttt{list}[R] & \xrightarrow{\texttt{cardinal\_l}_{\equiv_f}} & \mathbb{N} \end{array}$$

Figura 6.15: Refinamiento de `card`

Para ello, hay que considerar la especificación de la función cardinal realizada en el preludio, donde se define el cardinal de un conjunto finito S como el mínimo de los $n \in \mathbb{N}$ para los que existe una inyección de S en $\{0, 1, \ldots, n-1\}$.

Probamos que si una lista `l` no contiene elementos equivalentes por una determinada relación de equivalencia, entonces existe una biyección entre el conjunto `c(f)(ls)` y el conjunto $\{0, 1, \ldots, n-1\}$, siendo n la longitud de `l`.

279
```
sin_duplicados?(rel)(l): RECURSIVE bool =
    CASES l OF
     null: TRUE,
     cons(x, l1): NOT member(rel)(x, l1) & sin_duplicados?(rel)(l1)
    ENDCASES
    MEASURE length(l)

sin_duplicados_rel_bij_conj_2: LEMMA
    sin_duplicados?(igual?(f))(l) ⇒
    ∃ (g: [(c(f)(l)) -> below[length(l)]]): bijective?(g)
```

Demostración:

La prueba se hace por inducción en l

- `l = null`:

  Basta tener en cuenta que `c(f)(null)` es el conjunto vacío y considerar la función g idénticamente nula.

- `l = (r, ls)` y se verifica la hipótesis de inducción para `ls`.

  Si `l` no tiene elementos repetidos, por la definición de `sin_duplicados?`, `ls` no tiene elementos repetidos. Entonces, por hipótesis de inducción, existe una biyección g_1 : $\mathtt{c(f)(ls)} \to \mathtt{below[length(ls)]}$. Usando esta función, definimos $g : \mathtt{c(f)(cons(r, ls))} \to \mathtt{below[1 + length(ls)]}$ como sigue:

$$g(y) = \begin{cases} \mathtt{length(ls)}, & \text{si } \mathtt{y} = \mathtt{f(r)} \\ g_1(\mathtt{y}), & \text{si } \mathtt{y} \in \mathtt{c(f)(ls)} \end{cases}$$

  y probamos que g es biyectiva.

□

Probamos también que la lista $\mathtt{elimina\_duplicados}_{\equiv_f}\mathtt{(ls)}$ no contiene elementos equivalentes y que representa el mismo conjunto que la lista `ls`.

280
```
elimina_rel_sin_duplicados: LEMMA
   sin_duplicados?(rel)(elimina_duplicados(rel)(l))

elimina_duplicados_rel_igual_conj: LEMMA
   c(f)(elimina_duplicados(igual?(f))(l)) = c(f)(l)
```

Por último, probamos que $\mathtt{cardinal\_l}_{\equiv_f}$ es un refinamiento de `card` y, si f es injectivo, que `cardinal_l` también lo es, pues en ese caso, $\mathtt{cardinal\_l}_{\equiv_f} = \mathtt{cardinal\_l}$.

281
```
cardinal_l_rel_es_refinamiento_card: THEOREM
   es_refinamiento_op?[finite_set[T], nat,
                       list[R], nat, c(f), id[nat]]
                      (card, cardinal_l(igual?(f)))

cardinal_l_caardinal_l_rel: LEMMA
   injective?(f) ⇒ cardinal_l(igual?(f)) = cardinal_l

cardinal_l_es_refinamiento_card: COROLLARY
   injective?(f) ⇒
   es_refinamiento_op?[finite_set[T], nat,
                       list[R], nat, c(f), id[nat]]
                      (card, cardinal_l)
```

6.3.2.12. Refinamiento de `Union`

En el preludio, la función `Union` tiene como dominio el tipo `set[set[T]]`. Para poder refinarla mediante listas, es necesario restringir el dominio de la función `Union` a conjuntos finitos de conjuntos finitos, y el rango a conjuntos finitos. Por esta razón, definimos la función `union_f` como sigue:

282
```
union_f(S: finite_set[finite_set[T]]): finite_set[T] =
    Union(S)
```

Para refinarla, definimos recursivamente la concatenación generalizada de una lista de listas y probamos que es un refinamiento de `union_f`:

283
```
Append(LL): RECURSIVE list[T] =
    CASES LL OF
     null: null,
     cons(ls, LL2): append(ls, Append(LL2))
    ENDCASES
    MEASURE length(LL)

append_g_es_refinamiento_union: THEOREM
    es_refinamiento_op?[finite_set[finite_set[T]], finite_set[T],
                        list[list[R]], list[R],
                        c(c[T, R](f)), c(f)]
                       (union_f, Append)
```

$$\begin{array}{ccc} \texttt{finite\_set}[\texttt{finite\_set}[T]] & \xrightarrow{\texttt{union\_f}} & \texttt{finite\_set}[T] \\ \mathsf{c}(\mathsf{c}(f))\uparrow & \# & \mathsf{c}(f)\uparrow \\ \texttt{list}[\texttt{list}[R]] & \xrightarrow{\texttt{Append}} & \texttt{list}[R] \end{array}$$

Figura 6.16: Refinamiento de la unión generalizada

6.3.2.13. Refinamiento de `Interseccion`

En el preludio de PVS la función `Intersection` está definida para cualquier conjunto de conjuntos. Obviamente, se tiene que la intersección del conjunto vacío es el conjunto universal, representado en PVS por `fullset`. Para poder refinarla mediante unaa operación sobre listas, es necesario restringir el dominio de la función `Intersection` a conjuntos finitos no vacíos de conjuntos finitos, pues no se dispone de una lista que represente al conjunto universal. Por tanto, para poder expresar adecuadamente la noción de refinamiento, definimos la operación `intersection_f` como sigue:

284
```
intersection_f(S:non_empty_finite_set[finite_set[T]]): finite_set[T] =
    Intersection(S)
```

Para refinarla, definimos las operaciones $\texttt{Inter}_{\equiv}$ e `Inter` y probamos que `Inter` es un caso particular de $\texttt{Inter}_{\equiv}$.

285
```
Inter(rel)(LL: (cons?[list[T]])): RECURSIVE list[T] =
    IF null?(cdr(LL))
       THEN car(LL)
       ELSE inter(rel)(car(LL), Inter(rel)(cdr(LL))) ENDIF
    MEASURE length(LL)

Inter(LL: (cons?[list[T]])): RECURSIVE list[T] =
    IF null?(cdr(LL))
       THEN car(LL)
       ELSE inter(car(LL), Inter(cdr(LL))) ENDIF
    MEASURE length(LL)

Inter_rel_id: LEMMA
    Inter(=) = Inter
```

Probamos que $\texttt{Inter}_{\equiv_f}$ es un refinamiento de `intersection_f` y, como caso particular en el caso de que f sea inyectivo, que `Inter` también lo es.

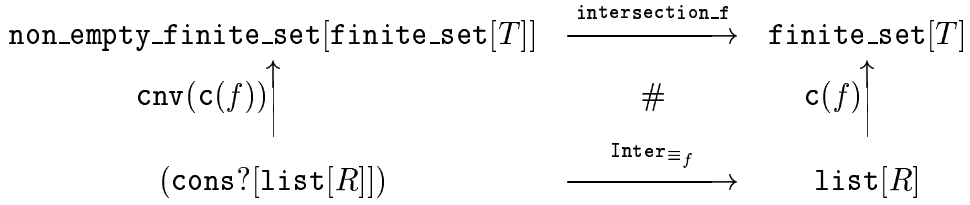


Figura 6.17: Refinamiento de la intersección generalizada

286
```
inter_g_es_refinamiento_intersection_rel: THEOREM
    es_refinamiento_op?[non_empty_finite_set[finite_set[T]],
                        finite_set[T],
                        (cons?[list[R]]), list[R],
                        cnv(c[T,R](f)), c(f)]
                       (intersection_f , Inter(igual?(f)))
```

6.3.2.14. Refinamiento del conjunto potencia

El dominio de la operación `powerset` definida en el preludio es el tipo de los conjuntos. Queremos refinar esta operación mediante listas. Por ello, es necesario restringir el dominio y el rango a conjuntos finitos. Lo hacemos definiendo la operación `powerset_f` como sigue:

287
```
powerset_f(A: finite_set[T]): finite_set[finite_set[T]] =
    powerset(A)
```

Para refinarla, hemos de disponer de la noción de potencia de una lista. Dada una lista `ls`, diremos que una lista de listas LL es una **potencia** de `ls` si para toda lista contenida en `ls`, existe una lista en LL con exactamente los mismos elementos que ella, y para cada elemento de LL, existe una lista contenida en `ls` con los mismos elementos. Obviamente, dada `ls` no existe una única LL que sea su potencia. Sólo podemos considerar que la potencia de una lista es única, módulo la relación de igualdad que considera como listas iguales las que tienen los mismos elementos.

Especificamos un prodecimiento para obtener una potencia de una lista, de la forma siguiente [4] :

[4] La expresión en PVS de una lista, detallando sus elementos se hace mediante una expresión de la forma `(: a, b, c :)`.

288
```
incluye_en_cada(x,LL): RECURSIVE list[list[T]] =
    CASES LL OF
     null: null,
     cons(l1, LL2): cons(cons(x, l1), incluye_en_cada(x, LL2))
    ENDCASES
    MEASURE length(LL)

powerset_ref_2(ls): RECURSIVE list[list[T]] =
    CASES ls OF
     null: (: null :),
     cons(x, l2): LET aux = powerset_ref_2(l2) IN
                    append(incluye_en_cada(x, aux), aux)
    ENDCASES
    MEASURE length(ls)
```

Probamos que, en efecto, esta especificación construye una potencia de una lista dada:

289
```
powerset_ref_2_cs1_rel: LEMMA
     sublista?(igual?(f))(l1, l2) ⇒
     ∃ l3: member(igual?(c(f)))(l3, powerset_ref_2(l2)) &
                igual?(c(f))(l1, l3)
```

Como consecuencia, probamos que `powerset_ref_2` es un refinamiento de `powerset_f`.

$$\begin{array}{ccc} \texttt{finite\_set}[T] & \xrightarrow{\texttt{powerset\_f}} & \texttt{finite\_set}[\texttt{finite\_set}[T]] \\ \texttt{c}(f)\uparrow & \# & \texttt{c}(\texttt{c}(f))\uparrow \\ \texttt{list}[R] & \xrightarrow{\texttt{powerset\_ref\_2}} & \texttt{list}[\texttt{list}[R]] \end{array}$$

Figura 6.18: Refinamiento del conjunto potencia

290
```
powerset_ref_2_es_refinamiento_g: THEOREM
    es_refinamiento_op?[finite_set[T], finite_set[finite_set[T]],
                        list[R], list[list[R]],
                        c(f), c(c(f))](powerset_f, powerset_ref_2)
```

6.3.2.15. Refinamiento del conjunto imagen

Dada una función $g : T_1 \to T_2$ y un conjunto de elementos de T_1, S, la función `image` del preludio obtiene el conjunto imagen de S por g, `image(g)(S)`. En el caso de conjuntos finitos, se tiene la aplicación

$$\texttt{image(g)} : \texttt{finite\_set[T1]} \to \texttt{finite\_set[T2]}$$

Probamos que si $f_i : R_i \to T_i$, $i = 1, 2$ son refinamientos de tipos y $h : R_1 \to R_2$ refina a g, entonces la función `map(h)` refina a la función `image(g)`:

291

```
map_es_refinamiento_image: THEOREM
  es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) ⇒
  es_refinamiento_op?[finite_set[T1], finite_set[T2],
                      list[R1], list[R2],
                      c(f1), c(f2)]
                     (image(g), map(h))
```

$$\begin{array}{ccc} \texttt{finite\_set}[T_1] & \xrightarrow{\texttt{image(g)}} & \texttt{finite\_set}[T_2] \\ \texttt{c}(f_1)\uparrow & \# & \texttt{c}(f_2)\uparrow \\ \texttt{list}[R_1] & \xrightarrow{\texttt{map(h)}} & \texttt{list}[R_2] \end{array}$$

Figura 6.19: Refinamiento del conjunto imagen

Por último, comentemos que una vez establecido que determinadas operaciones entre listas refinan a operaciones sobre conjuntos finitos, muchas de las propiedades de dichas operaciones sobre listas se han probado a través del refinamiento. Entre ellas, destacamos las siguientes:

```
cardinal_l_plus: THEOREM
  cardinal_l(l1) + cardinal_l(l2) =
  cardinal_l(append(l1,l2)) + cardinal_l(inter(l1, l2))

sublista_estricto?_cardinal_l: LEMMA
   sublista_estricto?(l1, l2) ⇒ cardinal_l(l1) < cardinal_l(l2)

map_conserva_igual_l?: LEMMA
  es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g,h) ⇒
  (igual_l?(igual?(f1))(l1,l2) ⇒
  igual_l?(igual?(f2))(map(h)(l1), map(h)(l2)))
```

Capítulo 7

Una formalización evaluable de la lógica proposicional

Las especificaciones de algoritmos realizadas en la formalización de la lógica proposicional, descrita en los capítulos 3 y 4 de este trabajo, no son evaluables. La razón es que hemos representado las cláusulas y los programas usando el tipo de los conjuntos finitos. Nuestro objetivo ahora es construir especificaciones evaluables de dichos algoritmos, basándonos en el marco para refinamientos descrito en el capítulo 6. Para ello, vamos a seguir los siguientes pasos:

1. Realizaremos una representación de las cláusulas de Horn, los programas definidos y las interpretaciones, usando listas (desarrollada en la teoría `clausulas_l` [páginas 338–339]).

2. Presentaremos una especificación genérica de funciones que transforman las representaciones anteriores en las representaciones de los mismos conceptos, basada en conjuntos finitos (en la teoría `relacion_rep_clausulas` [páginas 339–344]).

3. A través de dichas funciones de transformación, introduciremos la semántica de cláusulas representadas mediante listas (desarrollada en la teoría `semantica_clausulas_l` [páginas 347–350]).

4. Realizaremos especificaciones evaluables de los algoritmos de la lógica proposicional, descritos en el capítulo 4, usando las nociones de refinamiento del capítulo anterior. Concretamente:

   - Especificaremos un algoritmo de cálculo de la base de Herbrand de un programa definido, y probaremos que es un refinamiento evaluable de la especificación de la base de Herbrand, descrita previamente (desarrollada en la teoría `base_herbrand_l` [páginas 350–352]).
   - Análogamente para el menor modelo de Herbrand de un programa (desarrollada en la teoría `menor_modelo_herbrand_l` [páginas 352–355]).

- Especificaremos un algoritmo de cálculo del operador de consecuencia inmediata y probaremos que es un refinamiento evaluable del operador de consecuencia, especificado usando conjuntos finitos (desarrollado en la teoría `consecuencia_i_l` [páginas 355–356]).
- Especificaremos dos algoritmos para calcular el menor punto fijo de un programa definido y probaremos que son refinamientos evaluables de la función que define el menor punto fijo de un programa, especificada en la teoría `semantica_p_f` (desarrollados en las teorías `menor_punto_fijo_l` y `menor_punto_fijo_iterado_l` [pág. 357–360]).
- En cuanto a los algoritmos de refutación, en el capítulo 4 hemos descrito en qué consiste el proceso de resolución, pero no se ha especificado un algoritmo o una función, susceptible de ser refinada. Ahora bien, sí hemos probado la independencia de la regla de computación. Por tanto, los distintos algoritmos de refutación dependerán, fundamentalmente, del proceso de búsqueda usado para encontrar el objetivo vacío. En este capítulo, especificaremos distintos procesos de refutación, usando búsqueda en anchura, en profundidad y en profundidad iterativa (desarrollados `sld_resolucion_l`, `sld_resolucion_l_3` y `sld_resolucion_l_via_r_2` (páginas 360–367).

7.1. Una formalización evaluable

7.1.1. Representación

Consideremos un tipo no vacío `D`, que representa directamente el conjunto de átomos o símbolos proposicionales. Representamos las cláusulas como pares cuyos elementos son listas de átomos, considerando el primer elemento del par como la *cabeza* de la cláusula y el segundo elemento como su *cuerpo*:

292
```
clausula: TYPE = [list[T], list[T]]
cabeza(C: clausula): list[T] = PROJ_1(C)
cuerpo(C: clausula): list[T] = PROJ_2(C)
```

Entendemos que una cláusula es de Horn si su cabeza tiene como máximo un elemento, es una cláusula definida si su cabeza tiene exactamente un elemento y es un objetivo definido si su cabeza es la lista vacía. Si, además, su cuerpo es la lista vacía, entonces representa el objetivo vacío.

293
```
es_cl_horn(cl:clausula): bool = length(cabeza(cl)) <= 1
cl_horn: TYPE = (es_cl_horn)

es_cl_definida(cl:cl_horn): bool = length(cabeza(cl)) = 1
cl_definida: TYPE = (es_cl_definida)
```

```
es_objetivo_definido(cl:cl_horn): bool = null?(cabeza(cl))
objetivo_definido: TYPE = (es_objetivo_definido)

es_vacio(G:objetivo_definido): bool = null?(cuerpo(G))
```
294

Representamos un programa definido mediante una lista de cláusulas definidas:

```
programa: TYPE = list[cl_definida]
```
295

En cuanto a las interpretaciones, desde el punto de vista de la lógica proposicional son, esencialmente, conjuntos de átomos. Si los algoritmos que se desea hacer evaluables, no dependen directamente del uso de una representación evaluable de la noción de interpretación, podemos seguir considerándolas como conjuntos o conjuntos finitos de átomos. En cualquier caso, dispondremos también de una representación evaluable del concepto de interpretación como lista de átomos, puesto que el algoritmo para calcular el menor punto fijo de un programa hace uso de ellas.

7.1.2. Relación entre las dos representaciones

En esta subsección se presenta una especificación genérica de las transformaciones de las dos representaciones de cláusulas de Horn, programas definidos e interpretaciones establecidas: una basada en el tipo de dato listas y otra basada en conjuntos finitos. Además, se describen mediante axiomas las propiedades que deben verificar dichas transformaciones.

Denotamos por $\mathcal{T}_1$ la teoría desarrollada en `clausulas_l[T]`, que contiene la representación basada en listas, y por $\mathcal{T}_2$ la teoría desarrollada en `clausulas[T]`, que contiene la representación basada en conjuntos finitos.

Básicamente, hemos de establecer una relación entre las dos representaciones de los átomos y de los distintos tipos de cláusulas.

Concepto	Tipo	Tipo	Función
Átomo	D	atomo[T]	transf_atomo
Cláusula definida	cl_definida[D]	clausula_def[T]	transf_clausula_def
Objetivo definido	objetivo_definido[D]	objetivo_def[T]	transf_objetivo_def
Cláusula de Horn	cl_horn[D]	clausula_horn[T]	transf_clausula_horn

La representación basada en conjuntos finitos es más general que la que usa listas. O, dicho de otra forma, la representación basada en listas es más fina. Así, un mismo concepto (p.e. interpretación finita) tiene más de una representación como lista y sólo una como conjunto finito. Por ello, las funciones que transformen

la representación de un concepto usando listas en otra representación del mismo usando conjuntos finitos habrán de ser, en general, sobreyectivas.

Ahora bien, los átomos son las unidades básicas de cada representación. Como queremos que el universo sobre el que se construye los programas sea el mismo, exigimos que la función que relaciona las dos representaciones de átomos sea biyectiva.

```
transf_atomo_ax1: AXIOM surjective?(transf_atomo)
transf_atomo_ax2: AXIOM injective?(transf_atomo)

transf_clausula_def_ax1: AXIOM surjective?(transf_clausula_def)

transf_objetivo_def_ax1: AXIOM surjective?(transf_objetivo_def)

transf_clausula_horn_ax1: AXIOM surjective?(transf_clausula_horn)
```

También es necesario especificar axiomas que relacionan, básicamente, estas funciones. Por ejemplo, como el tipo `cl_definida[D]` es un subtipo de `cl_horn[D]`, establecemos como axioma que si C es una cláusula definida de $\mathcal{T}_1$, sus transformadas por `transf_clausula_def` y `transf_clausula_horn` coinciden, y el correspondiente axioma de subtipo. Análogamente, para objetivos definidos y para el objetivo vacío.

```
transf_cabeza_clausula_def: AXIOM
    ∀(C: cl_definida[D]):
     transf_atomo(car(cabeza(C))) = cabeza(transf_clausula_def(C))

transf_clausula_horn_ax2: AXIOM
  ∀(C:cl_definida[D]):transf_clausula_horn(C) = transf_clausula_def(C)

transf_clausula_horn_ax3: AXIOM
  ∀ (C: objetivo_definido[D]):
    transf_clausula_horn(C) = transf_objetivo_def(C)

transf_clausula_horn_def_ax: AXIOM
  ∀ (C: cl_definida[D]): es_clausula_def(transf_clausula_horn(C))

transf_clausula_horn_obj_ax: AXIOM
  ∀ (C: objetivo_definido[D]):es_objetivo_def(transf_clausula_horn(C))

transf_clausula_horn_obj_vacio_ax: AXIOM
  ∀ (G: (es_vacio[D])): vacia?(transf_clausula_horn(G))
```

Por último, los siguientes axiomas hacen referencia a los transformados de los átomos que forman la cabeza y el cuerpo de las cláusulas.

```
transf_cabeza_cl_def: AXIOM
  ∀(C: cl_definida[D]):
     transf_atomo(car(cabeza(C))) = cabeza(transf_clausula_horn(C))

transf_cuerpo: AXIOM
  ∀(CH: cl_horn[D], x:D):
  x∈cuerpo(CH) ⇒ transf_atomo(x) ∈ cuerpo(transf_clausula_horn(CH))

transf_cuerpo_2: AXIOM
  ∀(CH: cl_horn[D], x:D):
  transf_atomo(x) ∈ cuerpo(transf_clausula_horn(CH)) ⇒ x∈cuerpo(CH)
```

A partir de éstas, usando la función *255* que transforma listas en conjuntos finitos, tendremos determinadas las funciones que transforman programas, listas de cláusulas de Horn e interpretaciones.

Tipo	Tipo	Función
clausulas_l[D].programa	clausulas[T].programa	c(transf_clausula_def)
list[cl_horn[D]]	finite_set[clausula_horn[T]]	c(transf_clausula_horn)
list[D]	interpretacion_finita[T]	c(transf_atomo)

En este caso, las propiedades de la función *255* determinan las propiedades de estas funciones. El único axioma que imponemos es para asegurar que a través de las funciones `transf_clausula_horn` y `transf_interp` se conserva la propiedad de que el cuerpo de una cláusula esté contenido en una interpretación.

```
transf_interp_ax3: AXIOM
  ∀ (I: list[D], C: cl_definida[D]):
    sublista?(cuerpo(C), I) ⇔
    cuerpo(transf_clausula_horn(C)) ⊆ transf_interp(I)
```

7.1.3. Semántica

En esta subsección se introduce la semántica de la teoría representada mediante listas, a través de la semántica establecida para la representación basada en conjuntos finitos, utilizando para ello las funciones de transformación descritas previamente. La formalización en PVS se encuentra desarrollada en la teoría `semantica_clausulas_l` (páginas 347–350).

Obsérvese que en la lógica proposicional, una interpretación es, en esencia, un conjunto finito o infinito, de átomos. Así lo hemos considerado en la representación de la lógica proposicional realizada en la teoría $\mathcal{T}_2$. En la teoría $\mathcal{T}_1$ estamos desarrollando otra representación de la lógica proposicional, de forma que:

- sea equivalente a la anterior (lo que, en este caso, quiere decir que las especificaciones contenidas en $\mathcal{T}_1$ sean refinamientos de las correspondientes especificaciones de $\mathcal{T}_2$).
- sea evaluable.

Ahora bien, la cuestión es si todas las nociones que definamos en $\mathcal{T}_1$ han de ser evaluables, o bien si disponer de algunas definiciones no evaluables que sólo usaremos para razonar, y únicamente tener definiciones evaluables de algunas funciones. En este sentido, en relación con la definición de modelo de una cláusula (o de un programa) podemos:

- especificar cuándo un conjunto de átomos de $\mathcal{T}_1$ es modelo (no evaluable)
- especificar cuándo una lista de átomos de $\mathcal{T}_1$ es modelo (no evaluable)
- especificar cuándo una lista de átomos de $\mathcal{T}_1$ es modelo (evaluable)

En el caso de las especificaciones no evaluables lo hacemos a través de las funciones que constituyen el refinamiento, con lo que se tiene, directamente, que son equivalentes a las definiciones realizadas en $\mathcal{T}_2$. En el caso de la especificación evaluable se define directamente, usando la estructura de lista en la que está basada la representación, y después se prueba que es refinamiento de la especificación hecha en $\mathcal{T}_2$.

Como hemos comentado, la especificación evaluable de modelo se necesita para la especificación de un algoritmo que calcula el menor modelo de Herbrand de un programa de $\mathcal{T}_1$. Para los demás algoritmos es suficiente disponer de la especificación no evaluable de dicho concepto.
En lo que sigue, usaremos la siguiente notación:

```
CH:       VAR cl_horn[D]
C:        VAR cl_definida[D]
P,P1,P2: VAR clausulas_l[D].programa
I,I1,I2: VAR interpretacion[T] % conjunto de átomos de T2
Int,Int1:VAR set[D] % interpretaciones de T1
L,L1,L2: VAR list[D] % interpretaciones (evaluables) de T1
FSC:      VAR finite_set[cl_horn[D]]
LSC:      VAR list[cl_horn[D]]
```

Definición 7.1 *Modelo de una cláusula:*

- *Int es modelo de CH si su imagen por* `transf_atomo` *es modelo de la transformada de CH por* `transf_clausula_horn`*:*

```
es_modelo(Int,CH): bool =
  es_modelo(image(transf_atomo)(Int), transf_clausula_horn(CH))
```

- *L es modelo de CH si el conjunto* `transf_interp(L)` *es modelo de la transformada de CH por* `transf_clausula_horn`*:*

```
es_modelo(L,CH): bool =
  es_modelo(transf_interp(L), transf_clausula_horn(CH))
```

- *L es modelo (noción evaluable) de CH si:*
  - *existe algún átomo del cuerpo de CH que no pertenece a L, en el caso en que la cabeza de CH sea la lista nula, o bien*
  - *el átomo de la cabeza de CH pertenece a L o existe algún átomo del cuerpo de CH que no pertenece a L, en otro caso.*

```
es_modelo_e(L,CH): bool =
    IF null?(cabeza(CH))
    THEN some(LAMBDA(x): x ∉ L)(cuerpo(CH))
    ELSE car(cabeza(CH)) ∈ L ∨
         some(LAMBDA(x): x ∉ L)(cuerpo(CH))
    ENDIF
```

Análogamente, para cláusulas definidas y objetivos definidos:

```
es_modelo_e(L,C): bool =
  car(cabeza(C)) ∈ L ∨ some(LAMBDA(x): x ∉ L)(cuerpo(C))

es_modelo_e(L,G): bool = some(LAMBDA(x): x ∉ L)(cuerpo(G))
```

Definición 7.2 *Modelo de un programa:*

- *L es modelo de P si el conjunto* `transf_interp(L)` *es modelo del transformado de P por* `transf_programa`*.*
- *L es modelo (noción evaluable) de P si L es modelo de todas las cláusulas de P.*

```
es_modelo(L,P): bool = es_modelo(transf_interp(L),transf_programa(P))

es_modelo_e(L,P): bool = every(LAMBDA(C): es_modelo_e(L,C))(P)
```

Probamos que las definiciones evaluables de *modelo* son refinamientos de las nociones especificadas en el capítulo 4, como se muestra en las figuras 7.1 y 7.2.

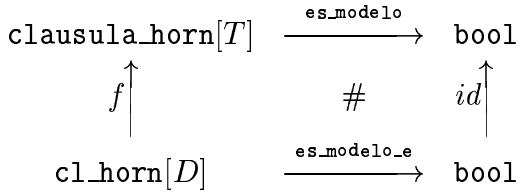


Figura 7.1: Refinamiento de la función `es_modelo`. Se ha denotado por f la función `transf_clausula_horn`.

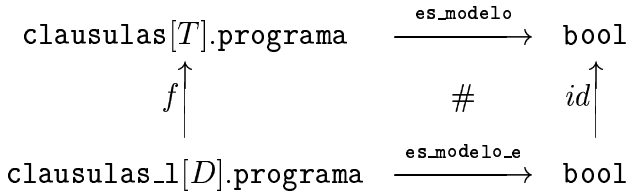


Figura 7.2: Refinamiento de la función `es_modelo`. Se ha denotado por f la función `transf_programa`.

Por último, especificamos las nociones (no evaluables) de consecuencia lógica e insatisfacibilidad, a través de las funciones de transformación, como sigue:

```
es_cons_logica(x,LSC): bool =
  es_cons_logica(transf_atomo(x), transf_lista_cl_horn(LSC))

es_cons_logica(x,P): bool =
  es_cons_logica(transf_atomo(x), transf_programa(P))

es_insatisfacible(LSC): bool =
  es_insatisfacible(transf_lista_cl_horn(LSC))
```

7.1.4. Interpretación

En esta subsección usamos la capacidad de PVS de interpretar teorías para establecer una interpretación de la teoría `relacion_rep_clausulas` (páginas 339–344), en la que concretamos las funciones no interpretadas declaradas en dicha teoría. Dicha interpretación se ha especificado en PVS en la teoría `relacion_rep_clausulas_interpretacion` (páginas 344–347).

Definimos las siguientes funciones que usamos para interpretar las funciones declaradas previamente:

```
296
transf_a_1(x:T):atomo[T] = a(x)

transf_clausula_def_1(C:cl_definida[T]): clausula_def[T] =
    (# cabeza:= transf_a_1(car(cabeza(C))),
       cuerpo:= re_atomo.c(transf_a_1)(cuerpo(C)) #)

transf_objetivo_def_1(C: objetivo_definido[T]): objetivo_def[T] =
    (# cabeza:= falso,
       cuerpo:= re_atomo.c(transf_a_1)(cuerpo(C)) #)

transf_cl_h_1(C:cl_horn[T]):  clausula_horn[T] =
    (# cabeza:= IF cons?(proj_1(C))
                THEN transf_a_1(car(cabeza(C)))
                ELSE falso ENDIF,
       cuerpo:= re_atomo.c(transf_a_1)(cuerpo(C)) #)
```

Con ello, la interpretación de la teoría `relacion_rep_clausulas` es:

```
297
IMPORTING relacion_rep_clausulas[T,T]
            {{transf_atomo:=         transf_a_1,
              transf_clausula_def:=  transf_clausula_def_1,
              transf_objetivo_def:=  transf_objetivo_def_1,
              transf_clausula_horn:= transf_cl_h_1}}
```

Al realizar esta interpretación, todos los axiomas impuestos sobre estas funciones se transforman en obligaciones de prueba, que hemos de demostrar (son las 17 obligaciones `IMP_*_TCCn` de las páginas 344–347). Las pruebas se hacen usando las propiedades de la función 255, que definimos en el capítulo 6 para refinar el tipo de los conjuntos finitos.

7.2. Especificaciones evaluables

7.2.1. Algoritmo de cálculo de la base de Herbrand

En esta subsección se especifica un algoritmo de cálculo de la base de Herbrand de un programa, y se prueba que es un refinamiento evaluable de la especificación de la base de Herbrand del capítulo 4.

En primer lugar, definimos una función que calcula una lista con los símbolos proposicionales de una cláusula de Horn, teniendo en cuenta que dicha lista se obtiene concatenando la cabeza de la cláusula con su cuerpo.

```
298
simb_prop_1(CH): list[D] = append(cabeza(CH),cuerpo(CH))
```

Y probamos que es un refinamiento de la función `simb_prop`:

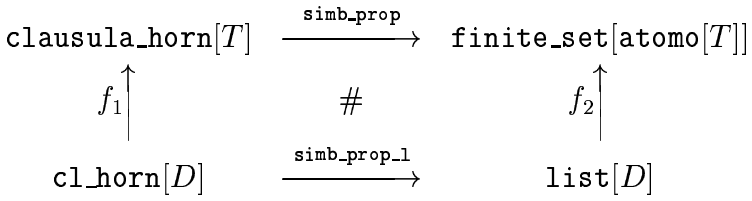


Figura 7.3: Refinamiento de la función simb_prop. Se ha denotado por f_1 la función transf_clausula_horn y por f_2 la función transf_interp

Con ella, podemos especificar la función que calcula la base de Herbrand de una lista LCH de cláusulas de Horn, concatenando las listas formadas por los símbolos proposicionales de las cláusulas de LCH.

```
base_herbrand_l(LCH): list[D] = Append(map(simb_prop_l)(LCH))
```
299

Probamos que la función `base_herbrand_l` refina a la función `base_herbrand`, como mostramos en el diagrama siguiente:

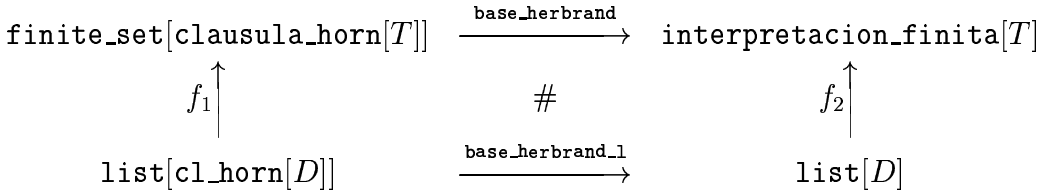


Figura 7.4: Refinamiento de la función base_herbrand. Se ha denotado por f_1 la función `transf_lista_cl_horn` y por f_2 la función `transf_interp`.

Particularizando, obtenemos la base de Herbrand de un programa P, restringiendo los dominios y los rangos de ambas funciones, de la siguiente forma:

```
base_herbrand_p(P: clausulas[T].programa): interpretacion_finita[T] =
    base_herbrand(P)

base_herbrand_l_p(P: clausulas_l[D].programa): list[D] =
    base_herbrand_l(P)
```
300

Finalmente, probamos que `base_herbrand_l_p` es refinamiento de la función que calcula la base de Herbrand de un programa.

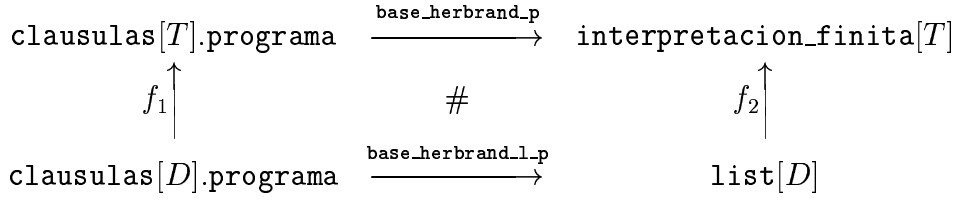


Figura 7.5: Refinamiento de la función `base_herbrand`. Se ha denotado por f_1 la función `transf_programa` y por f_2 la función `transf_interp`.

Como consecuencia, las propiedades de la primera especificación de la base de Herbrand se transmiten a la segunda. De esta forma, obtenemos los siguientes resultados, que se prueban haciendo uso de las propiedades de los refinamientos de operaciones:

301

```
base_herbrand_l_es_modelo: COROLLARY
    es_modelo(base_herbrand_l(P), P)

cardinal_base_herbrand_l_p: COROLLARY
    cardinal_l(base_herbrand_l(P)) =
    card(base_herbrand(transf_programa(P)))
```

Terminamos la subsección mostrando un ejemplo de cálculo de la base de Herbrand. Para ello, consideremos el programa P_1 y los objetivos G_1 y G_2 siguientes:

$$
\begin{array}{lll}
P_1: & p & \leftarrow s \\
 & p & \leftarrow q, r \\
 & r & \leftarrow \\
 & q & \leftarrow \\
 & p_1 & \leftarrow s \\
 & p_2 & \leftarrow s
\end{array}
\qquad G_1 :\leftarrow p \quad G_2 :\leftarrow p, s
$$

```
c1: cl_definida[string] = ((: "p" :), (: "s" :))
c2: cl_definida[string] = ((: "p" :), (: "q", "r" :))
c3: cl_definida[string] = ((: "r" :), null )
c4: cl_definida[string] = ((: "q" :), null )
c5: cl_definida[string] = ((: "p1" :), (: "s" :) )
c6: cl_definida[string] = ((: "p2" :), (: "s" :) )
P1: clausulas_l[string].programa = (: c1, c2, c3, c4, c5, c6 :)

G1: objetivo_definido[string] = (null, (: "p" :) )
G2: objetivo_definido[string] = (null, (: "p" , "s":) )
```

Consideremos también el programa P_2 y los objetivos siguientes:

$$\begin{array}{lll} P_2: & q & \leftarrow e \\ & e & \leftarrow a \\ & e & \leftarrow \end{array} \qquad G_{21} :\leftarrow a \quad G_{22} :\leftarrow e$$

```
c2_1: cl_definida[string] = ((: "a" :), (: "e" :))
c2_2: cl_definida[string] = ((: "e" :), (: "a" :))
c2_3: cl_definida[string] = ((: "e" :), null)
P2:  clausulas_l[string].programa = (: c2_1, c2_2, c2_3 :)
G2_1: objetivo_definido[string] = (null, (: "a" :) )
G2_2: objetivo_definido[string] = (null, (: "e" :) )
```

La sesión de cálculo en el evaluador básico de PVS es:

```
<GndEval> "base_herbrand_l(P1)"
==> (: "p", "s", "p", "q", "r", "r", "q", "p1", "s", "p2", "s" :)

<GndEval> "elimina_duplicados(base_herbrand_l(P1))"
==> (: "p", "r", "q", "p1", "p2", "s" :)

<GndEval> "cardinal_l(base_herbrand_l(P1))"
==> 6

<GndEval> "base_herbrand_l(P2)"
 ==> (: "a", "e", "e", "a", "e" :)

<GndEval> "cardinal_l(base_herbrand_l(P2))"
 ==> 2
```

7.2.2. Algoritmo de cálculo del menor modelo de Herbrand

En esta subsección se especifica un algoritmo de cálculo del menor modelo de Herbrand de un programa, y se prueba que es un refinamiento evaluable de la especificación del menor modelo de Herbrand, realizada en el capítulo 4.

Recordemos, en primer lugar, la especificación realizada:

302
```
menor_modelo_herbrand(P): finite_set[atomo] =
  Intersection(conj_modelos_herbrand(P))
```

En el capítulo 6 hemos definido la función `Inter` y hemos probado que refina a la función que obtiene la intersección de un conjunto finito de conjuntos. Luego, es suficiente definir una función que obtenga una lista con los modelos de

Herbrand de un programa P y construir la especificación evaluable, sustituyendo ambas funciones. Para ello, comenzamos especificando una función recursiva, que extrae los modelos de P de una lista cuyos elementos son listas de átomos de D. Obsérvese que, para que esta función sea evaluable, hemos de usar la especificación evaluable del concepto de modelo (`es_modelo_e`):

303
```
extrae_modelos_e(P,LL): RECURSIVE list[list[D]] =
  CASES LL OF
  null: null,
  cons(L,LL2): IF es_modelo_e(L,P) THEN cons(L,extrae_modelos_e(P,LL2))
               ELSE extrae_modelos_e(P,LL2)
               ENDIF
  ENDCASES
  MEASURE length(LL)
```

Con ella, podemos definir una función que obtiene una lista con las sublistas de la base de Herbrand de P que son modelos de P, y definir la función que obtiene el menor modelo de Herbrand de P[1].

304
```
conj_modelos_herbrand_e(P): (cons?[list[D]]) =
    extrae_modelos_e(P, powerset_ref_2(base_herbrand_l_p(P)))

menor_modelo_herbrand_e(P): list[D] =
    Inter(conj_modelos_herbrand_e(P))
```

Con esta definición y, usando las propiedades de `Inter`, `powerset_ref_2` y `base_herbrand_l_p` como refinamientos de operaciones, se prueba que, en efecto, la especificación `menor_modelo_herbrand_e` es un refinamiento de la función `menor_modelo_herbrand`. Así las propiedades de la primera se transmitirán a la segunda.

$$\begin{array}{ccc} \texttt{clausulas}[T].\texttt{programa} & \xrightarrow{\texttt{menor\_modelo\_herbrand}} & \texttt{interpretacion\_finita}[T] \\ \uparrow f_1 & \# & \uparrow f_2 \\ \texttt{clausulas}[D].\texttt{programa} & \xrightarrow{\texttt{menor\_modelo\_herbrand\_e}} & \texttt{list}[D] \end{array}$$

Figura 7.6: Refinamiento de la función `menor_modelo_herbrand`. Se ha denotado por f_1 la función `transf_programa` y por f_2 la función `transf_interp`.

[1]Nótese que las funciones evaluables obtienen una representación de la base de Herbrand o del menor modelo de Herbrand que, evidentemente, no es única.

Por último, mostramos el resultado del cálculo del menor modelo de Herbrand de los programas especificados en la subsección previa. La sesión de cálculo en el evaluador básico de PVS es:

```
<GndEval> "menor_modelo_herbrand_e(P1)
==> (: "p", "p", "q", "r", "r", "q" :)
<GndEval> "elimina_duplicados(menor_modelo_herbrand_e(P1))"
==> (: "p", "r", "q" :)
<GndEval> "elimina_duplicados(menor_modelo_herbrand_e(P2))"
==> (: "a", "e" :)
```

7.2.3. Algoritmo de cálculo del operador de consecuencia

En esta subsección se especifica un algoritmo de cálculo del operador consecuencia inmediata y se prueba que es un refinamiento evaluable del operador especificado en el capítulo 4. Para ello, definimos una función recursiva que, dado un programa P y una lista de átomos de D, obtiene una lista con las consecuencias inmediatas de dicha lista:

305

```
consecuencia_i(P)(I): RECURSIVE list[D] =
    CASES P OF
     null: null,
     cons(C, P2): IF sublista?(cuerpo(C), I)
                  THEN cons(car(cabeza(C)), consecuencia_i(P2)(I))
                  ELSE consecuencia_i(P2)(I) ENDIF
   ENDCASES
   MEASURE length(P)
```

Usando las propiedades de `sublista?` y de `transf_interp`, se caracteriza esta función, probando el siguiente resultado, que conduce a establecer que, para todo programa P, la función `consecuencia_i(P)` es un refinamiento de la función `c_i_f(transf_programa(P))`.

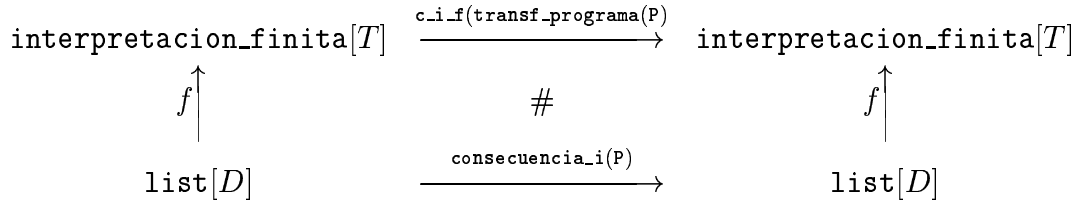


Figura 7.7: Refinamiento de la función c_i_f(transf_programa(P). Se ha denotado por f la función transf_interp

```
consecuencia_i_correcto_p_f: THEOREM
    transf_interp(consecuencia_i(P)(I)) =
    c_i_f(transf_programa(P))(transf_interp(I))
```
306

7.2.4. Algoritmo de cálculo del menor punto fijo

En esta subsección especificamos dos algoritmos para calcular el menor punto fijo de un programa, y probamos que ambos son refinamientos evaluables de la función que define el menor punto fijo de un programa, especificada en el capítulo 4. En dicho capítulo, definimos el menor punto fijo de un programa definido, usando que el operador de consecuencia es monótono, y la construcción del menor punto fijo para funciones monótonas descrita en la teoría de dominios [8].

```
mpf(P): interpretacion = u(c_i(P))
```
307

Además, probamos que el menor punto fijo de un programa P es la n–ésima potencia del operador de consecuencia inmediata restringido, siendo n el cardinal de la base de Herbrand de P.

```
mpf_potencia_c_i_corol_f: COROLLARY
    mpf(P) = iterate(c_i_f(P),card(base_herbrand(P)))(emptyset)
```
308

Como tenemos refinamientos evaluables de cada una de las funciones que intervienen en esta segunda especificación del menor punto fijo de un programa, la definición de una función evaluable que la refine se obtiene directamente sustituyendo cada una de ellas por la correspondiente función refinada. Así, una especificación evaluable del menor punto fijo es:

```
menor_punto_fijo_i(P): list[D] =
    iterate(consecuencia_i(P), cardinal_l(base_herbrand_l(P)))(null)
```
309

Aplicando la propiedada que garantiza que la composición de refinamientos de funciones es un refinamiento 252, probamos que la función `menor_punto_fijo_i` es un refinamiento evaluable de la función `mpf` (figura 7.8).

También hemos probado que el menor punto fijo de un programa P puede alcanzarse en una potencia del operador de consecuencia, menor que el cardinal de la base de Herbrand de P 115. Teniendo en cuenta esta propiedad, podemos describir un algoritmo que calcule las potencias sucesivas del operador de consecuencia y obtenga así el menor punto fijo de P. La especificación realizada en PVS es la siguiente:

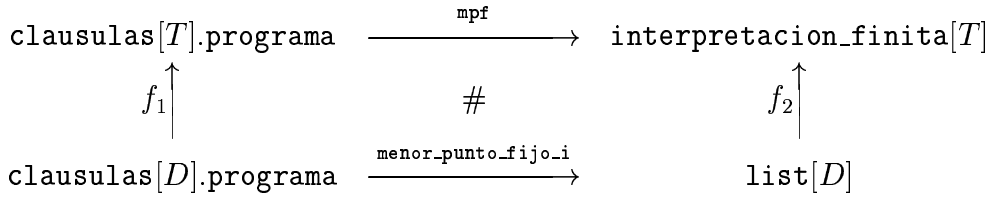


Figura 7.8: Refinamiento de la función `mpf`. Se ha denotado por f_1 la función `transf_programa` y por f_2 la función `transf_interp`.

310
```
menor_punto_fijo_aux(P)(I:list[D]): RECURSIVE list[D] =
    LET I1 = append(consecuencia_i(P)(I),I) IN
      IF igual_l?(I1, I)
      THEN I
      ELSE menor_punto_fijo_aux(P)(I1) ENDIF
    MEASURE ...

menor_punto_fijo(P): list[D] = menor_punto_fijo_aux(P)(null)
```

Como la función auxiliar es recursiva, hemos de proporcionar una función de medida que decrezca estríctamente en cada llamada recursiva, hasta alcanzar el punto fijo. Obsérvese que, en el uso de dicha función auxiliar para calcular el menor punto fijo, comenzamos con la lista vacía y, en cada paso, añadimos los elementos que son consecuencias de los que ya se tienen. Además, el crecimiento de esta lista está acotado por la base de Herbrand del programa. Es decir, una función de medida adecuada podría ser la diferencia entre el cardinal de la base de Herbrand de P y el cardinal de la interpretación. Ahora bien, si comenzamos con una interpretación cualquiera no podemos garantizar que dicha diferencia sea un número natural. Por ello, introducimos la noción de interpretación principal relativa a un programa P y definimos la función `menor_punto_fijo_aux(P)` sobre dichas interpretaciones, proporcionando así la función de medida aludida.

311
```
interpretacion_l_principal(P): TYPE =
    {I: list[D] | sublista?(I,base_herbrand_l(P))}

medida_p_f(P)(I: interpretacion_l_principal(P)): nat =
    cardinal_l(base_herbrand_l(P)) - cardinal_l(I)
```

312
```
menor_punto_fijo_aux(P)(I:interpretacion_l_principal(P)):
                        RECURSIVE list[D] =
    LET I1 = append(consecuencia_i(P)(I),I) IN
      IF igual_l?(I1, I)
      THEN I
      ELSE menor_punto_fijo_aux(P)(I1) ENDIF
    MEASURE medida_p_f(P)(I)

menor_punto_fijo(P): list[D] = menor_punto_fijo_aux(P)(null)
```

En este caso, no hemos obtenido una especificación a partir de otra previa, mediante la sustitución de las funciones que intervienen en la descripción de la primera por sus funciones refinadas. Por tanto, no podemos hacer uso de las propiedades de la composición de refinamientos de funciones. Entonces, para probar que la función `menor_punto_fijo` es un refinamiento evaluable de `mpf`, probamos lo siguiente:

1. Si J es el resultado de `menor_punto_fijo(P)`, entonces J es el menor punto fijo de `consecuencia_i(P)`, bajo las relaciones `igual_l?` y `sublista?`.

313
```
menor_punto_fijo_correcto: THEOREM
  menor_punto_fijo(P) = J
  ⇒ igual_l?(consecuencia_i(P)(J), J)  &
    (∀I1: igual_l?(consecuencia_i(P)(I1),I1) ⇒ sublista?(J,I1))
```

2. Si I es un punto fijo del operador `consecuencia_i(P)` bajo las relaciones `igual_l?` y `sublista?`, entonces `transf_interp(I)` es un punto fijo del operador `c_i(transf_programa(P))`.

314
```
conserva_puntos_fijos: LEMMA
  igual_l?[D](consecuencia_i(P)(I), I)
  ⇒ fixpoint?(c_i(transf_programa(P)))(transf_interp(I))

conserva_menor_punto_fijo: COROLLARY
  fixpoint?(c_i(transf_programa(P)))
           (transf_interp(menor_punto_fijo(P)))
```

3. El transformado de `menor_punto_fijo(P)` es subconjunto de cualquier punto fijo del operador de consecuencia inmediata del transformado del programa P.

```
315
transf_menor_punto_fijo_es_el_menor: THEOREM
  fixpoint?(c_i(transf_programa(P)))(IC)
  ⇒ subset?(transf_interp(menor_punto_fijo(P)), IC)
```

4. El transformado de `menor_punto_fijo(P)` es el menor punto fijo del operador consecuencia inmediata del transformado del programa `P`.

```
316
transf_menor_punto_fijo_es_menor_punto_fijo: COROLLARY
  least_fixpoint?(c_i(transf_programa(P)))
                 (transf_interp(menor_punto_fijo(P)))
```

Con todo ello, probamos el resultado que nos garantiza que `menor_punto_fijo` es un refinamiento de `mpf`:

```
317
menor_punto_fijo_es_refinamiento_mpf: THEOREM
    transf_interp(menor_punto_fijo(P)) = mpf(transf_programa(P))
```

Por último, comentemos que los resultados probados sobre las funciones que especifican el menor punto fijo de un programa P, o el menor modelo de Herbrand, se "trasladan" de manera natural, a través del refinamiento. Por ejemplo, se prueba fácilmente, el siguiente resultado:

```
mmh_igual_mpf: COROLLARY
  igual_l?(menor_modelo_herbrand_e(P), menor_punto_fijo_i(P))
```

Veamos, para terminar, el cálculo del menor punto fijo del programa P_1. La sesión de cálculo en el evaluador básico de PVS es:

```
<GndEval> "menor_punto_fijo(P1)"
==> (: "p", "r", "q", "r", "q" :)

<GndEval> "elimina_duplicados(menor_punto_fijo(P1))"
==> (: "p", "r", "q" :)

<GndEval> "menor_punto_fijo_i(P1)"
==> (: "p", "r", "q" :)
```

7.2.5. Algoritmos de resolución SLD

En esta subsección presentamos especificaciones de algoritmos de refutación que difieren, esencialmente, en el proceso de búsqueda utilizado. El resultado *160* muestra que, a la hora de especificar un proceso para obtener una refutación, la regla de computación que utilicemos es irrelevante. Así, una vez fijada una regla de computación, podemos construir el correspondiente espacio de búsqueda como un árbol, llamado *árbol SLD*.

En general, dados un programa P y un objetivo G, un **árbol SLD** para $P \cup \{G\}$ es un árbol que verifica las siguientes condiciones:

- Cada nodo del árbol es un objetivo definido.
- La raiz es G.
- Si $\leftarrow \{A_1, \ldots, A_j, \ldots, A_m\}$ con $m \geq 1$ es un nodo del árbol, y A_j es el átomo seleccionado, entonces para cada cláusula de P de la forma $A_j \leftarrow \{B_1, \ldots, B_k\}$, dicho nodo tiene como hijo a $\leftarrow \{A_1, \ldots, B_1, \ldots, B_k, \ldots, A_m\}$
- Si un nodo es el objetivo vacío, no tiene hijos.

En particular, dada una regla de computación R, el árbol SLD para $P \cup \{G\}$ vía R es un árbol en el que la función R selecciona los átomos. Entonces, por el teorema *160*, si $P \cup \{G\}$ es insatisfacible, el árbol SLD para $P \cup \{G\}$ vía R contiene el objetivo vacío. Un procedimiento de refutación SLD queda, pues, determinado por una regla de computación junto con un proceso de búsqueda. Por tanto, la completitud de dicho procedimiento dependerá de la completitud del método de búsqueda correspondiente.

El sistema *Prolog* usa como regla de computación la selección del átomo más a la izquierda en el objetivo, junto el proceso de búsqueda en profundidad en el árbol. Por tanto, no se puede garantizar su terminación. Para especificar este procedimiento en PVS, comenzamos por definir una función que calcula la resolvente de un objetivo propio (no vacío) y una cláusula, suponiendo que el átomo seleccionado es el primero del cuerpo de dicho objetivo:

```
resolvente((G: (es_objetivo_propio)),C): objetivo_definido =
    (null, append(cuerpo(C), cdr(cuerpo(G))))
```

Nótese que esta definición de resolvente es impropia, puesto que no exigimos que tenga sentido su cálculo. Es decir, no exigimos que la cabeza de la cláusula C coincida con el primer átomo del cuerpo del objetivo G. Por ello, cuando se use esta función en un proceso de resolución SLD, habrá que garantizar que C y G verifiquen esta condición.

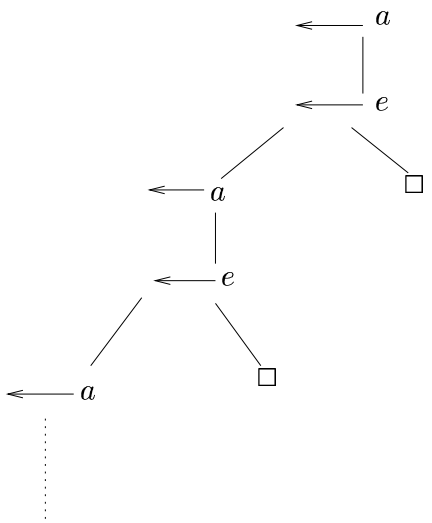
Como el proceso de búsqueda puede no terminar, hemos de establecer una medida para garantizar su terminación en PVS, mediante el axioma adecuado:

```
medida(P,G): nat

medida_ax: AXIOM
    ∀ (G: objetivo_definido[T], P: clausulas_l[T].programa):
     NOT es_vacio(G) ⇒
     (∀ (C: cl_definida[T]):
      car(cabeza(C)) = car(cuerpo(G)) ⇒
      medida(P, resolvente(G, C)) < medida(P, G))
```

De esta forma, estamos en condiciones de especificar el siguiente procedimiento para determinar la demostrabilidad de G a partir de P, usando la función `some` del preludio de PVS para implementar la búsqueda en profundidad.

```
demostrable_por_sld(P,G): RECURSIVE bool =
  IF es_vacio(G)
     THEN TRUE
     ELSE some(LAMBDA(C): car(cabeza(C)) = car(cuerpo(G)) &
                          demostrable_por_sld(P, resolvente(G,C)))(P)
  ENDIF
  MEASURE medida(P,G)
```

Nótese que en el caso del programa P_2 y el objetivo G_{21}, el proceso de búsqueda en profundidad no encuentra la solución.



Podemos observar la evaluación de `demostrable_por_sld` con los ejemplos anteriores. La sesión de cálculo en el evaluador básico de PVS es:

```
<GndEval> "demostrable_por_sld(P1,G1)"
==> TRUE
<GndEval> "demostrable_por_sld(P1,G2)"
==> FALSE
<GndEval> "demostrable_por_sld(P2,G2_1)"
Error: Stack overflow (signal 1000)
```

Especificamos ahora un procedimiento de refutación SLD, manteniendo la regla de computación consistente en elegir el primer átomo del cuerpo de G, y realizando la búsqueda en anchura. La implementación de la búsqueda en anchura puede hacerse mediante una "cola" de nodos por analizar. Esta cola estará representada por una lista y se actualizará introduciendo los nuevos nodos al final de dicha lista. Así, la especificación en PVS queda:

```
medida_res_3(P,G,LG): nat

medida_res_3_ax: AXIOM
  ∀ (G: objetivo_definido[T], LG: list[objetivo_definido[T]],
             P: clausulas_1[T].programa):
    NOT es_vacio(G) ⇒
     (∀ (L: list[objetivo_definido[T]]):
        L = lista_resolventes(P, G) ⇒
         (∀ (NLG: list[objetivo_definido[T]]):
            NLG = append(LG, L) & NOT null?(NLG) ⇒
             medida_res_3(P, car[objetivo_definido[T]](NLG),
                          cdr[objetivo_definido[T]](NLG))
              < medida_res_3(P, G, LG)))

demostrable_por_sld_3_aux(P,G,LG): RECURSIVE bool =
    IF es_vacio(G) THEN
       TRUE
    ELSE LET L = lista_resolventes(P,G),
             NLG = append(LG,L) IN
     IF null?(NLG) THEN
          FALSE

       ELSE demostrable_por_sld_3_aux(P,car(NLG),cdr(NLG))
       ENDIF
     ENDIF
    MEASURE medida_res_3(P,G,LG)

demostrable_por_sld_3(P,G): bool =
    demostrable_por_sld_3_aux(P,G,null)
```

donde:

```
lista_resolventes(P,(G: (es_objetivo_propio))):
                    RECURSIVE list[objetivo_definido] =
    CASES P OF
     null: null,
     cons(C1,P1): IF car(cabeza (C1)) = car(cuerpo(G)) THEN
                     cons (resolvente(G,C1), lista_resolventes(P1,G))
                  ELSE lista_resolventes(P1,G) ENDIF
   ENDCASES
   MEASURE length(P)
```

Nota 7.3 Es conocido que la búsqueda en anchura en un árbol es un procedimiento completo. La prueba de la completitud se basa en demostrar, por inducción, que se encuentra un determinado nodo en un número finito de pasos. El orden en el que se hace la inducción es un orden lexicográfico $(n, r(n))$, en el que n representa el nivel y $r(n)$ el lugar que ocupa un nodo en el nivel n. Ahora bien, la implementación que especificamos aquí no contempla directamente el nivel correspondiente a cada nodo. Por otra parte, no estamos interesados en mostrar ahora la completitud de la búsqueda en anchura, sino sólo en mostrar un ejemplo evaluable. Por esa razón, hemos declarado una medida e impuesto el axioma correspondiente.

La sesión de cálculo en el evaluador básico de PVS muestra que, en este caso, el proceso de búsqueda sí encuentra solución en el caso del programa P_2 y el objetivo G_{21}:

```
<GndEval> "demostrable_por_sld_3(P1,G1)"
==> TRUE
<GndEval> "demostrable_por_sld_3(P1,G2)"
==> FALSE
<GndEval> "demostrable_por_sld_3(P2,G2_2)"
==> TRUE
<Gndeval> "demostrable_por_sld_3(P2,G2_1)"
==> TRUE
```

Por último, especificamos otro procedimiento de refutación SLD, usando cualquier regla de computación, siendo la búsqueda mediante profundidad iterativa. Además, vamos a realizar, en primer lugar, una nueva especificación de la función que calcula la resolvente, eliminando del cuerpo del objetivo todos los átomos iguales al seleccionado:

```
R: VAR [(es_objetivo_propio) -> T]

es_regla_computacion(R): bool =
  ∀ (G: (es_objetivo_propio)): R(G) ∈ cuerpo(G)

tiene_resolvente_1(G,C): bool = car(cabeza(C)) ∈ cuerpo(G)

PD_1: TYPE = {par: [(es_objetivo_propio), cl_definida] |
                   LET (G, C) = par IN tiene_resolvente_1(G,C)}

resolvente_g(par:PD_1): objetivo_definido[T] =
    LET (G, C) = par IN
     (null, append(elimina(car(cabeza(C)),cuerpo(G)),cuerpo(C)))
```

En segundo lugar, especificamos una función que determina si $P \cup \{G\}$ tiene una refutación de longitud n vía R, usando búsqueda en profundidad. Y, aplicándola sucesivamente, comprobamos si $P \cup \{G\}$ tiene una refutación:

```
demostrable_por_sld_long_via_r_2(R:(es_regla_computacion))(P,G,n):
                                  RECURSIVE bool =
  IF n = 0
  THEN es_vacio(G)
  ELSE es_objetivo_propio(G) &
       some(LAMBDA(C): car(cabeza(C)) = R(G) &
       demostrable_por_sld_long_via_r_2(R)(P,resolvente_g(G,C),n-1))(P)
  ENDIF
    MEASURE n

medida_aux((R:(es_regla_computacion)),P,G,n): nat

comprueba_demostrable_aux((R:(es_regla_computacion)),P,G,n):
                                  RECURSIVE bool =
   IF demostrable_por_sld_long_via_r_2(R)(P,G,n)
   THEN TRUE
   ELSE comprueba_demostrable_aux(R,P,G,n+1)
   ENDIF
  MEASURE medida_aux(R,P,G,n)

demostrable_por_sld_via_r_2(R: (es_regla_computacion))(P,G): bool =
    IF es_vacio(G)
    THEN TRUE
    ELSE
    comprueba_demostrable_aux(R,P,G,1)
    ENDIF
```

Consideremos la regla de computación R_1 que consiste en elegir el primer átomo del cuerpo de un objetivo, y veamos el comportamiento del algoritmo demostrable_por_sld_long_via_r_2 en los programas y objetivos de los ejemplos anteriores. La sesión de cálculo en el evaluador básico de PVS es:

```
R_1(G:(es_objetivo_propio)): string = car(cuerpo(G))

<GndEval> "demostrable_por_sld_via_r_2(R_1)(P2,G2_2)"
==> TRUE

<Gndeval> "demostrable_por_sld_via_r_2(R_1)(P2,G2_1)"
==> TRUE

<GndEval> "demostrable_por_sld_via_r_2(R_1)(P1,G1)"
==> TRUE

<GndEval> "demostrable_por_sld_via_r_2(R_1)(P1,G2)"
==> no termina
```

Para terminar, mostramos una especificación de un algoritmo para obtener una prueba por resolución SLD de un objetivo a partir de un programa, en el caso de que sea demostrable. Para ello, en primer lugar, especificamos la noción de demostración o refutación: dados P y G, una **refutación por resolución** SLD de G a partir de P es una lista de pares $< G_0, C_0 >, \ldots, < G_i, C_i >, \ldots, < G_n, C_n >$, tal que $G_0 = G$, $C_0 \in P$, la cabeza de C_0 coincide con el primer átomo del cuerpo de G_0, $G_1 =$ resolvente(G_0, C_0) y la lista $< G_1, C_1 >, \ldots, < G_n, C_n >$ es una prueba por resolución SLD de G_1.

```
  % Tipo de paso de demostración: par_de es el tipo de los pasos de
  % demostración.
  par_de: TYPE = [objetivo_definido,cl_definida]

  % Tipo de demostración: de es el tipo de las demostraciones o
  % pruebas por resolución SLD.
  de: VAR list[par_de]

es_refutacion(de,P,G): RECURSIVE bool =
    CASES de OF
     null: es_vacio(G),
     cons(par, de2): es_objetivo_propio(G) &
                     LET (G1,C1) = par
                         IN G1 = G & C1 ∈ P &
                            car(cabeza (C1)) = car(cuerpo(G)) &
                            es_refutacion(de2,P,resolvente(G,C1))
    ENDCASES
    MEASURE length(de)
```

Para construir una función que obtiene una prueba por resolución, definimos una función recursiva auxiliar `prueba_por_sld_aux_b` que toma como argumentos el programa P, el objetivo a resolver an cada momento G, y una lista P_1 con las cláusulas de P que quedan para resolver el objetivo actual. Esta función busca una prueba por resolución para el objetivo G, usando el programa P. Termina cuando $G = \square$ o $P_1 = ()$. Si $G = \square$, devuelve un par cuyo primer elemento es `TRUE` y el segundo es una lista de pares de demostración, que constituye una prueba por resolución SLD de G, a partir de P. Si $P_1 = ()$, devuelve el par formado por `FALSE` y la lista vacía. Para ello, declaramos previamente una función de medida e imponemos el axioma necesario para probar su terminación.

```
prueba_por_sld_b(P,G): list[par_de] =
    PROJ_2(prueba_por_sld_aux_b(P,G,P))

medida_c(P,G,P1): nat

prueba_por_sld_aux_b(P,G,P1): RECURSIVE [bool, list[par_de]] =
  IF es_vacio(G) THEN
     (TRUE, null)
     ELSIF null?(P1) THEN
           (FALSE, null)
           ELSIF car(cabeza (car (P1))) = car(cuerpo(G)) THEN
           LET temp=prueba_por_sld_aux_b(P,resolvente(G,car(P1)),P)
                IN IF PROJ_1(temp) THEN
                      (TRUE, cons((G, car(P1)),PROJ_2(temp)))
                      ELSE prueba_por_sld_aux_b(P, G, cdr(P1))
                   ENDIF
           ELSE prueba_por_sld_aux_b(P, G, cdr(P1))
   ENDIF
   MEASURE medida_c(P,G,P1)
```

Mediante los resultados siguientes, probamos la corrección de este algoritmo en la teoría `sld_resolucion_1_correccion` (páginas 367–369).

Teorema 7.4 *Si G es el objetivo vacío, entonces* `prueba_por_sld_b(P,G)` *es una refutación de $P \cup \{G\}$.*

318

```
prueba_por_sld_b_correcto_vacio: THEOREM
  es_vacio(G) ⇒ es_refutacion(prueba_por_sld_b(P,G), P, G)
```

Teorema 7.5 *Sea P un programa definido. Si G es un objetivo propio y el resultado de* `prueba_por_sld_b(P,G)` *es una lista no vacía, entonces dicha lista es una refutación de $P \cup \{G\}$.*

```
                                                                    319
prueba_por_sld_b_correcto: THEOREM
  es_objetivo_propio(G) & cons?(prueba_por_sld_b(P,G))
  ⇒ es_refutacion(prueba_por_sld_b(P,G), P, G)
```

Demostración:

La prueba se basa en los dos resultados siguientes, que establecemos sobre la función auxiliar `prueba_por_sld_aux_b`:

1. Si el segundo elemento del par `prueba_por_sld_aux_b(P,G,P1)` es una lista no vacía, entonces el primer elemento de dicho par es `TRUE`.

   ```
   prueba_por_sld_aux_correcto_l1: LEMMA
     es_objetivo_propio(G) &
     cons?(PROJ_2(prueba_por_sld_aux_b(P, G, P1)))
     ⇒ PROJ_1(prueba_por_sld_aux_b(P, G, P1))
   ```

   La prueba se hace por inducción en `medida_b(P,G,P1)`, distinguiendo si el átomo de la cabeza de la primera cláusula de P_1 coincide o no con el primer átomo del cuerpo de G.

2. Sea P un programa y P_1 una lista de cláusulas de P. Si el primer elemento de `prueba_por_sld_aux_b(P, G, P1)` es `TRUE`, entonces el segundo elemento es una refutación de $P \cup \{G\}$.

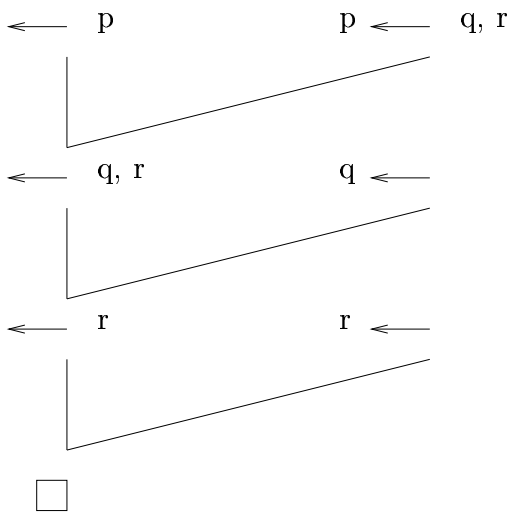
   ```
   prueba_por_sld_aux_b_correcto: THEOREM
     (∀ C: C ∈ P1 ⇒ C ∈ P) ⇒
     (PROJ_1(prueba_por_sld_aux_b(P,G,P1)) ⇒
     es_refutacion(PROJ_2(prueba_por_sld_aux_b(P,G,P1)), P, G))
   ```

   La prueba se hace por inducción en `medida_b(P,G,P1)`, distinguiendo los cuatro casos que contiene la definición de `prueba_por_sld_aux_b`.

□

Por último, mostramos la sesión de cálculo en el evaluador básico de PVS y un gráfico con la prueba del ejemplo:

```
<GndEval> "prueba_por_sld_b(P1, G1)"
==> (: (((: :), (: "p" :)), ((: "p" :), (: "q", "r" :))),
       (((: :), (: "q", "r" :)), ((: "q" :), (: :))),
       (((: :), (: "r" :)), ((: "r" :), (: :))) :)
<GndEval> "prueba_por_sld_b(P1, G2)"
==> (: :)
```


p
p
q, r
q, r
q
r
r

Capítulo 8

Una formalización evaluable del AFC

Las especificaciones de algoritmos del análisis formal de conceptos, descrita en el capítulo 5 de este trabajo, no son evaluables. La razón es que para facilitar el razonamiento hemos usado el tipo `finite_set[T]` en la representación de los elementos básicos de la teoría y en la especificación de los algoritmos. Nuestro objetivo ahora es construir especificaciones evaluables de dichos algoritmos, usando el marco para refinamientos descrito en el capítulo 6. Para ello, vamos a seguir el siguiente esquema:

1. Realizaremos representaciones de los tipos sobre los que se sustenta la formalización de la teoría, usando para ello el tipo `list[T]`, y probaremos que refinan a los anteriores.

2. Realizaremos especificaciones evaluables de las funciones y de los algoritmos del análisis formal de conceptos, usando las nociones de refinamiento. Concretamente:

   - Definiremos especificaciones evaluables de los operadores de derivación, de las nociones de concepto y subconcepto, y de las funciones que calculan el supremo y el ínfimo de una lista de conceptos (desarrollado en la teoría `contextos_formales_ref` [páginas 392–404]).
   - Construiremos un refinamiento evaluable del algoritmo que calcula los conceptos de un contexto formal finito (desarrollado en la teoría `conceptos_contexto_ref` [páginas 404–407]).
   - Describiremos la construcción de un refinamiento evaluable del algoritmo que genera la base de Duquenne–Guigues de un contexto formal finito (en la teoría `base_DG_ref` [páginas 409–413]).

8.1. Representación de los contextos formales

En esta sección definimos refinamientos de los tipos declarados en la formalización del AFC realizada en el capítulo 5, usando el refinamiento de los conjuntos finitos mediante listas, realizado previamente. Es decir, representaremos los conjuntos finitos mediante listas, y sustituiremos las operaciones sobre conjuntos finitos por las operaciones sobre listas que las refinan, según la tabla 6.3.2.

La representación de un contexto formal finito generalizado *161* es:

320
```
FFCT_rep: TYPE = [# obj_rep: list[T1],
                    atrib_rep: (cons?[T2]),
                    relacion_rep: list[[T1, T2]] #]
```

y la representación de un contexto formal finito *162* es:

321
```
 FFC_rep: TYPE = {R: FFCT_rep | LET Re_rep = relacion_rep(R) IN
                      every(LAMBDA (par: [T1, T2]):
                              PROJ_1(par) ∈ obj_rep(R) &
                              PROJ_2(par) ∈ atrib_rep(R))(Re_rep)}
```

La función que transforma la representación de contextos formales finitos generalizados usando listas, en la representación de contextos formales finitos generalizados usando conjuntos finitos es:

322
```
trans(R: FFCT_rep): FFCT[T1,T2] =
  (# obj:= c(id[T1])(obj_rep(R)),
     atrib:= c(id[T2])(atrib_rep(R)),
     relacion:= c(id[[T1,T2]])(relacion_rep(R)) #)
```

Lema 8.1 *La función* `trans` *establece un refinamiento del tipo* `FFCT` *por el tipo* `FFCT_rep`.

323
```
trans_es_ref_FFCT: LEMMA
   es_refinamiento?[FFCT[T1,T2], FFCT_rep](trans)
```

Demostración:

La prueba se hace usando que si $C = (O, A, I)$ es un contexto formal finito, los conjuntos O, A e I son finitos y, por tanto, existen listas l_1, l_2 y l_3 que los representan. Luego, basta considerar

```
R = (# obj_rep:= l1, atrib_rep:= l2, relacion_rep:= l3 #)
```

como representante de C. Es decir, `trans(R) = C`.

□

En lo que sigue, usaremos las variable C, C_1, C_2 para denotar contextos formales finitos representados por el tipo `FFC`, y R, R_1, R_2 para los contextos representados por el tipo `FFC_rep`.

A partir de la función `trans` definimos la función que transforma un elemento de `FFC_rep` en otro de `FFC`, y probamos que es un refinamiento de tipos, usando *323*.

324
```
trans_f(R: FFC_rep): FFC[T1, T2] = trans(R)

trans_f_es_ref_FFC: LEMMA
  es_refinamiento?[FFC[T1, T2], FFC_rep](trans_f)
```

8.2. Especificaciones evaluables de los operadores de derivación

La especificación del operador intención realizada en el capítulo 5 es declarativa, es decir, la intención de un conjunto X se declara como un conjunto de elementos que verifican una propiedad. Por tanto, no podemos construir un refinamiento suyo "sustituyendo sus componentes". En este caso, hemos de realizar una definición recursiva del operador intención y probar que refina a la especificación anterior.

Si R es un contexto formal finito, l_1 una lista de preobjetos y l_2 una lista de preatributos, definimos la función `intencion_rep_aux(R)`, que obtiene recursivamente una lista con los elementos de l_2 relacionados con todos los elementos de l_1, según el contexto R:

```
intencion_rep_aux(R)(l1: list[T1], l2: list[T2]):
  RECURSIVE list[T2] =
  CASES l2 OF
    null: null,
    cons(a,ls): IF every(LAMBDA (d:T1): (d,a) ∈ relacion_rep(R))(l1)
                THEN cons(a, intencion_rep_aux(R)(l1, ls)) ELSE
                intencion_rep_aux(R)(l1, ls) ENDIF
 ENDCASES
 MEASURE length(l2)
```

Usando esta función auxiliar, definimos la **intención** de una lista de preobjetos X respecto de un contexto formal R como una lista de atributos de R, relacionados con todos los elementos de X, definida a partir de la función auxiliar como sigue:

```
intencion_rep(R)(X: list[T1]): list[T2] =
  intencion_rep_aux(R)(X, atrib_rep(R))
```
325

Y probamos que, para cada contexto, el operador `intencion_rep(R)` es un refinamiento del operador `intencion(trans_f(R))` (figura 8.1):

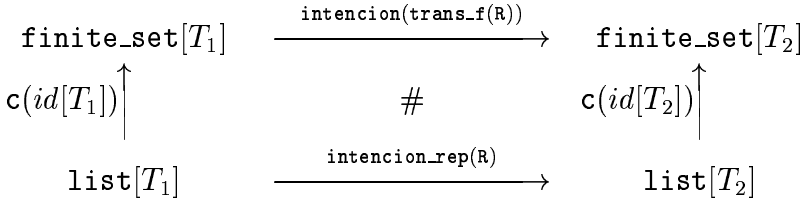


Figura 8.1: Refinamiento del operador `intención`

La prueba se realiza caracterizando, por inducción, los elementos de la lista `intencion_rep_aux(R)(l1, l2)` como los elementos de l_2 relacionados con todos los de l_1.

De la misma forma, se define la función evaluable que especifica el operador extensión y se prueba que `extension_rep(R)` es un refinamiento del operador `extension(trans_f(R))`.

```
extension_rep_aux(R)(l1: list[T1], l2: list[T2]):
  RECURSIVE list[T1] =
  CASES l1 OF
    null: null,
    cons(d,ls): IF every(LAMBDA (a:T2): (d,a) ∈ relacion_rep(R))(l2)
                THEN cons(d, extension_rep_aux(R)(ls, l2)) ELSE
                extension_rep_aux(R)(ls, l2) ENDIF
 ENDCASES
 MEASURE length(l1)

extension_rep(R)(Y: list[T2]): list[T1] =
  extension_rep_aux(R)(obj_rep(R), Y)
```
326

Una vez refinados los operadores de derivación asociados a un contexto formal, podemos construir especificaciones evaluables de los operadores de clausura, sin más que sustituir cada función por la correspondiente función refinada, como sigue:

```
                                                                    327
clausura_o_rep(R)(X: list[T1]): list[T1] =
  extension_rep(R)(intencion_rep(R)(X))

clausura_a_rep(R)(Y: list[T2]): list[T2] =
  intencion_rep(R)(extension_rep(R)(Y))
```

Se prueba fácilmente que estos operadores son refinamientos de los operadores de clausura, especificados en el capítulo 5 (figura 8.2). Además, todas las propiedades de las especificaciones conjuntistas de los operadores de derivación y de clausura, se "trasladan" a las correspondientes especificaciones evaluables a través de la noción de refinamiento.

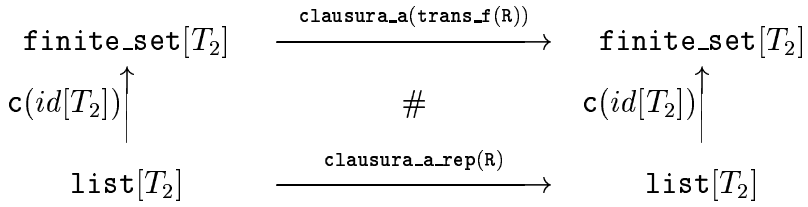


Figura 8.2: Refinamiento del operador `clausura_a`

Terminamos esta subsección mostrando un ejemplo de la evaluación de estos operadores en el evaluador básico de PVS. Para ello, en primer lugar, especificamos el ejemplo relativo a las relaciones binarias del capítulo 5:

```
objetos_r:list[string] = (: "S1", "S2", "S3", "S4", "S5", "S6" :)

atributos_r: list[string] = (: "R", "S", "Tr", "To", "E" :)

relacion_r: list[[string, string]] =
 (: ("S1", "R"), ("S1", "S"), ("S1", "Tr"), ("S1", "E"),
    ("S2", "S"), ("S2", "To"),
    ("S3", "S"), ("S3", "Tr"), ("S3", "To"),
    ("S4", "Tr"), ("S4", "To"),
    ("S5", "R"), ("S5", "Tr"), ("S5", "To"),
    ("S6", "R"), ("S6", "S"), ("S6", "To") :)

C_r: FFC_rep[string, string] =
  (# obj_rep:= objetos_r,
     atrib_rep:= atributos_r,
     relacion_rep:= relacion_ #)
```

La sesión de cálculo en el evaluador básico de PVS es:

```
% Consideramos las siguientes listas de objetos y de atributos:

l1: list[string] = (: "S1", "S2" :)
l2: list[string] = (: "S6" :)
l3: list[string] = (: "S" :)
l4: list[string] = (: "Tr", "To", "E" :)

<GndEval> "intencion_rep(C_r)(l1)"
==> (: "S" :)

<GndEval> "intencion_rep(C_r)(l2)"
==> (: "R", "S", "To" :)

<GndEval> "extension_rep(C_r)(l3)"
==> (: "S1", "S2", "S3", "S6" :)

<GndEval> "extension_rep(C_r)(l4)"
==> (: :)

<GndEval> "clausura_a_rep(C_r)(l4)"
==> (: "R", "S", "Tr", "To", "E" :)

<GndEval> "clausura_o_rep(C_r)(l1)"
==> (: "S1", "S2", "S3", "S6" :)
```

8.3. Refinamiento del tipo de los conceptos

Para definir una función que refine al predicado `es_concepto_c?` usamos que `sublista?` es un refinamiento evaluable de `subset?`, y que la función `igual_l?` refina la igualdad de conjuntos finitos. Por tanto, para definir un refinamiento evaluable del predicado `es_concepto_c?` será suficiente sustituir cada operación entre conjuntos finitos por la correspondiente función refinada. La especificación en PVS queda como sigue:

328

```
es_concepto_c_rep?(R)(par:[list[T1], list[T2]]): bool =
 LET (l1, l2) = par IN sublista?[T1](l1, obj_rep(R)) &
                       sublista?[T2](l2, atrib_rep(R)) &
                       igual_l?[T2](intencion_rep(R)(l1), l2) &
                       igual_l?[T1](extension_rep(R)(l2), l1)
```

A partir de las propiedades que aseguran que cada una de las operaciones que intervienen en la definición de `es_concepto_c_rep?` refina a cada una de las usadas en la definición de `es_concepto_c?` se prueba que el predicado

es_concepto_c_rep?(R) refina al predicado es_concepto_c?(trans_f(R)) (figura 8.3).

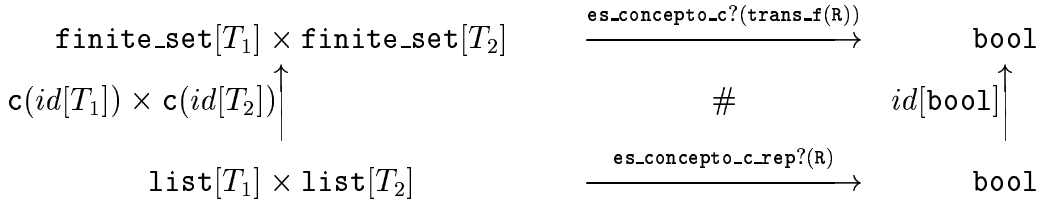


Figura 8.3: Refinamiento del predicado es_concepto_c?

Usando 175, probamos que dado el contexto $R = (O, A, I)$, el par (A', A'') es un concepto de R, por lo que establecemos el tipo de los conceptos de un contexto formal como sigue:

```
concepto_c_rep(R): NONEMPTY_TYPE = (es_concepto_c_rep?(R))
```
329

y probamos que el tipo concepto_c_rep(R) refina al tipo concepto_c(trans_f(R)), mediante la función $c(id[T_1]) \times c(id[T_2])$, que denotamos por trans_concepto(R)

```
trans_concepto(R)(par: concepto_c_rep(R)): concepto_c(trans_f(R)) =
  prod_cart(c(id[T1]), c(id[T2]))(par)
```

Con respecto a la igualdad inducida en el tipo refinado, sabemos que es de la forma igual?(trans_concepto(R)), pero que ésta no es evaluable. Por otra parte, es conveniente tener una definición equivalente y evaluable, definida directamente sobre el tipo concepto_c_rep(R). Para definirla, usamos la función igual_l? que sí es evaluable:

```
igual_concepto_rep_e(R)(par1,par2: [list[T1], list[T2]]): bool =
  igual_l?(PROJ_1(par1),PROJ_1(par2)) &
  igual_l?(PROJ_2(par1),PROJ_2(par2))
```
330

y probamos, usando 276, que son equivalentes:

```
igual_concepto_rep_e_igual?: LEMMA
  igual_concepto_rep_e(R) = igual?(prod_cart(c(id[T1]), c(id[T2])))
```
331

Obsérvese que la relación `igual_concepto_rep_e(R)` es evaluable, mientras que `igual?(prod_cart(c(id[T1]), c(id[T2])))` no lo es, aunque ambas especificaciones tienen el mismo "valor".

Pasamos a definir una especificación evaluable que refine a la relación de subconcepto. Para ello, definimos previamente las funciones que refinan a las funciones de acceso a las componentes de un concepto, y sustituimos cada operación por la correspondiente función refinada:

```
                                                                    332
intencion_concepto_rep(R)(par:concepto_c_rep(R)):list[T2]=proj_2(par)

extension_concepto_rep(R)(par:concepto_c_rep(R)):list[T1]=proj_1(par)

subconcepto_c_rep?(R)(par1, par2: concepto_c_rep(R)): bool =
  sublista?(extension_concepto_rep(R)(par1),
            extension_concepto_rep(R)(par2))
```

Obsérvese que la relación `subconcepto_c_rep?(R)` no es una relación de orden parcial, pues la propiedad de antisimetría no se conserva a través del refinamiento, por no ser inyectivo. En el caso de considerar la relación en el conjunto cociente, generado mediante la relación de igualdad inducida en `concepto_c_rep(R)` por la función `trans_concepto`, sí se verificaría dicha propiedad.

Por último, probamos que la relación `subconcepto_c_rep?(R)` refina a la relación `subconcepto_c?(trans_f(R))` (figura 8.4):

$$\begin{array}{ccc} \texttt{concepto\_c(trans\_f(R))} \times \texttt{concepto\_c(trans\_f(R))} & \xrightarrow{\texttt{subconcepto\_c?(trans\_f(R))}} & \texttt{bool} \\ \uparrow f_1 & \# & \uparrow f_2 \\ \texttt{concepto\_c\_rep(R)} \times \texttt{concepto\_c\_rep(R)} & \xrightarrow{\texttt{subconcepto\_c\_rep?(R)}} & \texttt{bool} \end{array}$$

Figura 8.4: Refinamiento del predicado `es_concepto_c?`, donde `f_1` denota la función `trans_concepto(R)` $\times$ `trans_concepto(R)` y `f_2` denota la función `id[bool]`

8.4. Especificaciones evaluables del ínfimo y el supremo

De la misma forma que hemos construido una especificación evaluable de la relación de subconcepto, construimos especificaciones evaluables que calculan el ínfimo y el supremo de una lista de conceptos, sustituyendo cada función por las

correspondientes funciones refinadas. Para ello definimos, en primer lugar, funciones que refinan a las que obtienen el conjunto de intenciones (resp. extensiones) de un conjunto de conceptos, usando que la función `map` refina a la función `image`.

Si R un contexto formal finito y LL una lista de conceptos de R, definimos la función `intenciones_conj_conceptos_rep(R)(LL)` que obtiene una lista con las intenciones de los conceptos de LL. De igual forma, se obtiene una lista con las extensiones de los conceptos de LL, mediante la expresión `extensiones_conj_conceptos_rep(R)(LL)`

333
```
intenciones_conj_conceptos_rep(R)(LL: list[concepto_c_rep(R)]):
                                  list[list[T2]]=
  map(intencion_concepto_rep(R))(LL)

extensiones_conj_conceptos_rep(R)(LL: list[concepto_c_rep(R)]):
                                  list[list[T1]]=
   map(extension_concepto_rep(R))(LL)
```

Probamos que ambas funciones son refinamientos, usando *291*.

Así, estamos en condiciones de especificar las funciones que calculan, efectivamente, el ínfimo y el supremo de una lista no vacía de conceptos de R. Si LL una lista no vacía de conceptos de R, definimos las funciones `infimo_rep(R)(LL)` y `supremo_rep(R)(LL)` como sigue:

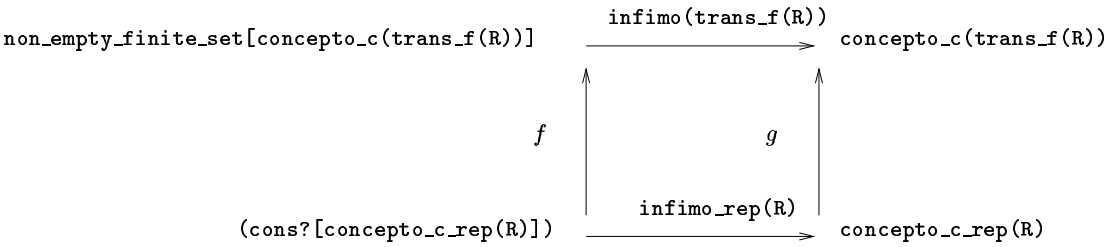
334
```
infimo_rep(R)(LL:(cons?[concepto_c_rep(R)])): concepto_c_rep(R) =
 (Inter(extensiones_conj_conceptos_rep(R)(LL)),
  clausura_a_rep(R)(Append(intenciones_conj_conceptos_rep(R)(LL))))

supremo_rep(R)(LL: (cons?[concepto_c_rep(R)])): concepto_c_rep(R) =
 (clausura_o_rep(R)(Append(extensiones_conj_conceptos_rep(R)(LL))),
  Inter(intenciones_conj_conceptos_rep(R)(LL)))
```

Para asegurar que se verifican las condiciones de tipo generadas por estas definiciones, hemos probado previamente los resultados siguientes, que garantizan que el rango de ambas funciones es el tipo `concepto_c_rep(R)`. En la prueba, usamos que cada función que interviene en la especificación refina a la correspondiente función usada en las especificaciones de las funciones `infimo` y `supremo`, y los lemas que prueban las condiciones descritas en *186*.

Lema 8.2 *Sea R un contexto formal finito y LL una lista no vacía de conceptos de R. Entonces:*

335
```
infimo_rep_l1: LEMMA
 ∀ (LL: (cons?[concepto_c_rep(R)])):
  es_concepto_c_rep?(R)
   (Inter(extensiones_conj_conceptos_rep(R)(LL)),
    intencion_rep(R)
     (extension_rep(R)(Append(intenciones_conj_conceptos_rep(R)(LL)))))

supremo_rep_lema_1: LEMMA
 ∀(LL:(cons?[concepto_c_rep(R)])):
  es_concepto_c_rep?(R)
    (extension_rep(R)
     (intencion_rep(R)(Append(extensiones_conj_conceptos_rep(R)(LL)))),
     Inter(intenciones_conj_conceptos_rep(R)(LL)))
```

Probamos que, en efecto, las funciones `infimo_rep(R)` y `supremo_rep(R)` refinan a las funciones `infimo(trans_f(R))` y `supremo(trans_f(R))`, respectivamente.



Refinamiento de la función `infimo(trans_f(R))`, donde `f` denota la función `c(trans_concepto(R))` y `g` a la función `trans_concepto(R)`.

Terminamos esta subsección mostrando los resultados de la evaluación de estas funciones en el evaluador básico de PVS, usando el ejemplo *8.2*:

335
```
% Consideremos las siguientes listas de conceptos:
S1: list[[list[string], list[string]]] =
    (: ((: "S1", "S2", "S3", "S6" :), (: "S" :)),
       ((: "S1", "S6" :), (: "R", "S" :)) :)

S2: list[[list[string], list[string]]] =
    (: ((: "S1" :), (: "R", "S", "Tr", "E" :)),
       ((: "S1", "S5" :), (: "R", "Tr" :)) :)

S3: list[[list[string], list[string]]] =
    (: ((: "S1", "S5" :), (: "R", "Tr" :)),
       ((: "S1", "S3" :), (: "S", "Tr" :)) :)
```

```
<GndEval> "infimo_rep(C_r)(S1)"
==> ((: "S1", "S6" :), (: "R", "S" :))

<GndEval> "infimo_rep(C_r)(S2)"
==> ((: "S1" :), (: "R", "S", "Tr", "E" :))

<GndEval> "supremo_rep(C_r)(S1)"
==> ((: "S1", "S2", "S3", "S6" :), (: "S" :))

<GndEval> "supremo_rep(C_r)(S2)"
==> ((: "S1", "S5" :), (: "R", "Tr" :))

<GndEval> "infimo_rep(C_r)(S3)"
==> ((: "S1" :), (: "R", "S", "Tr", "E" :))

<GndEval> "supremo_rep(C_r)(S3)"
==> ((: "S1", "S3", "S4", "S5" :), (: "Tr" :))
```

8.5. Algoritmo de cálculo de los conceptos.

En esta sección se describe la construcción de un refinamiento evaluable de la especificación del algoritmo para generar el conjunto de los conceptos de un contexto formal finito, descrito en el capítulo 5. Lo haremos sustituyendo cada una de las funciones que intervienen en su definición por las correspondientes funciones refinadas. El proceso seguido es el siguiente:

- Refinamiento de la función `agrega_extension_elto`:

336
```
agrega_extension_elto_rep(R)(a: T2,
                           LL:list[list[T1]]):list[list[T1]] =
 LET fun=LAMBDA(ls:list[T1]): inter(ls,extension_rep(R)((: a :)))
    IN  append(LL, map(fun)(LL))
```

- Refinamiento de la función `genera_extensiones_aux`:

337
```
genera_extensiones_aux_rep(R)(l2:list[T2], LL:list[list[T1]]):
                           RECURSIVE list[list[T1]] =
 CASES l2 OF
 null: LL,
 cons(a, l1): genera_extensiones_aux_rep(R)(rest_l(l2),
                    agrega_extension_elto_rep(R)(car(l2), LL))
 ENDCASES
 MEASURE length(l2)
```

- Refinamiento de la función genera_extensiones:

```
genera_extensiones_rep(R): list[list[T1]] =                    338
  genera_extensiones_aux_rep(R)(atrib_rep(R), (: obj_rep(R) :))
```

- Refinamiento de la función genera_conceptos:

```
genera_conceptos_rep(R): list[[list[T1], list[T2]]] =          339
 LET fun = LAMBDA(ls: list[T1]): (ls, intencion_rep(R)(ls)) IN
     map(fun)(genera_extensiones_rep(R))
```

Abordamos ahora dos cuestiones. En primer lugar, probar que la función genera_conceptos_rep es un refinamiento de la función genera_conceptos. Para ello, es suficiente tener en cuenta que cada una de las funciones que se usan son refinamientos de las funciones correspondientes.

Teorema 8.3 *La función* genera_conceptos_rep *refina a* genera_conceptos.

```
genera_conceptos_rep_es_refinamiento: THEOREM                  340
 es_refinamiento_op?[FFC[T1, T2],
                     finite_set[[finite_set[T1], finite_set[T2]]],
                     FFC_rep[T1, T2],
                     list[[list[T1], list[T2]]],
                     trans_f[T1,T2],c(prod_cart(c(id[T1]),c(id[T2])))]
  (genera_conceptos, genera_conceptos_rep)
```

$$\begin{array}{ccc} \texttt{FFC} & \xrightarrow{\texttt{genera\_conceptos}} & \texttt{finite\_set}[[\texttt{finite\_set}[T1], \texttt{finite\_set}[T2]]] \\ \texttt{trans\_f}\uparrow & \# & \uparrow g \\ \texttt{FCC\_rep} & \xrightarrow{\texttt{genera\_conceptos\_rep}} & \texttt{list}[[\texttt{list}[T1], \texttt{list}[T2]]] \end{array}$$

Figura 8.5: Refinamiento de la función genera_conceptos, donde g la función c(prod_cart(c(id[T1]), c(id[T2]))

Demostración:

La demostración se basa en probar que cada una de las funciones que hemos definido para construir la especificación genera_conceptos_rep es un refinamiento de la función correspondiente, y usar el resultado 246. Para ello, probamos previamente los lemas siguientes:

- La función agrega_extension_elto_rep(R) refina a la función agrega_extension_elto(trans_f(R)).
- La función genera_extensiones_aux_rep(R) refina a la función genera_extensiones_aux(trans_f(R)).
- La función genera_extensiones_rep refina a genera_extensiones.

□

En segundo lugar, probar la corrección del algoritmo genera_conceptos_rep, para lo cual usaremos la corrección de genera_conceptos, que está expresada mediante la igualdad de conjuntos siguiente:

genera_conceptos(C) = CONCEPTOS(C) = {par| es_concepto_c?(C)(par)}

Ahora bien, un concepto (o un par de conjuntos finitos) no tiene una representación única como par de listas. Por tanto, así como el conjunto CONCEPTOS(C) es único, la lista formada por los conceptos de un contexto formal finito R no es única. Es más, tiene dos grados por los que pierde la unicidad: por ser una lista que representa a un conjunto, con lo que distintas listas con los elementos ordenados de forma diferente o repetidos representan al mismo conjunto; y porque los elementos de CONCEPTOS(C) también admiten más de una representación. Entonces, la expresión de la corrección del algoritmo genera_conceptos_rep sería de la forma siguiente: "*par* es un concepto de R si y sólo si en la lista genera_conceptos_rep(R) hay un elemento equivalente a él (i.e., igual módulo la relación de igualdad igual_concepto_rep_e(R))". Para ello, definimos el siguiente predicado, que reconoce, dado un contexto formal finito R y una lista LL, si para cada concepto de R hay un elemento en LL que lo representa:

341
```
es_conj_todos_conceptos_e_rep?(R:FFC_rep[T1, T2],
                               LL:list[[list[T1], list[T2]]]): bool =
  ∀ (par1:[list[T1], list[T2]]):
  es_concepto_c_rep?(R)(par1) ⇔
  ∃ (par2:[list[T1], list[T2]]):  par2 ∈ LL &
  prod_cart(c(id[T1]),c(id[T2]))(par1) =
  prod_cart(c(id[T1]),c(id[T2]))(par2)
```

Entonces, enunciamos el teorema de la forma siguiente:

Teorema 8.4 *Corrección de* genera_conceptos_rep*:*

342
```
correccion_genera_conceptos_e_se_preserva: THEOREM
  es_conj_todos_conceptos_e_rep?(R, genera_conceptos_rep(R))
```

Demostración:

Usando la definición *341*, los resultados relativos a los refinamientos de las funciones que intervienen y el teorema de corrección *199*.

□

Obsérvese que la función `genera_conceptos_rep` obtiene una lista con elementos que pueden ser repetidos. Mediante la función `genera_conceptos_rep_sd(R)` calculamos una lista sin elementos duplicados, con los conceptos de R:

343
```
genera_conceptos_rep_sd(R): list[[list[T1], list[T2]]] =
  elimina_duplicados(igual_concepto_rep_e(R))(genera_conceptos_rep(R))
```

Probamos que las listas obtenidas mediante `genera_conceptos_rep_sd(R)` y `genera_conceptos_rep(R)` son iguales, módulo la relación de igualdad `igual_concepto_rep_e(R)`:

344
```
elimina_duplicados_genera_conceptos_rep_igual: LEMMA
  igual_l?(igual_concepto_rep_e(R))(genera_conceptos_rep_sd(R),
                                    genera_conceptos_rep(R))
```

Terminamos mostrando los resultados de la evaluación de la función que calcula los conceptos de R en el evaluador básico de PVS, usando el ejemplo *8.2*:

344
```
 <GndEval> "genera_conceptos_rep_sd(C2_rep)"
 ==>
(: ((: "S1", "S2", "S3", "S4", "S5", "S6" :), (: :)),
   ((: "S1", "S5", "S6" :), (: "R" :)),
   ((: "S1", "S2", "S3", "S6" :), (: "S" :)),
   ((: "S1", "S6" :), (: "R", "S" :)),
   ((: "S1", "S3", "S4", "S5" :), (: "Tr" :)),
   ((: "S1", "S5" :), (: "R", "Tr" :)),
   ((: "S1", "S3" :), (: "S", "Tr" :)),
   ((: "S2", "S3", "S4", "S5", "S6" :), (: "To" :)),
   ((: "S5", "S6" :), (: "R", "To" :)),
   ((: "S2", "S3", "S6" :), (: "S", "To" :)),
   ((: "S6" :), (: "R", "S", "To" :)),
   ((: "S3", "S4", "S5" :), (: "Tr", "To" :)),
   ((: "S5" :), (: "R", "Tr", "To" :)),
   ((: "S3" :), (: "S", "Tr", "To" :)),
   ((: "S1" :), (: "R", "S", "Tr", "E" :)),
   ((: :), (: "R", "S", "Tr", "To", "E" :)) :)
```

```
% Cálculo del supremo y el ínfimo de todos los conceptos:

 <GndEval> "infimo_rep(C2_rep)(genera_conceptos_rep_sd(C2_rep))"
 ==> ((: "S1" :), (: "R", "S", "Tr", "To", "E" :))

 <GndEval> "supremo_rep(C2_rep)(genera_conceptos_rep_sd(C2_rep))"
 ==> ((: "S1", "S2", "S3", "S4", "S5", "S6" :), (: "To" :))
```

8.6. Generación de la base de Duquenne–Guigues

En esta sección se describe la construcción de un refinamiento evaluable del algoritmo (genera_imps) que calcula la base de Duquenne–Guigues de un contexto formal finito, especificado en el capítulo 3. Lo haremos sustituyendo cada una de las funciones por las correspondientes funciones refinadas. Para ello, hemos de construir refinamientos de las siguientes funciones:

- gen_s
- gen_s_pasos
- pseudo_restringidas
- es_pseudo_int_rest?

Para definirlas, usaremos a su vez, los refinamientos que tenemos de las funciones que se usan en las respectivas definiciones.

Comenzamos definiendo los tipos que representan a las implicaciones entre atributos de un contexto formal, usando listas, que refinan a los tipos definidos en el capítulo 3.

La representación de las implicaciones entre preatributos usando listas es:

345
```
implicacion_gen_ref: TYPE = [# antecedente_ref: list[T2],
                               consecuente_ref: list[T2] #]
```

Definimos una función que transforma la representación de las implicaciones entre preatributos usando listas (implicacion_gen_ref), en la representacion basada en el uso de conjuntos finitos (implicacion_gen), y probamos que dicha función es un refinamiento de tipos.

346
```
transf_imp_gen(imp_g_r:implicacion_gen_ref): implicacion_gen[T1,T2] =
  (# antecedente:= c(id[T2])(antecedente_ref(imp_g_r)),
     consecuente:= c(id[T2])(consecuente_ref(imp_g_r)) #)

implicacion_gen_ref_es_refinamiento: LEMMA
  es_refinamiento?[implicacion_gen[T1, T2], implicacion_gen_ref]
       (transf_imp_gen)
```

Demostración:

La prueba se basa en que si `imp` $= Y_1 \to Y_2$ es una implicación entre atributos, tanto Y_1 como Y_2 son conjuntos finitos y, por tanto, existen listas l_1 y l_2 que los representan. Luego, es suficiente considerar la implicación

```
imp_rep = (# antecedente_ref: l1,  consecuente_ref: l2 #)
```

y probar que `transf_imp_gen(imp_rep) = imp`.

□

La representación de las implicaciones entre atributos de un contexto formal finito R, usando listas, es:

347
```
implicacion_ref(R): TYPE =
  {imp_r: implicacion_gen_ref |
          sublista?[T2](antecedente_ref(imp_r), atrib_rep(R)) &
          sublista?[T2](consecuente_ref(imp_r), atrib_rep(R))}
```

Nótese que, para todo contexto R, `implicacion_ref(R)` es un subtipo del tipo `implicacion_gen_ref`, al igual que `implicacion(C)` es un subtipo del tipo `implicacion_gen`. Entonces, a partir de la función `transf_imp_gen` definimos una función específica para cada contexto, `transf_imp_gen(R)`, y probamos que dicha función es un refinamiento de tipos:

348
```
transf_imp_gen(R)(imp_r:implicacion_ref(R)): implicacion(trans_f(R))=
  transf_imp_gen(imp_r)

implicacion_ref_es_refinamiento: LEMMA
  es_refinamiento?[implicacion(trans_f(R)), implicacion_ref(R)]
       (transf_imp_gen(R))
```

Describimos los pasos seguidos en la construcción del refinamiento de la función `genera_imps`:

- Refinamiento de la función `es_pseudo_int_rest?`:

```
es_pseudo_int_rest_ref?(R)(ls:list[T2],LL:list[list[T2]]): bool =
  NOT sublista?[T2](clausura_a_rep(R)(ls), ls) &
  every(LAMBDA(l: list[T2]):
        IF sublista_estricto?[T2](l, ls)
        THEN sublista?[T2](clausura_a_rep(R)(l), ls)
        ELSE TRUE ENDIF) (LL)
```

- Refinamiento de la función `pseudo_restringidas`:

```
pseudo_restringidas_ref(R)(LL, S: list[list[T2]]):
                          RECURSIVE list[list[T2]] =
  CASES LL OF
   null: null,
   cons(ls,LL2): IF es_pseudo_int_rest_ref?(R)(ls, S)
                 THEN cons(ls,pseudo_restringidas_ref(R)(LL2,S))
                 ELSE pseudo_restringidas_ref(R)(LL2, S) ENDIF
  ENDCASES
  MEASURE length(LL)
```

- Refinamiento de la función gen_s_pasos:

```
gen_s_pasos_ref(R)(ls:list[T2], k:upto(cardinal_l[T2](ls)),
                   S:list[list[T2]]): RECURSIVE list[list[T2]] =
  LET NS = pseudo_restringidas_ref(R)(sublistas_card[T2](ls,k),S)
  IN  IF k = cardinal_l[T2](ls) THEN
      append(S, NS) ELSE
      gen_s_pasos_ref(R)(ls, k+1, append(S, NS))
      ENDIF
  MEASURE cardinal_l[T2](ls)-k
```

- Refinamiento de la función gen_s:

```
gen_s_ref(R): list[list[T2]] =
   gen_s_pasos_ref(R)(atrib_rep(R), 0, null)
```

- Refinamiento de la función genera_imps:

```
genera_imps_ref(R): list[implicacion_gen_ref[T1,T2]] =
  LET fun = LAMBDA (Y: list[T2]):
            (# antecedente_ref:= Y,
               consecuente_ref:= clausura_a_rep(R)(Y) #)
  IN map(fun)(gen_s_ref(R))
```

A partir de los lemas que garantizan que las funciones usadas en la especificación de genera_imps_ref refinan a las usadas en la definición de genera_imps_g, se prueba el siguiente teorema.

Teorema 8.5 *La función* genera_imps_ref *refina a la función* genera_imps_g.

```
genera_imps_ref_es_refinamiento: THEOREM                                    349
  es_refinamiento_op?[FFC[T1,T2], finite_set[implicacion_gen[T1,T2]],
                      FFC_rep[T1,T2], list[implicacion_gen_ref[T1,T2]],
                      trans_f[T1,T2],
                      c(transf_imp_gen[T1,T2])]
       (genera_imps_g, genera_imps_ref)
```

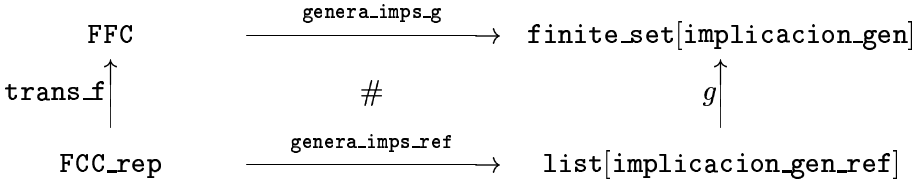


Figura 8.6: Refinamiento de la función genera_imps_g, donde g la función c(transf_imp_gen)

Nótese que la función gen_s_ref obtiene una lista con elementos posiblemente repetidos. Como consecuencia, genera_imps_ref genera una lista que puede contener implicaciones iguales. Ahora bien, igual que hemos hecho para la función que calcula los conceptos de un contexto, definimos la función genera_imps_ref_b que obtiene una lista, sin elementos repetidos, con las implicaciones que constituyen la base de Duquenne–Guigues del contexto.

```
genera_imps_ref_b(R): list[implicacion_gen_ref[T1,T2]] =
  LET fun = LAMBDA (Y: list[T2]):
              (# antecedente_ref:= Y,
                 consecuente_ref:= clausura_a_rep(R)(Y) #)
  IN map(fun)(elimina_duplicados[list[T2]](igual_l?[T2])(gen_s_ref(R)))
```

Finalizamos mostrando el cálculo de la base de Duquenne–Guigues del ejemplo 8.2:

```
<GndEval> "genera_imps_ref_b(C_r)"
==> (: (# antecedente_ref := (: "E" :),
          consecuente_ref := (: "R", "S", "Tr", "E" :) #),
       (# antecedente_ref := (: "Tr", "S", "R" :),
          consecuente_ref := (: "R", "S", "Tr", "E" :) #) :)
```

Capítulo 9

Conclusiones y trabajo futuro

En este capítulo presentamos algunas consideraciones en relación con los objetivos planteados al comienzo de este trabajo, así como posibles líneas de continuación del mismo.

Los objetivos planteados al comienzo del trabajo eran la formalización de teorías matemáticas en sistemas de razonamiento automático, la verificación formal de algoritmos de dichas teorías y la realización de dicha verificación de forma que el razonamiento dentro del sistema no se viera complicado de manera artificial.

En este sentido, podemos decir que hemos hecho realidad nuestras expectativas iniciales, puesto que:

- Se ha mostrado que es posible desarrollar teorías matemáticas formalizadas en un entorno como el que proporciona PVS, de forma que el razonamiento realizado sobre las especificaciones sea, esencialmente, del mismo nivel de dificultad del que se haría fuera del sistema de razonamiento. Además, las demostraciones se han realizado guiando al demostrador mediante indicaciones que corresponden a grandes pasos de prueba, y dejando que, a partir de ellas, el demostrador actúe de forma automática.

- Los guiones de prueba generados son similares a las demostraciones presentadas en en un texto matemático que desarrolle la misma teoría.

- Se ha construido un marco genérico en el que relacionar especificaciones de un mismo concepto o algoritmo.

- Se ha elaborado una batería de operaciones sobre listas y se ha probado que refinan a operaciones entre conjuntos finitos. Estas operaciones se podrá usar para refinar algoritmos cuyas especificaciones estén basadas en el uso de conjuntos finitos.

- Se ha realizado una formalización de la programación lógica proposicional, probándose la equivalencia entre las distintas interpretaciones semánticas

de los programas lógicos. Asimismo, se ha probado la completitud fuerte de la resolución SLD, usando una representación abstracta de los elementos que intervienen en el proceso de resolución. Finalmente, se han construido refinamientos evaluables de los los algoritmos de la programación lógica proposicional.

- Se ha formalizado la teoría del análisis formal de conceptos, de forma abstracta y se han obtenido refinamientos evaluables de los algoritmos de dicha teoría.

En la consecución de dichos objetivos hay dos factores que han jugado un papel fundamental: el sistema de razonamiento elegido y la forma de abordar la formalización.

Como ya hemos comentado, la motivación fundamental para elegir PVS ha sido la capacidad para combinar la potencia expresiva y la naturalidad en el razonamiento con la posibilidad de obtener algoritmos evaluables. Estas características se han puesto de manifiesto ampliamente en el desarrollo de esta memoria. En particular, es de destacar la capacidad de PVS para cuantificar sobre proposiciones y predicados, para tratar la igualdad sobre funciones no interpretadas. Además, ha sido de gran utilidad el uso de las teorías construidas en el preludio, de la biblioteca de conjuntos finitos y el mecanismo de empaquetar teorías en bibliotecas para ser usadas posteriormente.

Sin embargo, hemos encontrado también algunas dificultades en el uso del sistema, que podemos concretar en los aspectos siguientes:

- Cuando la formalización de las teorías se distribuyen en distintos directorios (lo que conlleva la creación por parte del sistema de distintos contextos), hemos detectado algunos problemas relacionados con la importación de teorías, que no ocurre si todas las teorías pertenecen a un mismo contexto. En este sentido, hemos optado por mantener una jerarquía entre los directorios (y, por tanto, entre los contextos), aunque hayamos tenido que anular un determinado contexto antes de acceder a otro que lo use, mediante el mecanismo de importación.

- Como hemos comentado en el capítulo 2, en la definición de una función se exige declarar el tipo, tanto del dominio como del rango de dicha función. Por otra parte, una función se puede aplicar sobre cualquier subtipo de su dominio, usando la conversión `restrict`. Esto tiene la clara ventaja de no tener que realizar distintas definiciones de una misma función. Sin embargo, puede hacer que algunas pruebas se compliquen de forma artificiosa, pues el sistema extenderá o restringirá el tipo, según el caso. Para evitarlo, en algunos casos hemos optado por especializar las definiciones en los subtipos correspondientes, haciendo uso de la posibilidad que ofrece PVS de sobrecargar un símbolo.

- Aunque a partir de la versión 2 de PVS, se ha incorporado un evaluador básico, que permite la evaluación de funciones de un fragmento del lenguaje, éste se encuentra aún poco desarrollado. Entre las dificultades encontradas en su uso, una de ellas ha sido la sintaxis de la entrada de las expresiones a evaluar, pues los delimitadores de la expresión son las dobles comillas, lo que dificulta la evaluación de expresiones que incluyen comillas. Por otra parte, para que el resultado de la evaluación sea correcto, la expresión ha debido ser semánticamente verificada, cosa que no puede hacerse desde el evaluador. Por ello, en la mayoría de los casos hay que declarar en la teoría una constante que represente la expresión a evaluar, comprobar su corrección semántica y, posteriormente, realizar su evaluación. Otra limitación del evaluador es la imposibilidad de admitir teorías parametrizadas.

En los planes de desarrollo del sistema (ver el documento [31]), se contemplan soluciones a estas dificultades, así como, la posibilidad de integrar PVS con otros sistemas.

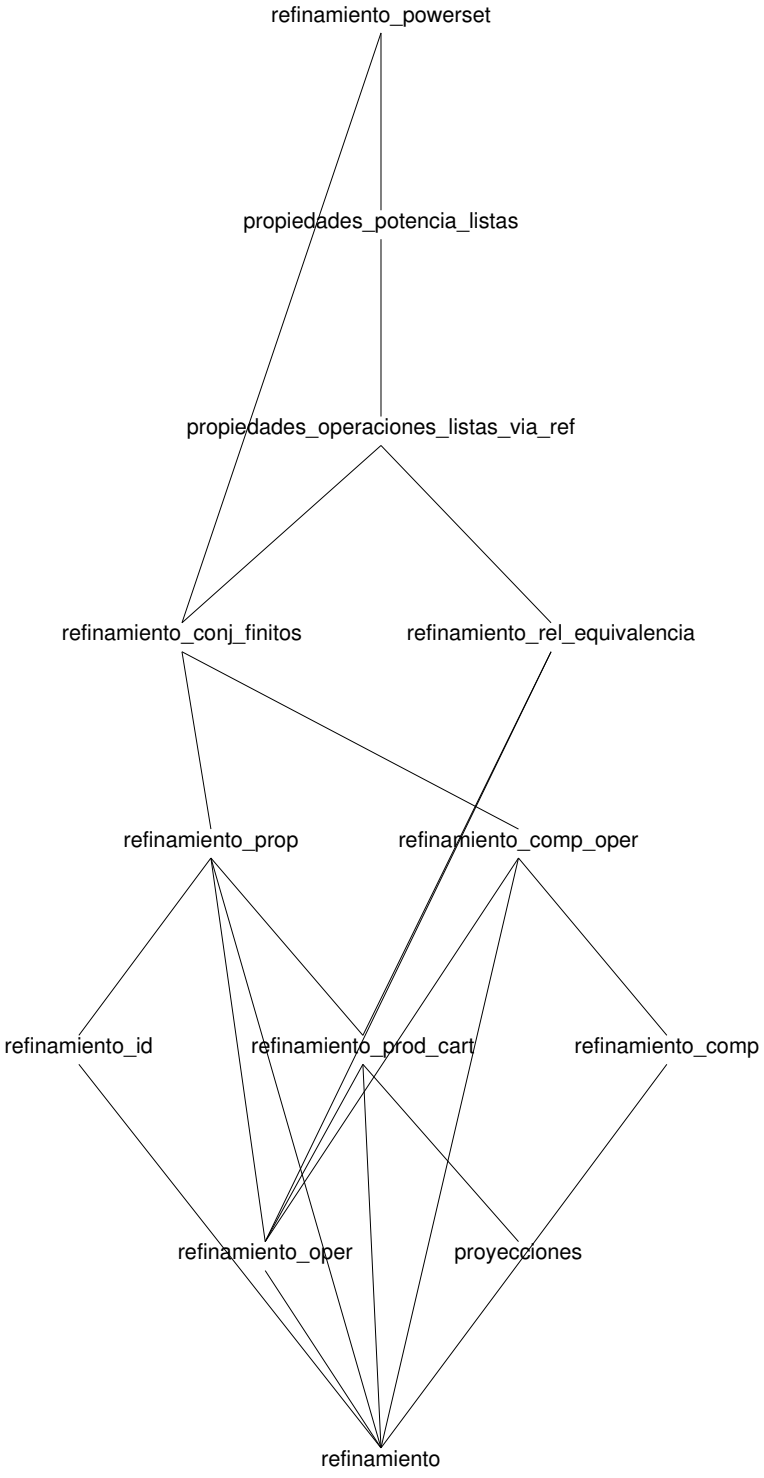
Una idea del trabajo de formalización desarrollado se puede apreciar en la siguiente tabla:

Biblioteca	Lemas	Sintaxis	Tipos	Pruebas
Conjuntos	66	0.01	27.29	9.69
Listas	114	0.00	22.19	16.66
Marco para refinamiento	234	0.01	150.52	53.63
Programación lógica proposicional	479	0.01	564.26	202.64
Análisis formal de conceptos	296	0.02	335.29	191.00

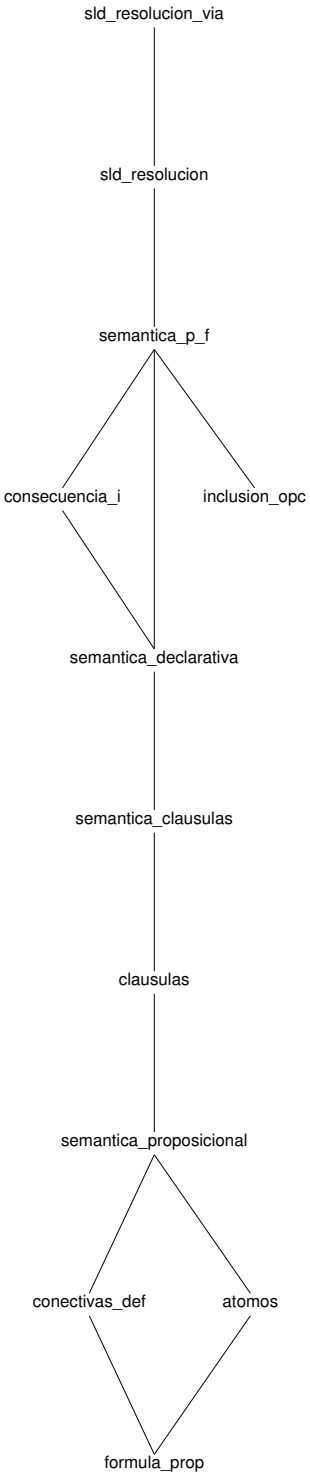
donde la primera columna corresponde a las bibliotecas de teorías recogidas en los apéndices, la segunda corresponde al número de lemas probados en cada una de ellas; y las siguientes al tiempo, en segundos, empleado por el sistema en el análisis sintáctico, la verificación de tipos y la prueba de los lemas de cada una de las teorías.

En esta tabla puede observarse que el tiempo empleado en la verificación de tipos es dos o tres veces superior al tiempo usado en la prueba de los lemas de una teoría. Esto es debido a que las pruebas de las condiciones de corrección de tipos (TCCs) se hacen de manera automática por el sistema, usando básicamente la estrategia `grind`, mientras que las pruebas de los lemas han sido optimizadas por el usuario.

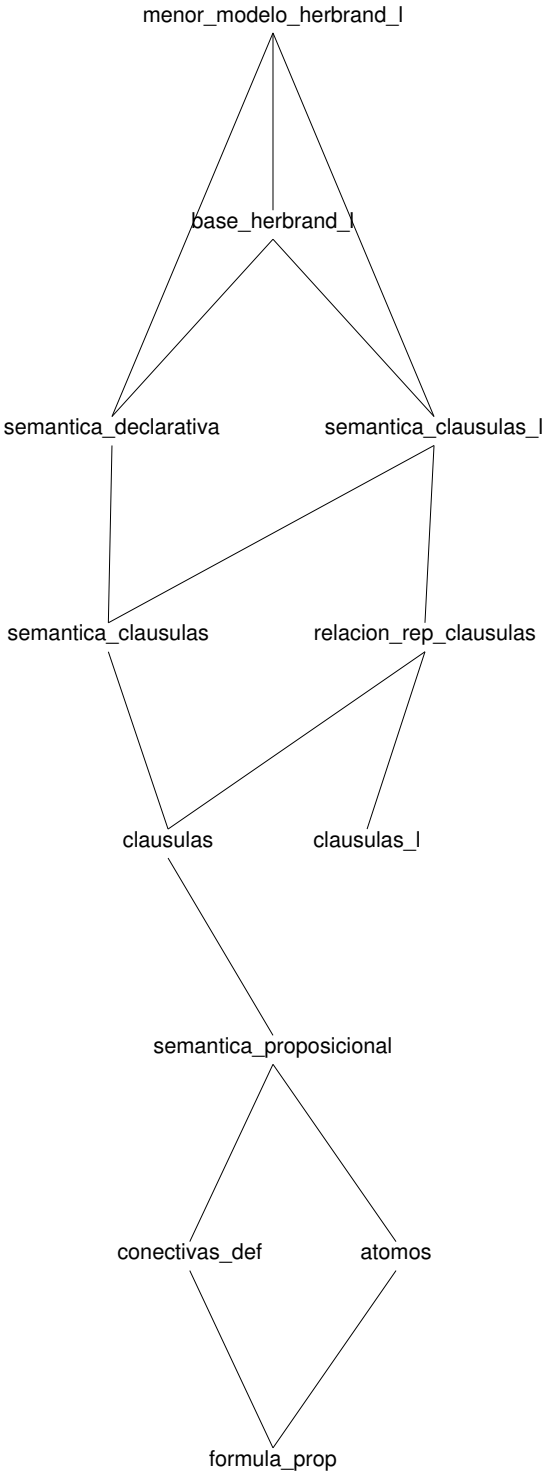
Estas bibliotecas están constituidas por teorías relacionadas entre sí por el mecanismo de importación de PVS. En los gráficos que incluimos en las páginas siguientes, se muestra el grado de dependencia de dichas teorías.



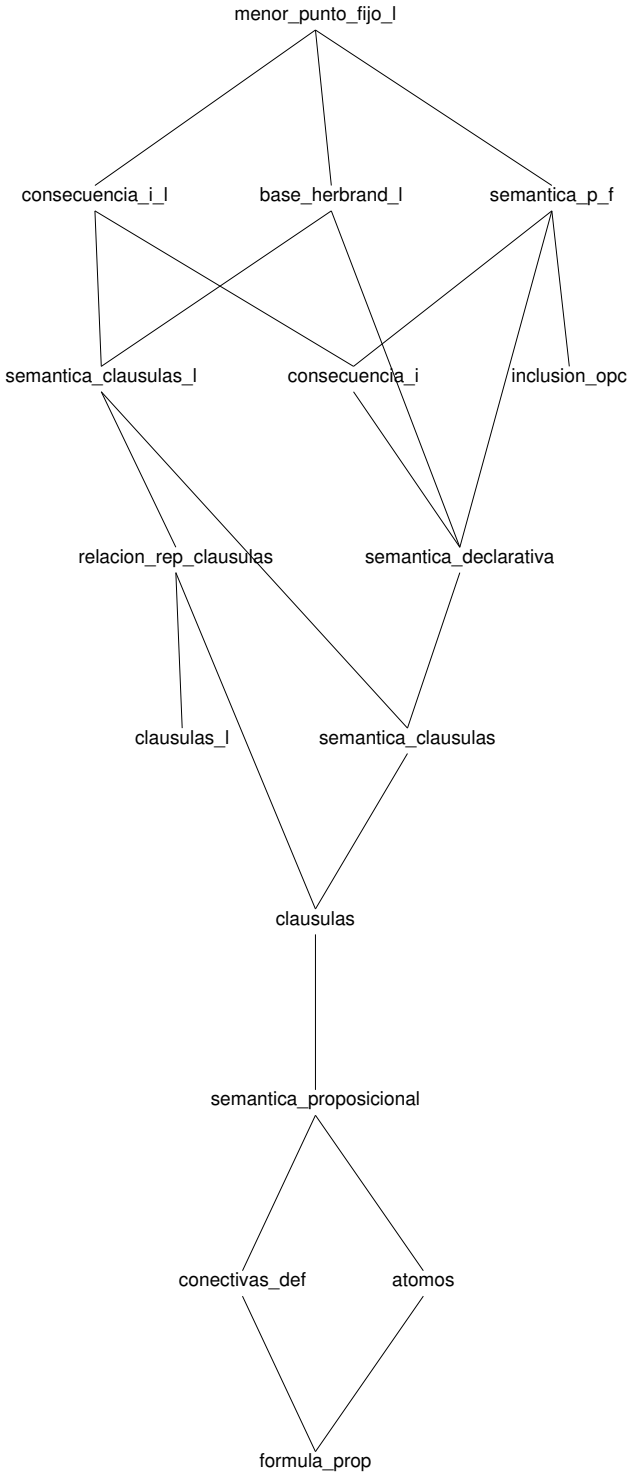
Jerarquía de teorías correspondiente al refinamiento del conjunto potencia.



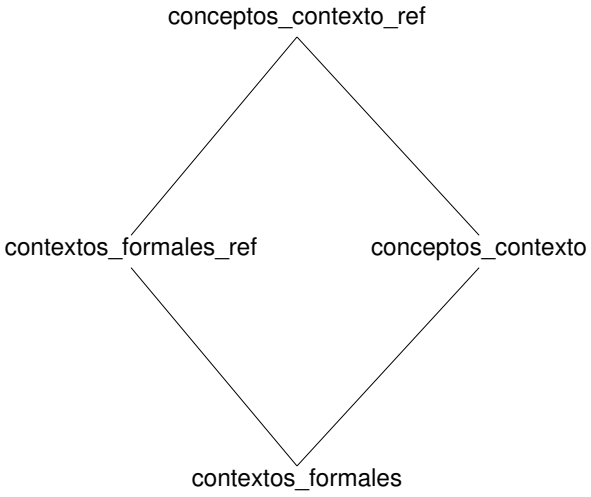
Jerarquía de teorías correspondiente a la prueba de la completitud fuerte de la resolución SLD.



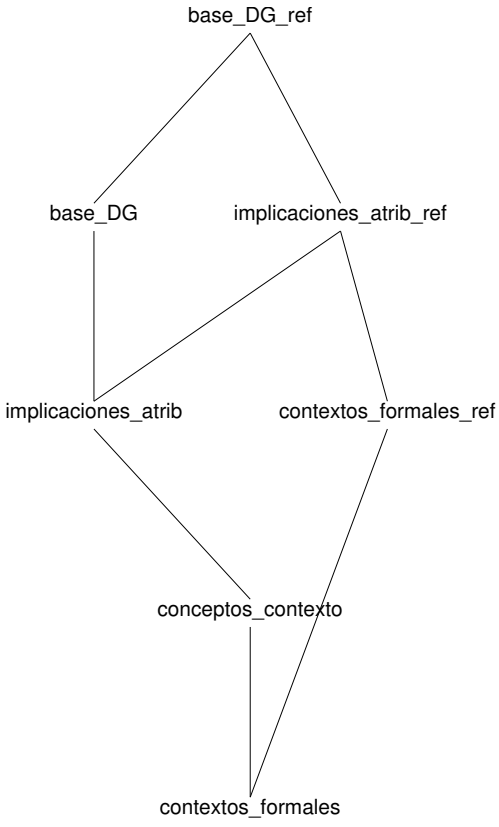
Jerarquía de teorías correspondiente al algoritmo que calcula el menor modelo de Herbrand.



Jerarquía de teorías correspondiente al algoritmo que calcula el menor punto fijo.



Jerarquía de teorías correspondiente al algoritmo de generación de los conceptos de un contexto formal.



Jerarquía de teorías correspondiente al algoritmo de cálculo de una base de implicaciones.

El trabajo presentado en esta memoria puede continuarse en las siguientes líneas:

- En la formalización de la programación lógica nos hemos limitado a desarrollar la programación lógica proposicional. Este trabajo puede extenderse, como se hace en [50], para incluir negación, predicados de primer orden (*datalog*), funciones (*Prolog puro*), corte y predicados aritméticos (*Prolog*). Para estas extensiones es esencial la formalización del algoritmo de unificación del que, actualmente, hemos realizado una primera formalización basada en el algoritmo de Martelli–Montanari. Una vez realizada una verificación formal de este algoritmo, abordaríamos la tarea de obtener, mediante refinamientos de las funciones que se usan, otras especificaciones del mismo, correctas y evaluables.

  Por otra parte, los distintos algoritmos de resolución dependen fundamentalmente de los algoritmos de búsqueda en árboles de computación. En este sentido, sería interesente elaborar especificaciones de los distintos procesos de búsqueda evaluables y correctas. En [24], Dold desarrolla un tratamiento formal en PVS de los pasos de transformación en el desarrollo de programas. Sería interesente analizar si la metodología de refinamientos presentada se puede aplicar a los esquemas de procesos desarrollados por Dold, para obtener algoritmos evaluables. Estos se podrán usar, posteriormente, para obtener algoritmos que implementen el proceso de resolución SLD.

- El análisis formal de conceptos es una teoría sobre la que se han desarrollado algoritmos que se aplican al descubrimiento del conocimiento o a la minería de datos, como comentamos en el capítulo 5. El trabajo iniciado en esta memoria va encaminado hacia la obtención de implementaciones de dichos algoritmos, verificadas formalmente. En este sentido, el trabajo desarrollado puede continuarse en tres direcciones:

  - La verificación formal de otros algoritmos de la teoría, o de otros algoritmos más eficientes, tanto para la generación de los conceptos como para la obtención de una base de implicaciones.
  - La ampliación del fragmento de teoría formalizado.
  - La inmersión de los fundamentos del análisis formal de conceptos en la teoría de operadores de clausura.

- Otra línea de trabajo es la búsqueda de relaciones entre las bases de reglas de un contexto formal finito y los programas lógicos proposicionales. Como consecuencia, se podrían obtener resultados de una de las teorías a partir de la otra.

- En cuanto a la metodología de refinamientos desarrollada para relacionar especificaciones de un mismo algoritmo, creemos que es lo suficientemente

robusta para garantizar su aplicabilidad a otras teorías, con alto contenido matemático y algorítmico. En particular, a teorías ya formalizadas en PVS, como las desarrolladas por un equipo de desarrolladores de la NASA, encabezados por R. Butler [15]. O bien, a las extensiones de las bibliotecas de la programación lógica proposicional y el análisis formal de conceptos, que hemos comentado en los puntos anteriores. Queda también abierta la posibilidad de construir otros refinamientos de los algoritmos desarrollados en este trabajo, usando tipos de datos más eficientes, bien directamente a partir de las especificaciones más abstractas, o bien a partir de especificaciones ya refinadas.

Bibliografía

[1] Amphion.
`http://ase.arc.nasa.gov/docs/amphion.html/`.

[2] FDL: A Prototype Formal Digital Library.
`http://www.nuprl.org/FDLproject`.

[3] Logosphere. A Formal Digital Library.
`http://www.logosphere.org`.

[4] The QED Manifesto.
`http://www-unix.mcs.anl.gov/qed`.

[5] K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[6] K. R. Apt y M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[7] G. Arévalo y T. Mens. Analysing object-oriented application frameworks using concept analysis. En *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 53–63. Springer-Verlag, 2002.

[8] F. Bartels, A. Dold, P. H., F. W. Henke, y H. Ruess. Formalizing fixed–point theory in PVS. Technical Report UIB–96–10, Fakultät für Informatik, 1996.

[9] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, y L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2):66–75, 2000.

[10] G. Boole. The calculus of logic. *The Cambridge and Dublin Mathematical Journal*, 3:183–198, 1848.

[11] N. Bourbaki. *Theory of Sets (Elements of Mathematics).* Addison–Wesley, 1968.

[12] J. Bowen. Formal Methods.
`http://vl.fmnet.info`.

[13] R. Boyer y J. S. Moore. Nqthm, the Boyer–Moore theorem prover. `http://www.cs.utexas.edu/users/boyer/ftp/nqthm/index.html`.

[14] S. Burris. *Logic for Mathematics and Computer Science.* Prentice–Hall, 1998.

[15] R. Butler. NASA Langley PVS Libraries. `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[16] J. L. Caldwell. Classical propositional decidability via Nuprl proof extraction. *Lecture Notes in Computer Science*, 1479, 1998.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, y J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[18] R. Cole, P. Eklund, y D. Wlaker. Using conceptual scaling in formal concept analysis for knowledge and data discovery in medical texts. En *International Symposium on Knowledge Retrieval, Use, and Storage for Efficiency*, pages 151–164, 1997.

[19] R. Cole y G. Stumme. CEM: A conceptual email manager. En *7th International Conference on Conceptual Structures, ICCS'2000.* Springer-Verlag, 2000.

[20] R. Constable, S. Allen, H. Bromely, W. Cleveland, y otros. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, Inc., 1986.

[21] M. Davis y H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215.

[22] K. Doets. *From Logic to Logic Programming.* MIT Press, 1994.

[23] A. Dold. Representing, verifying and applying software development steps using the PVS system. En V. S. Alagar y M. Nivat, editors, *Algebraic Methodology and Software Technology, AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 431–445, Montreal, Canada, July 1995. Springer-Verlag.

[24] A. Dold. *Formal Software Development using Generic Development Steps.* PhD thesis, Universität Ulm, Germany, 2000.

[25] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, P. C., C. Paulin-Mohring, y B. Werner. The Coq proof assistant. `http://coq.inria.fr`.

[26] B. Elspas, M. Green, M. Moriconi, y R. Shostak. A JOVIAL verifier. Technical report, Computer Science Laboratory, SRI International, 1979.

[27] H. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[28] M. Erdmann. Formal concept analysis to learn from the Sisyphus-III material. En *Proceedings of the 11th knowledge Adquisition for Knowledge–Based Systems Workshop (KAN'98)*, 1998.

[29] W. M. Farmer. Theory interpretation in simple type theory. *In Proceedings, International Workshop on Higher Order Algebra, Logic and Term Rewriting (HOA '93)*, 1994.

[30] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, New York, 1990.

[31] Formal Methods Program. *Formal Methods Roadmap: PVS, ICS, and SAL.* Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 2003.

[32] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, y J. Meseguer. Principles of OBJ2. En *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 52–66. ACM Press, 1985.

[33] B. Ganter y R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer Verlag, 1999.

[34] H. Gelernter. Realization of a geometry–theorem proving machine. En U. House, editor, *Proc. Intl. Conf. on Information Processing*, pages 273–282, 1959.

[35] P. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal Research Development.*, 4:28–35.

[36] M. Gordon y T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[37] M. Gordon, R. Milner, y C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer–Verlag, 1979.

[38] C. Graciani-Díaz. *Especificación y verificación de programas moleculares en PVS.* PhD thesis, Universidad de Sevilla, 2003.

[39] J. L. Guigues y V. Duquenne. Familles minimales d'implications informatives resultant d'un tableau de donnees binaires. *Mathèmatiques et Sciences Humaines*, pages 1–18, 1986.

[40] J. Hooman. Correctness of real time systems by construction. En H. Langmaack, W.-P. de Roever, y J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, Lübeck, Germany, Sept. 1994. Springer-Verlag.

[41] M. Jaume. A full formalisation of SLD-resolution in the calculus of inductive constructions. *Journal of Automated Reasoning, Special Issue on Formal Proof*, 23(3–4):347–371, 1999.

[42] S. D. Johnson, P. S. Miner, y A. Camlleri. Studies of the single-pulser in various reasoning systems. En R. Kumar y T. Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, Sept. 1994. Springer-Verlag.

[43] C. B. Jones. *Systematic Software Development using VDM (second edition).* Prentice Hall, 1990.

[44] S. Kahrs, D. Sannella, y A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[45] M. Kaufmann y J. S. Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciencies, University of Texas at Austin, 1998.

[46] D. E. Knuth y P. B. Bendix. Simple word problems in universal algebras. En J. Leech, editor, *Computational problems in abstract algebras*, pages 263–297. Pergamon Press, 1970.

[47] L. Lamport y L. C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[48] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 2 edition, 1987.

[49] D. W. Loveland. Automated Theorem Proving: A quarter century review. En W. W. Bledsoe y D. W. Loveland, editors, *Contemporary Mathematics: Automated Theorem Proving - After 25 Years*, pages 1–45. American Mathematical Society, Providence, RI, 1984.

[50] D. Maier y D. S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings Publishing Co., Menlo Park, CA, 1988.

[51] F. Martín Mateos. *Teoría computacional (en ACL2) sobre cálculos proposicionales.* PhD thesis, Universidad de Sevilla, 2002.

[52] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, pages 263–276.

[53] W. McCune. Otter: An Automated Deduction System, 2001. `http://www-unix.mcs.anl.gov/AR/otter/`.

[54] I. Medina-Bulo. *Verificación formal en ACL2 del algoritmo de Buchberger.* PhD thesis, Universidad de Sevilla, 2003.

[55] P. M. Melliar-Smith y J. Rushby. The enhanced HDM system for specification and verification. En *Proc. VerkShop III*, pages 41–43, Watsonville, CA, Feb. 1985. Publicado en ACM Software Engineering Notes, Vol. 10, No. 4, Agosto. 85.

[56] R. Milner, M. Tofte, y R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[57] A. Nerode y R. A. Shore. *Logic for Applications.* Springer-Verlag, New York, NY, 1993.

[58] J. S. Newell, J. Shaw. Empirical explorations of the logic theory machine. *Fiegenbaum and Feldman (eds), Computers and Thought.*

[59] S. Owre, J. Rushby, y S. N. PVS: A propotype of verification system. `http://pvs.csl.sri.com`.

[60] S. Owre, J. Rushby, N. Shankar, y F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[61] S. Owre y N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 1997.

[62] S. Owre y N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, Abril 2001.

[63] S. Owre, N. Shankar, J. M. Rushby, y D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[64] S. Owre, N. Shankar, J. M. Rushby, y D. W. J. Stringer-Calvert. *PVS System Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[65] N. Pasquier, Y. Bastide, R. Taouil, y L. Lakhal. Discovering frequent closed itemsets for association rules. *Lecture Notes in Computer Science*, 1540:398–416, 1999.

[66] L. Paulson. Isabelle.
`http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[67] L. Paulson y T. Nipkow. Isabelle: a generic theorem prover. *Lecture Notes in Computer Science*, 825.

[68] J. Pei, J. Han, y R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. En *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[69] P. S. Rajan. Transformations in high-level synthesis: formal specification and efficient mechanical verification. Technical Report SRI-CSL-94-10, Computer Science Laboratory, SRI International, 1994.

[70] W. Reif. The KIV system: systematic construction of verified software. En D. Kapur, editor, *11th Conference on Automated Deduction. Proceedings*, Lecture Notes in Computer Science. Albany, NY, USA, Springer, 1992.

[71] J. A. Robinson. A machine-oriented logic based on resolution principle. *Journal of the ACM*, 12(1):23–49, Jan. 1965.

[72] L. Robinson y K. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, Apr. 1976.

[73] L. Robinson, K. Levitt, y B. A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.

[74] J. Ruiz-Reina. *Una teoría computacional acerca de la lógica ecuacional*. PhD thesis, Universidad de Sevilla, 2001.

[75] J. Ruiz-Reina, J. Alonso, M. Hidalgo, y F. Martín-Mateos. Formalizing rewriting in the ACL2 Theorem Prover. En *Artificial Intelligence and Symbolic Computation - AISC'2000*, number 1930 in LNAI, pages 92–106, Berlin, 2000. Springer–Verlag.

[76] J. Rushby, F. von Henke, y S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1991.

[77] C. Schwarzweller. Mizar formalization of concept lattices. *Mechanized Mathematics and its Applications*, 1(1):1–10, 2000.

[78] N. Shankar. Verification of real-time systems using PVS. En C. Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, Junio/Julio 1993. Springer-Verlag.

[79] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

[80] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 1999.

[81] N. Shankar, S. Owre, J. M. Rushby, y D. W. J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[82] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, 1967.

[83] R. E. Shostak, R. Schwartz, y P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. En D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

[84] J. U. Skakkebæk y N. Shankar. Towards a duration calculus proof assistant in PVS. En H. Langmaack, W.-P. de Roever, y J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, Sept. 1994. Springer-Verlag.

[85] D. R. Smith. KIDS: A knowledge-based software development system. En M. Lowry y R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[86] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[87] Y. V. Srinivas y R. Jullig. Specware: formal support for composing software. En *Mathematics of Program Construction*, pages 399–422, 1995.

[88] M. K. Srivas y S. P. Miller. Applying formal verification to the AAMP5 microprocessor: a case study in the industrial use of formal methods. *Form. Methods Syst. Des.*, 8(2):153–188, 1996.

[89] R. Stärk. A direct proof for the completeness of SLD-Resolution. En E. Börger, H. K. Büning, y M. M. Richter, editors, *Computer Science Logic, selected papers from CSL '89*, pages 382–383. Springer-Verlag, Lecture Notes in Computer Science 440, 1990.

[90] G. Stumme. Formal concept analysis on its way from mathematics to computer science. En *Proceedings of the 10th International Conference on Conceptual Structures*, pages 2–19. Springer-Verlag, 2002.

[91] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, y L. Lakhal. Computing iceberg concept lattices with TITANIC. *Data Knowl. Eng.*, 42(2):189–222, 2002.

[92] G. Stumme, R. Wille, y U. Wille. Conceptual knowledge discovery in databases using formal concept analysis methods. En *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 450–458. Springer-Verlag, 1998.

[93] A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[94] A. Trybulec y H. Blair. Computer assisted reasoning with MIZAR. En A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.

[95] Z. Trybulec y H. Swiczkowska. The language of mathematical texts. *Studies in Logic, Grammar and Rhetoric*, (10/11):103–124, 1992.

[96] M. H. van Emden y R. Kowalski. The semantics of predicate logic as programming language. *Journal of ACM*, 23(4):733–743, 1976.

[97] F. Vogt y R. Wille. TOSCANA - a graphical tool for analyzing and exploring data. En *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 226–233. Springer-Verlag, 1995.

[98] H. Wang. Towars mechanical mathematics. *IBM Journal Res. Develop.*, 4:2–22, 1960.

[99] A. Whitehead y B. Russell. *Principia Mathematica.* Cambridge University Press, 1910.

[100] L. Wos y G. Robinson. The concept of demodulation in theorem proving. *Journal of the ACM*, 14:698–709, 1967.

[101] L. Wos y G. Robinson. Paramodulation and set of support. pages 276–310, 1970.

Apéndice A

Extensión de la teoría de conjuntos

A.1. Operaciones sobre conjuntos finitos

```
operaciones_conj_finitos[T: TYPE]: THEORY
 BEGIN

  x: VAR T

  A, Y: VAR finite_set[T]

  k: VAR nat

  union_f_TCC1: OBLIGATION
    FORALL (S: finite_set[finite_set[T]]):
      is_finite[T]
          (Union[T](extend[setof[T], finite_set[T], bool, FALSE](S)));

  union_f(S: finite_set[finite_set[T]]): finite_set[T] =
      Union(extend[setof[T], finite_set[T], bool, FALSE](S))

  union_f_bd: LEMMA
    FORALL (S1, S2: finite_set[finite_set[T]]):
      S1 = S2 IMPLIES union_f(S1) = union_f(S2)

  intersection_f_TCC1: OBLIGATION
    FORALL (S: non_empty_finite_set[finite_set[T]]):
      is_finite[T]
          (Intersection[T]
               (extend[setof[T], finite_set[T], bool, FALSE](S)));

  intersection_f(S: non_empty_finite_set[finite_set[T]]): finite_set[T] =
      Intersection(extend[setof[T], finite_set[T], bool, FALSE](S))

  intersection_f_bd: LEMMA
```

```
    FORALL (S1, S2: non_empty_finite_set[finite_set[T]]):
      S1 = S2 IMPLIES intersection_f(S1) = intersection_f(S2)

  powerset_f(A: finite_set[T]): finite_set[finite_set[T]] =
      restrict[setof[T], finite_set[T], boolean](powerset(A))

  subconjuntos(A, k): set[finite_set[T]] =
      {Y | subset?(Y, A) AND card(Y) = k}

  subconjuntos_finito: LEMMA
    FORALL (A: finite_set[T], k: upto(card(A))):
      is_finite(subconjuntos(A, k))

  singleton_f(x): finite_set[T] = singleton(x)
 END operaciones_conj_finitos
```

A.2. Propiedades de conjuntos

```
propiedades_conjuntos[T: TYPE]: THEORY
 BEGIN

  x, y: VAR T

  S, S1, S2: VAR set[T]

  FS, FS1: VAR finite_set[T]

  P: VAR pred[T]

  A, B: VAR set[set[T]]

  FS1, FS2: VAR finite_set[finite_set[T]]

  SS1, SS2: VAR set[finite_set[T]]

  cardinal_subconjunto_estricto: LEMMA
    strict_subset?(FS, FS1) IMPLIES card[T](FS) < card[T](FS1)

  vacuidad_pred_falso: LEMMA ({x | FALSE}) = emptyset

  vacuidad_pred_falso_gen: LEMMA
    (FORALL x: NOT P(x)) IMPLIES ({x | P(x)}) = emptyset

  union_vacios: LEMMA empty?(S1) AND empty?(S2) IFF empty?(union(S1, S2))

  union_vacio1: LEMMA empty?(S1) IMPLIES union(S1, S2) = S2

  union_vacio2: LEMMA empty?(S2) IMPLIES union(S1, S2) = S1

  Union_emptyset: LEMMA Union(emptyset[set[T]]) = emptyset[T]
```

```
union_vacia: LEMMA
  FORALL (F: set[set[T]]):
    empty?(Union(F)) IMPLIES (FORALL (X: (F)): empty?(X))

extend_emptyset: LEMMA
  extend[setof[T], finite_set[T], bool, FALSE](emptyset) = emptyset

pertenencia_unitario: LEMMA member(x, singleton(x))

no_vacuidad_unitario: LEMMA NOT empty?(singleton(x))

member_union: CLAIM
  member(x, union(S1, S2)) IFF (member(x, S1) OR member(x, S2))

cs1_subconjunto: LEMMA
  subset?(S1, add(x, S2)) AND NOT member(x, S1) IMPLIES subset?(S1, S2)

cn_add_subset: LEMMA
  subset?(add(x, S1), S2) IMPLIES subset?(S1, S2) AND member(x, S2)

cs_add_subset: LEMMA
  subset?(S1, S2) AND member(x, S2) IMPLIES subset?(add(x, S1), S2)

cs1_member: LEMMA subset?(S1, S2) AND member(x, S1) IMPLIES member(x, S2)

remove_subset: LEMMA
  subset?(S, S1) AND NOT S(x) IMPLIES subset?(S, remove(x, S1))

cs_propiedad_subconjunto: LEMMA
  (FORALL x: member(x, S2) IMPLIES P(x)) AND subset?(S1, S2) IMPLIES
   (FORALL x: member(x, S1) IMPLIES P(x))

cs_propiedad_union_1: LEMMA
  (FORALL x: member(x, union(S1, S2)) IMPLIES P(x)) IMPLIES
   (FORALL x: member(x, S1) IMPLIES P(x))

cs_propiedad_union_2: LEMMA
  (FORALL x: member(x, union(S1, S2)) IMPLIES P(x)) IMPLIES
   (FORALL x: member(x, S2) IMPLIES P(x))

CNS_elemento_conjunto_finito_TCC1: OBLIGATION
  FORALL (FS: finite_set[T], x: T):
    NOT empty?(FS) AND NOT member(x, rest(FS)) IMPLIES nonempty?[T](FS);

CNS_elemento_conjunto_finito: CLAIM
  NOT empty?(FS) IMPLIES
   (member(x, FS) IFF member(x, rest(FS)) OR x = choose(FS))

CNS_elemento_conjunto_finito_2: CLAIM
  NOT empty?(FS) IMPLIES
   (FORALL x: (member(x, FS) IFF member(x, rest(FS)) OR x = choose(FS)))
```

```
descomposicion_conj_no_vacio_TCC1: OBLIGATION
  FORALL (S: set[T]): NOT empty?(S) IMPLIES nonempty?[T](S);

descomposicion_conj_no_vacio: LEMMA
  NOT empty?(S) IMPLIES union(rest(S), singleton[T](choose(S))) = S

prop_todos_conjunto_TCC1: OBLIGATION
  FORALL (FS: finite_set[T]): NOT empty?(FS) IMPLIES nonempty?[T](FS);

prop_todos_conjunto: CLAIM
  NOT empty?(FS) IMPLIES
   (FORALL (P: pred[T]):
      P(choose(FS)) AND (FORALL x: member(x, rest(FS)) IMPLIES P(x)) IFF
       (FORALL x: member(x, FS) IMPLIES P(x)))

CS_Union_subset: LEMMA
  (FORALL (S: (A)): subset?(S, S1)) IMPLIES subset?(Union(A), S1)

Union_union_distributiva_s: LEMMA
  Union(union(A, B)) = union(Union(A), Union(B))

Union_singleton: LEMMA Union(singleton(S)) = S

Union_extend_singleton: LEMMA
  Union(extend[set[T], finite_set[T], bool, FALSE](singleton(FS))) = FS

union_extend_distributiva: LEMMA
  extend[set[T], finite_set[T], bool, FALSE](union(FS1, FS2)) =
   union(extend[set[T], finite_set[T], bool, FALSE](FS1),
         extend[set[T], finite_set[T], bool, FALSE](FS2))

union_extend_distributiva_s: LEMMA
  extend[set[T], finite_set[T], bool, FALSE](union(SS1, SS2)) =
   union(extend[set[T], finite_set[T], bool, FALSE](SS1),
         extend[set[T], finite_set[T], bool, FALSE](SS2))

Intersection_subset: LEMMA
  member(S, A) IMPLIES subset?(Intersection(A), S)

subset_Intersection: LEMMA
  (FORALL (S: (A)): subset?(S1, S)) IMPLIES subset?(S1, Intersection(A))

cs3_subconjunto: LEMMA
  subset?(union(remove(x, S), S1), S2) AND member(x, S2) IMPLIES
   subset?(S, S2)

cs4_subconjunto: LEMMA
  subset?(remove(x, S), S1) AND member(x, S1) IMPLIES subset?(S, S1)

union_remove_add: LEMMA
  member(y, S1) AND NOT member(x, S1) IMPLIES
```

```
    union(remove(y, add(x, S1)), S2) = add(x, union(remove(y, S1), S2))

  choose_subset_1: LEMMA
    FORALL (Z, A: set[T]):
      subset?(Z, A) AND nonempty?(A) AND NOT member(choose(A), Z) IMPLIES
       subset?(Z, rest(A))

  choose_subset_2: LEMMA
    FORALL (Z, A: set[T]):
      subset?(Z, A) AND nonempty?(A) AND member(choose(A), Z) IMPLIES
       (EXISTS (Z1: set[T]):
          subset?(Z1, rest(A)) AND Z = add(choose(A), Z1))

  cardinal_no_unitario: LEMMA
    NOT empty?(FS) AND NOT singleton?(FS) IMPLIES card(FS) > 1

  pertenece_union_1: LEMMA
    member(x, S1) AND NOT member(x, S2) IMPLIES
     remove(x, union(S1, S2)) = union(remove(x, S1), S2)

  pertenece_union_2: LEMMA
    member(x, S2) AND NOT member(x, S1) IMPLIES
     remove(x, union(S1, S2)) = union(S1, remove(x, S2))

  pertenece_union_3: LEMMA
    member(x, S1) AND member(x, S2) IMPLIES
     remove(x, union(S1, S2)) = union(remove(x, S1), remove(x, S2))

  remove_commutativo: LEMMA
    member(x, S) AND member(y, S) AND x /= y IMPLIES
     remove(y, remove(x, S)) = remove(x, remove(y, S))

  cn_pertenece_union: LEMMA
    member(x, union(S1, S2)) IMPLIES
     (member(x, S1) AND NOT member(x, S2)) OR
      (member(x, S2) AND NOT member(x, S1)) OR
       (member(x, S1) AND member(x, S2))

  pertenece_union: LEMMA
    remove(x, union(S1, S2)) = union(remove(x, S1), remove(x, S2))
 END propiedades_conjuntos


propiedades_conjuntos_ref[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x: VAR T

  A, B, C: VAR set[T]
```

```
  AF, BF, CF: VAR finite_set[T]

  cs_remove_subset: LEMMA
    member(x, A) AND subset?(A, add(x, B)) IMPLIES subset?(remove(x, A), B)

  cn_igual_add: LEMMA
    member(x, A) AND remove(x, A) = C IMPLIES A = add(x, C)

  card_subset_strict: LEMMA
    strict_subset?(AF, BF) IMPLIES card(AF) < card(BF)

  subset_neg_l1: LEMMA
    subset?(A, add(x, B)) AND NOT subset?(A, B) IMPLIES member(x, A)

  subset_neg: LEMMA
    subset?(A, add(x, B)) AND NOT subset?(A, B) IMPLIES
     (EXISTS C: subset?(C, B) AND A = add(x, C))

  subset_neg_finito: COROLLARY
    subset?(AF, add(x, BF)) AND NOT subset?(AF, BF) IMPLIES
     (EXISTS CF: subset?(CF, BF) AND AF = add(x, CF))
 END propiedades_conjuntos_ref
```

A.3. Propiedades del mínimo y el máximo

```
min_nat_extension[T: TYPE FROM nat]: THEORY
 BEGIN

  IMPORTING min_nat

  S, S1, S2: VAR non_empty_finite_set[T]

  a, b, x: VAR T

  pertenece_min: LEMMA member(x, S) IMPLIES min(S) <= x

  cota_inferior_min: LEMMA (FORALL (x: (S)): b <= x) IMPLIES b <= min(S)

  IMPORTING finite_sets@finite_sets_minmax[T,
                                           restrict[[real, real],
                                                    [T, T],
                                                    boolean]
                                           (<=)]

  pertenece_max: LEMMA member(x, S) IMPLIES max(S) >= x

  cota_superior_max: LEMMA (FORALL (x: (S)): x <= b) IMPLIES max(S) <= b

  max_singleton: LEMMA max(singleton(x)) = x
```

```
  min_singleton: LEMMA min(singleton(x)) = x

  max_subset: LEMMA subset?(S1, S2) IMPLIES max(S1) <= max(S2)

  min_subset: LEMMA subset?(S1, S2) IMPLIES min(S2) <= min(S1)
END min_nat_extension
```

Apéndice B

Extensión de la teoría de listas

B.1. Operaciones sobre listas

```
operaciones_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x, x1, x2: VAR T

  l, l1, l2, l3, ls: VAR list[T]

  rel: VAR (equivalence?[T])

  member_TCC1: OBLIGATION
    FORALL (a: T, l2: list[T], l: list[T], rel: (equivalence?[T]), x: T):
      l = cons(a, l2) AND NOT rel(a, x) IMPLIES
       length[T](l2) < length[T](l);

  member(rel)(x, l): RECURSIVE bool =
    CASES l OF null: FALSE, cons(a, l2): rel(a, x) OR member(rel)(x, l2)
      ENDCASES
     MEASURE length(l)

  cs1_member_rel: LEMMA
    member(rel)(x1, l) AND rel(x1, x2) IMPLIES member(rel)(x2, l)

  cs2_member_rel: LEMMA member(x, l) IMPLIES member(rel)(x, l)

  cn1_member_rel: LEMMA
    member(rel)(x, l) IMPLIES (EXISTS x1: member(x1, l) AND rel(x, x1))

  igual_es_de_equivalencia: LEMMA (equivalence?[T](=))

  equal_TCC1: OBLIGATION equivalence?[T](=);
```

```
JUDGEMENT = HAS_TYPE (equivalence?[T])

member_rel_id_ext: LEMMA member(=)(x, l) = member(x, l)

member_rel_id: LEMMA member(=) = member

inter_TCC1: OBLIGATION
  FORALL (ls: list[T], x: T, l1: list[T], l2: list[T]):
    l1 = cons(x, ls) AND member(x, l2) IMPLIES
     length[T](ls) < length[T](l1);

inter_TCC2: OBLIGATION
  FORALL (ls: list[T], x: T, l1: list[T], l2: list[T]):
    l1 = cons(x, ls) AND NOT member(x, l2) IMPLIES
     length[T](ls) < length[T](l1);

inter(l1, l2): RECURSIVE list[T] =
  CASES l1
    OF null: null,
       cons(x, ls):
         IF member(x, l2) THEN cons(x, inter(ls, l2))
         ELSE inter(ls, l2)
         ENDIF
    ENDCASES
   MEASURE length(l1)

inter_TCC3: OBLIGATION
  FORALL (ls: list[T], x: T, l1: list[T], l2: list[T],
          rel: (equivalence?[T])):
    l1 = cons(x, ls) AND member(rel)(x, l2) IMPLIES
     length[T](ls) < length[T](l1);

inter_TCC4: OBLIGATION
  FORALL (ls: list[T], x: T, l1: list[T], l2: list[T],
          rel: (equivalence?[T])):
    l1 = cons(x, ls) AND NOT member(rel)(x, l2) IMPLIES
     length[T](ls) < length[T](l1);

inter(rel)(l1, l2): RECURSIVE list[T] =
  CASES l1
    OF null: null,
       cons(x, ls):
         IF member(rel)(x, l2) THEN cons(x, inter(rel)(ls, l2))
         ELSE inter(rel)(ls, l2)
         ENDIF
    ENDCASES
   MEASURE length(l1)

inter_rel_id_ext: LEMMA inter(=)(l1, l2) = inter(l1, l2)

inter_rel_id: LEMMA inter(=) = inter
```

```
elimina_TCC1: OBLIGATION
  FORALL (l2: list[T], y: T, l: list[T], x: T):
    l = cons(y, l2) AND x = y IMPLIES length[T](l2) < length[T](l);

elimina_TCC2: OBLIGATION
  FORALL (l2: list[T], y: T, l: list[T], x: T):
    l = cons(y, l2) AND NOT x = y IMPLIES length[T](l2) < length[T](l);

elimina(x, l): RECURSIVE list[T] =
  CASES l
    OF null: null,
       cons(y, l2):
         IF x = y THEN elimina(x, l2) ELSE cons(y, elimina(x, l2)) ENDIF
    ENDCASES
   MEASURE length(l)

elimina_TCC3: OBLIGATION
  FORALL (l2: list[T], y: T, l: list[T], rel: (equivalence?[T]), x: T):
    l = cons(y, l2) AND rel(x, y) IMPLIES length[T](l2) < length[T](l);

elimina_TCC4: OBLIGATION
  FORALL (l2: list[T], y: T, l: list[T], rel: (equivalence?[T]), x: T):
    l = cons(y, l2) AND NOT rel(x, y) IMPLIES
     length[T](l2) < length[T](l);

elimina(rel)(x, l): RECURSIVE list[T] =
  CASES l
    OF null: null,
       cons(y, l2):
         IF rel(x, y) THEN elimina(rel)(x, l2)
         ELSE cons(y, elimina(rel)(x, l2))
         ENDIF
    ENDCASES
   MEASURE length(l)

elimina_rel_id_ext: LEMMA elimina(=)(x, l) = elimina(x, l)

elimina_rel_id: LEMMA elimina(=) = elimina

rest_l_TCC1: OBLIGATION
  FORALL (l: list[T]): NOT null?(l) IMPLIES cons?[T](l);

rest_l(l): list[T] = IF null?(l) THEN null ELSE elimina(car(l), l) ENDIF

rest_l(rel)(l): list[T] =
    IF null?(l) THEN null ELSE elimina(rel)(car(l), l) ENDIF

rest_l_rel_id_ext: LEMMA rest_l(=)(l) = rest_l(l)

rest_l_rel_id: LEMMA rest_l(=) = rest_l

sublista?(l1, l2): RECURSIVE bool =
```

```
  CASES l1 OF null: TRUE, cons(x, l): member(x, l2) AND sublista?(l, l2)
    ENDCASES
   MEASURE length(l1)

sublista?(rel)(l1, l2): RECURSIVE bool =
  CASES l1
    OF null: TRUE,
       cons(x, l): member(rel)(x, l2) AND sublista?(rel)(l, l2)
    ENDCASES
   MEASURE length(l1)

sublista_rel_id_ext: LEMMA sublista?(=)(l1, l2) = sublista?(l1, l2)

sublista_rel_id: LEMMA sublista?(=) = sublista?

sublista_implica_sublista_rel: LEMMA
  sublista?(l1, l2) IMPLIES sublista?(rel)(l1, l2)

elimina_duplicados_TCC1: OBLIGATION
  FORALL (l1: list[T], x: T, l: list[T]):
    l = cons(x, l1) AND member(x, l1) IMPLIES
     length[T](l1) < length[T](l);

elimina_duplicados_TCC2: OBLIGATION
  FORALL (l1: list[T], x: T, l: list[T]):
    l = cons(x, l1) AND NOT member(x, l1) IMPLIES
     length[T](l1) < length[T](l);

elimina_duplicados(l): RECURSIVE list[T] =
  CASES l
    OF null: null,
       cons(x, l1):
         IF member(x, l1) THEN elimina_duplicados(l1)
         ELSE cons(x, elimina_duplicados(l1))
         ENDIF
    ENDCASES
   MEASURE length(l)

sin_duplicados?(l): RECURSIVE bool =
  CASES l
    OF null: TRUE, cons(x, l1): NOT member(x, l1) AND sin_duplicados?(l1)
    ENDCASES
   MEASURE length(l)

cardinal_l(l): nat = length(elimina_duplicados(l))

igual_l?(l1, l2): bool = sublista?(l1, l2) AND sublista?(l2, l1)

igual_l?(rel)(l1, l2): bool =
    sublista?(rel)(l1, l2) AND sublista?(rel)(l2, l1)

igual_l_rel_id: LEMMA igual_l?(=) = igual_l?
```

```
igual_l_implica_igual_l_rel: LEMMA
  igual_l?(l1, l2) IMPLIES igual_l?(rel)(l1, l2)

sublista_estricto?(l1, l2): bool =
    sublista?(l1, l2) AND NOT sublista?(l2, l1)

sublista_estricto?(rel)(l1, l2): bool =
    sublista?(rel)(l1, l2) AND NOT sublista?(rel)(l2, l1)

sublista_estricto_rel_id: LEMMA sublista_estricto?(=) = sublista_estricto?

elimina_duplicados_TCC3: OBLIGATION
  FORALL (l1: list[T], x: T, l: list[T], rel: (equivalence?[T])):
    l = cons(x, l1) AND member(rel)(x, l1) IMPLIES
     length[T](l1) < length[T](l);

elimina_duplicados_TCC4: OBLIGATION
  FORALL (l1: list[T], x: T, l: list[T], rel: (equivalence?[T])):
    l = cons(x, l1) AND NOT member(rel)(x, l1) IMPLIES
     length[T](l1) < length[T](l);

elimina_duplicados(rel)(l): RECURSIVE list[T] =
  CASES l
    OF null: null,
       cons(x, l1):
         IF member(rel)(x, l1) THEN elimina_duplicados(rel)(l1)
         ELSE cons(x, elimina_duplicados(rel)(l1))
         ENDIF
    ENDCASES
   MEASURE length(l)

elimina_duplicados_rel_id_ext: LEMMA
  elimina_duplicados(=)(l) = elimina_duplicados(l)

elimina_duplicados_rel_id: LEMMA
  elimina_duplicados(=) = elimina_duplicados

sin_duplicados?(rel)(l): RECURSIVE bool =
  CASES l
    OF null: TRUE,
       cons(x, l1): NOT member(rel)(x, l1) AND sin_duplicados?(rel)(l1)
    ENDCASES
   MEASURE length(l)

sin_duplicados_rel_id_ext: LEMMA
  sin_duplicados?(=)(l) = sin_duplicados?(l)

sin_duplicados_rel_id: LEMMA sin_duplicados?(=) = sin_duplicados?

cardinal_l(rel)(l): nat = length(elimina_duplicados(rel)(l))
```

```
  cardinal_1_rel_id_ext: LEMMA cardinal_1(=)(l) = cardinal_1(l)

  cardinal_1_rel_id: LEMMA cardinal_1(=) = cardinal_1
 END operaciones_listas

operaciones_gen_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING operaciones_listas[T]

  x, x1, x2: VAR T

  l, l1, l2, l3, ls: VAR list[T]

  LL: VAR list[list[T]]

  rel: VAR (equivalence?[T])

  Append_TCC1: OBLIGATION
    FORALL (LL2: list[list[T]], ls: list[T], LL: list[list[T]]):
      LL = cons(ls, LL2) IMPLIES length[list[T]](LL2) < length[list[T]](LL);

  Append(LL): RECURSIVE list[T] =
    CASES LL OF null: null, cons(ls, LL2): append(ls, Append(LL2)) ENDCASES
     MEASURE length(LL)

  Inter_TCC1: OBLIGATION
    FORALL (LL: (cons?[list[T]])):
      NOT null?(cdr(LL)) IMPLIES cons?[list[T]](cdr[list[T]](LL));

  Inter_TCC2: OBLIGATION
    FORALL (LL: (cons?[list[T]])):
      NOT null?(cdr(LL)) IMPLIES
       length[list[T]](cdr[list[T]](LL)) < length[list[T]](LL);

  Inter(LL: (cons?[list[T]])): RECURSIVE list[T] =
    IF null?(cdr(LL)) THEN car(LL) ELSE inter(car(LL), Inter(cdr(LL))) ENDIF
     MEASURE length(LL)

  Inter(rel)(LL: (cons?[list[T]])): RECURSIVE list[T] =
    IF null?(cdr(LL)) THEN car(LL)
    ELSE inter(rel)(car(LL), Inter(rel)(cdr(LL)))
    ENDIF
     MEASURE length(LL)

  Inter_rel_id_ext: LEMMA
    FORALL (LL: (cons?[list[T]])): Inter(=)(LL) = Inter(LL)

  Inter_rel_id: LEMMA Inter(=) = Inter
```

```
 END operaciones_gen_listas
```

B.1.1. Potencia de una lista

```
potencia_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x, y: VAR T

  l, l1, l2, ls: VAR list[T]

  LL: VAR list[list[T]]

  powerset_ref_TCC1: OBLIGATION
    FORALL (l2: list[T], x: T, ls: list[T]):
      ls = cons(x, l2) IMPLIES length[T](l2) < length[T](ls);

  powerset_ref(ls): RECURSIVE list[list[T]] =
    CASES ls
      OF null: (: null :),
         cons(x, l2):
           append(map(LAMBDA (l1: list[T]): cons(x, l1))(powerset_ref(l2)),
                  powerset_ref(l2))
      ENDCASES
     MEASURE length(ls)

  incluye_en_cada_TCC1: OBLIGATION
    FORALL (LL2: list[list[T]], l1: list[T], LL: list[list[T]]):
      LL = cons(l1, LL2) IMPLIES length[list[T]](LL2) < length[list[T]](LL);

  incluye_en_cada(x, LL): RECURSIVE list[list[T]] =
    CASES LL
      OF null: null,
         cons(l1, LL2): cons(cons(x, l1), incluye_en_cada(x, LL2))
      ENDCASES
     MEASURE length(LL)

  powerset_ref_2(ls): RECURSIVE list[list[T]] =
    CASES ls
      OF null: (: null :),
         cons(x, l2):
           LET aux = powerset_ref_2(l2) IN
             append(incluye_en_cada(x, aux), aux)
      ENDCASES
     MEASURE length(ls)
 END potencia_listas
```

B.1.2. Sublistas de tamaño fijo

```
sublistas_cardinal_def[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING propiedades_operaciones_listas[T]

  x, y: VAR T

  l, l1, l2, ls: VAR list[T]

  LL: VAR list[list[T]]

  sublistas_card_aux2_TCC1: OBLIGATION
    FORALL (LL2: list[list[T]], ls: list[T], LL: list[list[T]], x: T):
      LL = cons(ls, LL2) AND member(x, ls) IMPLIES
       length[list[T]](LL2) < length[list[T]](LL);

  sublistas_card_aux2_TCC2: OBLIGATION
    FORALL (LL2: list[list[T]], ls: list[T], LL: list[list[T]], x: T):
      LL = cons(ls, LL2) AND NOT member(x, ls) IMPLIES
       length[list[T]](LL2) < length[list[T]](LL);

  sublistas_card_aux2(x, LL): RECURSIVE list[list[T]] =
    CASES LL
      OF null: null,
         cons(ls, LL2):
           IF member(x, ls) THEN sublistas_card_aux2(x, LL2)
           ELSE cons(cons(x, ls), sublistas_card_aux2(x, LL2))
           ENDIF
      ENDCASES
     MEASURE length(LL)

  sublistas_card_aux1_TCC1: OBLIGATION
    FORALL (a: T, l2: list[T], ls: list[T]):
      ls = cons(a, l2) IMPLIES length[T](l2) < length[T](ls);

  sublistas_card_aux1(ls, LL): RECURSIVE list[list[T]] =
    CASES ls
      OF null: null,
         cons(a, l2):
           append(sublistas_card_aux2(a, LL), sublistas_card_aux1(l2, LL))
      ENDCASES
     MEASURE length(ls)

  sublistas_card_TCC1: OBLIGATION
    FORALL (ls: list[T], k: upto(cardinal_l[T](ls))):
      NOT k = 0 IMPLIES k - 1 >= 0 AND k - 1 <= cardinal_l[T](ls);
```

```
  sublistas_card_TCC2: OBLIGATION
    FORALL (ls: list[T], k: upto(cardinal_l[T](ls))):
      NOT k = 0 IMPLIES k - 1 < k;

  sublistas_card(ls: list[T], k: upto(cardinal_l[T](ls))): RECURSIVE
  list[list[T]] =
    IF k = 0 THEN (: null :)
    ELSE sublistas_card_aux1(ls, sublistas_card(ls, k - 1))
    ENDIF
     MEASURE k
 END sublistas_cardinal_def
```

B.2. Esquemas de inducción

```
induccion_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING operaciones_listas[T]

  l, l1, l2: VAR list[T]

  p: VAR pred[list[T]]

  n: VAR nat

  induction_long: LEMMA
    (FORALL l: p(l)) IFF (FORALL n, l: length(l) = n IMPLIES p(l))

  induccion_f_l: LEMMA
    (FORALL l:
       (FORALL l2: length(l2) < length(l) IMPLIES p(l2)) IMPLIES p(l))
     IMPLIES (FORALL l: p(l))

  induction_cardinal_l: LEMMA
    (FORALL l: p(l)) IFF (FORALL n, l: cardinal_l(l) = n IMPLIES p(l))

  induccion_cardinal_l: LEMMA
    (FORALL l:
       (FORALL l2: cardinal_l(l2) < cardinal_l(l) IMPLIES p(l2)) IMPLIES
        p(l))
     IMPLIES (FORALL l: p(l))
 END induccion_listas
```

B.3. Propiedades

```
propiedades_operaciones_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING operaciones_listas[T]

  x, x1, x2: VAR T

  l, l1, l2, l3, ls: VAR list[T]

  rel: VAR (equivalence?[T])

  member_rel_caract: LEMMA
    member(rel)(x, l) IFF (EXISTS x1: member(x1, l) AND rel(x1, x))

  cn_pertenece_rel: COROLLARY
    member(rel)(x, l) AND rel(x, x1) IMPLIES member(rel)(x1, l)

  inter_cons_1: LEMMA
    member(x, l2) IMPLIES inter(cons(x, l1), l2) = cons(x, inter(l1, l2))

  inter_cons_2: LEMMA
    NOT member(x, l2) IMPLIES inter(cons(x, l1), l2) = inter(l1, l2)

  caract_inter: LEMMA
    member(x, inter(l1, l2)) IFF member(x, l1) AND member(x, l2)

  member_rel_member: LEMMA member(x, l) IMPLIES member(rel)(x, l)

  caract_inter_rel: LEMMA
    member(x, inter(rel)(l1, l2)) IFF member(x, l1) AND member(rel)(x, l2)

  elimina_cons_1: LEMMA
    x = x1 IMPLIES elimina(x, cons(x1, l)) = elimina(x, l)

  elimina_cons_2: LEMMA
    x /= x1 IMPLIES elimina(x, cons(x1, l)) = cons(x1, elimina(x, l))

  pertenece_elimina_lista: LEMMA
    member(x, elimina(x1, l)) IMPLIES member(x, l)

  pertenece_elimina_distinto: LEMMA
    member(x, elimina(x1, l)) IMPLIES x /= x1

  pertenece_lista_elimina: LEMMA
    member(x, l) AND x /= x1 IMPLIES member(x, elimina(x1, l))

  pertenece_rel_elimina_lista: LEMMA
```

```
  member(x, elimina(rel)(x1, l)) IMPLIES member(x, l)

pertenece_rel_elimina_distinto: LEMMA
  member(x, elimina(rel)(x1, l)) IMPLIES NOT rel(x1, x)

pertenece_rel_lista_elimina: LEMMA
  (member(x, l) AND NOT rel(x1, x)) IMPLIES member(x, elimina(rel)(x1, l))

rest_l_cons: LEMMA
  member(x, rest_l(cons(x1, l))) IFF x /= x1 AND member(x, l)

cn_member_rest_l: LEMMA member(x, rest_l(l)) IMPLIES member(x, l)

cn_member_rest_l_rel: LEMMA member(x, rest_l(rel)(l)) IMPLIES member(x, l)

member_or_TCC1: OBLIGATION
  FORALL (l: list[T], x: T): member(x, l) IMPLIES cons?[T](l);

member_or: LEMMA member(x, l) IMPLIES x = car(l) OR member(x, cdr(l))

elimina_no_pertenece: LEMMA NOT (member(x, l)) IMPLIES elimina(x, l) = l

elimina_pertenece_length: LEMMA
  cons?(l) AND member(x, l) IMPLIES length(elimina(x, l)) < length(l)

rest_l_length: LEMMA cons?(l) IMPLIES length(rest_l(l)) < length(l)

caract_sublista?_every: LEMMA
  sublista?(l1, l2) IFF every(LAMBDA (x): member(x, l2))(l1)

caract_sublista?_rel_every: LEMMA
  sublista?(rel)(l1, l2) IFF every(LAMBDA (x): member(rel)(x, l2))(l1)

sublista_cons: LEMMA
  sublista?(l1, l2) IMPLIES sublista?(cons(x, l1), cons(x, l2))

sublista_member_1: LEMMA
  sublista?(l1, cons(x, l2)) AND NOT member(x, l1) IMPLIES
   sublista?(l1, l2)

sublista_car: LEMMA
  cons?(l1) AND sublista?(cdr(l1), l2) AND member(car(l1), l2) IMPLIES
   sublista?(l1, l2)

sublista_null: LEMMA sublista?(null, l)

elimina_duplicados_member: LEMMA
  member(x, elimina_duplicados(l)) IMPLIES member(x, l)

elimina_duplicados_member_2: LEMMA
  member(x, l) IMPLIES member(x, elimina_duplicados(l))
```

```
  elimina_sin_duplicados: LEMMA sin_duplicados?(elimina_duplicados(l))

  elimina_duplicados_member_rel: LEMMA
    member(rel)(x, elimina_duplicados(rel)(l)) IMPLIES member(rel)(x, l)

  elimina_duplicados_member_rel_2: LEMMA
    member(rel)(x, l) IMPLIES member(rel)(x, elimina_duplicados(rel)(l))

  elimina_rel_sin_duplicados: LEMMA
    sin_duplicados?(rel)(elimina_duplicados(rel)(l))

  sin_duplicados_rel_cons: LEMMA
    sin_duplicados?(rel)(cons(x, l1)) AND member(rel)(x1, l1) IMPLIES
     NOT rel(x, x1)

  member_car: LEMMA cons?(l) IMPLIES member(car(l), l)
 END propiedades_operaciones_listas

propiedades_operaciones_gen_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING operaciones_gen_listas[T], induccion_listas

  x, x1, x2: VAR T

  l, l1, l2, l3, ls: VAR list[T]

  LL: VAR list[list[T]]

  rel: VAR (equivalence?[T])

  append_2_cons: LEMMA
    member(x, append(l1, l2)) IFF member(x, l1) OR member(x, l2)

  append_cons: LEMMA
    member(x, Append(cons(l, LL))) IFF member(x, l) OR member(x, Append(LL))

  caract_append_exist: LEMMA
    member(x, Append(LL)) IFF (EXISTS l: member(l, LL) AND member(x, l))

  caract_inter_rel: LEMMA
    member(rel)(x, inter(rel)(l1, l2)) IFF
     member(rel)(x, l1) AND member(rel)(x, l2)

  caract_inter: LEMMA
    member(x, inter(l1, l2)) IFF member(x, l1) AND member(x, l2)

  caract_inter_every_1_rel: LEMMA
    FORALL (LL: (cons?[list[T]]), x: T):
```

```
      member(rel)(x, Inter(rel)(LL)) IMPLIES
       every(LAMBDA (l: list[T]): member(rel)(x, l))(LL)

  caract_Inter_every_1_rel: LEMMA
    FORALL (LL: (cons?[list[T]]), x: T):
      member(rel)(x, Inter(rel)(LL)) IMPLIES
       (FORALL l: member(l, LL) IMPLIES member(rel)(x, l))

  caract_inter_every_1: LEMMA
    FORALL (LL: (cons?[list[T]]), x: T):
      member(x, Inter(LL)) IMPLIES
       every(LAMBDA (l: list[T]): member(x, l))(LL)

  caract_inter_every_2_rel: LEMMA
    FORALL (LL: (cons?[list[T]]), x: T):
      every(LAMBDA (l: list[T]): member(rel)(x, l))(LL) IMPLIES
       member(rel)(x, Inter(rel)(LL))

  caract_inter_every_2: LEMMA
    FORALL (LL: (cons?[list[T]]), x: T):
      every(LAMBDA (l: list[T]): member(x, l))(LL) IMPLIES
       member(x, Inter(LL))
 END propiedades_operaciones_gen_listas

listas_prop_1[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x: VAR T

  l, ls: VAR list[T]

  p, q: VAR pred[T]

  some_exists: LEMMA
    some(p)(ls) IMPLIES (EXISTS (x: T): (member(x, ls) AND p(x)))

  some_exists_2: LEMMA
    (EXISTS (x: T): (member(x, ls) AND p(x))) IMPLIES some(p)(ls)

  every_implies: LEMMA
    ((FORALL (x: T): p(x) IMPLIES q(x)) AND every(p)(ls)) IMPLIES
     every(q)(ls)

  every_car: LEMMA cons?(ls) AND every(p)(ls) IMPLIES p(car(ls))

  every_cdr: LEMMA cons?(ls) AND every(p)(ls) IMPLIES every(p)(cdr(ls))

  every_member: LEMMA member(x, ls) AND every(p)(ls) IMPLIES p(x)
```

```
  cn_longitud_0: LEMMA length(l) = 0 IMPLIES NOT member(x, l)

  cn_longitud_1_TCC1: OBLIGATION
    FORALL (l: list[T]): length(l) = 1 IMPLIES cons?[T](l);

  cn_longitud_1: LEMMA length(l) = 1 IMPLIES (member(x, l) IFF x = car(l))
 END listas_prop_1

pertenece_tipo[T: TYPE+, S: TYPE FROM T]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x: VAR S

  ls: VAR list[S]

  y: VAR T

  member_tipo_lista: LEMMA
    member[T](y, ls) AND S_pred(y) IMPLIES member[S](y, ls)

  member_tipo_lista_1: LEMMA member[T](x, ls) IMPLIES member[S](x, ls)

  member_tipo_lista_2: LEMMA member[S](x, ls) IMPLIES member[T](x, ls)
 END pertenece_tipo
```

Apéndice C

Marco para refinamientos

C.1. Clases de equivalencia

```
equivalence_class[T: TYPE, ==: (equivalence?[T])]: THEORY
 BEGIN

  x, y: VAR T

  equiv_class(x): setof[T] = {y | y == x}

  E: TYPE = {A: setof[T] | EXISTS x: A = equiv_class(x)}

  rep_TCC1: OBLIGATION FORALL (A: E): EXISTS (x: T): TRUE;

  rep_TCC2: OBLIGATION FORALL (A: E): A(epsilon[T](A));

  rep(A: E): (A) = epsilon(A)

  CONVERSION equiv_class


  CONVERSION equivalence_class.rep


  equiv_class_covers: LEMMA FORALL x: EXISTS (A: E): member(x, A)

  equiv_class_separates: LEMMA
    NOT (x == y) => disjoint?(equiv_class(x), equiv_class(y))

  equiv_class_igual: LEMMA
    NOT (disjoint?(equiv_class(x), equiv_class(y))) =>
     equiv_class(x) = equiv_class(y)

  equiv_class_elto: LEMMA member(x, equiv_class(x))
 END equivalence_class
```

C.2. Refinamiento de tipos

```
refinamiento[T, R: TYPE]: THEORY
 BEGIN

  f: VAR [R -> T]

  es_refinamiento?(f): bool = surjective?(f)

  g: VAR (es_refinamiento?[T, R])

  igual?(g)(r1, r2: R): bool = g(r1) = g(r2)

  igual?_es_de_equivalencia: LEMMA equivalence?[R](igual?(g))

  igual?_TCC1: OBLIGATION FORALL (g): equivalence?[R](igual?(g));

  JUDGEMENT igual?(g) HAS_TYPE (equivalence?[R])

  igual?_injective: LEMMA injective?(g) => igual?(g) = =
 END refinamiento

refinamiento_id[T: TYPE]: THEORY
 BEGIN

  IMPORTING refinamiento

  identidad_ref: LEMMA es_refinamiento?[T, T](id)

  id_TCC1: OBLIGATION es_refinamiento?[T, T](id[T]);

  JUDGEMENT id HAS_TYPE (es_refinamiento?[T, T])
 END refinamiento_id
```

C.2.1. Refinamiento inducido en el tipo cociente

```
refinamiento_equiv[T, R: TYPE]: THEORY
 BEGIN

  IMPORTING refinamiento

  IMPORTING equivalence_class

  g: VAR (es_refinamiento?[T, R])

  f_equiv_TCC1: OBLIGATION FORALL (g): equivalence?[R](igual?[T, R](g));

  f_equiv(g)(A: E[R, igual?(g)]): T = g(rep(A))

  f_equiv_bd: LEMMA
```

```
    FORALL (r1, r2: R):
      member(r2, equiv_class[R, igual?(g)](r1)) IMPLIES g(r1) = g(r2)

  clases_equiv_refinamiento: LEMMA
    es_refinamiento?[T, E[R, igual?(g)]](f_equiv(g))

  f_equiv_TCC2: OBLIGATION
    FORALL (g): es_refinamiento?[T, E[R, igual?[T, R](g)]](f_equiv(g));

  JUDGEMENT f_equiv(g) HAS_TYPE (es_refinamiento?[T, E[R, igual?(g)]])

  f_equiv_bd_reciproco: LEMMA
    FORALL (r1, r2: R):
      g(r1) = g(r2) IMPLIES member(r2, equiv_class[R, igual?(g)](r1))

  f_equiv_inyectiva: LEMMA injective?(f_equiv(g))

  f_equiv_biyectiva: LEMMA bijective?(f_equiv(g))
 END refinamiento_equiv
```

C.2.2. Composición de refinamientos

```
refinamiento_comp[T, R, Q: TYPE]: THEORY
 BEGIN

  IMPORTING refinamiento

  f: VAR [R -> T]

  g: VAR [Q -> R]

  comp_es_refinamiento: LEMMA
    es_refinamiento?[T, R](f) AND es_refinamiento?[R, Q](g) =>
     es_refinamiento?[T, Q](f o g)

  f1: VAR (es_refinamiento?[T, R])

  f2: VAR (es_refinamiento?[R, Q])

  oh_TCC1: OBLIGATION
    FORALL (f1, f2): es_refinamiento?[T, Q](o[Q, R, T](f1, f2));

  JUDGEMENT o(f1, f2) HAS_TYPE (es_refinamiento?[T, Q])
 END refinamiento_comp
```

C.2.3. Refinamiento del producto de tipos

```
proyecciones[T1, T2: TYPE]: THEORY
 BEGIN
```

```
  proyeccion_1(par: [T1, T2]): T1 = PROJ_1(par)

  proyeccion_2(par: [T1, T2]): T2 = PROJ_2(par)
 END proyecciones



refinamiento_prod_cart[T1, T2, R1, R2: TYPE]: THEORY
 BEGIN

  f1: VAR [R1 -> T1]

  f2: VAR [R2 -> T2]

  IMPORTING refinamiento

  IMPORTING proyecciones

  IMPORTING refinamiento_oper

  prod_cart(f1, f2)(par: [R1, R2]): [T1, T2] =
      (f1(PROJ_1(par)), f2(PROJ_2(par)))

  prod_cart_es_refinamiento: LEMMA
    es_refinamiento?[T1, R1](f1) AND es_refinamiento?[T2, R2](f2) IMPLIES
     es_refinamiento?[[T1, T2], [R1, R2]](prod_cart(f1, f2))

  g1: VAR (es_refinamiento?[T1, R1])

  g2: VAR (es_refinamiento?[T2, R2])

  prod_cart_TCC1: OBLIGATION
    FORALL (g1, g2):
      es_refinamiento?[[T1, T2], [R1, R2]](prod_cart(g1, g2));

  JUDGEMENT prod_cart(g1, g2) HAS_TYPE
      (es_refinamiento?[[T1, T2], [R1, R2]])

  proyeccion_1_rep_refinamiento: LEMMA
    es_refinamiento_op?[[T1, T2], T1, [R1, R2], R1, prod_cart(g1, g2), g1]
        (proyeccion_1[T1, T2], proyeccion_1[R1, R2])

  proyeccion_2_rep_refinamiento: LEMMA
    es_refinamiento_op?[[T1, T2], T2, [R1, R2], R2, prod_cart(g1, g2), g2]
        (proyeccion_2[T1, T2], proyeccion_2[R1, R2])
 END refinamiento_prod_cart
```

C.3. Refinamiento de operaciones

C.3.1. Definición

```
refinamiento_oper[T1, T2, R1, R2: TYPE, (IMPORTING refinamiento) f1:
                  (es_refinamiento?[T1, R1]), f2:
                  (es_refinamiento?[T2, R2])]: THEORY
 BEGIN

  op: VAR [T1 -> T2]

  op_ref: VAR [R1 -> R2]

  es_refinamiento_op?(op, op_ref): bool =
      FORALL (r1: R1): op(f1(r1)) = f2(op_ref(r1))
 END refinamiento_oper
```

C.3.2. Operación inducida sobre los tipos cociente

```
refinamiento_oper_equiv[T1, T2, R1, R2: TYPE+, (IMPORTING refinamiento) f1:
                        (es_refinamiento?[T1, R1]), f2:
                        (es_refinamiento?[T2, R2])]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;

   R1_TCC1: ASSUMPTION EXISTS (x: R1): TRUE;

   R2_TCC1: ASSUMPTION EXISTS (x: R2): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_oper, refinamiento_equiv, equivalence_class

  op: VAR [T1 -> T2]

  op_ref: VAR [R1 -> R2]

  g1: VAR (es_refinamiento?[T1, R1])

  g2: VAR (es_refinamiento?[T2, R2])

  op_ref_equiv_bd_TCC1: OBLIGATION
    FORALL (op: [T1 -> T2], op_ref: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) IMPLIES
       equivalence?[R1](igual?[T1, R1](f1));

  op_ref_equiv_bd_TCC2: OBLIGATION
    FORALL (op: [T1 -> T2], op_ref: [R1 -> R2]):
```

```
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) IMPLIES
       (FORALL (r, s: R1):
          member(s, equiv_class[R1, igual?(f1)](r)) IMPLIES
           equivalence?[R2](igual?[T2, R2](f2)));

  op_ref_equiv_bd: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) =>
     (FORALL (r, s: R1):
        member(s, equiv_class[R1, igual?(f1)](r)) =>
         equiv_class[R2, igual?(f2)](op_ref(r)) =
          equiv_class[R2, igual?(f2)](op_ref(s)))

  op_ref_equiv_TCC1: OBLIGATION equivalence?[R1](igual?[T1, R1](f1));

  op_ref_equiv_TCC2: OBLIGATION equivalence?[R2](igual?[T2, R2](f2));

  op_ref_equiv_TCC3: OBLIGATION
    FORALL (A: E[R1, igual?[T1, R1](f1)], op_ref: [R1 -> R2]):
      EXISTS (x: R2):
        equiv_class[R2, igual?[T2, R2](f2)](op_ref(rep[R1, igual?(f1)](A)))
         = equiv_class[R2, igual?[T2, R2](f2)](x);

  op_ref_equiv(op, op_ref)(A: E[R1, igual?(f1)]): E[R2, igual?(f2)] =
      equiv_class[R2, igual?(f2)](op_ref(rep(A)))

  op_ref_equiv_es_refinamiento_TCC1: OBLIGATION
    FORALL (op: [T1 -> T2], op_ref: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) IMPLIES
       es_refinamiento?[T1, E[R1, igual?[T1, R1](f1)]](f_equiv[T1, R1](f1));

  op_ref_equiv_es_refinamiento_TCC2: OBLIGATION
    FORALL (op: [T1 -> T2], op_ref: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) IMPLIES
       es_refinamiento?[T2, E[R2, igual?[T2, R2](f2)]](f_equiv[T2, R2](f2));

  op_ref_equiv_es_refinamiento: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) =>
     es_refinamiento_op?[T1, T2, E[R1, igual?(f1)], E[R2, igual?(f2)],
                         f_equiv[T1, R1](f1), f_equiv[T2, R2](f2)]
        (op, op_ref_equiv(op, op_ref))
 END refinamiento_oper_equiv
```

C.3.3. Composición de operaciones refinadas

```
refinamiento_comp_oper[T1, T2, T3, R1, R2, R3: TYPE,
                       (IMPORTING refinamiento) f1:
                       (es_refinamiento?[T1, R1]), f2:
                       (es_refinamiento?[T2, R2]), f3:
                       (es_refinamiento?[T3, R3])]: THEORY
 BEGIN
```

```
  IMPORTING refinamiento_oper, refinamiento_comp

  op1: VAR [T1 -> T2]

  op1_ref: VAR [R1 -> R2]

  op2: VAR [T2 -> T3]

  op2_ref: VAR [R2 -> R3]

  comp_oper_es_refinamiento: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op1, op1_ref) AND
     es_refinamiento_op?[T2, T3, R2, R3, f2, f3](op2, op2_ref)
     IMPLIES
     es_refinamiento_op?[T1, T3, R1, R3, f1, f3]
         (op2 o op1, op2_ref o op1_ref)
 END refinamiento_comp_oper

refinamiento_iteracion[T, R: TYPE, (IMPORTING refinamiento) f:
                       (es_refinamiento?[T, R])]: THEORY
 BEGIN

  IMPORTING refinamiento_oper[T, T, R, R, f, f]

  IMPORTING refinamiento_comp_oper[T, T, T, R, R, R, f, f, f]

  op: VAR [T -> T]

  op_ref: VAR [R -> R]

  n: VAR nat

  iteracion_es_refinamiento: THEOREM
    es_refinamiento_op?(op, op_ref) IMPLIES
     es_refinamiento_op?(iterate(op, n), iterate(op_ref, n))
 END refinamiento_iteracion
```

C.3.4. Conservación de la corrección

```
refinamiento_prop[T1, T2, R1, R2: TYPE, (IMPORTING refinamiento) f1:
                  (es_refinamiento?[T1, R1]), f2:
                  (es_refinamiento?[T2, R2])]: THEORY
 BEGIN

  IMPORTING refinamiento_id, refinamiento_oper, refinamiento_prod_cart

  op: VAR [T1 -> T2]

  op_ref: VAR [R1 -> R2]
```

```
  p: VAR [T1 -> bool]

  p_ref: VAR [R1 -> bool]

  q: VAR pred[[T1, T2]]

  q_ref: VAR pred[[R1, R2]]

  r: VAR pred[T2]

  r_ref: VAR pred[R2]

  ref_preserva_correccion_TCC1: OBLIGATION
    FORALL (op: [T1 -> T2], op_ref: [R1 -> R2], p: [T1 -> bool],
            p_ref: [R1 -> bool], f: [[R1, R2] -> [T1, T2]]):
      f = prod_cart[T1, T2, R1, R2](f1, f2) AND
       es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) AND
        es_refinamiento_op?[T1, bool, R1, bool, f1, id[bool]](p, p_ref)
       IMPLIES es_refinamiento?[[T1, T2], [R1, R2]](f);

  ref_preserva_correccion: LEMMA
    LET f = prod_cart[T1, T2, R1, R2](f1, f2) IN
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref) AND
       es_refinamiento_op?[T1, bool, R1, bool, f1, id[bool]](p, p_ref) AND
        es_refinamiento_op?[[T1, T2], bool, [R1, R2], bool, f, id[bool]]
            (q, q_ref)
         AND (FORALL (x: T1): p(x) IMPLIES q(x, op(x)))
       IMPLIES (FORALL (z: R1): p_ref(z) IMPLIES q_ref(z, op_ref(z)))

  ref_preserva_completitud_relativa: LEMMA
    LET f = prod_cart[T1, T2, R1, R2](f1, f2) IN
           es_refinamiento_op?[T1, T2, R1, R2, f1, f2](op, op_ref)
       AND es_refinamiento_op?[T1, bool, R1, bool, f1, id[bool]](p, p_ref)
       AND es_refinamiento_op?[T2, bool, R2, bool, f2, id[bool]](r, r_ref)
       AND injective?(f2)
       AND (FORALL (y: T2):
              r(y) IMPLIES (EXISTS (x: T1): p(x) AND y = op(x)))
       IMPLIES
       (FORALL (u: R2):
          r_ref(u) IMPLIES (EXISTS (z: R1): p_ref(z) AND u = op_ref(z)))
 END refinamiento_prop
```

C.3.5. Refinamiento de las relaciones de equivalencia

```
refinamiento_rel_equivalencia[T, R: TYPE]: THEORY
 BEGIN

  IMPORTING refinamiento_prod_cart, refinamiento_oper
```

```
  f: VAR (es_refinamiento?[T, R])

  ro: VAR pred[[T, T]]

  phi: VAR pred[[R, R]]

  refinamiento_preserva_rel_equivalencia_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R])):
      es_refinamiento?[[T, T], [R, R]](prod_cart[T, T, R, R](f, f));

  refinamiento_preserva_rel_equivalencia_TCC2: OBLIGATION
    es_refinamiento?[boolean, boolean](id[bool]);

  refinamiento_preserva_rel_equivalencia: THEOREM
    es_refinamiento_op?[[T, T], bool, [R, R], bool, prod_cart(f, f),
                        id[bool]]
        (ro, phi)
     AND equivalence?(ro)
     IMPLIES equivalence?(phi)
 END refinamiento_rel_equivalencia
```

C.4. Refinamiento: conjuntos finitos

```
refinamiento_conj_finitos[T, R: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   R_TCC1: ASSUMPTION EXISTS (x: R): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_prop

  f: VAR (es_refinamiento?[T, R])

  x, x1, x2: VAR R

  y, y1, y2: VAR T

  l, l1, l2, l3, ls: VAR list[R]

  rel: VAR (equivalence?[R])

  IMPORTING finite_sets@finite_sets_inductions

  TL: LIBRARY = "../auxiliares/listas"

  IMPORTING TL@propiedades_operaciones_listas[R]

  c_TCC1: OBLIGATION
```

```
  FORALL (l1: list[R], x: R, f: (es_refinamiento?[T, R]), l: list[R]):
    l = cons(x, l1) IMPLIES length[R](l1) < length[R](l);

c(f)(l: list[R]): RECURSIVE finite_set[T] =
  CASES l OF null: emptyset, cons(x, l1): add(f(x), c(f)(l1)) ENDCASES
   MEASURE length(l)

imagen_c_caract: LEMMA
  member(y, c(f)(l)) IFF (EXISTS x: member(x, l) AND y = f(x))

c_elimina_duplicados: LEMMA c(f)(elimina_duplicados(l)) = c(f)(l)

imagen_c_caract_rel_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])): equivalence?[R](igual?[T, R](f));

imagen_c_caract_rel: LEMMA
  member(y, c(f)(l)) IFF (EXISTS x: member(igual?(f))(x, l) AND y = f(x))

c_elimina_duplicados_rel: LEMMA
  c(f)(elimina_duplicados(igual?(f))(l)) = c(f)(l)

c_es_refinamiento: LEMMA es_refinamiento?[finite_set[T], list[R]](c(f))

c_TCC2: OBLIGATION
  FORALL (f): es_refinamiento?[finite_set[T], list[R]](c(f));

JUDGEMENT c(f) HAS_TYPE (es_refinamiento?[finite_set[T], list[R]])

null?_es_refinamiento_empty?: THEOREM
  es_refinamiento_op?[finite_set[T], bool, list[R], bool, c(f), id[bool]]
      (restrict[set[T], finite_set[T], boolean](empty?), null?)

null?_es_refinamiento_empty?_corol: COROLLARY null?(l) IFF empty?(c(f)(l))

cons_es_refinamiento_add_l1: LEMMA add(f(x), c(f)(l)) = c(f)(cons(x, l))

cons_es_refinamiento_add_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])):
    es_refinamiento?[[T, finite_set[T]], [R, list[R]]]
        (prod_cart[T, finite_set[T], R, list[R]](f, c(f)));

cons_es_refinamiento_add: THEOREM
  es_refinamiento_op?[[T, finite_set[T]], finite_set[T], [R, list[R]],
                      list[R], prod_cart(f, c(f)), c(f)]
      (restrict[[T, set[T]], [T, finite_set[T]], (nonempty?[T])](add),
       cons)

member_es_refinamiento_member_l2: LEMMA
  member(x, l) IMPLIES member(f(x), c(f)(l))

member_rel_es_refinamiento_member_lema: LEMMA
  member(f(x), c(f)(l)) IFF member(igual?(f))(x, l)
```

```
member_rel_es_refinamiento_member: THEOREM
  es_refinamiento_op?[[T, finite_set[T]], bool, [R, list[R]], bool,
                      prod_cart(f, c(f)), id[bool]]
      (restrict[[T, set[T]], [T, finite_set[T]], boolean](member),
       member(igual?(f)))

member_member_rel: LEMMA injective?(f) IMPLIES member(igual?(f)) = member

member_es_refinamiento_member: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[[T, finite_set[T]], bool, [R, list[R]], bool,
                       prod_cart(f, c(f)), id[bool]]
       (restrict[[T, set[T]], [T, finite_set[T]], boolean](member),
        member)

append_es_refinamiento_union_l1: LEMMA
  union(c(f)(l1), c(f)(l2)) = c(f)(append(l1, l2))

append_es_refinamiento_union_TCC1: OBLIGATION
  FORALL (x1: [finite_set[T], finite_set[T]]): is_finite[T](union[T](x1));

append_es_refinamiento_union_TCC2: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])):
    es_refinamiento?[[finite_set[T], finite_set[T]], [list[R], list[R]]]
        (prod_cart[finite_set[T], finite_set[T], list[R], list[R]]
             (c(f), c(f)));

append_es_refinamiento_union: LEMMA
  es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                      [list[R], list[R]], list[R], prod_cart(c(f), c(f)),
                      c(f)]
      (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]], set[T]]
           (union),
       append)

intersection_singleton: LEMMA
  FORALL (B: set[T], x: T):
    member(x, B) IMPLIES intersection(B, singleton(x)) = singleton(x)

intersection_singleton_2: LEMMA
  FORALL (B: set[T], x: T):
    NOT member(x, B) IMPLIES intersection(B, singleton(x)) = emptyset

intersection_add_1: LEMMA
  FORALL (A, B: set[T], x: T):
    member(x, B) IMPLIES
     intersection(add(x, A), B) = add(x, intersection(A, B))

intersection_add_2: LEMMA
  FORALL (A, B: set[T], x: T):
    NOT member(x, B) IMPLIES
```

```
      intersection(add(x, A), B) = intersection(A, B)

  inter_rel_es_refinamiento_intersection_lema: LEMMA
    c(f)(inter(igual?(f))(l1, l2)) = intersection(c(f)(l1), c(f)(l2))

  inter_rel_es_refinamiento_intersection_TCC1: OBLIGATION
    FORALL (x1: [finite_set[T], finite_set[T]]):
      is_finite[T](intersection[T](x1));

  inter_rel_es_refinamiento_intersection: THEOREM
    es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                        [list[R], list[R]], list[R], prod_cart(c(f), c(f)),
                        c(f)]
       (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]], set[T]]
            (intersection),
        inter(igual?(f)))

  inter_inter_rel: LEMMA injective?(f) => inter(igual?(f)) = inter

  inter_es_refinamiento_intersection: COROLLARY
    injective?(f) IMPLIES
     es_refinamiento_op?[[finite_set[T], finite_set[T]], finite_set[T],
                         [list[R], list[R]], list[R],
                         prod_cart(c(f), c(f)), c(f)]
        (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]], set[T]]
             (intersection),
         inter)

  elimina_rel_es_refinamiento_remove_lema: LEMMA
    remove(f(x), c(f)(l)) = c(f)(elimina(igual?(f))(x, l))

  elimina_rel_es_refinamiento_remove_TCC1: OBLIGATION
    FORALL (x1: [T, finite_set[T]]): is_finite[T](remove[T](x1));

  elimina_rel_es_refinamiento_remove: THEOREM
    es_refinamiento_op?[[T, finite_set[T]], finite_set[T], [R, list[R]],
                        list[R], prod_cart(f, c(f)), c(f)]
       (restrict[[T, set[T]], [T, finite_set[T]], set[T]](remove),
        elimina(igual?(f)))

  elimina_elimina_rel: LEMMA
    injective?(f) IMPLIES elimina(igual?(f)) = elimina

  elimina_es_refinamiento_remove: THEOREM
    injective?(f) IMPLIES
     es_refinamiento_op?[[T, finite_set[T]], finite_set[T], [R, list[R]],
                         list[R], prod_cart(f, c(f)), c(f)]
        (restrict[[T, set[T]], [T, finite_set[T]], set[T]](remove),
         elimina)

  car_epsilon: POSTULATE cons?(l) IMPLIES epsilon(c(f)(l)) = f(car(l))
```

```
cnv_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R]), l: (cons?[R])):
    NOT empty?[T](c(f)(l));

cnv(f)(l: (cons?[R])): non_empty_finite_set[T] = c(f)(l)

c_es_refinamiento_no_vacio: LEMMA
  es_refinamiento?[non_empty_finite_set[T], (cons?[R])]
      (restrict[list[R], ((cons?[R])), finite_set[T]](c(f)))

cnv_es_refinamiento_no_vacio: THEOREM
  es_refinamiento?[non_empty_finite_set[T], (cons?[R])](cnv(f))

cnv_TCC2: OBLIGATION
  FORALL (f):
    es_refinamiento?[non_empty_finite_set[T], (cons?[R])](cnv(f));

JUDGEMENT cnv(f) HAS_TYPE
    (es_refinamiento?[non_empty_finite_set[T], (cons?[R])])

JUDGEMENT non_empty_finite_set[T] SUBTYPE_OF (nonempty?[T])

car_es_refinamiento_choose_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])):
    (FORALL (x1: ((cons?[R]))): NOT empty?[T](c(f)(x1))) AND
     es_refinamiento?[non_empty_finite_set[T], ((cons?[R]))]
         (restrict[list[R], ((cons?[R])), finite_set[T]](c(f)));

car_es_refinamiento_choose: THEOREM
  es_refinamiento_op?[non_empty_finite_set[T], T, (cons?[R]), R,
                      restrict[list[R], ((cons?[R])), finite_set[T]]
                          (c(f)),
                      f]
      (restrict[(nonempty?[T]), non_empty_finite_set[T], T](choose), car)

car_es_refinamiento_choose_corol_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R]), l: (cons?[R])):
    nonempty?[T](c(f)(l));

car_es_refinamiento_choose_corol: COROLLARY
  FORALL (l: (cons?[R])): choose(c(f)(l)) = f(car(l))

car_es_refinamiento_choose_2: THEOREM
  es_refinamiento_op?[non_empty_finite_set[T], T, (cons?[R]), R, cnv(f),
                      f]
      (restrict[(nonempty?[T]), non_empty_finite_set[T], T](choose), car)

rest_l_rel_es_refinamiento_rest_lema: LEMMA
  c(f)(rest_l(igual?(f))(l)) = rest(c(f)(l))

rest_l_rel_es_refinamiento_rest_TCC1: OBLIGATION
  FORALL (x1: finite_set[T]): is_finite[T](rest[T](x1));
```

```
rest_l_rel_es_refinamiento_rest: THEOREM
  es_refinamiento_op?[finite_set[T], finite_set[T], list[R], list[R],
                      c(f), c(f)]
      (restrict[set[T], finite_set[T], set[T]](rest), rest_l(igual?(f)))

rest_l_rest_l_rel: LEMMA injective?(f) => rest_l(igual?(f)) = rest_l

rest_l_es_refinamiento_rest: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[finite_set[T], finite_set[T], list[R], list[R],
                       c(f), c(f)]
       (restrict[set[T], finite_set[T], set[T]](rest), rest_l)

ro: VAR pred[T]

phi: VAR pred[R]

every_equiv_forall_ref: LEMMA
  es_refinamiento_op?[T, bool, R, bool, f, id[bool]](ro, phi) IMPLIES
   (every(phi)(l) IFF (FORALL (y: (c(f)(l))): ro(y)))

sublista?_implica_subset?: LEMMA
  sublista?(l1, l2) IMPLIES subset?(c(f)(l1), c(f)(l2))

sublista?_rel_es_refinamiento_subset?_lema: LEMMA
  sublista?(igual?(f))(l1, l2) IFF subset?(c(f)(l1), c(f)(l2))

sublista?_rel__es_refinamiento_subset?: THEOREM
  es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                      [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                      id[bool]]
      (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]], boolean]
           (subset?),
       sublista?(igual?(f)))

cn_sublista_rel: LEMMA
  sublista?(igual?(f))(l1, l2) IMPLIES
   (EXISTS l3: sublista?(l3, l2) AND igual?(c(f))(l1, l3))

sublista_sublista_rel: LEMMA
  injective?(f) => sublista?(igual?(f)) = sublista?

sublista?_es_refinamiento_subset?: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                       [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                       id[bool]]
       (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]],
                 boolean]
            (subset?),
        sublista?)
```

```
sublista?_es_refinamiento_subset?_l1: COROLLARY
  injective?(f) IMPLIES
   (sublista?(l1, l2) IFF subset?(c(f)(l1), c(f)(l2)))

p: VAR pred[list[R]]

n: VAR nat

sin_duplicados_bij_conj_l1: CLAIM
  injective?(f) AND NOT member(x, l) AND member(y, c(f)(l)) IMPLIES
   y /= f(x)

cns_menor_sucesor: CLAIM FORALL (m, n: nat): m < n + 1 IFF m < n OR m = n

sin_duplicados_bij_conj_2: LEMMA
  sin_duplicados?(l) AND injective?(f) IMPLIES
   (EXISTS (g: [(c(f)(l)) -> below[length(l)]]): bijective?(g))

elimina_duplicados_igual_conj: LEMMA
  injective?(f) IMPLIES c(f)(elimina_duplicados(l)) = c(f)(l)

sin_duplicados_rel_bij_conj_l1: LEMMA
  NOT member(igual?(f))(x, l) AND member(y, c(f)(l)) IMPLIES y /= f(x)

sin_duplicados_rel_bij_conj_2: LEMMA
  sin_duplicados?(igual?(f))(l) IMPLIES
   (EXISTS (g: [(c(f)(l)) -> below[length(l)]]): bijective?(g))

elimina_duplicados_rel_igual_conj: LEMMA
  c(f)(elimina_duplicados(igual?(f))(l)) = c(f)(l)

cardinal_l_rel_es_refinamiento_card: THEOREM
  es_refinamiento_op?[finite_set[T], nat, list[R], nat, c(f), id[nat]]
      (card, cardinal_l(igual?(f)))

cardinal_l_caardinal_l_rel: LEMMA
  injective?(f) => cardinal_l(igual?(f)) = cardinal_l

cardinal_l_es_refinamiento_card: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[finite_set[T], nat, list[R], nat, c(f), id[nat]]
       (card, cardinal_l)

igual_c?(A, B: finite_set[T]): bool = A = B

cns_igual_c?: LEMMA
  FORALL (A, B: finite_set[T]):
    igual_c?(A, B) IFF subset?(A, B) AND subset?(B, A)

igual_c?_es_de_equivalencia: LEMMA equivalence?[finite_set[T]](igual_c?)
```

```
igual_l?_rel_es_refinamiento_igual_c: THEOREM
  es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                      [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                      id[bool]]
      (igual_c?, igual_l?(igual?(f)))

igual_l_elimina_duplicados_rel: COROLLARY
  igual_l?(igual?(f))(elimina_duplicados(igual?(f))(l), l)

igual_l_implica_igual_rel: LEMMA
  igual_l?(l1, l2) IMPLIES igual?(c(f))(l1, l2)

igual_l_igual_l_rel: LEMMA injective?(f) => igual_l?(igual?(f)) = igual_l?

igual_l?_es_refinamiento_igual_c: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                       [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                       id[bool]]
       (igual_c?, igual_l?)

igual_l_elimina_duplicados_rel_inyectivo: COROLLARY
  injective?(f) IMPLIES igual_l?(elimina_duplicados(igual?(f))(l), l)

sublista_estricto?_rel_es_refinamiento_strict_subset?: THEOREM
  es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                      [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                      id[bool]]
      (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]], boolean]
           (strict_subset?),
       sublista_estricto?(igual?(f)))

sublista_estricto_sublista_estricto_rel: LEMMA
  injective?(f) => sublista_estricto?(igual?(f)) = sublista_estricto?

sublista_estricto?_es_refinamiento_strict_subset?: COROLLARY
  injective?(f) IMPLIES
   es_refinamiento_op?[[finite_set[T], finite_set[T]], bool,
                       [list[R], list[R]], bool, prod_cart(c(f), c(f)),
                       id[bool]]
       (restrict[[set[T], set[T]], [finite_set[T], finite_set[T]],
                 boolean]
            (strict_subset?),
        sublista_estricto?)

sublista_estricto?_es_refinamiento_strict_subset?_corol: COROLLARY
  injective?(f) IMPLIES
   (sublista_estricto?(l1, l2) IFF strict_subset?(c(f)(l1), c(f)(l2)))

igualdad_equiv: LEMMA
  injective?(f) IMPLIES
   (igual_l?(l1, l2) IFF igual?[finite_set[T], list[R]](c(f))(l1, l2))
```

```
  append_commutativa_igual: LEMMA
    igual?[finite_set[T], list[R]](c(f))(append(l1, l2), append(l2, l1))

 END refinamiento_conj_finitos


refinamiento_conj_finitos_gen[T, R: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   R_TCC1: ASSUMPTION EXISTS (x: R): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_conj_finitos

  IMPORTING TL@propiedades_operaciones_gen_listas[R]

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@operaciones_conj_finitos[T]

  f: VAR (es_refinamiento?[T, R])

  append_g_es_refinamiento_union_l1_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R])):
      es_refinamiento?[finite_set[T], list[R]](c[T, R](f));

  append_g_es_refinamiento_union_l1: LEMMA
    FORALL (LL: list[list[R]]):
      Union(extend[setof[T], finite_set[T], bool, FALSE]
                (c[finite_set[T], list[R]](c(f))(LL)))
       = c(f)(Append(LL))

  append_g_es_refinamiento_union_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R])):
      es_refinamiento?[finite_set[finite_set[T]], list[list[R]]]
          (c[finite_set[T], list[R]](c[T, R](f)));

  append_g_es_refinamiento_union: THEOREM
    es_refinamiento_op?[finite_set[finite_set[T]], finite_set[T],
                        list[list[R]], list[R], c(c[T, R](f)), c(f)]
      (union_f, Append)

  igual?_es_de_equivalencia: LEMMA equivalence?[R](igual?[T, R](f))

  inter_g_es_refinamiento_intersection_l1_lema_rel_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R]), LL: (cons?[list[R]]), y: T):
      (FORALL (A:
                 (extend[set[T], finite_set[T], bool, FALSE]
                      (c(c[T, R](f))(LL)))):
```

```
       A(y))
     IMPLIES equivalence?[R](igual?[T, R](f));

inter_g_es_refinamiento_intersection_l1_lema_rel: LEMMA
  FORALL (LL: (cons?[list[R]]), y: T):
    (FORALL (A:
               (extend[set[T], finite_set[T], bool, FALSE]
                    (c(c[T, R](f))(LL)))):
       A(y))
     IMPLIES
     (EXISTS (x: R):
        member(igual?(f))(x, Inter(igual?(f))(LL)) AND y = f(x))

inter_g_es_refinamiento_intersection_l1_rel_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])): equivalence?[R](igual?[T, R](f));

inter_g_es_refinamiento_intersection_l1_rel: LEMMA
  FORALL (LL: (cons?[list[R]])):
    subset?(Intersection(extend[setof[T], finite_set[T], bool, FALSE]
                            (c(c[T, R](f))(LL))),
            c(f)(Inter(igual?(f))(LL)))

every_elto: CLAIM
  FORALL (LL: list[list[R]], l: list[R], ro: pred[list[R]]):
    every(ro)(LL) AND member(l, LL) IMPLIES ro(l)

inter_g_es_refinamiento_intersection_l2_rel: LEMMA
  FORALL (LL: (cons?[list[R]])):
    subset?(c(f)(Inter(igual?(f))(LL)),
            Intersection(extend[setof[T], finite_set[T], bool, FALSE]
                            (c(c[T, R](f))(LL))))

inter_g_es_refinamiento_intersection_rel_TCC1: OBLIGATION
  FORALL (f: (es_refinamiento?[T, R])):
    es_refinamiento?[non_empty_finite_set[finite_set[T]],
                     ((cons?[list[R]]))]
        (cnv[finite_set[T], list[R]](c[T, R](f)));

inter_g_es_refinamiento_intersection_rel: THEOREM
  es_refinamiento_op?[non_empty_finite_set[finite_set[T]], finite_set[T],
                      (cons?[list[R]]), list[R], cnv(c[T, R](f)), c(f)]
      (intersection_f, Inter(igual?(f)))

inter_g_es_refinamiento_intersection_l2: LEMMA
  FORALL (LL: (cons?[list[R]])):
    subset?(c(f)(Inter(LL)),
            Intersection(extend[setof[T], finite_set[T], bool, FALSE]
                            (c(c[T, R](f))(LL))))

inter_g_es_refinamiento_intersection_2: THEOREM
  injective?(f) IMPLIES
   es_refinamiento_op?[non_empty_finite_set[finite_set[T]],
```

```
                        finite_set[T], (cons?[list[R]]), list[R],
                        cnv(c[T, R](f)), c(f)]
        (intersection_f, Inter)

  Union_union_distributiva: LEMMA
    FORALL (A, B: finite_set[finite_set[T]]):
      Union(extend[setof[T], finite_set[T], bool, FALSE](union(A, B))) =
       union(Union(extend[setof[T], finite_set[T], bool, FALSE](A)),
             Union(extend[setof[T], finite_set[T], bool, FALSE](B)))

  append_append_gen: LEMMA
    FORALL (LL1, LL2: list[list[R]]):
      igual?[finite_set[T], list[R]]
          (c[T, R](f))
          (Append(append(LL1, LL2)), append(Append(LL1), Append(LL2)))
 END refinamiento_conj_finitos_gen
```

C.4.1. Operaciones generalizadas

```
refinamiento_conj_finitos_image[T1, T2, R1, R2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;

   R1_TCC1: ASSUMPTION EXISTS (x: R1): TRUE;

   R2_TCC1: ASSUMPTION EXISTS (x: R2): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_conj_finitos

  IMPORTING finite_sets@finite_sets_eq

  f1: VAR (es_refinamiento?[T1, R1])

  f2: VAR (es_refinamiento?[T2, R2])

  g: VAR [T1 -> T2]

  h: VAR [R1 -> R2]

  l1: VAR list[R1]

  l2: VAR list[R2]

  caract_map: LEMMA
    FORALL (r2: R2):
      member(r2, map(h)(l1)) IFF
       (EXISTS (r1: R1): member(r1, l1) AND r2 = h(r1))
```

```
  map_es_refinamiento_image_TCC1: OBLIGATION
    FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
            g: [T1 -> T2], h: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
       (FORALL (x1: finite_set[T1]): is_finite[T2](image[T1, T2](g)(x1)));

  map_es_refinamiento_image_TCC2: OBLIGATION
    FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
            g: [T1 -> T2], h: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
       es_refinamiento?[finite_set[T1], list[R1]](c[T1, R1](f1));

  map_es_refinamiento_image_TCC3: OBLIGATION
    FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
            g: [T1 -> T2], h: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
       es_refinamiento?[finite_set[T2], list[R2]](c[T2, R2](f2));

  map_es_refinamiento_image: THEOREM
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     es_refinamiento_op?[finite_set[T1], finite_set[T2], list[R1],
                         list[R2], c(f1), c(f2)]
         (restrict[set[T1], finite_set[T1], set[T2]](image(g)), map(h))

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@conjuntos_finitos_image

  map_es_refinamiento_image_f: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     es_refinamiento_op?[finite_set[T1], finite_set[T2], list[R1],
                         list[R2], c(f1), c(f2)]
         (image_f(g), map(h))
 END refinamiento_conj_finitos_image
```

C.4.2. Refinamiento de la función imagen

```
refinamiento_conj_finitos_image[T1, T2, R1, R2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;

   R1_TCC1: ASSUMPTION EXISTS (x: R1): TRUE;

   R2_TCC1: ASSUMPTION EXISTS (x: R2): TRUE;
  ENDASSUMING
```

```
IMPORTING refinamiento_conj_finitos

IMPORTING finite_sets@finite_sets_eq

f1: VAR (es_refinamiento?[T1, R1])

f2: VAR (es_refinamiento?[T2, R2])

g: VAR [T1 -> T2]

h: VAR [R1 -> R2]

l1: VAR list[R1]

l2: VAR list[R2]

caract_map: LEMMA
  FORALL (r2: R2):
    member(r2, map(h)(l1)) IFF
     (EXISTS (r1: R1): member(r1, l1) AND r2 = h(r1))

map_es_refinamiento_image_TCC1: OBLIGATION
  FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
          g: [T1 -> T2], h: [R1 -> R2]):
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     (FORALL (x1: finite_set[T1]): is_finite[T2](image[T1, T2](g)(x1)));

map_es_refinamiento_image_TCC2: OBLIGATION
  FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
          g: [T1 -> T2], h: [R1 -> R2]):
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     es_refinamiento?[finite_set[T1], list[R1]](c[T1, R1](f1));

map_es_refinamiento_image_TCC3: OBLIGATION
  FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
          g: [T1 -> T2], h: [R1 -> R2]):
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     es_refinamiento?[finite_set[T2], list[R2]](c[T2, R2](f2));

map_es_refinamiento_image: THEOREM
  es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
   es_refinamiento_op?[finite_set[T1], finite_set[T2], list[R1],
                       list[R2], c(f1), c(f2)]
      (restrict[set[T1], finite_set[T1], set[T2]](image(g)), map(h))

TC: LIBRARY = "../auxiliares/conjuntos"

IMPORTING TC@conjuntos_finitos_image

map_es_refinamiento_image_f: LEMMA
  es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
   es_refinamiento_op?[finite_set[T1], finite_set[T2], list[R1],
```

```
                        list[R2], c(f1), c(f2)]
        (image_f(g), map(h))
 END refinamiento_conj_finitos_image
```

C.4.3. Propiedades de operaciones sobre listas vía refinamiento

```
propiedades_operaciones_listas_via_ref[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_conj_finitos[T, T], refinamiento_rel_equivalencia

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@propiedades_conjuntos_ref[T]

  x: VAR T

  l, l1, l2, l3: VAR list[T]

  f: VAR [T -> T]

  A, B, C: VAR set[T]

  id_inyectiva: LEMMA injective?(id[T])

  c_lista_unitaria: LEMMA c(id[T])((: x :)) = singleton(x)

  JUDGEMENT id[T] HAS_TYPE (injective?[T, T])

  null?_es_refinamiento_empty?_id: LEMMA null?(l) IFF empty?(c(id[T])(l))

  sublista?_es_refinamiento_subset?_id: LEMMA
    sublista?(l1, l2) IFF subset?(c(id[T])(l1), c(id[T])(l2))

  sublista_reflexiva: LEMMA sublista?(l, l)

  sublista_transitiva: LEMMA
    sublista?(l1, l2) AND sublista?(l2, l3) IMPLIES sublista?(l1, l3)

  cons_es_refinamiento_add_id: LEMMA
    c(id[T])(cons(x, l)) = add(x, c(id[T])(l))

  sublista_extension: LEMMA sublista?(l, cons(x, l))

  rest_l_es_refinamiento_rest_id: LEMMA
    c(id[T])(rest_l(l)) = rest(c(id[T])(l))
```

```
sublista_rest_l: LEMMA cons?(l) IMPLIES sublista?(rest_l(l), l)

elimina_es_refinamiento_remove_id: LEMMA
  c(id[T])(elimina(x, l)) = remove(x, c(id[T])(l))

member_es_refinamiento_member_id: LEMMA
  member(x, l) IFF member(x, c(id[T])(l))

sublista_elimina_1: LEMMA
  sublista?(l1, l2) IMPLIES sublista?(elimina(x, l1), l2)

sublista_elimina: LEMMA
  member(x, l1) AND sublista?(l1, cons(x, l2)) IMPLIES
   sublista?(elimina(x, l1), l2)

cardinal_l_es_refinamiento_card_id: LEMMA
  cardinal_l(l) = card(c(id[T])(l))

CS_cardinal_l_nulo: LEMMA cardinal_l(l) = 0 IMPLIES null?(l)

CS_cardinal_l_null: LEMMA cardinal_l(l) = 0 IMPLIES l = null

cardinal_elimina: LEMMA
  member(x, l) IMPLIES cardinal_l(elimina(x, l)) = cardinal_l(l) - 1

CN_cons_cardinal_l: LEMMA cardinal_l(l) >= 1 IMPLIES cons?(l)

inter_es_refinamiento_intersection_id: LEMMA
  c(id[T])(inter(l1, l2)) = intersection(c(id[T])(l1), c(id[T])(l2))

append_es_refinamiento_union_id: LEMMA
  c(id[T])(append(l1, l2)) = union(c(id[T])(l1), c(id[T])(l2))

cardinal_l_plus: THEOREM
  cardinal_l(l1) + cardinal_l(l2) =
   cardinal_l(append(l1, l2)) + cardinal_l(inter(l1, l2))

igual_l?_es_refinamiento_igual_c_id: LEMMA
  igual_l?(l1, l2) IFF igual_c?(c(id[T])(l1), c(id[T])(l2))

igual_l?_es_de_equivalencia: LEMMA equivalence?[list[T]](igual_l?)

sublista_elimina_igual: LEMMA
  member(x, l1) AND igual_l?(elimina(x, l1), l3) IMPLIES
   igual_l?(l1, cons(x, l3))

elimina_car_igual: LEMMA
  cons?(l1) AND igual_l?(elimina(car(l1), l1), l2) IMPLIES
   igual_l?(l1, cons(car(l1), l2))

elimina_igual_no_pertenencia: LEMMA
```

```
    igual_l?(elimina(x, l1), l2) IMPLIES NOT member(x, l2)

  sublista_pertenece_car: LEMMA
    sublista?(l1, l2) AND cons?(l1) IMPLIES member(car(l1), l2)

  sublista?_cardinal_l: LEMMA
    sublista?(l1, l2) IMPLIES cardinal_l(l1) <= cardinal_l(l2)

  append_sublista?_1: LEMMA sublista?(l1, append(l1, l2))

  append_sublista?_2: LEMMA sublista?(l2, append(l1, l2))

  sublista_estricto?_es_refinamiento_strict_subset?_id: LEMMA
    sublista_estricto?(l1, l2) IFF
     strict_subset?(c(id[T])(l1), c(id[T])(l2))

  sublista_estricto?_cardinal_l: LEMMA
    sublista_estricto?(l1, l2) IMPLIES cardinal_l(l1) < cardinal_l(l2)

  sublista_neg: LEMMA
    sublista?(l1, cons(x, l2)) AND NOT sublista?(l1, l2) IMPLIES
     (EXISTS l3: sublista?(l3, l2) AND igual_l?(l1, cons(x, l3)))

  append_commutativa_l: LEMMA igual_l?(append(l1, l2), append(l2, l1))

  sublista_igual_l_2: LEMMA
    igual_l?(l1, l2) AND sublista?(l, l1) IMPLIES sublista?(l, l2)

  igual_l_sublista_1: LEMMA
    igual_l?(l1, l2) AND sublista?(l2, l) IMPLIES sublista?(l1, l)

  sublista_append_2: LEMMA
    sublista?(l1, l2) IMPLIES igual_l?(append(l2, l1), l2)

  cs_append_sublista: LEMMA
    sublista?(l1, l) AND sublista?(l2, l) IMPLIES
     sublista?(append(l1, l2), l)

  CN_cardinal_l_cdr: LEMMA
    cons?(l) AND NOT member(car(l), cdr(l)) IMPLIES
     cardinal_l(l) = cardinal_l(cdr(l)) + 1
 END propiedades_operaciones_listas_via_ref

propiedades_map_via_ref[T1, T2, R1, R2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;

   R1_TCC1: ASSUMPTION EXISTS (x: R1): TRUE;
```

```
   R2_TCC1: ASSUMPTION EXISTS (x: R2): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_conj_finitos_image

  f1: VAR (es_refinamiento?[T1, R1])

  f2: VAR (es_refinamiento?[T2, R2])

  g: VAR [T1 -> T2]

  h: VAR [R1 -> R2]

  l1, l2: VAR list[R1]

  map_conserva_igual_l?_TCC1: OBLIGATION
    FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
            g: [T1 -> T2], h: [R1 -> R2]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
       equivalence?[R1](igual?[T1, R1](f1));

  map_conserva_igual_l?_TCC2: OBLIGATION
    FORALL (f1: (es_refinamiento?[T1, R1]), f2: (es_refinamiento?[T2, R2]),
            g: [T1 -> T2], h: [R1 -> R2], l1, l2: list[R1]):
      es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) AND
       igual_l?(igual?(f1))(l1, l2)
       IMPLIES equivalence?[R2](igual?[T2, R2](f2));

  map_conserva_igual_l?: LEMMA
    es_refinamiento_op?[T1, T2, R1, R2, f1, f2](g, h) IMPLIES
     (igual_l?(igual?(f1))(l1, l2) IMPLIES
       igual_l?(igual?(f2))(map(h)(l1), map(h)(l2)))
 END propiedades_map_via_ref

propiedades_potencia_listas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING propiedades_operaciones_listas_via_ref

  IMPORTING TL@potencia_listas, TL@propiedades_operaciones_gen_listas

  x, y: VAR T

  l, l1, l2, ls: VAR list[T]

  LL: VAR list[list[T]]

  incluye_en_cada_cn1: LEMMA
    member(l2, incluye_en_cada(x, LL)) IMPLIES
```

```
     (EXISTS l1: member(l1, LL) AND l2 = cons(x, l1))

  powerset_ref_2_cn1: LEMMA
    member(l1, powerset_ref_2(l2)) IMPLIES sublista?(l1, l2)

  powerset_ref_2_cs1_l1: LEMMA
    member(l2, powerset_ref_2(l1)) IMPLIES
     member(l2, powerset_ref_2(cons(x, l1)))

  incluye_en_cada_cons: LEMMA
    member(ls, LL) IMPLIES member(cons(x, ls), incluye_en_cada(x, LL))
 END propiedades_potencia_listas

sublistas_cardinal[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING propiedades_operaciones_listas_via_ref

  IMPORTING TL@propiedades_operaciones_gen_listas, TL@sublistas_cardinal_def

  x, y: VAR T

  l, l1, l2, ls: VAR list[T]

  LL: VAR list[list[T]]

  pertenece_sublistas_card_aux2_cons: LEMMA
    member(l1, sublistas_card_aux2(x, LL)) IMPLIES cons?(l1)

  CN_pertenece_sublistas_card_aux2_TCC1: OBLIGATION
    FORALL (LL: list[list[T]], l1: list[T], x: T):
      member(l1, sublistas_card_aux2(x, LL)) IMPLIES cons?[T](l1);

  CN_pertenece_sublistas_card_aux2: LEMMA
    member(l1, sublistas_card_aux2(x, LL)) IMPLIES
     car(l1) = x AND member(cdr(l1), LL)

  CN_pertenece_sublistas_card_aux2_car: LEMMA
    member(l1, sublistas_card_aux2(x, LL)) IMPLIES
     NOT member(car(l1), cdr(l1))

  CN_pertenece_sublistas_card_aux2_cardinal_b: LEMMA
    member(l1, sublistas_card_aux2(x, LL)) IMPLIES
     cardinal_1(l1) = cardinal_1(cdr(l1)) + 1

  CS1_pertenece_sublistas_card_aux2: LEMMA
    member(ls, LL) AND NOT member(x, ls) IMPLIES
     member(cons(x, ls), sublistas_card_aux2(x, LL))
```

```
  CN_pertenece_sublistas_card_aux1: LEMMA
    member(l1, sublistas_card_aux1(ls, LL)) IMPLIES
     (EXISTS x: member(x, ls) AND member(l1, sublistas_card_aux2(x, LL)))

  CS1_pertenece_sublistas_card_aux1: LEMMA
    member(l2, LL) AND member(x, ls) AND NOT member(x, l2) IMPLIES
     member(cons(x, l2), sublistas_card_aux1(ls, LL))

  sublistas_card_cn1: LEMMA
    FORALL (ls: list[T], k: upto(cardinal_l[T](ls))):
      member(l1, sublistas_card(ls, k)) IMPLIES sublista?[T](l1, ls)

  sublistas_card_cn2: LEMMA
    FORALL (ls: list[T], k: upto(cardinal_l[T](ls))):
      member(l1, sublistas_card(ls, k)) IMPLIES cardinal_l(l1) = k

  cs_sublistas_cardinal_0_TCC1: OBLIGATION
    FORALL (l1, ls: list[T]): null?(l1) IMPLIES 0 <= cardinal_l[T](ls);

  cs_sublistas_cardinal_0: LEMMA
    null?(l1) IMPLIES member(l1, sublistas_card(ls, 0))

  sublistas_card_cs: LEMMA
    FORALL (ls: list[T], k: upto(cardinal_l[T](ls))):
      sublista?(l1, ls) AND cardinal_l(l1) = k IMPLIES
       (EXISTS l2: member(l2, sublistas_card(ls, k)) AND igual_l?(l1, l2))
 END sublistas_cardinal
```

C.4.4. Refinamiento: conjunto potencia

```
refinamiento_powerset[T, R: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   R_TCC1: ASSUMPTION EXISTS (x: R): TRUE;
  ENDASSUMING

  IMPORTING refinamiento_conj_finitos, propiedades_potencia_listas

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@operaciones_conj_finitos[T]

  l, ls, l1, l2, l3: VAR list[R]

  f: VAR (es_refinamiento?[T, R])

  A, B: VAR finite_set[T]
```

```
  CNS_pertenencia_powerset_f: LEMMA
    member(B, powerset_f(A)) IFF subset?(B, A)

  powerset_ref_2_cs1: LEMMA
    sublista?[R](l1, l2) IMPLIES
     (EXISTS l3: member(l3, powerset_ref_2(l2)) AND igual_l?[R](l1, l3))

  powerset_ref_2_cs1_rel_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R]), l1, l2: list[R]):
      sublista?(igual?(f))(l1, l2) IMPLIES
       es_refinamiento?[finite_set[T], list[R]](c[T, R](f));

  powerset_ref_2_cs1_rel: LEMMA
    sublista?(igual?(f))(l1, l2) IMPLIES
     (EXISTS l3:
        member(igual?(c(f)))(l3, powerset_ref_2(l2)) AND
         igual?(c(f))(l1, l3))

  powerset_ref_2_es_refinamiento_l1_g_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R])):
      es_refinamiento?[finite_set[T], list[R]](c[T, R](f));

  powerset_ref_2_es_refinamiento_l1_g: LEMMA
    subset?(powerset_f(c(f)(l)), c(c(f))(powerset_ref_2(l)))

  powerset_ref_2_es_refinamiento_l1: LEMMA
    injective?(f) IMPLIES
     subset?(powerset_f(c(f)(l)), c(c(f))(powerset_ref_2(l)))

  powerset_ref_2_es_refinamiento_l2_g: LEMMA
    subset?(c(c(f))(powerset_ref_2(l)), powerset_f(c(f)(l)))

  powerset_ref_2_es_refinamiento_l2: LEMMA
    injective?(f) IMPLIES
     subset?(c(c(f))(powerset_ref_2(l)), powerset_f(c(f)(l)))

  powerset_ref_2_es_refinamiento_g_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R])):
      es_refinamiento?[finite_set[finite_set[T]], list[list[R]]]
          (c[finite_set[T], list[R]](c[T, R](f)));

  powerset_ref_2_es_refinamiento_g: THEOREM
    es_refinamiento_op?[finite_set[T], finite_set[finite_set[T]], list[R],
                        list[list[R]], c(f), c(c(f))]
       (powerset_f, powerset_ref_2)

  powerset_ref_2_es_refinamiento: THEOREM
    injective?(f) IMPLIES
     es_refinamiento_op?[finite_set[T], finite_set[finite_set[T]], list[R],
                         list[list[R]], c(f), c(c(f))]
        (powerset_f, powerset_ref_2)
END refinamiento_powerset
```

C.4.5. Refinamiento: subconjuntos de cardinal dado

```
refinamiento_subconjuntos_card[T, R: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   R_TCC1: ASSUMPTION EXISTS (x: R): TRUE;
  ENDASSUMING

  IMPORTING sublistas_cardinal[R], refinamiento_conj_finitos

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@operaciones_conj_finitos[T]

  x: VAR T

  A, X, Y: VAR finite_set[T]

  k: VAR nat

  l, ls, l1, l2: VAR list[R]

  f: VAR (es_refinamiento?[T, R])

  subconjuntos_finito: LEMMA
    FORALL (A: finite_set[T], k: upto(card(A))):
      is_finite(subconjuntos(A, k))

  sublistas_card_es_refinamiento_corol_l1: LEMMA
    FORALL (ls: list[R], k: upto(cardinal_l[R](ls))):
      injective?(f) IMPLIES
       member(l1, sublistas_card(ls, k)) IMPLIES
        member(c(f)(l1), subconjuntos(c(f)(ls), k))

  sublistas_card_es_refinamiento_corol_l2_TCC1: OBLIGATION
    FORALL (f: (es_refinamiento?[T, R]), l1: list[R], ls: list[R],
            k: upto(cardinal_l[R](ls))):
      injective?(f) AND member(c(f)(l1), subconjuntos(c(f)(ls), k)) IMPLIES
       (FORALL (l2):
          member(l2, sublistas_card(ls, k)) IMPLIES
           es_refinamiento?[finite_set[T], list[R]](c[T, R](f)));

  sublistas_card_es_refinamiento_corol_l2: LEMMA
    FORALL (ls: list[R], k: upto(cardinal_l[R](ls))):
      injective?(f) IMPLIES
       member(c(f)(l1), subconjuntos(c(f)(ls), k)) IMPLIES
        (EXISTS l2:
          member(l2, sublistas_card(ls, k)) AND igual?(c(f))(l1, l2))

  sublistas_card_es_refinamiento_corol_b_TCC1: OBLIGATION
```

```
    FORALL (f: (es_refinamiento?[T, R])):
      injective?(f) IMPLIES
       es_refinamiento?[finite_set[T], list[R]](c[T, R](f));

  sublistas_card_es_refinamiento_corol_b: THEOREM
    FORALL (ls: list[R], k: upto(cardinal_1[R](ls))):
      injective?(f) IMPLIES
       subconjuntos(c(f)(ls), k) = c(c(f))(sublistas_card(ls, k))
END refinamiento_subconjuntos_card
```

Apéndice D

Programación lógica proposicional

D.1. Formalización usando conjuntos finitos

D.1.1. Tipo de dato: fórmula proposicional

```
formula_prop[T: TYPE+]: DATATYPE
 BEGIN
  falso: falsa?
  a(simb: T): atomo?
  ~(fla: formula_prop): negacion?
  &(fla1, fla2: formula_prop): conjuncion?
 END formula_prop


formula_prop_adt[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  formula_prop: TYPE

  falsa?, atomo?, negacion?, conjuncion?: [formula_prop -> boolean]

  falso: (falsa?)

  a: [T -> (atomo?)]

  ~: [formula_prop -> (negacion?)]

  &: [[formula_prop, formula_prop] -> (conjuncion?)]

  simb: [(atomo?) -> T]
```

```
fla: [(negacion?) -> formula_prop]

fla1: [(conjuncion?) -> formula_prop]

fla2: [(conjuncion?) -> formula_prop]

formula_prop_ord: [formula_prop -> upto(3)]

formula_prop_ord_defaxiom: AXIOM
  formula_prop_ord(falso) = 0 AND
   (FORALL (simb: T): formula_prop_ord(a(simb)) = 1) AND
    (FORALL (fla: formula_prop): formula_prop_ord(~fla) = 2) AND
     (FORALL (fla1: formula_prop, fla2: formula_prop):
        formula_prop_ord(fla1 & fla2) = 3);

ord(x: formula_prop): upto(3) =
    CASES x
      OF falso: 0,
         a(a1_var): 1,
         ~(tilde1_var): 2,
         &(and1_var, and2_var): 3
      ENDCASES

formula_prop_falso_extensionality: AXIOM
  FORALL (falsa?_var: (falsa?), falsa?_var2: (falsa?)):
    falsa?_var = falsa?_var2;

formula_prop_a_extensionality: AXIOM
  FORALL (atomo?_var: (atomo?), atomo?_var2: (atomo?)):
    simb(atomo?_var) = simb(atomo?_var2) IMPLIES atomo?_var = atomo?_var2;

formula_prop_a_eta: AXIOM
  FORALL (atomo?_var: (atomo?)): a(simb(atomo?_var)) = atomo?_var;

formula_prop_tilde_extensionality: AXIOM
  FORALL (negacion?_var: (negacion?), negacion?_var2: (negacion?)):
    fla(negacion?_var) = fla(negacion?_var2) IMPLIES
     negacion?_var = negacion?_var2;

formula_prop_tilde_eta: AXIOM
  FORALL (negacion?_var: (negacion?)):
    (~fla(negacion?_var)) = negacion?_var;

formula_prop_and_extensionality: AXIOM
  FORALL (conjuncion?_var: (conjuncion?),
          conjuncion?_var2: (conjuncion?)):
    fla1(conjuncion?_var) = fla1(conjuncion?_var2) AND
     fla2(conjuncion?_var) = fla2(conjuncion?_var2)
     IMPLIES conjuncion?_var = conjuncion?_var2;

formula_prop_and_eta: AXIOM
  FORALL (conjuncion?_var: (conjuncion?)):
```

```
    (fla1(conjuncion?_var) & fla2(conjuncion?_var)) = conjuncion?_var;

formula_prop_simb_a: AXIOM FORALL (a1_var: T): simb(a(a1_var)) = a1_var;

formula_prop_fla_tilde: AXIOM
  FORALL (tilde1_var: formula_prop): fla(~tilde1_var) = tilde1_var;

formula_prop_fla1_and: AXIOM
  FORALL (and1_var: formula_prop, and2_var: formula_prop):
    fla1(and1_var & and2_var) = and1_var;

formula_prop_fla2_and: AXIOM
  FORALL (and1_var: formula_prop, and2_var: formula_prop):
    fla2(and1_var & and2_var) = and2_var;

formula_prop_inclusive: AXIOM
  FORALL (formula_prop_var: formula_prop):
    falsa?(formula_prop_var) OR
     atomo?(formula_prop_var) OR
      negacion?(formula_prop_var) OR conjuncion?(formula_prop_var);

formula_prop_induction: AXIOM
  FORALL (p: [formula_prop -> boolean]):
    (p(falso) AND
      (FORALL (a1_var: T): p(a(a1_var))) AND
       (FORALL (tilde1_var: formula_prop):
          p(tilde1_var) IMPLIES p(~tilde1_var))
        AND
        (FORALL (and1_var: formula_prop, and2_var: formula_prop):
           p(and1_var) AND p(and2_var) IMPLIES p(and1_var & and2_var)))
     IMPLIES
     (FORALL (formula_prop_var: formula_prop): p(formula_prop_var));

every(p: PRED[T])(a1: formula_prop):  boolean =
    CASES a1
      OF falso: TRUE,
         a(a1_var): p(a1_var),
         ~(tilde1_var): every(p)(tilde1_var),
         &(and1_var, and2_var): every(p)(and1_var) AND every(p)(and2_var)
      ENDCASES;

every(p: PRED[T], a1: formula_prop):  boolean =
    CASES a1
      OF falso: TRUE,
         a(a1_var): p(a1_var),
         ~(tilde1_var): every(p, tilde1_var),
         &(and1_var, and2_var): every(p, and1_var) AND every(p, and2_var)
      ENDCASES;

some(p: PRED[T])(a1: formula_prop):  boolean =
    CASES a1
      OF falso: FALSE,
```

```
          a(a1_var): p(a1_var),
          ~(tilde1_var): some(p)(tilde1_var),
          &(and1_var, and2_var): some(p)(and1_var) OR some(p)(and2_var)
      ENDCASES;

some(p: PRED[T], a1: formula_prop):  boolean =
    CASES a1
      OF falso: FALSE,
         a(a1_var): p(a1_var),
         ~(tilde1_var): some(p, tilde1_var),
         &(and1_var, and2_var): some(p, and1_var) OR some(p, and2_var)
      ENDCASES;

subterm(x, y: formula_prop):  boolean =
    x = y OR
     CASES y
       OF falso: FALSE,
          a(a1_var): FALSE,
          ~(tilde1_var): subterm(x, tilde1_var),
          &(and1_var, and2_var):
            subterm(x, and1_var) OR subterm(x, and2_var)
       ENDCASES;

<<:  (well_founded?[formula_prop]) =
    LAMBDA (x, y: formula_prop):
      CASES y
        OF falso: FALSE,
           a(a1_var): FALSE,
           ~(tilde1_var): x = tilde1_var OR x << tilde1_var,
           &(and1_var, and2_var):
             (x = and1_var OR x << and1_var) OR
              x = and2_var OR x << and2_var
        ENDCASES;

formula_prop_well_founded: AXIOM well_founded?[formula_prop](<<);

reduce_nat(falsa?_fun: nat, atomo?_fun: [T -> nat],
           negacion?_fun: [nat -> nat],
           conjuncion?_fun: [[nat, nat] -> nat]):
 [formula_prop -> nat] =
    LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> nat] =
            reduce_nat(falsa?_fun, atomo?_fun, negacion?_fun,
                       conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun,
             a(a1_var): atomo?_fun(a1_var),
             ~(tilde1_var): negacion?_fun(red(tilde1_var)),
             &(and1_var, and2_var):
               conjuncion?_fun(red(and1_var), red(and2_var))
          ENDCASES;
```

```
REDUCE_nat(falsa?_fun: [formula_prop -> nat],
           atomo?_fun: [[T, formula_prop] -> nat],
           negacion?_fun: [[nat, formula_prop] -> nat],
           conjuncion?_fun: [[nat, nat, formula_prop] -> nat]):
 [formula_prop -> nat] =
    LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> nat] =
            REDUCE_nat(falsa?_fun, atomo?_fun, negacion?_fun,
                       conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun(formula_prop_adtvar),
             a(a1_var): atomo?_fun(a1_var, formula_prop_adtvar),
             ~(tilde1_var):
               negacion?_fun(red(tilde1_var), formula_prop_adtvar),
             &(and1_var, and2_var):
               conjuncion?_fun(red(and1_var), red(and2_var),
                               formula_prop_adtvar)
          ENDCASES;

reduce_ordinal(falsa?_fun: ordinal, atomo?_fun: [T -> ordinal],
               negacion?_fun: [ordinal -> ordinal],
               conjuncion?_fun: [[ordinal, ordinal] -> ordinal]):
 [formula_prop -> ordinal] =
    LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> ordinal] =
            reduce_ordinal(falsa?_fun, atomo?_fun, negacion?_fun,
                           conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun,
             a(a1_var): atomo?_fun(a1_var),
             ~(tilde1_var): negacion?_fun(red(tilde1_var)),
             &(and1_var, and2_var):
               conjuncion?_fun(red(and1_var), red(and2_var))
          ENDCASES;

REDUCE_ordinal(falsa?_fun: [formula_prop -> ordinal],
               atomo?_fun: [[T, formula_prop] -> ordinal],
               negacion?_fun: [[ordinal, formula_prop] -> ordinal],
               conjuncion?_fun:
                 [[ordinal, ordinal, formula_prop] -> ordinal]):
 [formula_prop -> ordinal] =
    LAMBDA (formula_prop_adtvar: formula_prop):
      LET red: [formula_prop -> ordinal] =
            REDUCE_ordinal(falsa?_fun, atomo?_fun, negacion?_fun,
                           conjuncion?_fun)
        IN
        CASES formula_prop_adtvar
          OF falso: falsa?_fun(formula_prop_adtvar),
             a(a1_var): atomo?_fun(a1_var, formula_prop_adtvar),
```

```
              ~(tilde1_var):
                negacion?_fun(red(tilde1_var), formula_prop_adtvar),
              &(and1_var, and2_var):
                conjuncion?_fun(red(and1_var), red(and2_var),
                                formula_prop_adtvar)
          ENDCASES;
 END formula_prop_adt
```

D.1.2. Conectivas

```
conectivas_def[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING formula_prop_adt[T]

  F1, F2: VAR formula_prop[T]

  verdad: formula_prop = ~falso;

  \/(F1, F2): formula_prop = ~(~F1 & ~F2);

  =>(F1, F2): formula_prop = ~F1 \/ F2;

  <=>(F1, F2): formula_prop = (F1 => F2) & (F2 => F1)
 END conectivas_def
```

D.1.3. Átomos

```
atomos[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING formula_prop_adt[T]

  atomo: TYPE = (atomo?)

  JUDGEMENT atomo SUBTYPE_OF formula_prop
 END atomos
```

D.1.4. Cláusulas

```
clausulas[T: TYPE+]: THEORY
 BEGIN
```

```
ASSUMING
 T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
ENDASSUMING

IMPORTING semantica_proposicional[T]

A: VAR atomo[T]

t_0: T

cabeza_cl: TYPE = {F: formula_prop | falsa?(F) OR atomo?(F)}

clausula_horn_TCC1: OBLIGATION
  EXISTS (x: [# cabeza: cabeza_cl, cuerpo: finite_set[atomo[T]] #]): TRUE;

clausula_horn: TYPE+ = [# cabeza: cabeza_cl, cuerpo: finite_set[atomo] #]

CH: VAR clausula_horn

SC: VAR set[clausula_horn]

FSC: VAR finite_set[clausula_horn]

es_clausula_def(CH): bool = atomo?(cabeza(CH))

clausula_def_TCC1: OBLIGATION
  EXISTS (x: {x: clausula_horn | es_clausula_def(x)}): TRUE;

clausula_def: TYPE+ = (es_clausula_def)

JUDGEMENT clausula_def SUBTYPE_OF clausula_horn

C, C1, C2: VAR clausula_def

cabeza_TCC1: OBLIGATION FORALL (C): atomo?[T](cabeza(C));

JUDGEMENT cabeza(C) HAS_TYPE atomo[T]

es_objetivo_def(CH): bool = falsa?(cabeza(CH))

objetivo_def_TCC1: OBLIGATION
  EXISTS (x: {x: clausula_horn | es_objetivo_def(x)}): TRUE;

objetivo_def: TYPE+ = (es_objetivo_def)

JUDGEMENT objetivo_def SUBTYPE_OF clausula_horn

G, G1, G2: VAR objetivo_def

clausula_horn_tipo: LEMMA es_clausula_def(CH) OR es_objetivo_def(CH)

programa: TYPE = finite_set[clausula_def]
```

```
P, P1, P2: VAR programa

JUDGEMENT cabeza(C) HAS_TYPE atomo

regla?(C): bool = nonempty?(cuerpo(C))

hecho?(C): bool = empty?(cuerpo(C))

vacia?(G): bool = empty?(cuerpo(G))

clausula_horn(F: cabeza_cl, FSA: finite_set[atomo]): clausula_horn =
    (# cabeza := F, cuerpo := FSA #)

cabeza_clausula_horn: LEMMA
  FORALL (F: cabeza_cl, FSA: finite_set[atomo]):
    cabeza(clausula_horn(F, FSA)) = F

cuerpo_clausula_horn: LEMMA
  FORALL (F: cabeza_cl, FSA: finite_set[atomo]):
    cuerpo(clausula_horn(F, FSA)) = FSA

objetivo_TCC1: OBLIGATION falsa?[T](falso[T]) OR atomo?[T](falso[T]);

objetivo_TCC2: OBLIGATION
  FORALL (FSA: finite_set[atomo[T]]):
    es_objetivo_def(clausula_horn(falso[T], FSA));

objetivo(FSA: finite_set[atomo]): objetivo_def = clausula_horn(falso, FSA)

cs1_igualdad_objetivos: CLAIM
  vacia?(G) IMPLIES objetivo(singleton(A)) = objetivo(add(A, cuerpo(G)))

objetivo_cuerpo_objetivo: CLAIM objetivo(cuerpo(G)) = G

cuerpo_objetivo_add: CLAIM
  cuerpo(objetivo(add(A, cuerpo(G)))) = add(A, cuerpo(G))

cuerpo_objetivo: CLAIM
  FORALL (FSA: finite_set[atomo]): cuerpo(objetivo(FSA)) = FSA

hecho_TCC1: OBLIGATION FORALL (A: atomo[T]): falsa?[T](A) OR atomo?[T](A);

hecho_TCC2: OBLIGATION
  FORALL (A: atomo[T]):
    es_clausula_def(clausula_horn(A, emptyset[atomo[T]]));

hecho(A: atomo): clausula_def = clausula_horn(A, emptyset)

hecho_b_d: LEMMA FORALL (A: atomo): hecho?(hecho(A))

fla(CH): formula_prop[T] =
```

```
      conjuncion(extend[formula_prop[T], atomo[T], bool, FALSE]
                    (cuerpo(CH)))
       => cabeza(CH)

  conj_fla(SC): set[formula_prop[T]] = image(fla)(SC)

  IMPORTING finite_sets@finite_sets_eq

  conj_fla_TCC1: OBLIGATION
    FORALL (FSC): is_finite[formula_prop[T]](conj_fla(FSC));

  JUDGEMENT conj_fla(FSC) HAS_TYPE finite_set[formula_prop[T]]

  conj_fla_add: LEMMA conj_fla(add(CH, SC)) = add(fla(CH), conj_fla(SC))
 END clausulas
```

D.1.5. Semántica proposicional

```
semantica_proposicional[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING conectivas_def[T], atomos[T], finite_sets@finite_sets_inductions

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@propiedades_conjuntos, TC@propiedades_image

  x: VAR T

  F, F1, F2: VAR formula_prop[T]

  A, A1, A2: VAR atomo

  S, S1, S2: VAR set[formula_prop]

  FS: VAR finite_set[formula_prop]

  simb_prop_TCC1: OBLIGATION
    FORALL (F1: formula_prop[T], F: formula_prop[T]):
      F = ~F1 IMPLIES <<[T](F1, F);

  simb_prop_TCC2: OBLIGATION
    FORALL (F1, F2: formula_prop[T], F: formula_prop[T]):
      F = (F1 & F2) IMPLIES <<[T](F1, F);

  simb_prop_TCC3: OBLIGATION
    FORALL (F1, F2: formula_prop[T], F: formula_prop[T]):
```

```
    F = (F1 & F2) IMPLIES <<[T](F2, F);

simb_prop(F): RECURSIVE finite_set[atomo] =
  CASES F
    OF falso: emptyset,
       a(x): singleton[atomo[T]](a(x)),
       ~(F1): simb_prop(F1),
       &(F1, F2): union(simb_prop(F1), simb_prop(F2))
    ENDCASES
   MEASURE F BY <<

simb_prop_finito: LEMMA is_finite(simb_prop(F))

JUDGEMENT simb_prop(F) HAS_TYPE finite_set[atomo]

simb_prop_falsa: LEMMA falsa?(F) IMPLIES simb_prop(F) = emptyset

atomo_simb_prop: LEMMA atomo?(A) IMPLIES member(A, simb_prop(A))

simb_prop_atomo_igual: CLAIM member(A1, simb_prop(A)) IMPLIES A1 = A

simb_prop_negacion: LEMMA simb_prop(~F) = simb_prop(F)

simb_prop_verdad: LEMMA simb_prop(verdad) = emptyset

simb_prop_conjuncion_bin: LEMMA
  simb_prop(F1 & F2) = union(simb_prop(F1), simb_prop(F2))

simb_prop_disyuncion_bin: LEMMA
  simb_prop(F1 \/ F2) = union(simb_prop(F1), simb_prop(F2))

simb_prop_implicacion: LEMMA
  simb_prop(F1 => F2) = union(simb_prop(F1), simb_prop(F2))

interpretacion: TYPE = set[atomo]

interpretacion_finita: TYPE = finite_set[atomo]

I, I1, I2: VAR interpretacion

es_modelo_TCC1: OBLIGATION
  FORALL (x: T, F: formula_prop[T]): F = a(x) IMPLIES atomo?[T](F);

es_modelo(I, F): RECURSIVE bool =
  CASES F
    OF falso: FALSE,
       a(x): member(F, I),
       ~(F1): NOT es_modelo(I, F1),
       &(F1, F2): es_modelo(I, F1) AND es_modelo(I, F2)
    ENDCASES
   MEASURE F BY <<
```

```
no_es_modelo_falso: LEMMA NOT es_modelo(I, falso)

es_modelo_atomo: LEMMA es_modelo(I, A) = member(A, I)

es_modelo_negacion: LEMMA es_modelo(I, ~F) IFF NOT es_modelo(I, F)

es_modelo_conjuncion_binaria: LEMMA
  es_modelo(I, F1 & F2) IFF (es_modelo(I, F1) AND es_modelo(I, F2))

es_modelo_verdad: LEMMA es_modelo(I, verdad)

es_modelo_disyuncion: LEMMA
  es_modelo(I, F1 \/ F2) IFF (es_modelo(I, F1) OR es_modelo(I, F2))

es_modelo_implicacion: LEMMA
  es_modelo(I, F1 => F2) IFF ((NOT es_modelo(I, F1)) OR es_modelo(I, F2))

es_modelo_equivalencia: LEMMA
  es_modelo(I, F1 <=> F2) IFF (es_modelo(I, F1) IFF es_modelo(I, F2))

es_modelo_simb_prop_interpretacion: LEMMA
  (FORALL A:
     member(A, simb_prop(F)) IMPLIES
      (es_modelo(I1, A) = es_modelo(I2, A)))
   IMPLIES es_modelo(I1, F) = es_modelo(I2, F)

modelo_restringido: LEMMA
  es_modelo(I, F) = es_modelo(intersection(I, simb_prop(F)), F)

es_satisfacible(F): bool = EXISTS I: es_modelo(I, F)

es_valida(F): bool = FORALL I: es_modelo(I, F)

ejemplo1: LEMMA es_valida((F1 => F2) \/ (F2 => F1))

ejemplo2: LEMMA es_valida(verdad \/ F)

ejemplo3: LEMMA NOT (es_valida(falso & F))

equivalentes(F1, F2): bool =
    FORALL I: es_modelo(I, F1) IFF es_modelo(I, F2)

verdad_neutro_conjuncion: CLAIM equivalentes(F & verdad, F)

equivalentes_falso: LEMMA equivalentes(F, F \/ falso)

equivalentes_neg: LEMMA
  equivalentes(F1, F2) IMPLIES equivalentes(~F1, ~F2)

equivalentes_disy: LEMMA
  equivalentes(F1, F2) IMPLIES equivalentes(F1 \/ F, F2 \/ F)
```

```
equivalentes_reflexiva: LEMMA reflexive?(equivalentes)

equivalentes_simetrica: LEMMA symmetric?(equivalentes)

equivalentes_transitiva: LEMMA transitive?(equivalentes)

equivalentes_equivalencia: LEMMA equivalence?(equivalentes)

es_modelo(I, S): bool = FORALL F: member(F, S) IMPLIES es_modelo(I, F)

modelo_de_subconjuntos: LEMMA
  es_modelo(I, S1) AND subset?(S2, S1) IMPLIES es_modelo(I, S2)

modelo_singleton: LEMMA es_modelo(I, singleton(F)) IFF es_modelo(I, F)

modelo_union: LEMMA
  es_modelo(I, union(S1, S2)) IFF (es_modelo(I, S1) AND es_modelo(I, S2))

modelo_add: LEMMA
  es_modelo(I, add(F, S)) IFF (es_modelo(I, S) AND es_modelo(I, F))

modelo_add_neg: LEMMA
  es_modelo(I, add(~F, S)) IFF (es_modelo(I, S) AND NOT es_modelo(I, F))

es_satisfacible(S): bool = EXISTS I: es_modelo(I, S)

es_insatisfacible(S): bool = FORALL I: NOT es_modelo(I, S)

es_valido(S): bool = FORALL I: es_modelo(I, S)

es_no_valido(S): bool = EXISTS I: NOT es_modelo(I, S)

equivalentes_insatisfacible: LEMMA
  equivalentes(F1, F2) IMPLIES
   (FORALL S:
      es_insatisfacible(add(F1, S)) IFF es_insatisfacible(add(F2, S)))

es_cons_logica(F, S): bool =
    FORALL I: es_modelo(I, S) IMPLIES es_modelo(I, F)

equivalentes_cons_log: LEMMA
  equivalentes(F1, F2) IMPLIES
   (FORALL S: es_cons_logica(F1, S) IFF es_cons_logica(F2, S))

cons_logica_insatisfacible: LEMMA
  es_cons_logica(F, S) IFF es_insatisfacible(add(~F, S))

cons_log_conj_binaria: LEMMA
  es_cons_logica(F1 & F2, S) IFF
   (es_cons_logica(F1, S) AND es_cons_logica(F2, S))

cons_log_verdad: CLAIM es_cons_logica(verdad, S)
```

```
conjuncion_TCC1: OBLIGATION
  FORALL (FS: finite_set[formula_prop]):
    NOT empty?(FS) IMPLIES nonempty?[formula_prop[T]](FS);

conjuncion_TCC2: OBLIGATION
  FORALL (FS: finite_set[formula_prop]):
    NOT empty?(FS) IMPLIES
     card[formula_prop[T]](rest[formula_prop[T]](FS)) <
      card[formula_prop[T]](FS);

conjuncion(FS): RECURSIVE formula_prop =
  IF empty?(FS) THEN verdad ELSE choose(FS) & conjuncion(rest(FS)) ENDIF
   MEASURE card(FS)

conjuncion_vacio: LEMMA conjuncion(emptyset[formula_prop[T]]) = verdad

conjuncion_unitario_aux: LEMMA conjuncion(singleton(F)) = (F & verdad)

conjuncion_unitario: LEMMA equivalentes(conjuncion(singleton(F)), F)

conjuncion_unitario_atomo: LEMMA
  equivalentes(conjuncion(singleton[formula_prop](A)), A)

disyuncion(FS): RECURSIVE formula_prop =
  IF empty?(FS) THEN falso ELSE choose(FS) \/ disyuncion(rest(FS)) ENDIF
   MEASURE card(FS)

es_modelo_conjuncion: LEMMA
  es_modelo(I, conjuncion(FS)) IFF
   (FORALL F: member(F, FS) IMPLIES es_modelo(I, F))

no_es_modelo_conjuncion: COROLLARY
  NOT es_modelo(I, conjuncion(FS)) IFF
   (EXISTS F: member(F, FS) AND NOT es_modelo(I, F))

simb_prop_conjuncion_s_l1: LEMMA
  simb_prop(conjuncion(FS)) =
   Union(extend[setof[atomo[T]], finite_set[atomo[T]], bool, FALSE]
             (image(simb_prop)(FS)))

cons_log_conjuncion_s: LEMMA
  es_cons_logica(conjuncion(FS), S) IFF
   (FORALL F: member(F, FS) IMPLIES es_cons_logica(F, S))

lenguaje: TYPE = set[atomo]

LE: VAR lenguaje

formula_prop_l_TCC1: OBLIGATION
  FORALL (LE: lenguaje): subset?[atomo[T]](simb_prop(falso[T]), LE);
```

```
formula_prop_l(LE): TYPE = {F: formula_prop | subset?(simb_prop(F), LE)}
CONTAINING falso

JUDGEMENT formula_prop_l(LE) SUBTYPE_OF formula_prop

es_interpretacion_herbrand(LE)(I): bool = subset?(I, LE)

interpretacion_l(LE): TYPE = (es_interpretacion_herbrand(LE))

interpretacion_restringida: LEMMA
  es_interpretacion_herbrand(LE)(intersection(I, LE))

es_satisfacible(LE)(F: formula_prop_l(LE)): bool =
    EXISTS (I: interpretacion_l(LE)): es_modelo(I, F)

es_satisfacible_le: LEMMA
  FORALL (F: formula_prop_l(LE)):
    es_satisfacible(LE)(F) IFF es_satisfacible(F)

es_valida(LE)(F: formula_prop_l(LE)): bool =
    FORALL (I: interpretacion_l(LE)): es_modelo(I, F)

es_valida_le: LEMMA
  FORALL (F: formula_prop_l(LE)): es_valida(LE)(F) IFF es_valida(F)

es_cons_logica(LE)(F: formula_prop_l(LE), S: set[formula_prop_l(LE)]):
bool =
    FORALL (I: interpretacion_l(LE)):
      es_modelo(I,
                extend[formula_prop[T], formula_prop_l(LE), bool, FALSE]
                    (S))
       IMPLIES es_modelo(I, F)

interpretacion_restringida_es_modelo_formula: LEMMA
  FORALL (F: formula_prop_l(LE)):
    es_modelo(I, F) IFF es_modelo(intersection(I, LE), F)

interpretacion_restringida_es_modelo_conjunto: LEMMA
  FORALL (S: set[formula_prop_l(LE)]):
    es_modelo(I,
              extend[formula_prop[T], formula_prop_l(LE), bool, FALSE]
                  (S))
     IFF
     es_modelo(intersection(I, LE),
                extend[formula_prop[T], formula_prop_l(LE), bool, FALSE]
                    (S))

es_cons_logica_le: LEMMA
  FORALL (F: formula_prop_l(LE), S: set[formula_prop_l(LE)]):
    es_cons_logica(LE)(F, S) IFF
     es_cons_logica(F,
                    extend[formula_prop[T], formula_prop_l(LE), bool,
```

```
                              FALSE]
                            (S))

  base_herbrand(LE): set[atomo] = LE

  es_modelo_herbrand(LE)(I: interpretacion, F: formula_prop_l(LE)): bool =
      es_interpretacion_herbrand(LE)(I) AND es_modelo(I, F)

  es_modelo_herbrand(LE)(I: interpretacion, S: set[formula_prop_l(LE)]):
  bool =
      es_interpretacion_herbrand(LE)(I) AND
       es_modelo(I,
                 extend[formula_prop[T], formula_prop_l(LE), bool, FALSE]
                     (S))
 END semantica_proposicional
```

D.1.6. Semántica de cláusulas

```
semantica_clausulas[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING clausulas[T]

  IMPORTING TC@propiedades_extend

  A, A1, A2: VAR atomo[T]

  CH: VAR clausula_horn[T]

  C: VAR clausula_def[T]

  G: VAR objetivo_def[T]

  P, P1, P2: VAR programa[T]

  I, I1, I2: VAR interpretacion[T]

  SC: VAR set[clausula_horn[T]]

  FSC: VAR finite_set[clausula_horn[T]]

  LE: VAR lenguaje[T]

  es_modelo(I, CH): bool = es_modelo(I, fla(CH))

  es_modelo_cd((I: interpretacion_finita[T]), C): bool = es_modelo(I, C)
```

```
ej_1_es_modelo: LEMMA es_modelo(singleton(A1), A1)

ej_2_es_modelo: LEMMA A1 /= A2 IMPLIES NOT es_modelo(singleton(A1), A2)

es_modelo_clausula_def_caract_TCC1: OBLIGATION
  FORALL (C: clausula_def[T]): atomo?[T](cabeza(C));

es_modelo_clausula_def_caract: LEMMA
  es_modelo(I, C) IFF
   member(cabeza(C), I) OR
    (EXISTS A: member(A, cuerpo(C)) AND NOT member(A, I))

cs1_es_modelo_clausula_def: COROLLARY
  member(cabeza(C), I) IMPLIES es_modelo(I, C)

cs2_es_modelo_clausula_def: COROLLARY
  member(A, cuerpo(C)) AND NOT member(A, I) IMPLIES es_modelo(I, C)

cs3_es_modelo_clausula_def: COROLLARY
  NOT subset?(cuerpo(C), I) IMPLIES es_modelo(I, C)

cn1_es_modelo_clausula_def: COROLLARY
  es_modelo(I, C) AND NOT member(cabeza(C), I) IMPLIES
   (EXISTS A: member(A, cuerpo(C)) AND NOT member(A, I))

cn2_es_modelo_clausula_def: COROLLARY
  es_modelo(I, C) AND NOT member(cabeza(C), I) IMPLIES
   NOT subset?(cuerpo(C), I)

es_modelo_objetivo_def_caract: LEMMA
  es_modelo(I, G) IFF
   (EXISTS A: member(A, cuerpo(G)) AND NOT member(A, I))

no_es_modelo_de_vacia: LEMMA vacia?(G) IMPLIES NOT es_modelo(I, G)

cns_modelo_objetivo_unitario: COROLLARY
  es_modelo(I, objetivo(singleton(A))) IFF NOT member(A, I)

cs1_es_modelo_objetivo_def_neg: LEMMA
  NOT es_modelo(I, G) IMPLIES subset?(cuerpo(G), I)

cn1_es_modelo_objetivo_def: LEMMA
  es_modelo(I, G) IMPLIES
   (EXISTS A: member(A, cuerpo(G)) AND NOT member(A, I))

es_modelo(I, SC): bool =
    FORALL CH: member(CH, SC) IMPLIES es_modelo(I, CH)

es_modelo_cl_fla: LEMMA es_modelo(I, SC) IFF es_modelo(I, conj_fla(SC))

modelo_add_cl: LEMMA
  es_modelo(I, add(CH, SC)) IFF (es_modelo(I, SC) AND es_modelo(I, CH))
```

```
es_modelo(I, P): bool = FORALL C: member(C, P) IMPLIES es_modelo(I, C)

es_modelo_extend: LEMMA
  es_modelo(I, extend[clausula_horn[T], clausula_def[T], bool, FALSE](P))
   IFF es_modelo(I, P)

es_cons_logica(A, SC): bool =
    FORALL I: es_modelo(I, SC) IMPLIES es_modelo(I, A)

es_cons_logica_atomo_cl_fla: LEMMA
  es_cons_logica(A, SC) IFF es_cons_logica(A, conj_fla(SC))

es_cons_logica(A, P): bool =
    FORALL I: es_modelo(I, P) IMPLIES es_modelo(I, A)

es_cons_logica_extend: LEMMA
  es_cons_logica(A,
                 extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                     (P))
   IFF es_cons_logica(A, P)

es_cons_logica_atomo_cl_fla_programa: LEMMA
  es_cons_logica(A, P) IFF
   es_cons_logica(A,
                  conj_fla(extend[clausula_horn[T],
                                  clausula_def[T],
                                  bool,
                                  FALSE]
                                 (P)))

es_cons_logica(CH, SC): bool =
    FORALL I: es_modelo(I, SC) IMPLIES es_modelo(I, CH)

es_cons_logica_cl_fla: LEMMA
  es_cons_logica(CH, SC) IFF es_cons_logica(fla(CH), conj_fla(SC))

es_cons_logica(C, P): bool =
    FORALL I: es_modelo(I, P) IMPLIES es_modelo(I, C)

es_cons_logica_extend_cl: LEMMA
  es_cons_logica(C,
                 extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                     (P))
   IFF es_cons_logica(C, P)

es_cons_logica_cl_fla_programa: LEMMA
  es_cons_logica(C, P) IFF
   es_cons_logica(fla(C),
                  conj_fla(extend[clausula_horn[T],
                                  clausula_def[T],
                                  bool,
```

```
                                  FALSE]
                               (P)))

  es_satisfacible(CH): bool = EXISTS I: es_modelo(I, CH)

  es_satisfacible(SC): bool = EXISTS I: es_modelo(I, SC)

  es_satisfacible_cl_fla: LEMMA
    es_satisfacible(SC) IFF es_satisfacible(conj_fla(SC))

  es_insatisfacible(SC): bool = FORALL I: NOT es_modelo(I, SC)

  es_insatisfacible_cl_fla: LEMMA
    es_insatisfacible(SC) IFF es_insatisfacible(conj_fla(SC))

  es_insatisfacible_vacio: LEMMA
    vacia?(G) IMPLIES
     es_insatisfacible(add[clausula_horn[T]]
                           (G,
                            extend[clausula_horn[T], clausula_def[T], bool,
                                   FALSE]
                                (P)))

  log_equivalente_hecho: CLAIM equivalentes(A, fla(hecho(A)))

  cons_log_l_hecho: LEMMA
    es_cons_logica(A, P) IFF es_cons_logica(hecho(A), P)

  objetivo_unit_log_equiv: CLAIM
    equivalentes(fla(objetivo(singleton(A))), ~A)

  cons_log_insatisfacilidad: THEOREM
    es_cons_logica(A, SC) IFF
     es_insatisfacible(add(objetivo(singleton(A)), SC))

  cons_log_insatisfacilidad_p: THEOREM
    es_cons_logica(A, P) IFF
     es_insatisfacible(add[clausula_horn[T]]
                           (objetivo(singleton(A)),
                            extend[clausula_horn[T], clausula_def[T], bool,
                                   FALSE]
                                (P)))

  equivalente_fla_objetivo: CLAIM
    equivalentes(fla(G),
                 ~conjuncion(extend[formula_prop[T], atomo[T], bool, FALSE]
                                (cuerpo(G))))

  cons_log_objetivo: THEOREM
    es_insatisfacible(add(G, SC)) IFF
     (FORALL A: member(A, cuerpo(G)) IMPLIES es_cons_logica(A, SC))
```

```
simb_prop(CH): finite_set[atomo] = simb_prop(fla(CH))

simb_prop_contiene_cabeza: LEMMA member(cabeza(C), simb_prop(C))

simb_prop_conjuncion_cuerpo: CLAIM
  simb_prop(conjuncion(extend[formula_prop[T], atomo[T], bool, FALSE]
                           (cuerpo(CH))))
   = cuerpo(CH)

CNS_simb_prop: LEMMA
  simb_prop(CH) = union(simb_prop(cabeza(CH)), cuerpo(CH))

simb_prop_objetivo_unitario: COROLLARY
  simb_prop(objetivo(singleton(A))) = singleton(A)

clausula_horn_1(LE): TYPE =
{CH: clausula_horn | subset?(simb_prop(CH), LE)}

LE(SC): set[atomo] =
    Union(extend[setof[atomo[T]], finite_set[atomo[T]], bool, FALSE]
              (image(simb_prop)(SC)))

LE_TCC1: OBLIGATION FORALL (FSC): is_finite[atomo[T]](LE(FSC));

JUDGEMENT LE(FSC) HAS_TYPE finite_set[atomo]

base_herbrand(SC): set[atomo] = base_herbrand(LE(SC))

base_herbrand_TCC1: OBLIGATION
  FORALL (FSC): is_finite[atomo[T]](base_herbrand(FSC));

JUDGEMENT base_herbrand(FSC) HAS_TYPE finite_set[atomo]

JUDGEMENT base_herbrand(FSC) HAS_TYPE interpretacion_finita

LE_add: LEMMA LE(add(CH, SC)) = union(simb_prop(CH), LE(SC))

LE_add_subset: COROLLARY subset?(LE(SC), LE(add(CH, SC)))

es_interpretacion_herbrand(I, SC): bool =
    es_interpretacion_herbrand(LE(SC))(I)

cs1_es_interpretacion_herbrand: LEMMA
  es_interpretacion_herbrand(I, add(objetivo(singleton(A)), SC)) AND
   NOT member(A, I)
   IMPLIES es_interpretacion_herbrand(I, SC)

es_modelo_herbrand(I, SC): bool =
    es_modelo(I, SC) AND es_interpretacion_herbrand(I, SC)

es_modelo_simb_prop_modelos_cl: LEMMA
  (FORALL A:
```

```
      member(A, simb_prop(CH)) IMPLIES
       (es_modelo(I1, A) = es_modelo(I2, A)))
   IMPLIES (es_modelo(I1, CH) IFF es_modelo(I2, CH))

modelo_c_simb: LEMMA
  es_modelo(I, CH) IMPLIES es_modelo(intersection(I, simb_prop(CH)), CH)

simb_prop_clausula_contenidos_b_herbrand: CLAIM
  member(CH, SC) IMPLIES subset?(simb_prop(CH), base_herbrand(SC))

CS_modelo_simb_prop_clausula_horn: CLAIM
  member(CH, SC) IMPLIES
   (FORALL A:
      member(A, simb_prop(CH)) IMPLIES
       (es_modelo(I, A) =
         es_modelo(intersection(I, base_herbrand(SC)), A)))

modelo_c_base_h: LEMMA
  es_modelo(I, CH) AND member(CH, SC) IMPLIES
   es_modelo(intersection(I, base_herbrand(SC)), CH)

modelo_imp_modelo_herbrand: THEOREM
  es_modelo(I, SC) IMPLIES
   es_modelo_herbrand(intersection(I, base_herbrand(SC)), SC)

modelo_imp_modelo_herbrand_p: COROLLARY
  es_modelo(I, P) IMPLIES
   es_modelo_herbrand(intersection(I,
                                   base_herbrand
                                   (extend[clausula_horn[T],
                                           clausula_def[T],
                                           bool,
                                           FALSE]
                                    (P))),
                      extend[clausula_horn[T], clausula_def[T], bool,
                             FALSE]
                         (P))

insatisfacible_mod_herbrand: THEOREM
  es_insatisfacible(SC) IFF (FORALL I: NOT es_modelo_herbrand(I, SC))

insatisfacible_mod_herbrand_alt: COROLLARY
  es_insatisfacible(SC) IFF
   (FORALL I:
      es_interpretacion_herbrand(I, SC) IMPLIES NOT es_modelo(I, SC))

insatisfacible_mod_herbrand_programa: COROLLARY
  es_insatisfacible(extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                        (P))
   IFF
   (FORALL I:
      NOT es_modelo_herbrand(I,
```

```
                                extend[clausula_horn[T],
                                       clausula_def[T],
                                       bool,
                                       FALSE]
                                   (P)))

  insatisfacible_mod_herbrand_programa_alt: COROLLARY
    es_insatisfacible(extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                           (P))
     IFF
     (FORALL (I:
                (powerset(LE(extend[clausula_horn[T],
                                    clausula_def[T],
                                    bool,
                                    FALSE]
                                  (P))))):
        NOT es_modelo(I, P))
 END semantica_clausulas
```

D.1.7. Semántica declarativa

```
semantica_declarativa[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_clausulas[T]

  A, A1, A2: VAR atomo

  CH: VAR clausula_horn

  C: VAR clausula_def

  P, P1, P2: VAR programa

  I, I1, I2: VAR interpretacion

  SC: VAR set[clausula_horn[T]]

  CI: VAR set[interpretacion]

  cabeza_pertenece_base_herbrand_g_TCC1: OBLIGATION
    FORALL (C: clausula_def[T], SC: set[clausula_horn[T]]):
      member(C, SC) IMPLIES atomo?[T](cabeza(C));

  cabeza_pertenece_base_herbrand_g: CLAIM
    member(C, SC) IMPLIES member(cabeza(C), base_herbrand(SC))
```

```
cabeza_pertenece_base_herbrand_TCC1: OBLIGATION
  FORALL (C: clausula_def[T], P: programa[T]):
    member(C, P) IMPLIES atomo?[T](cabeza(C));

cabeza_pertenece_base_herbrand: CLAIM
  member(C, P) IMPLIES
   member(cabeza(C),
          base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                              (P)))

base_herbrand_es_modelo: LEMMA
  es_modelo(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                                (P)),
            P)

BH_es_modelo_de_Herbrand: THEOREM
  es_modelo_herbrand(base_herbrand(extend[clausula_horn[T],
                                          clausula_def[T],
                                          bool,
                                          FALSE]
                                       (P)),
                     extend[clausula_horn[T], clausula_def[T], bool,
                            FALSE]
                         (P))

CNS_modelo_herbrand: LEMMA
  es_modelo_herbrand(I,
                     extend[clausula_horn[T], clausula_def[T], bool,
                            FALSE]
                         (P))
   IFF
   powerset(LE(extend[clausula_horn[T], clausula_def[T], bool, FALSE](P)))
           (I)
    AND es_modelo(I, P)

conj_modelos_herbrand_TCC1: OBLIGATION
  FORALL (P: programa[T]):
    is_finite[set[atomo[T]]]
        ({I: set[atomo[T]] |
            powerset[atomo[T]]
                (LE[T]
                     (extend[clausula_horn[T], clausula_def[T], bool,
                             FALSE]
                            (P)))
                (I)
             AND es_modelo[T](I, P)});

conj_modelos_herbrand(P): finite_set[set[atomo]] =
    {I: set[atomo[T]] |
       powerset(LE(extend[clausula_horn[T], clausula_def[T], bool, FALSE]
```

```
                        (P)))
                (I)
         AND es_modelo(I, P)}

  conj_modelos_herbrand_finitos: LEMMA
    every(is_finite)(conj_modelos_herbrand(P))

  menor_modelo_herbrand_TCC1: OBLIGATION
    FORALL (P: programa[T]):
      is_finite[atomo[T]](Intersection[atomo[T]](conj_modelos_herbrand(P)));

  menor_modelo_herbrand(P): finite_set[atomo] =
      Intersection(conj_modelos_herbrand(P))

  CNS_pertenencia_menor_modelo_herbrand: LEMMA
    member(A, menor_modelo_herbrand(P)) IFF
     (FORALL I:
        es_modelo_herbrand(I,
                           extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                               (P))
         IMPLIES member(A, I))

  es_conjunto_de_modelos?(CI, C): bool =
      FORALL I: member(I, CI) IMPLIES es_modelo(I, C)

  inters_familia_modelo_c_d: LEMMA
    es_conjunto_de_modelos?(CI, C) IMPLIES es_modelo(Intersection(CI), C)

  es_conjunto_de_modelos?(CI, P): bool =
      FORALL I: member(I, CI) IMPLIES es_modelo(I, P)

  conj_modelos_herbrand_es_conj_de_modelos: LEMMA
    es_conjunto_de_modelos?(conj_modelos_herbrand(P), P)

  conj_modelos_p_imp_conj_mod_c: CLAIM
    es_conjunto_de_modelos?(CI, P) AND member(C, P) IMPLIES
     es_conjunto_de_modelos?(CI, C)

  inters_familia_modelo_conj_c_d: LEMMA
    es_conjunto_de_modelos?(CI, P) IMPLIES es_modelo(Intersection(CI), P)

  menor_modelo_es_modelo: THEOREM es_modelo(menor_modelo_herbrand(P), P)

  menor_modelo_es_interpretacion_herbrand: LEMMA
    es_interpretacion_herbrand(menor_modelo_herbrand(P),
                               extend[clausula_horn[T],
                                      clausula_def[T],
                                      bool,
                                      FALSE]
                                  (P))
```

```
menor_modelo_es_modelo_herbrand: THEOREM
  es_modelo_herbrand(menor_modelo_herbrand(P),
                     extend[clausula_horn[T], clausula_def[T], bool,
                            FALSE]
                         (P))

menor_modelo_es_el_menor: THEOREM
  es_modelo_herbrand(I,
                     extend[clausula_horn[T], clausula_def[T], bool,
                            FALSE]
                         (P))
   IMPLIES subset?(menor_modelo_herbrand(P), I)

menor_modelo_es_el_menor_corol: COROLLARY
  subset?(menor_modelo_herbrand(P),
          base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                           (P)))

JUDGEMENT menor_modelo_herbrand(P) HAS_TYPE interpretacion_finita

mmh_menor_todos_los_modelos: COROLLARY
  es_modelo(I, P) IMPLIES subset?(menor_modelo_herbrand(P), I)

menor_modelo_con_log: LEMMA
  member(A, menor_modelo_herbrand(P)) IMPLIES
   member(A,
          base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                           (P)))
    AND es_cons_logica(A, P)

con_log_menor_modelo: LEMMA
  member(A,
         base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                              FALSE]
                          (P)))
   AND es_cons_logica(A, P)
   IMPLIES member(A, menor_modelo_herbrand(P))

con_log_equiv_menor_modelo: THEOREM
  menor_modelo_herbrand(P) =
   ({A |
       member(A,
              base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                               (P)))
        AND es_cons_logica(A, P)})

CL_TCC1: OBLIGATION
```

```
    FORALL (P: programa[T]):
      is_finite[atomo[T]]
          ({A |
              member[atomo[T]]
                  (A,
                   base_herbrand[T]
                       (extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                              (P)))
               AND es_cons_logica[T](A, P)});

  CL(P): finite_set[atomo[T]] =
      {A |
         member(A,
                base_herbrand(extend[clausula_horn[T],
                                     clausula_def[T],
                                     bool,
                                     FALSE]
                                    (P)))
          AND es_cons_logica(A, P)}

  con_log_equiv_menor_modelo_c: COROLLARY menor_modelo_herbrand(P) = CL(P)
 END semantica_declarativa
```

D.1.8. Operador de consecuencia inmediata

```
consecuencia_i[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_declarativa[T]

  CH: VAR clausula_horn

  C: VAR clausula_def

  A: VAR atomo

  I, J: VAR interpretacion

  FI: VAR interpretacion_finita

  P, P1: VAR programa

  n, m: VAR nat

  c_i(P)(I): interpretacion =
      {A |
```

```
       EXISTS C: member(C, P) AND A = cabeza(C) AND subset?(cuerpo(C), I)}

cns_pertenece_c_i: LEMMA
  member(A, c_i(P)(I)) IFF
   (EXISTS C: member(C, P) AND A = cabeza(C) AND subset?(cuerpo(C), I))

cs_pertenece_c_i: COROLLARY
  A = cabeza(C) AND member(C, P) AND subset?(cuerpo(C), I) IMPLIES
   member(A, c_i(P)(I))

cabeza_pertenece_c_i_TCC1: OBLIGATION
  FORALL (C: clausula_def[T], I: interpretacion[T], P: programa[T]):
    subset?(cuerpo(C), I) AND member(C, P) IMPLIES atomo?[T](cabeza(C));

cabeza_pertenece_c_i: COROLLARY
  member(C, P) AND subset?(cuerpo(C), I) IMPLIES
   member(cabeza(C), c_i(P)(I))

cn_pertenece_c_i: COROLLARY
  member(A, c_i(P)(I)) IMPLIES
   (EXISTS C: A = cabeza(C) AND member(C, P) AND subset?(cuerpo(C), I))

c_i_en_base_herbrand: LEMMA
  subset?(c_i(P)(I),
          base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                              (P)))

c_i_interp_herbrand: COROLLARY
  es_interpretacion_herbrand(c_i(P)(I),
                             extend[clausula_horn[T],
                                    clausula_def[T],
                                    bool,
                                    FALSE]
                                   (P))

c_i_TCC1: OBLIGATION FORALL (I, P): is_finite[atomo[T]](c_i(P)(I));

JUDGEMENT c_i(P)(I) HAS_TYPE interpretacion_finita

c_i_f(P)(FI): interpretacion_finita = c_i(P)(FI)

c_i_monotona: LEMMA subset?(I, J) IMPLIES subset?(c_i(P)(I), c_i(P)(J))

potencia_c_i(P, n): interpretacion = iterate(c_i(P), n)(emptyset)

base_herbrand_contiene_potencias: LEMMA
  subset?(iterate(c_i(P), n)(emptyset),
          base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                              (P)))
```

```
  potencia_c_i_finito: OBLIGATION
    FORALL (P, n): is_finite[atomo[T]](potencia_c_i(P, n));

  potencia_c_i_finito: JUDGEMENT potencia_c_i(P, n) HAS_TYPE
      finite_set[atomo[T]]

  potencia_c_i_finito_b: JUDGEMENT potencia_c_i(P, n) HAS_TYPE
      interpretacion_finita

  c_i_iterate_vacio: LEMMA
    subset?(iterate(c_i(P), n)(emptyset), iterate(c_i(P), 1 + n)(emptyset))

  c_i_iterate_vacio_gen: COROLLARY
    m <= n IMPLIES
     subset?(iterate(c_i(P), m)(emptyset), iterate(c_i(P), n)(emptyset))

  c_i_modelos_h: THEOREM es_modelo(I, P) IFF subset?(c_i(P)(I), I)
 END consecuencia_i
```

D.1.9. La inclusión como orden parcial completo

```
inclusion_opc[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  PF: LIBRARY = "../auxiliares/dominios"

  IMPORTING PF@po[set[T], sets[T].subset?], PF@cpo_defs[set[T]]

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@propiedades_conjuntos

  a: VAR T

  S, S1: VAR set[T]

  SS, Cad: VAR set[set[T]]

  subset?_es_precop_l1: LEMMA chain?(SS) IMPLIES ub?(Union(SS), SS)

  subset?_es_precop_l2: LEMMA chain?(SS) IMPLIES lub?(Union(SS), SS)

  subset?_es_precop_l3: LEMMA chain?(Cad) IMPLIES lub_exists?(Cad)

  subset?_es_precop: LEMMA precpo?(subset?)

  subset?_TCC1: OBLIGATION precpo?[set[T]](subset?[T]);
```

```
  JUDGEMENT subset? HAS_TYPE pCPO[set[T]]

  vacio_es_bottom: LEMMA bottom?(subset?)(emptyset)

  subset?_es_cpo: THEOREM cpo?(subset?, emptyset)

  IMP_fixpoints_cont_TCC1: OBLIGATION
    bottom?[set[T]](sets[T].subset?)(sets[T].emptyset);

  IMPORTING PF@fixpoints_cont[set[T], sets[T].subset?, sets[T].emptyset]
 END inclusion_opc
```

D.1.10. Semántica del punto fijo

```
semantica_p_f[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING consecuencia_i[T], semantica_declarativa[T],
            inclusion_opc[atomo[T]],
            finite_sets@finite_sets_inductions[atomo]

  PF: LIBRARY = "../auxiliares/dominios"

  IMPORTING PF@propiedades_cadenas

  A, B: VAR atomo

  Cl: VAR clausula_def

  P, P1: VAR programa

  I, J: VAR interpretacion

  FI: VAR interpretacion_finita

  SS: VAR set[set[atomo]]

  Cad: VAR po[set[atomo], sets[atomo].subset?].Chain

  m, n: VAR nat

  c_i_monotonic: LEMMA monotonic?(c_i(P))

  c_i_TCC1: OBLIGATION
    FORALL (P):
      monotonic?[set[atomo[T]], sets[atomo[T]].subset?, set[atomo[T]],
                 sets[atomo[T]].subset?]
```

```
        (c_i[T](P));

  JUDGEMENT c_i(P) HAS_TYPE Monotonic

  mpf_TCC1: OBLIGATION
    bottom?[set[atomo[T]]](sets[atomo[T]].subset?)(sets[atomo[T]].emptyset);

  mpf(P): interpretacion = u(c_i(P))

  mpf_subset_mmh_l1: LEMMA
    subset?(c_i(P)(menor_modelo_herbrand(P)), menor_modelo_herbrand(P))

  mpf_subset_mmh: THEOREM subset?(mpf(P), menor_modelo_herbrand(P))

  mpf_es_modelo: LEMMA es_modelo(mpf(P), P)

  mpf_es_interp_herbrand: LEMMA
    es_interpretacion_herbrand(mpf(P),
                               extend[clausula_horn[T],
                                      clausula_def[T],
                                      bool,
                                      FALSE]
                                  (P))

  mmh_subset_mpf: THEOREM subset?(menor_modelo_herbrand(P), mpf(P))

  mmh_es_mpf: THEOREM menor_modelo_herbrand(P) = mpf(P)

  mpf_TCC2: OBLIGATION FORALL (P): is_finite[atomo[T]](mpf(P));

  JUDGEMENT mpf(P) HAS_TYPE interpretacion_finita

  mmh_es_punto_fijo_c_i: COROLLARY
    fixpoint?(c_i(P))(menor_modelo_herbrand(P))

  mpf_es_cl: COROLLARY mpf(P) = CL(P)

  IMPORTING PF@po_lems[set[atomo[T]], sets[atomo[T]].subset?]

  c_i_continua_lema_1_rec: CLAIM
    ub?(I, SS) AND member(A, I) AND (FORALL (S: (SS)): NOT S(A)) IMPLIES
     ub?(remove(A, I), SS)

  c_i_continua_lema_1: LEMMA
    lub_exists?(SS) AND
     member(A, po[set[atomo], sets[atomo].subset?].lub(SS))
     IMPLIES (EXISTS (S: (SS)): S(A))

  c_i_es_continua_l1_2: COROLLARY
    member(A, lub(Cad)) IMPLIES (EXISTS (I: (Cad)): I(A))

  c_i_es_continua_l1: LEMMA
```

```
  FORALL (I: finite_set[atomo]):
    subset?(I, lub(Cad)) IMPLIES (EXISTS (J: (Cad)): subset?(I, J))

c_i_es_continua_l2: LEMMA
  subset?(c_i(P)(lub(Cad)),
          po[set[atomo], sets[atomo].subset?].lub(set_image(c_i(P))(Cad)))

c_i_es_continua_l3_1: LEMMA ub?(c_i(P)(lub(Cad)), set_image(c_i(P))(Cad))

c_i_es_continua_l3: LEMMA
  subset?(po[set[atomo], sets[atomo].subset?].lub
              (set_image(c_i(P))(Cad)),
          c_i(P)(lub(Cad)))

c_i_es_continua: THEOREM continuous?(c_i(P))

punto_fijo_c_i: THEOREM mu(c_i(P)) = lub(bottom_iterations(c_i(P)))

calculo_mpf: COROLLARY mpf(P) = lub(bottom_iterations(c_i(P)))

base_herbrand_finito: LEMMA
  is_finite(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                               (P)))

existencia_punto_fijo_finito: COROLLARY
  fixpoint?(c_i(P))(I) IMPLIES
   (EXISTS J: is_finite(J) AND fixpoint?(c_i(P))(J) AND subset?(J, I))

base_herbrand_cota_superior_c_i: COROLLARY
  ub?(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                        (P)),
      bottom_iterations(c_i(P)))

cardinal_potencias_c_i_l1_b: CLAIM
  strict_subset?(potencia_c_i(P, n + 1), potencia_c_i(P, n + 2)) IMPLIES
   strict_subset?(potencia_c_i(P, n), potencia_c_i(P, n + 1))

cardinal_potencias_c_i_b: LEMMA
  strict_subset?(potencia_c_i(P, n), potencia_c_i(P, n + 1)) IMPLIES
   card(potencia_c_i(P, n + 1)) >= n + 1

cardinal_potencias_c_i_l1: CLAIM
  strict_subset?(iterate(c_i(P), n + 1)(emptyset),
                 iterate(c_i(P), n + 2)(emptyset))
   IMPLIES
   strict_subset?(iterate(c_i(P), n)(emptyset),
                  iterate(c_i(P), n + 1)(emptyset))

cardinal_potencias_c_i_TCC1: OBLIGATION
  FORALL (P: programa[T], n: nat):
    strict_subset?(iterate(c_i(P), n)(emptyset),
```

```
                    iterate(c_i(P), 1 + n)(emptyset))
      IMPLIES
      is_finite[atomo[T]]
          (iterate[interpretacion[T]]
               (c_i[T](P), 1 + n)(emptyset[atomo[T]]));

cardinal_potencias_c_i: LEMMA
  strict_subset?(iterate(c_i(P), n)(emptyset),
                 iterate(c_i(P), 1 + n)(emptyset))
   IMPLIES card(iterate(c_i(P), 1 + n)(emptyset)) >= n + 1

CS_potencia_punto_fijo: LEMMA
  n =
   card(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                             FALSE]
                            (P)))
   IMPLIES fixpoint?(c_i(P))(iterate(c_i(P), n)(emptyset))

CN_potencia_punto_fijo: LEMMA
  fixpoint?(c_i(P))(iterate(c_i(P), m)(emptyset)) IMPLIES
   (FORALL n:
      reals.<=(m, n) IMPLIES
       iterate(c_i(P), n)(emptyset) = iterate(c_i(P), m)(emptyset))

CN_pertenencia_potencias_c_i: COROLLARY
  member(I, bottom_iterations(c_i(P))) IMPLIES
   (EXISTS n:
      reals.<=
          (n,
           card(base_herbrand(extend[clausula_horn[T],
                                     clausula_def[T],
                                     bool,
                                     FALSE]
                                    (P))))
       AND I = iterate(c_i(P), n)(emptyset))

potencias_c_i_finito: LEMMA is_finite(bottom_iterations(c_i(P)))

mpf_potencia_c_i: THEOREM
  EXISTS n:
    reals.<=
        (n,
         card(base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                  (P))))
     AND mpf(P) = iterate(c_i(P), n)(emptyset)

mpf_potencia_c_i_corol: COROLLARY
  mpf(P) =
   iterate(c_i(P),
```

```
            card(base_herbrand(extend[clausula_horn[T],
                                      clausula_def[T],
                                      bool,
                                      FALSE]
                                     (P))))
           (emptyset)

  iterate_c_i_f: LEMMA iterate(c_i_f(P), n)(FI) = iterate(c_i(P), n)(FI)

  mpf_potencia_c_i_corol_f: COROLLARY
    mpf(P) =
     iterate(c_i_f(P),
             card(base_herbrand(extend[clausula_horn[T],
                                      clausula_def[T],
                                      bool,
                                      FALSE]
                                     (P))))
           (emptyset)
 END semantica_p_f
```

D.1.11. Semántica procedimental: resolución SLD

```
sld_resolucion[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_p_f[T], finite_sets@finite_sets_inductions[atomo[T]]

  A, B: VAR atomo

  C: VAR clausula_def

  G, G1, G2: VAR objetivo_def

  P, P1: VAR programa

  I, J: VAR interpretacion

  F: VAR finite_set[atomo]

  S: VAR finite_set[objetivo_def]

  n, m, k: VAR nat

  tiene_resolvente_TCC1: OBLIGATION
    FORALL (C: clausula_def[T]): atomo?[T](cabeza(C));

  tiene_resolvente(G, C): bool = member(cabeza(C), cuerpo(G))
```

```
PD: TYPE =
{par: [objetivo_def, clausula_def] |
         LET (G, C) = par IN tiene_resolvente(G, C)}

resolvente(par: PD): objetivo_def[T] =
    LET (G, C) = par IN
      objetivo(union(remove(cabeza(C), cuerpo(G)), cuerpo(C)))

es_resolvente?_TCC1: OBLIGATION
  FORALL (C: clausula_def[T], G: objetivo_def[T]):
    member(cabeza(C), cuerpo(G)) IMPLIES tiene_resolvente(G, C);

es_resolvente?(G2, G, C): bool =
    member(cabeza(C), cuerpo(G)) AND G2 = resolvente(G, C)

tiene_refutacion_long_TCC1: OBLIGATION
  FORALL (G: objetivo_def[T], P: programa[T], n: nat):
    NOT n = 0 IMPLIES
     (FORALL (C):
        member(C, P) AND member(cabeza(C), cuerpo(G)) IMPLIES n - 1 >= 0);

tiene_refutacion_long_TCC2: OBLIGATION
  FORALL (G: objetivo_def[T], P: programa[T], n: nat):
    NOT n = 0 IMPLIES
     (FORALL (C):
        member(C, P) AND member(cabeza(C), cuerpo(G)) IMPLIES n - 1 < n);

tiene_refutacion_long(P, G, n): RECURSIVE bool =
  IF n = 0 THEN vacia?(G)
  ELSE EXISTS C:
         member(C, P) AND
          member(cabeza(C), cuerpo(G)) AND
           tiene_refutacion_long(P, resolvente(G, C), n - 1)
  ENDIF
   MEASURE n

tiene_refutacion(P, G): bool = EXISTS n: tiene_refutacion_long(P, G, n)

tiene_refutacion_long(P)(G, n): bool = tiene_refutacion_long(P, G, n)

tiene_refutacion(P)(G): bool = tiene_refutacion(P, G)

Exitos_TCC1: OBLIGATION
  FORALL (P: programa[T]):
    is_finite[atomo[T]]
        ({A |
            member[atomo[T]]
                (A,
                 base_herbrand[T]
                     (extend[clausula_horn[T], clausula_def[T], bool,
                             FALSE]
```

```
                          (P)))
             AND
             tiene_refutacion(P, objetivo[T](singleton[atomo[T]](A)))});

Exitos(P): finite_set[atomo[T]] =
    {A |
       member(A,
              base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                (P)))
        AND tiene_refutacion(P, objetivo(singleton(A)))}

CS1_pertenece_cuerpo_resolvente: LEMMA
  tiene_resolvente(G, C) AND member(A, cuerpo(C)) IMPLIES
   member(A, cuerpo(resolvente(G, C)))

CS2_pertenece_cuerpo_resolvente: LEMMA
  tiene_resolvente(G, C) AND member(A, cuerpo(G)) AND A /= cabeza(C)
   IMPLIES member(A, cuerpo(resolvente(G, C)))

CS1_modelo_resolvente: LEMMA
  es_modelo(I, C) AND es_modelo(I, G) AND tiene_resolvente(G, C) IMPLIES
   es_modelo(I, resolvente(G, C))

CS2_modelo_resolvente_aux: CLAIM
  es_modelo(I, P) AND member(C, P) IMPLIES es_modelo(I, C)

CS2_modelo_resolvente: LEMMA
  es_modelo(I, P) AND
   es_modelo(I, G) AND member(C, P) AND tiene_resolvente(G, C)
   IMPLIES es_modelo(I, resolvente(G, C))

CN_insatisfacible_add_resolvente: LEMMA
  member(C, P) AND
   tiene_resolvente(G, C) AND
    es_insatisfacible(add[clausula_horn[T]]
                          (resolvente(G, C),
                           extend[clausula_horn[T],
                                  clausula_def[T],
                                  bool,
                                  FALSE]
                               (P)))
   IMPLIES
   es_insatisfacible(add[clausula_horn[T]]
                         (G,
                          extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                              (P)))

CN_tiene_refutacion_long_mayor_0: LEMMA
```

```
  tiene_refutacion_long(P, G, n) AND n > 0 IMPLIES NOT vacia?(G)

tiene_refutacion_long_insatisfacible: LEMMA
  tiene_refutacion_long(P, G, n) IMPLIES
   es_insatisfacible(add[clausula_horn[T]]
                         (G,
                          extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                              (P)))

sld_adecuacion: THEOREM
  tiene_refutacion(P, G) IMPLIES
   es_insatisfacible(add[clausula_horn[T]]
                         (G,
                          extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                              (P)))

tiene_refutacion_cons_logica: COROLLARY
  tiene_refutacion(P, G) IMPLIES
   (FORALL A: member(A, cuerpo(G)) IMPLIES es_cons_logica(A, P))

exitos_contenido_menor_modelo_herbrand: COROLLARY
  subset?(Exitos(P), menor_modelo_herbrand(P))

CN_subconjunto_cuerpo_resolvente: CLAIM
  tiene_resolvente(G, C) AND subset?(cuerpo(resolvente(G, C)), I) IMPLIES
   subset?(remove(cabeza(C), cuerpo(G)), I) AND subset?(cuerpo(C), I)

tiene_refutacion_long_consecuencia_i: THEOREM
  tiene_refutacion_long(P, G, n) IMPLIES
   subset?(cuerpo(G), iterate(c_i(P), n)(emptyset))

resolvente_objetivo_add_TCC1: OBLIGATION
  FORALL (A: atomo[T], C: clausula_def[T], G: objetivo_def[T]):
    NOT member(A, cuerpo(G)) AND tiene_resolvente(G, C) IMPLIES
     tiene_resolvente(objetivo[T](add[atomo[T]](A, cuerpo(G))), C);

resolvente_objetivo_add: CLAIM
  tiene_resolvente(G, C) AND NOT member(A, cuerpo(G)) IMPLIES
   resolvente(objetivo(add(A, cuerpo(G))), C) =
    objetivo(add(A, cuerpo(resolvente(G, C))))

composicion_refutaciones_1: LEMMA
  tiene_refutacion_long(P, G, m) AND
   tiene_refutacion_long(P, objetivo(singleton(A)), k)
   IMPLIES
   (EXISTS n:
      reals.<=(n, m + k) AND
       tiene_refutacion_long(P, objetivo(add(A, cuerpo(G))), n))

composicion_refutaciones_1_corol: COROLLARY
```

```
    tiene_refutacion(P, G) AND tiene_refutacion(P, objetivo(singleton(A)))
     IMPLIES tiene_refutacion(P, objetivo(add(A, cuerpo(G))))

  composicion_refutaciones_2: LEMMA
    (FORALL A:
       member(A, F) IMPLIES tiene_refutacion(P, objetivo(singleton(A))))
     IMPLIES tiene_refutacion(P, objetivo(F))

  composicion_refutaciones_2_corol: COROLLARY
    (FORALL A:
       member(A, cuerpo(G)) IMPLIES
        tiene_refutacion(P, objetivo(singleton(A))))
     IMPLIES tiene_refutacion(P, G)

  resolvente_objetivo_singleton_TCC1: OBLIGATION
    FORALL (A: atomo[T], C: clausula_def[T]):
      A = cabeza(C) IMPLIES
       tiene_resolvente(objetivo[T](singleton[atomo[T]](A)), C);

  resolvente_objetivo_singleton: CLAIM
    A = cabeza(C) IMPLIES
     resolvente(objetivo(singleton(A)), C) = objetivo(cuerpo(C))

  CN_tiene_refutación_objetivo_unitario: CLAIM
    member(C, P) AND
     A = cabeza(C) AND tiene_refutacion(P, objetivo(cuerpo(C)))
     IMPLIES tiene_refutacion(P, objetivo(singleton(A)))

  consecuencia_i_tiene_refutacion: LEMMA
    member(A, iterate(c_i(P), n)(emptyset)) IMPLIES
     tiene_refutacion(P, objetivo(singleton(A)))

  menor_modelo_herbrand_contenido_exitos: LEMMA
    subset?(menor_modelo_herbrand(P), Exitos(P))

  menor_modelo_herbrand_exitos: THEOREM menor_modelo_herbrand(P) = Exitos(P)

  sld_completitud: THEOREM
    es_insatisfacible(add[clausula_horn[T]]
                          (G,
                           extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                               (P)))
     IMPLIES tiene_refutacion(P, G)
 END sld_resolucion
```

D.1.12. Completitud fuerte de la resolución SLD

```
sld_resolucion_via[T: TYPE+]: THEORY
 BEGIN
```

```
ASSUMING
 T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
ENDASSUMING

IMPORTING sld_resolucion[T]

IMPORTING TC@min_nat_extension[nat]

C, C1, C2: VAR clausula_def[T]

G, G1, G2: VAR objetivo_def[T]

A, B: VAR atomo[T]

I, J: VAR interpretacion[T]

P, P1: VAR programa[T]

S: VAR finite_set[objetivo_def[T]]

n, m, k: VAR nat

F, H: VAR finite_set[atomo[T]]

es_regla_computacion(f: [objetivo_def[T] -> atomo[T]]): bool =
    FORALL G: member(f(G), cuerpo(G))

regla_computacion: TYPE = (es_regla_computacion)

f: VAR regla_computacion

tiene_refutacion_long_via_TCC1: OBLIGATION
  FORALL (G: objetivo_def[T], P: programa[T], f: regla_computacion,
          n: nat):
    NOT n = 0 IMPLIES
     (FORALL (C):
        member(C, P) AND cabeza(C) = f(G) IMPLIES
         tiene_resolvente[T](G, C));

tiene_refutacion_long_via_TCC2: OBLIGATION
  FORALL (G: objetivo_def[T], P: programa[T], f: regla_computacion,
          n: nat):
    NOT n = 0 IMPLIES
     (FORALL (C): member(C, P) AND cabeza(C) = f(G) IMPLIES n - 1 >= 0);

tiene_refutacion_long_via_TCC3: OBLIGATION
  FORALL (G: objetivo_def[T], P: programa[T], f: regla_computacion,
          n: nat):
    NOT n = 0 IMPLIES
     (FORALL (C): member(C, P) AND cabeza(C) = f(G) IMPLIES n - 1 < n);

tiene_refutacion_long_via(P, G, n, f): RECURSIVE bool =
```

```
  IF n = 0 THEN vacia?(G)
  ELSE EXISTS C:
         member(C, P) AND
          cabeza(C) = f(G) AND
           tiene_refutacion_long_via(P, resolvente(G, C), n - 1, f)
  ENDIF
   MEASURE n

tiene_refutacion_via(P, G, f): bool =
    EXISTS n: tiene_refutacion_long_via(P, G, n, f)

IMP_finite_sets_sum_TCC1: OBLIGATION
  identity?[nat]
      (restrict[[numfield, numfield], [nat, nat], numfield](+))(0);

IMP_finite_sets_sum_TCC2: OBLIGATION
  associative?[nat]
      (restrict[[numfield, numfield], [nat, nat], numfield](+))
   AND
   commutative?[nat]
       (restrict[[numfield, numfield], [nat, nat], numfield](+));

IMPORTING finite_sets@finite_sets_sum[atomo[T],
                                      nat,
                                      0,
                                      restrict[[numfield, numfield],
                                               [nat, nat],
                                               numfield]
                                      (+)]

IMPORTING finite_sets@finite_sets_minmax[nat,
                                         restrict[[real, real],
                                                  [nat, nat],
                                                  boolean]
                                         (reals.<=)]

cons_logica_exp: LEMMA
  member(A, CL(P)) IMPLIES
   (EXISTS (k: nat):
      reals.<=
          (k,
           card(base_herbrand(extend[clausula_horn[T],
                                     clausula_def[T],
                                     bool,
                                     FALSE]
                                    (P))))
       AND member(A, iterate(c_i(P), k)(emptyset)))

cons_logica_exp_corol: LEMMA
  member(A, CL(P)) IMPLIES
   nonempty?({k: nat |
                reals.<=
```

```
                    (k,
                     card(base_herbrand(extend[clausula_horn[T],
                                               clausula_def[T],
                                               bool,
                                               FALSE]
                                          (P))))
                AND member(A, iterate(c_i(P), k)(emptyset))})

m_exponente_TCC1: OBLIGATION
  FORALL (A: atomo[T], P: programa[T]):
    member(A, CL(P)) IMPLIES
     is_finite[nat]
         ({k: nat |
             (reals.<=
                  (k,
                   card[atomo[T]]
                       (base_herbrand[T]
                            (extend[clausula_horn[T],
                                    clausula_def[T],
                                    bool,
                                    FALSE]
                                (P)))))
              AND
              member[atomo[T]]
                  (A,
                   iterate[interpretacion[T]]
                       (c_i[T](P), k)(emptyset[atomo[T]]))})
      AND
      NOT empty?[nat]
              ({k: nat |
                  (reals.<=
                       (k,
                        card[atomo[T]]
                            (base_herbrand[T]
                                 (extend[clausula_horn[T],
                                         clausula_def[T],
                                         bool,
                                         FALSE]
                                  (P)))))
                   AND
                   member[atomo[T]]
                       (A,
                        iterate[interpretacion[T]]
                            (c_i[T](P), k)(emptyset[atomo[T]]))});

m_exponente(A, P): nat =
    IF member(A, CL(P))
      THEN min({k: nat |
                  reals.<=
                      (k,
                       card(base_herbrand(extend[clausula_horn[T],
                                                 clausula_def[T],
```

```
                                                    bool,
                                                    FALSE]
                                               (P))))
                    AND member(A, iterate(c_i(P), k)(emptyset))})
    ELSE 0
    ENDIF

cota_superior_m_exponente: LEMMA
  reals.<=
      (m_exponente(A, P),
       card(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                          (P))))

cs_pertenencia_potencia_c_i: LEMMA
  member(A, iterate(c_i(P), k)(emptyset)) IMPLIES k >= 1

cons_logica_exp_clausula_l: LEMMA
  member(A, CL(P)) AND m_exponente(A, P) = k IMPLIES
   member(A, iterate(c_i(P), k)(emptyset))

cons_logica_m_exponente: LEMMA
  member(A, CL(P)) IMPLIES m_exponente(A, P) >= 1

cons_logica_exp_clausula_TCC1: OBLIGATION
  FORALL (A: atomo[T], P: programa[T], k: nat):
    m_exponente(A, P) = k AND member(A, CL(P)) IMPLIES
     (FORALL (C): member(C, P) AND cabeza(C) = A IMPLIES k - 1 >= 0);

cons_logica_exp_clausula: LEMMA
  member(A, CL(P)) AND m_exponente(A, P) = k IMPLIES
   (EXISTS C:
      member(C, P) AND
       cabeza(C) = A AND
        subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset)))

IMPORTING finite_sets@finite_sets_nat

m_exponente_cl_caract: LEMMA
  member(A, CL(P)) IMPLIES
   (m_exponente(A, P) = k IFF
     member(A, iterate(c_i(P), k)(emptyset)) AND
      (FORALL n:
         reals.<=
             (n,
              card(base_herbrand(extend[clausula_horn[T],
                                        clausula_def[T],
                                        bool,
                                        FALSE]
                                       (P))))
          AND member(A, iterate(c_i(P), n)(emptyset))
          IMPLIES reals.<=(k, n)))
```

```
cota_m_exponente_c_i: LEMMA
  member(A, iterate(c_i(P), k)(emptyset)) IMPLIES
   reals.<=(m_exponente(A, P), k)

cs_m_exponente_1: LEMMA m_exponente(A, P) = 1 IMPLIES member(hecho(A), P)

med_exp_TCC1: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) IMPLIES
     NOT empty?[nat]
             (image[atomo[T], nat](LAMBDA A: m_exponente(A, P), F));

med_exp(F, P): nat =
    IF empty?(F) THEN 0
    ELSE max(image((LAMBDA A: m_exponente(A, P)), F))
    ENDIF

med_exp_subconjunto: LEMMA
  subset?(F, H) IMPLIES reals.<=(med_exp(F, P), med_exp(H, P))

med_exp_singleton: LEMMA med_exp(singleton(A), P) = m_exponente(A, P)

med_exp_cons_logica: LEMMA
  member(A, CL(P)) IMPLIES med_exp(singleton(A), P) >= 1

med_exp_l1: LEMMA
       member(A, CL(P)) AND m_exponente(A, P) = k AND member(C, P)
   AND cabeza(C) = A
   AND subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))
   IMPLIES med_exp(cuerpo(C), P) < med_exp(singleton(A), P)

med_exp_l2: LEMMA
       member(A, CL(P)) AND m_exponente(A, P) = k AND member(A, F)
   AND member(C, P) AND cabeza(C) = A
   AND subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))
   IMPLIES med_exp(cuerpo(C), P) < med_exp(F, P)

medida_a_TCC1: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND singleton?(F) IMPLIES
     is_finite[atomo[T]]
         (base_herbrand[T]
              (extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                   (P)));

medida_a_TCC2: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND singleton?(F) IMPLIES
     (FORALL (n:
                {n: nat |
                         n =
```

```
                          Card[atomo[T]]
                              (base_herbrand[T]
                                   (extend[clausula_horn[T],
                                           clausula_def[T],
                                           bool,
                                           FALSE]
                                    (P)))}):
        n =
         card(base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                (P)))
         IMPLIES nonempty?[atomo[T]](F));

medida_a_TCC3: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND singleton?(F) IMPLIES
     (FORALL (n:
                {n: nat |
                         n =
                          Card[atomo[T]]
                              (base_herbrand[T]
                                   (extend[clausula_horn[T],
                                           clausula_def[T],
                                           bool,
                                           FALSE]
                                    (P)))}):
        n =
         card(base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                (P)))
         IMPLIES
         (FORALL (k: nat):
            k = m_exponente(choose(F), P) AND
             member(choose(F), CL(P)) AND NOT k = 1
             IMPLIES
             (FORALL (C):
                member(C, P) AND cabeza(C) = choose(F) IMPLIES
                 k - 1 >= 0)));

medida_a_TCC4: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND singleton?(F) IMPLIES
     (FORALL (n:
                {n: nat |
                         n =
                          Card[atomo[T]]
                              (base_herbrand[T]
                                   (extend[clausula_horn[T],
```

```
                                          clausula_def[T],
                                          bool,
                                          FALSE]
                                    (P))))}):
        n =
         card(base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                  (P)))
         IMPLIES
         (FORALL (k: nat):
            k = m_exponente(choose(F), P) AND
             member(choose(F), CL(P)) AND NOT k = 1
             IMPLIES
             (FORALL (C):
                member(C, P) AND
                 cabeza(C) = choose(F) AND
                  subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))
                 IMPLIES
                 lex2(med_exp(cuerpo(C), P), card[atomo[T]](cuerpo(C))) <
                  lex2(med_exp(F, P), card[atomo[T]](F)))));

medida_a_TCC5: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T],
          v:
            [{z: [finite_set[atomo[T]], programa[T]] |
                      lex2(med_exp(z`1, z`2), card(z`1)) <
                       lex2(med_exp(F, P), card(F))} ->
               nat]):
    NOT empty?(F) AND singleton?(F) IMPLIES
     (FORALL (n:
                {n: nat |
                         n =
                          Card[atomo[T]]
                              (base_herbrand[T]
                                   (extend[clausula_horn[T],
                                          clausula_def[T],
                                          bool,
                                          FALSE]
                                    (P)))}):
        n =
         card(base_herbrand(extend[clausula_horn[T],
                                   clausula_def[T],
                                   bool,
                                   FALSE]
                                  (P)))
         IMPLIES
         (FORALL (k: nat):
            k = m_exponente(choose(F), P) AND
             member(choose(F), CL(P)) AND NOT k = 1
             IMPLIES
```

```
            is_finite[nat]
                ({j: nat |
                    EXISTS C:
                      member[clausula_def[T]](C, P) AND
                       cabeza(C) = choose[atomo[T]](F) AND
                        subset?[atomo[T]]
                            (cuerpo(C),
                             iterate[interpretacion[T]]
                                 (c_i[T](P), k - 1)(emptyset[atomo[T]]))
                         AND j = v(cuerpo(C), P)})
             AND
             NOT empty?[nat]
                     ({j: nat |
                         EXISTS C:
                           member[clausula_def[T]](C, P) AND
                            cabeza(C) = choose[atomo[T]](F) AND
                             subset?[atomo[T]]
                                 (cuerpo(C),
                                  iterate[interpretacion[T]]
                                  (c_i[T](P), k - 1)
                                  (emptyset[atomo[T]]))
                              AND j = v(cuerpo(C), P)}));

medida_a_TCC6: OBLIGATION
  FORALL (F: finite_set[atomo[T]]):
    NOT empty?(F) AND NOT singleton?(F) IMPLIES nonempty?[atomo[T]](F);

medida_a_TCC7: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND NOT singleton?(F) IMPLIES
     lex2(med_exp(singleton[atomo[T]](choose[atomo[T]](F)), P),
          card[atomo[T]](singleton[atomo[T]](choose[atomo[T]](F))))
      < lex2(med_exp(F, P), card[atomo[T]](F));

medida_a_TCC8: OBLIGATION
  FORALL (F: finite_set[atomo[T]], P: programa[T]):
    NOT empty?(F) AND NOT singleton?(F) IMPLIES
     lex2(med_exp(rest[atomo[T]](F), P),
          card[atomo[T]](rest[atomo[T]](F)))
      < lex2(med_exp(F, P), card[atomo[T]](F));

medida_a(F, P): RECURSIVE nat =
  IF empty?(F) THEN 0
  ELSIF singleton?(F)
    THEN LET n =
               card(base_herbrand(extend[clausula_horn[T],
                                         clausula_def[T],
                                         bool,
                                         FALSE]
                                  (P))),
             k = m_exponente(choose(F), P)
           IN
```

```
            IF NOT member(choose(F), CL(P)) THEN expt(n, n)
            ELSIF k = 1 THEN 1
            ELSE 1 +
                  min({j: nat |
                         EXISTS C:
                           member(C, P) AND
                            cabeza(C) = choose(F) AND
                             subset?(cuerpo(C),
                                     iterate(c_i(P), k - 1)(emptyset))
                              AND j = medida_a(cuerpo(C), P)})
            ENDIF
  ELSE medida_a(singleton[atomo[T]](choose(F)), P) + medida_a(rest(F), P)
  ENDIF
   MEASURE lex2(med_exp(F, P), card(F))

medida_a_vacio: LEMMA empty?(F) IMPLIES medida_a(F, P) = 0

medida_a_vacio_b: LEMMA medida_a(emptyset, P) = 0

medida_a_descomp: LEMMA
  subset?(F, CL(P)) AND member(A, F) IMPLIES
   medida_a(singleton(A), P) + medida_a(remove(A, F), P) = medida_a(F, P)

medida_a_unitario: LEMMA
  member(A, CL(P)) IMPLIES medida_a(singleton(A), P) >= 1

med(F, H): nat = card(F) + card(H)

medida_a_union: LEMMA
  subset?(F, CL(P)) AND subset?(H, CL(P)) IMPLIES
   reals.<=(medida_a(union(F, H), P), medida_a(F, P) + medida_a(H, P))

conj_clausulas_resolv_TCC1: OBLIGATION
  FORALL (A: atomo[T], P: programa[T], k: nat):
    k = m_exponente(A, P) AND k >= 1 IMPLIES
     (FORALL (C: clausula_def[T]):
        member(C, P) AND cabeza(C) = A IMPLIES k - 1 >= 0);

conj_clausulas_resolv_TCC2: OBLIGATION
  FORALL (A: atomo[T], P: programa[T]):
    is_finite[clausula_def[T]]
        (LET k = m_exponente(A, P) IN
           IF k >= 1
             THEN {C |
                     member[clausula_def[T]](C, P) AND
                      cabeza(C) = A AND
                       subset?[atomo[T]]
                           (cuerpo(C),
                            iterate[interpretacion[T]]
                                (c_i[T](P), k - 1)(emptyset[atomo[T]]))}
           ELSE emptyset[clausula_def[T]]
           ENDIF);
```

```
conj_clausulas_resolv(A, P): finite_set[clausula_def[T]] =
    LET k = m_exponente(A, P) IN
      IF k >= 1
        THEN {C |
                member(C, P) AND
                 cabeza(C) = A AND
                  subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))}
      ELSE emptyset
      ENDIF

conj_clausulas_resolv_cl: LEMMA
  member(A, CL(P)) IMPLIES nonempty?(conj_clausulas_resolv(A, P))

conj_clausulas_resolv_med: COROLLARY
  member(A, CL(P)) IMPLIES
   nonempty?(image((LAMBDA (C): medida_a(cuerpo(C), P)),
                   conj_clausulas_resolv(A, P)))

medida_a_unitario_cl_TCC1: OBLIGATION
  FORALL (A: atomo[T], P: programa[T], k: nat):
    m_exponente(A, P) = k AND member(A, CL(P)) IMPLIES
     (FORALL (C):
        member(C, P) AND
         cabeza(C) = A AND
          subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))
         IMPLIES
         is_finite[nat]
             ({j: nat |
                 EXISTS C:
                   member[clausula_def[T]](C, P) AND
                    cabeza(C) = A AND
                     subset?[atomo[T]]
                         (cuerpo(C),
                          iterate[interpretacion[T]]
                              (c_i[T](P), k - 1)(emptyset[atomo[T]]))
                      AND j = medida_a(cuerpo(C), P)})
          AND
          NOT empty?[nat]
                  ({j: nat |
                      EXISTS C:
                        member[clausula_def[T]](C, P) AND
                         cabeza(C) = A AND
                          subset?[atomo[T]]
                              (cuerpo(C),
                               iterate[interpretacion[T]]
                                   (c_i[T](P), k - 1)
                                   (emptyset[atomo[T]]))
                           AND j = medida_a(cuerpo(C), P)}));

medida_a_unitario_cl: LEMMA
  member(A, CL(P)) AND m_exponente(A, P) = k IMPLIES
```

```
    (EXISTS C:
       member(C, P) AND
        cabeza(C) = A AND
         subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset)) AND
          medida_a(cuerpo(C), P) =
           min({j: nat |
                  EXISTS C:
                    member(C, P) AND
                     cabeza(C) = A AND
                      subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset))
                       AND j = medida_a(cuerpo(C), P)}))

  medida_a_unitario_2_TCC1: OBLIGATION
    FORALL (A: atomo[T], P: programa[T], k: nat):
      k > 1 AND m_exponente(A, P) = k AND member(A, CL(P)) IMPLIES
       is_finite[nat]
           ({j: nat |
               EXISTS C:
                 member[clausula_def[T]](C, P) AND
                  cabeza(C) = A AND
                   subset?[atomo[T]]
                       (cuerpo(C),
                        iterate[interpretacion[T]]
                            (c_i[T](P), k - 1)(emptyset[atomo[T]]))
                    AND j = medida_a(cuerpo(C), P)})
       AND
       NOT empty?[nat]
               ({j: nat |
                   EXISTS C:
                     member[clausula_def[T]](C, P) AND
                      cabeza(C) = A AND
                       subset?[atomo[T]]
                           (cuerpo(C),
                            iterate[interpretacion[T]]
                                (c_i[T](P), k - 1)(emptyset[atomo[T]]))
                        AND j = medida_a(cuerpo(C), P)});

  medida_a_unitario_2: LEMMA
    member(A, CL(P)) AND m_exponente(A, P) = k AND k > 1 IMPLIES
     medida_a(singleton(A), P) =
      1 +
       min({j: nat |
              EXISTS C:
                member(C, P) AND
                 cabeza(C) = A AND
                  subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset)) AND
                   j = medida_a(cuerpo(C), P)})

  medida_a_unitario_cl_mayor: COROLLARY
    member(A, CL(P)) AND m_exponente(A, P) = k AND k > 1 IMPLIES
     (EXISTS C:
        member(C, P) AND
```

```
        cabeza(C) = A AND
         subset?(cuerpo(C), iterate(c_i(P), k - 1)(emptyset)) AND
          medida_a(singleton(A), P) > medida_a(cuerpo(C), P))

  medida_g(G, P): nat = medida_a(cuerpo(G), P)

  insatisfacible_contenido_BH: LEMMA
    es_insatisfacible(add[clausula_horn[T]]
                          (G,
                           extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                              (P)))
     IMPLIES
     subset?(cuerpo(G),
             base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                              (P)))

  insatisfacible_contenido_CL: LEMMA
    es_insatisfacible(add[clausula_horn[T]]
                          (G,
                           extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                              (P)))
     IMPLIES subset?(cuerpo(G), CL(P))

  potencia_c_i_contenido_CL: LEMMA
    subset?(F, iterate(c_i(P), k)(emptyset)) IMPLIES subset?(F, CL(P))

  insatisfacible_medida_g_TCC1: OBLIGATION
    FORALL (G: objetivo_def[T], P: programa[T]):
      es_insatisfacible(add[clausula_horn[T]]
                            (G,
                             extend[clausula_horn[T],
                                    clausula_def[T],
                                    bool,
                                    FALSE]
                                (P)))
       IMPLIES
       (FORALL (A):
          member(A, cuerpo(G)) IMPLIES
           (FORALL (C):
              member(C, P) AND cabeza(C) = A IMPLIES
               tiene_resolvente[T](G, C)));

  insatisfacible_medida_g: THEOREM
    es_insatisfacible(add[clausula_horn[T]]
                          (G,
                           extend[clausula_horn[T], clausula_def[T], bool,
                                  FALSE]
                              (P)))
     IMPLIES
```

```
    (FORALL A:
       member(A, cuerpo(G)) IMPLIES
        (EXISTS C:
           member(C, P) AND
            cabeza(C) = A AND
             medida_g(resolvente(G, C), P) < medida_g(G, P) AND
              es_insatisfacible(add[clausula_horn[T]]
                                    (resolvente(G, C),
                                     extend[clausula_horn[T],
                                            clausula_def[T],
                                            bool,
                                            FALSE]
                                     (P)))))

 tiene_refutacion_medida_g_TCC1: OBLIGATION
   FORALL (G: objetivo_def[T], P: programa[T]):
     tiene_refutacion(P, G) IMPLIES
      (FORALL (A):
         member(A, cuerpo(G)) IMPLIES
          (FORALL (C):
             member(C, P) AND cabeza(C) = A IMPLIES
              tiene_resolvente[T](G, C)));

 tiene_refutacion_medida_g: THEOREM
   tiene_refutacion(P, G) IMPLIES
    (FORALL A:
       member(A, cuerpo(G)) IMPLIES
        (EXISTS C:
           member(C, P) AND
            cabeza(C) = A AND
             medida_g(resolvente(G, C), P) < medida_g(G, P) AND
              tiene_refutacion(P, resolvente(G, C))))

 sld_independiente_regla_comp: THEOREM
   tiene_refutacion(P, G) IMPLIES tiene_refutacion_via(P, G, f)

 sld_completitud_via: COROLLARY
   es_insatisfacible(add[clausula_horn[T]]
                         (G,
                          extend[clausula_horn[T], clausula_def[T], bool,
                                 FALSE]
                              (P)))
    IMPLIES tiene_refutacion_via(P, G, f)
END sld_resolucion_via
```

D.2. Formalización evaluable, usando listas

D.2.1. Cláusulas

```
clausulas_1[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  TL: LIBRARY = "../auxiliares/listas"

  IMPORTING TL@listas_prop_1

  x: VAR T

  clausula: TYPE = [list[T], list[T]]

  cabeza(C: clausula): list[T] = PROJ_1(C)

  cuerpo(C: clausula): list[T] = PROJ_2(C)

  es_cl_horn(cl: clausula): bool = length(cabeza(cl)) <= 1

  cl_horn: TYPE = (es_cl_horn)

  JUDGEMENT cl_horn SUBTYPE_OF clausula

  es_cl_definida(cl: cl_horn): bool = length(cabeza(cl)) = 1

  cl_definida: TYPE = (es_cl_definida)

  C, C1, C2: VAR cl_definida

  JUDGEMENT cl_definida SUBTYPE_OF cl_horn

  JUDGEMENT cl_definida SUBTYPE_OF clausula

  cabeza_TCC1: OBLIGATION FORALL (C): cons?[T](cabeza(C));

  JUDGEMENT cabeza(C) HAS_TYPE (cons?[T])

  cns_pertenece_cabeza: LEMMA member(x, cabeza(C)) IFF x = car(cabeza(C))

  t: T

  existe_cl_definida_TCC1: OBLIGATION es_cl_horn((: t :), null[T]);

  existe_cl_definida: LEMMA es_cl_definida((: t :), null)

  es_regla(C): bool = cons?(cuerpo(C))
```

```
  es_hecho(C): bool = null?(cuerpo(C))

  es_objetivo_definido(cl: cl_horn): bool = null?(cabeza(cl))

  objetivo_definido: TYPE = (es_objetivo_definido)

  G, G1, G2: VAR objetivo_definido

  JUDGEMENT objetivo_definido SUBTYPE_OF cl_horn

  JUDGEMENT objetivo_definido SUBTYPE_OF clausula

  es_vacio(G): bool = null?(cuerpo(G))

  es_objetivo_propio(G): bool = cons?(cuerpo(G))

  cl_horn_tipo: LEMMA
    FORALL (cl: cl_horn): es_cl_definida(cl) OR es_objetivo_definido(cl)

  programa: TYPE = list[cl_definida]

  JUDGEMENT programa SUBTYPE_OF list[cl_horn]

  P, P1, P2: VAR programa

  programa_no_vacio: TYPE = {P: programa | cons?(P)}

  resolvente_TCC1: OBLIGATION
    FORALL (C: cl_definida, (G: (es_objetivo_propio))): cons?[T](cuerpo(G));

  resolvente_TCC2: OBLIGATION
    FORALL (C: cl_definida, (G: (es_objetivo_propio))):
      es_cl_horn(null[T], append[T](cuerpo(C), cdr[T](cuerpo(G)))) AND
       es_objetivo_definido(null[T],
                            append[T](cuerpo(C), cdr[T](cuerpo(G))));

  resolvente((G: (es_objetivo_propio)), C): objetivo_definido =
      (null, append(cuerpo(C), cdr(cuerpo(G))))
 END clausulas_l
```

D.2.2. Relación formal entra las dos representaciones

```
relacion_rep_clausulas[D, T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;

   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING
```

```
IMPORTING clausulas_1[D] AS T1

IMPORTING clausulas[T] AS T2

REFINAMIENTO: LIBRARY = "../refinamiento"

IMPORTING REFINAMIENTO@refinamiento_conj_finitos

IMPORTING TL@operaciones_listas[D]

IMPORTING TL@pertenece_tipo

IMPORTING TL@listas_prop_1

transf_atomo: [D -> atomo[T]]

transf_atomo_ax1: AXIOM surjective?(transf_atomo)

transf_atomo_ax2: AXIOM injective?(transf_atomo)

transf_atomo_TCC1: OBLIGATION es_refinamiento?[atomo[T], D](transf_atomo);

JUDGEMENT transf_atomo HAS_TYPE (es_refinamiento?[atomo[T], D])

transf_clausula_def: [cl_definida[D] -> clausula_def[T]]

transf_clausula_def_ax1: AXIOM surjective?(transf_clausula_def)

transf_clausula_def_TCC1: OBLIGATION
  es_refinamiento?[clausula_def[T], cl_definida[D]](transf_clausula_def);

JUDGEMENT transf_clausula_def HAS_TYPE
    (es_refinamiento?[clausula_def[T], cl_definida[D]])

transf_cabeza_clausula_def: AXIOM
  FORALL (C: cl_definida[D]):
    transf_atomo(car(cabeza(C))) = cabeza(transf_clausula_def(C))

transf_objetivo_def: [objetivo_definido[D] -> objetivo_def[T]]

transf_objetivo_def_ax1: AXIOM surjective?(transf_objetivo_def)

transf_objetivo_def_TCC1: OBLIGATION
  es_refinamiento?[objetivo_def[T], objetivo_definido[D]]
      (transf_objetivo_def);

JUDGEMENT transf_objetivo_def HAS_TYPE
    (es_refinamiento?[objetivo_def[T], objetivo_definido[D]])

transf_clausula_horn: [cl_horn[D] -> clausula_horn[T]]

transf_clausula_horn_ax1: AXIOM surjective?(transf_clausula_horn)
```

```
transf_clausula_horn_TCC1: OBLIGATION
  es_refinamiento?[clausula_horn[T], cl_horn[D]](transf_clausula_horn);

JUDGEMENT transf_clausula_horn HAS_TYPE
    (es_refinamiento?[clausula_horn[T], cl_horn[D]])

transf_clausula_horn_ax2: AXIOM
  FORALL (C: cl_definida[D]):
    transf_clausula_horn(C) = transf_clausula_def(C)

transf_clausula_horn_ax3: AXIOM
  FORALL (C: objetivo_definido[D]):
    transf_clausula_horn(C) = transf_objetivo_def(C)

transf_clausula_horn_def_ax: AXIOM
  FORALL (C: cl_definida[D]): es_clausula_def(transf_clausula_horn(C))

transf_clausula_horn_TCC2: OBLIGATION
  FORALL (C: cl_definida[D]): es_clausula_def[T](transf_clausula_horn(C));

JUDGEMENT transf_clausula_horn(C: cl_definida[D]) HAS_TYPE clausula_def[T]

transf_clausula_horn_obj_ax: AXIOM
  FORALL (C: objetivo_definido[D]):
    es_objetivo_def(transf_clausula_horn(C))

transf_clausula_horn_TCC3: OBLIGATION
  FORALL (C: objetivo_definido[D]):
    es_objetivo_def[T](transf_clausula_horn(C));

JUDGEMENT transf_clausula_horn(C: objetivo_definido[D]) HAS_TYPE
    objetivo_def[T]

transf_clausula_horn_obj_vacio_ax: AXIOM
  FORALL (G: (es_vacio[D])): vacia?(transf_clausula_horn(G))

transf_clausula_horn_TCC4: OBLIGATION
  FORALL (G: (es_vacio[D])): vacia?[T](transf_clausula_horn(G));

JUDGEMENT transf_clausula_horn(G: (es_vacio[D])) HAS_TYPE (vacia?[T])

transf_cabeza_cl_def: AXIOM
  FORALL (C: cl_definida[D]):
    transf_atomo(car(cabeza(C))) = cabeza(transf_clausula_horn(C))

transf_cuerpo: AXIOM
  FORALL (CH: cl_horn[D], x: D):
    member(x, cuerpo(CH)) IMPLIES
     member(transf_atomo(x), cuerpo(transf_clausula_horn(CH)))

transf_cuerpo_2: AXIOM
```

```
  FORALL (CH: cl_horn[D], x: D):
    member(transf_atomo(x), cuerpo(transf_clausula_horn(CH))) IMPLIES
     member(x, cuerpo(CH))

transf_resolvente_cuerpo: AXIOM
  FORALL ((G: (es_objetivo_propio[D])), (C: cl_definida[D])):
    subset?[atomo[T]]
        (cuerpo(transf_clausula_horn(C)),
         cuerpo(transf_clausula_horn(resolvente(G, C))))

transf_resolvente_cuerpo_obj_TCC1: OBLIGATION
  FORALL (A: atomo[T], (C: cl_definida[D]),
          (G: (es_objetivo_propio[D]))):
    member[atomo[T]](A, cuerpo(transf_clausula_horn(G))) AND
     A /= cabeza(transf_clausula_horn(C))
     IMPLIES cons?[D](cuerpo[D](G));

transf_resolvente_cuerpo_obj: AXIOM
  FORALL ((G: (es_objetivo_propio[D])), (C: cl_definida[D]),
          A: atomo[T]):
    member[atomo[T]](A, cuerpo(transf_clausula_horn(G))) AND
     A /= cabeza(transf_clausula_horn(C)) AND
      car(cabeza(C)) = car(cuerpo(G))
     IMPLIES
     member[atomo[T]](A, cuerpo(transf_clausula_horn(resolvente(G, C))))

transf_programa_TCC1: OBLIGATION EXISTS (x: cl_definida[D]): TRUE;

transf_programa: [clausulas_l[D].programa -> clausulas[T].programa] =
    c(transf_clausula_def)

transf_programa_ax1: LEMMA surjective?(transf_programa)

transf_programa_TCC2: OBLIGATION
  es_refinamiento?[clausulas[T].programa, clausulas_l[D].programa]
      (transf_programa);

JUDGEMENT transf_programa HAS_TYPE
    (es_refinamiento?[clausulas[T].programa, clausulas_l[D].programa])

transf_lista_cl_horn:
[list[cl_horn[D]] -> finite_set[clausula_horn[T]]] =
    c(transf_clausula_horn)

transf_lista_cl_horn_ax1: LEMMA surjective?(transf_lista_cl_horn)

transf_lista_cl_horn_TCC1: OBLIGATION
  es_refinamiento?[finite_set[clausula_horn[T]], list[cl_horn[D]]]
      (transf_lista_cl_horn);

JUDGEMENT transf_lista_cl_horn HAS_TYPE
    (es_refinamiento?[finite_set[clausula_horn[T]], list[cl_horn[D]]])
```

```
transf_lista_cl_horn_ax2: LEMMA
  FORALL (CH: cl_horn[D], LCH: list[cl_horn[D]]):
    transf_lista_cl_horn(cons(CH, LCH)) =
     add(transf_clausula_horn(CH), transf_lista_cl_horn(LCH))

transf_programa_ax4_b: LEMMA
  FORALL (LCH: list[cl_horn[D]], CH: cl_horn[D]):
    member(CH, LCH) IMPLIES
     member(transf_clausula_horn(CH), transf_lista_cl_horn(LCH))

transf_programa_ax5: LEMMA
  FORALL (P: clausulas_l[D].programa, C: clausula_def[T]):
    member(C, transf_lista_cl_horn(P)) IMPLIES
     (EXISTS (C1: cl_horn[D]):
        es_cl_definida[D](C1) AND
         member(C1, P) AND transf_clausula_horn(C1) = C)

transf_lista_cl_horn_ax6: LEMMA
  FORALL (CH: clausula_horn[T], LCH: list[cl_horn[D]]):
    member(CH, transf_lista_cl_horn(LCH)) IMPLIES
     (EXISTS (C: cl_horn[D]):
        member(C, LCH) AND CH = transf_clausula_horn(C))

transf_programa_ax7: LEMMA
  FORALL (P: clausulas_l[D].programa, CH: clausula_horn[T]):
    member(CH, transf_lista_cl_horn(P)) IMPLIES es_clausula_def[T](CH)

transf_programa_ax8: LEMMA
  FORALL (P: T1.programa):
    transf_lista_cl_horn(P) =
     extend[clausula_horn[T], clausula_def[T], bool, FALSE]
         (transf_programa(P))

transf_interp: [list[D] -> interpretacion_finita[T]] = c(transf_atomo)

transf_interp_ax1: LEMMA
  FORALL (I: list[D], x: D):
    member(x, I) IMPLIES member(transf_atomo(x), transf_interp(I))

transf_interp_ax2: LEMMA
  FORALL (I: list[D], x: D):
    member(transf_atomo(x), transf_interp(I)) IMPLIES member(x, I)

transf_interp_ax3: AXIOM
  FORALL (I: list[D], C: cl_definida[D]):
    sublista?(cuerpo(C), I) IFF
     subset?(cuerpo(transf_clausula_horn(C)), transf_interp(I))

transf_interp_ax4: LEMMA
  FORALL (I, J: list[D]):
    sublista?(I, J) IFF subset?(transf_interp(I), transf_interp(J))
```

```
  transf_interp_ax5: LEMMA
    FORALL (I, J: list[D]):
      igual_l?(I, J) IFF transf_interp(I) = transf_interp(J)

  transf_interp_ax6: LEMMA surjective?(transf_interp)

  transf_interp_TCC1: OBLIGATION
    es_refinamiento?[interpretacion_finita[T], list[D]](transf_interp);

  JUDGEMENT transf_interp HAS_TYPE
      (es_refinamiento?[interpretacion_finita[T], list[D]])
 END relacion_rep_clausulas
```

D.2.3. Interpretación de la relación entre las dos representaciones

```
relacion_rep_clausulas_interpretacion[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  x_0: T

  IMPORTING clausulas[T] AS T2

  IMPORTING clausulas_l[T] AS T1

  REFINAMIENTO: LIBRARY = "../refinamiento"

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos[atomo[T], T] AS re_atomo

  re_cl_TCC1: OBLIGATION EXISTS (x: cl_horn[T]): TRUE;

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos[clausula_horn[T],
                                                   cl_horn[T]] AS re_cl

  re_cld_TCC1: OBLIGATION EXISTS (x: cl_definida[T]): TRUE;

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos[clausula_def[T],
                                                   cl_definida[T]] AS re_cld

  IMPORTING REFINAMIENTO@propiedades_operaciones_listas_via_ref

  IMPORTING TL@propiedades_operaciones_listas

  transf_a_1(x: T): atomo[T] = a(x)

  transf_a_1_TCC1: OBLIGATION es_refinamiento?[atomo[T], T](transf_a_1);
```

```
JUDGEMENT transf_a_1 HAS_TYPE (es_refinamiento?[atomo[T], T])

transf_clausula_def_1_TCC1: OBLIGATION
  FORALL (C: cl_definida[T]):
    es_clausula_def[T]
        ((# cabeza := transf_a_1(car[T](cabeza[T](C))),
            cuerpo
              := re_atomo[atomo[T], T].c(transf_a_1)(cuerpo[T](C)) #));

transf_clausula_def_1(C: cl_definida[T]): clausula_def[T] =
    (# cabeza := transf_a_1(car(cabeza(C))),
       cuerpo := re_atomo.c(transf_a_1)(cuerpo(C)) #)

transf_objetivo_def_1_TCC1: OBLIGATION
  FORALL (C: objetivo_definido[T]):
    es_objetivo_def[T]
        ((# cabeza := falso[T],
            cuerpo
              := re_atomo[atomo[T], T].c(transf_a_1)(cuerpo[T](C)) #));

transf_objetivo_def_1(C: objetivo_definido[T]): objetivo_def[T] =
    (# cabeza := falso, cuerpo := re_atomo.c(transf_a_1)(cuerpo(C)) #)

transf_cl_h_1_TCC1: OBLIGATION
  FORALL (C: cl_horn[T]): cons?(PROJ_1(C)) IMPLIES cons?[T](cabeza[T](C));

transf_cl_h_1(C: cl_horn[T]): clausula_horn[T] =
    (# cabeza
         := IF cons?(PROJ_1(C)) THEN transf_a_1(car(cabeza(C)))
            ELSE falso
            ENDIF,
       cuerpo := re_atomo.c(transf_a_1)(cuerpo(C)) #)

transf_cl_h_1_TCC2: OBLIGATION
  es_refinamiento?[clausula_horn[T], cl_horn[T]](transf_cl_h_1);

JUDGEMENT transf_cl_h_1 HAS_TYPE
    (es_refinamiento?[clausula_horn[T], cl_horn[T]])

transf_cl_h_1_cl_definida: LEMMA
  FORALL (C: cl_horn[T]):
    es_clausula_def(transf_cl_h_1(C)) IMPLIES es_cl_definida(C)

transf_cuerpo_cl_definida: LEMMA
  FORALL (C: cl_horn[T]):
    re_atomo.c(transf_a_1)(cuerpo(C)) = cuerpo(transf_cl_h_1(C))

IMP_relacion_rep_clausulas_transf_atomo_ax1_TCC1: OBLIGATION
  surjective?[T, atomo[T]](transf_a_1);

IMP_relacion_rep_clausulas_transf_atomo_ax2_TCC1: OBLIGATION
```

```
  injective?[T, atomo[T]](transf_a_1);

IMP_relacion_rep_clausulas_transf_clausula_def_ax1_TCC1: OBLIGATION
  surjective?[cl_definida[T], clausula_def[T]](transf_clausula_def_1);

IMP_relacion_rep_clausulas_transf_cabeza_clausula_def_TCC1: OBLIGATION
  FORALL (C: cl_definida[T]):
    transf_a_1(car[T](cabeza[T](C))) = cabeza(transf_clausula_def_1(C));

IMP_relacion_rep_clausulas_transf_objetivo_def_ax1_TCC1: OBLIGATION
  surjective?[objetivo_definido[T], objetivo_def[T]]
      (transf_objetivo_def_1);

IMP_relacion_rep_clausulas_transf_clausula_horn_ax1_TCC1: OBLIGATION
  surjective?[cl_horn[T], clausula_horn[T]](transf_cl_h_1);

IMP_relacion_rep_clausulas_transf_clausula_horn_ax2_TCC1: OBLIGATION
  FORALL (C: cl_definida[T]): transf_cl_h_1(C) = transf_clausula_def_1(C);

IMP_relacion_rep_clausulas_transf_clausula_horn_ax3_TCC1: OBLIGATION
  FORALL (C: objetivo_definido[T]):
    transf_cl_h_1(C) = transf_objetivo_def_1(C);

IMP_relacion_rep_clausulas_transf_clausula_horn_def_ax_TCC1: OBLIGATION
  FORALL (C: cl_definida[T]): es_clausula_def[T](transf_cl_h_1(C));

IMP_relacion_rep_clausulas_transf_clausula_horn_obj_ax_TCC1: OBLIGATION
  FORALL (C: objetivo_definido[T]): es_objetivo_def[T](transf_cl_h_1(C));

IMP_relacion_rep_clausulas_transf_clausula_horn_obj_vacio_ax_TCC1:
OBLIGATION FORALL (G: (es_vacio[T])): vacia?[T](transf_cl_h_1(G));

IMP_relacion_rep_clausulas_transf_cabeza_cl_def_TCC1: OBLIGATION
  FORALL (C: cl_definida[T]):
    transf_a_1(car[T](cabeza[T](C))) = cabeza(transf_cl_h_1(C));

IMP_relacion_rep_clausulas_transf_cuerpo_TCC1: OBLIGATION
  FORALL (CH: cl_horn[T], x: T):
    member[T](x, cuerpo[T](CH)) IMPLIES
     member[atomo[T]](transf_a_1(x), cuerpo(transf_cl_h_1(CH)));

IMP_relacion_rep_clausulas_transf_cuerpo_2_TCC1: OBLIGATION
  FORALL (CH: cl_horn[T], x: T):
    member[atomo[T]](transf_a_1(x), cuerpo(transf_cl_h_1(CH))) IMPLIES
     member[T](x, cuerpo[T](CH));

IMP_relacion_rep_clausulas_transf_resolvente_cuerpo_TCC1: OBLIGATION
  FORALL ((G: (es_objetivo_propio[T])), (C: cl_definida[T])):
    subset?[atomo[T]]
        (cuerpo(transf_cl_h_1(C)),
         cuerpo(transf_cl_h_1(resolvente[T](G, C))));
```

```
  IMP_relacion_rep_clausulas_transf_resolvente_cuerpo_obj_TCC1: OBLIGATION
    FORALL ((G: (es_objetivo_propio[T])), (C: cl_definida[T]),
            A: atomo[T]):
      member[atomo[T]](A, cuerpo(transf_cl_h_1(G))) AND
       A /= cabeza(transf_cl_h_1(C)) AND
        car[T](cabeza[T](C)) = car[T](cuerpo[T](G))
       IMPLIES
       member[atomo[T]](A, cuerpo(transf_cl_h_1(resolvente[T](G, C))));

  IMP_relacion_rep_clausulas_transf_interp_ax3_TCC1: OBLIGATION
    FORALL (I: list[T], C: cl_definida[T]):
      sublista?[T](cuerpo[T](C), I) IFF
       subset?[atomo[T]](cuerpo(transf_cl_h_1(C)), transf_interp[T, T](I));

  IMPORTING relacion_rep_clausulas[T, T]
                                  {{ transf_atomo := transf_a_1,
                                     transf_clausula_def
                                     := transf_clausula_def_1,
                                     transf_objetivo_def
                                     := transf_objetivo_def_1,
                                     transf_clausula_horn
                                     := transf_cl_h_1 }}
 END relacion_rep_clausulas_interpretacion
```

D.2.4. Semántica de cláusulas

```
semantica_clausulas_l[D, T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;

   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING relacion_rep_clausulas[D, T]

  IMPORTING semantica_clausulas[T]

  x, x1, x2: VAR D

  CH: VAR cl_horn[D]

  C: VAR cl_definida[D]

  P, P1, P2: VAR clausulas_l[D].programa

  I, I1, I2: VAR interpretacion[T]

  Int, Int1: VAR set[D]
```

```
L, L1, L2: VAR list[D]

FSC: VAR finite_set[cl_horn[D]]

LSC: VAR list[cl_horn[D]]

G: VAR objetivo_definido[D]

es_modelo(Int, CH): bool =
    es_modelo(image(transf_atomo)(Int), transf_clausula_horn(CH))

imagen_imagen_inversa: CLAIM
  image(transf_atomo)(image(inverse(transf_atomo))(I)) = I

es_modelo_l_caract: LEMMA
  es_modelo(I, transf_clausula_horn(CH)) IFF
   es_modelo(image(inverse(transf_atomo))(I), CH)

es_modelo(Int, LSC): bool =
    es_modelo(image(transf_atomo)(Int),
              restrict[clausula_horn[T], clausula_def[T], boolean]
                  (transf_lista_cl_horn(LSC)))

es_modelo(Int, P): bool =
    es_modelo(image(transf_atomo)(Int), transf_programa(P))

es_modelo_l_caract_p: LEMMA
  es_modelo(I,
            restrict[clausula_horn[T], clausula_def[T], boolean]
                (transf_lista_cl_horn(LSC)))
   IFF es_modelo(image(inverse(transf_atomo))(I), LSC)

modelo_p_es_modelo_cl: LEMMA
  es_modelo(Int, P) AND member(C, P) IMPLIES es_modelo(Int, C)

es_modelo(L, CH): bool =
    es_modelo(transf_interp(L), transf_clausula_horn(CH))

es_modelo_e_TCC1: OBLIGATION
  FORALL (CH: cl_horn[D]):
    NOT null?(cabeza(CH)) IMPLIES cons?[D](cabeza[D](CH));

es_modelo_e(L, CH): bool =
    IF null?(cabeza(CH))
      THEN some(LAMBDA (x): NOT member(x, L))(cuerpo(CH))
    ELSE member(car(cabeza(CH)), L) OR
          some(LAMBDA (x): NOT member(x, L))(cuerpo(CH))
    ENDIF

es_modelo_e(L, G): bool = some(LAMBDA (x): NOT member(x, L))(cuerpo(G))

es_modelo_e(L, C): bool =
```

```
    member(car(cabeza(C)), L) OR
     some(LAMBDA (x): NOT member(x, L))(cuerpo(C))

es_modelo_e_c_def_igual?: LEMMA
  igual_l?[D](L1, L2) IMPLIES (es_modelo_e(L1, C) IFF es_modelo_e(L2, C))

es_modelo(L, P): bool = es_modelo(transf_interp(L), transf_programa(P))

es_modelo_e(L, P): bool = every(LAMBDA (C): es_modelo_e(L, C))(P)

es_modelo_e_programa_igual?: LEMMA
  igual_l?[D](L1, L2) IMPLIES (es_modelo_e(L1, P) IFF es_modelo_e(L2, P))

es_cons_logica(x, LSC): bool =
    es_cons_logica(transf_atomo(x),
                   restrict[clausula_horn[T], clausula_def[T], boolean]
                       (transf_lista_cl_horn(LSC)))

es_cons_logica(x, P): bool =
    es_cons_logica(transf_atomo(x), transf_programa(P))

es_insatisfacible(LSC): bool =
    es_insatisfacible(transf_lista_cl_horn(LSC))

par_de: TYPE = [objetivo_definido, cl_definida]

de: VAR list[par_de]

es_refutacion_TCC1: OBLIGATION
  FORALL (de2: list[par_de], par: par_de, G: objetivo_definido[D],
          P: clausulas_l[D].programa, de: list[par_de]):
    de = cons(par, de2) AND es_objetivo_propio(G) IMPLIES
     (FORALL (C1: cl_definida[D], G1: objetivo_definido[D]):
        G1 = par`1 AND C1 = par`2 AND G1 = G AND member(C1, P) IMPLIES
         cons?[D](cuerpo[D](G)));

es_refutacion_TCC2: OBLIGATION
  FORALL (de2: list[par_de], par: par_de, G: objetivo_definido[D],
          P: clausulas_l[D].programa, de: list[par_de]):
    de = cons(par, de2) AND es_objetivo_propio(G) IMPLIES
     (FORALL (C1: cl_definida[D], G1: objetivo_definido[D]):
              G1 = par`1 AND C1 = par`2 AND G1 = G AND member(C1, P)
         AND car(cabeza(C1)) = car(cuerpo(G))
         IMPLIES length[par_de](de2) < length[par_de](de));

es_refutacion(de, P, G): RECURSIVE bool =
  CASES de
    OF null: es_vacio(G),
       cons(par, de2):
         es_objetivo_propio(G) AND
          LET (G1, C1) = par IN
            G1 = G AND
```

```
              member(C1, P) AND
               car(cabeza(C1)) = car(cuerpo(G)) AND
                es_refutacion(de2, P, resolvente(G, C1))
     ENDCASES
    MEASURE length(de)
 END semantica_clausulas_l
```

D.2.5. Cálculo de la base de Herbrand

```
base_herbrand_l[T, D: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;
  ENDASSUMING

  IMPORTING semantica_declarativa[T]

  IMPORTING semantica_clausulas_l[D, T]

  REFINAMIENTO: LIBRARY = "../refinamiento"

  IMPORTING REFINAMIENTO@refinamiento_oper

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos_gen[atomo[T], D]

  IMP_refinamiento_conj_finitos_image_TCC1: OBLIGATION
    EXISTS (x: cl_horn[D]): TRUE;

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos_image[clausula_horn[T],
                                                         finite_set[atomo[T]],
                                                         cl_horn[D],
                                                         list[D]]

  x: VAR D

  A: VAR atomo[T]

  CH, CH1, CH2: VAR cl_horn

  C: VAR cl_definida

  LCH: VAR list[cl_horn]

  P, P1, P2: VAR clausulas_l[D].programa

  PC: VAR clausulas[T].programa

  SCH: VAR set[clausula_horn]
```

```
simb_prop_l(CH): list[D] = append(cabeza(CH), cuerpo(CH))

simb_prop_l_correcto: THEOREM
  member(x, simb_prop_l(CH)) IFF
   member(transf_atomo(x), simb_prop(transf_clausula_horn(CH)))

simb_prop_l_refinamiento: COROLLARY
  es_refinamiento_op?[clausula_horn[T], finite_set[atomo[T]], cl_horn[D],
                      list[D], transf_clausula_horn, c(transf_atomo)]
      (simb_prop, simb_prop_l)

base_herbrand_l(LCH): list[D] = Append(map(simb_prop_l)(LCH))

base_herbrand_l_sd(LCH): list[D] =
    elimina_duplicados(base_herbrand_l(LCH))

cns_base_herbrand_l: LEMMA
  member(x, base_herbrand_l(LCH)) IFF
   (EXISTS CH: member(CH, LCH) AND member(x, simb_prop_l(CH)))

cns_base_herbrand: LEMMA
  member(A, base_herbrand(SCH)) IFF
   (EXISTS (CH: clausula_horn[T]):
      member(CH, SCH) AND member(A, simb_prop(CH)))

base_herbrand_l_correcto_g: THEOREM
  member(x, base_herbrand_l(LCH)) IFF
   member(transf_atomo(x), base_herbrand(transf_lista_cl_horn(LCH)))

base_herbrand_l_correcto: COROLLARY
  member(x, base_herbrand_l(P)) IFF
   member(transf_atomo(x), base_herbrand(transf_lista_cl_horn(P)))

cn_pertenencia_transf_interp: CLAIM
  FORALL (ls: list[D]):
    member(A, transf_interp(ls)) IMPLIES
     (EXISTS x: member(x, ls) AND A = transf_atomo(x))

base_herbrand_l_correcto_2: THEOREM
  transf_interp(base_herbrand_l(LCH)) =
   base_herbrand(transf_lista_cl_horn(LCH))

base_herbrand_l_correcto_2_p: COROLLARY
  transf_interp(base_herbrand_l(P)) =
   base_herbrand(transf_lista_cl_horn(P))

base_herbrand_l_es_refinamiento_TCC1: OBLIGATION
  FORALL (x1: finite_set[clausula_horn[T]]):
    is_finite[atomo[T]](base_herbrand[T](x1));

base_herbrand_l_es_refinamiento: LEMMA
```

```
    es_refinamiento_op?[finite_set[clausula_horn[T]],
                        interpretacion_finita[T], list[cl_horn[D]],
                        list[D], transf_lista_cl_horn, transf_interp]
        (restrict[set[clausula_horn[T]], finite_set[clausula_horn[T]],
                  set[atomo[T]]]
            (base_herbrand),
         base_herbrand_l)

  base_herbrand_p(P: clausulas[T].programa): interpretacion_finita[T] =
      base_herbrand(extend[clausula_horn[T], clausula_def[T], bool, FALSE]
                        (P))

  base_herbrand_l_p(P: clausulas_l[D].programa): list[D] =
      base_herbrand_l(P)

  base_herbrand_l_es_refinamiento_p: COROLLARY
    es_refinamiento_op?[clausulas[T].programa, interpretacion_finita[T],
                        clausulas_l[D].programa, list[D], transf_programa,
                        transf_interp]
       (base_herbrand_p, base_herbrand_l_p)

  base_herbrand_l_es_modelo: COROLLARY es_modelo(base_herbrand_l(P), P)

  cardinal_base_herbrand_l: COROLLARY
    cardinal_l(base_herbrand_l(LCH)) =
     card(base_herbrand(transf_lista_cl_horn(LCH)))

  cardinal_base_herbrand_l_p: COROLLARY
    cardinal_l(base_herbrand_l(P)) =
     card(base_herbrand(extend[clausula_horn[T], clausula_def[T], bool,
                               FALSE]
                          (transf_programa(P))))
 END base_herbrand_l
```

D.2.6. Cálculo del menor modelo de Herbrand

```
menor_modelo_herbrand_l[T, D: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;
  ENDASSUMING

  IMPORTING semantica_declarativa[T]

  IMPORTING semantica_clausulas_l[D, T]

  IMPORTING base_herbrand_l[T, D]
```

```
REFINAMIENTO: LIBRARY = "../refinamiento"

IMPORTING REFINAMIENTO@refinamiento_oper

IMPORTING REFINAMIENTO@refinamiento_powerset[atomo[T], D]

x: VAR D

A: VAR atomo[T]

CH, CH1, CH2: VAR cl_horn

C: VAR cl_definida

LCH: VAR list[cl_horn]

P, P1, P2: VAR clausulas_l[D].programa

PC: VAR clausulas[T].programa

SCH: VAR set[clausula_horn]

L, I, I1, I2, J: VAR list[D]

LL: VAR list[list[D]]

es_modelo_e_refina_l1: LEMMA
  es_modelo_e(L, C) IFF
   es_modelo(transf_interp(L), transf_clausula_def(C))

es_modelo_e_es_refinamiento: THEOREM
  es_refinamiento_op?[[interpretacion_finita[T], clausula_def[T]], bool,
                      [list[D], cl_definida[D]], bool,
                      prod_cart(transf_interp, transf_clausula_def),
                      id[bool]]
      (restrict[[interpretacion[T], clausula_horn[T]],
                [interpretacion_finita[T], clausula_def[T]], boolean]
           (es_modelo),
       es_modelo_e)

es_modelo_e_refina_l2: LEMMA
  es_modelo_e(L, P) IFF es_modelo(transf_interp(L), transf_programa(P))

es_modelo_p((I: interpretacion_finita[T]), PC): bool = es_modelo(I, PC)

es_modelo_e_es_refinamiento_p: THEOREM
  es_refinamiento_op?[[interpretacion_finita[T], clausulas[T].programa],
                      bool, [list[D], clausulas_l[D].programa], bool,
                      prod_cart(transf_interp, transf_programa), id[bool]]
      (es_modelo_p, es_modelo_e)

extrae_modelos_e_TCC1: OBLIGATION
```

```
  FORALL (L: list[D], LL2: list[list[D]], LL: list[list[D]],
          P: clausulas_l[D].programa):
    LL = cons(L, LL2) AND es_modelo_e(L, P) IMPLIES
     length[list[D]](LL2) < length[list[D]](LL);

extrae_modelos_e_TCC2: OBLIGATION
  FORALL (L: list[D], LL2: list[list[D]], LL: list[list[D]],
          P: clausulas_l[D].programa):
    LL = cons(L, LL2) AND NOT es_modelo_e(L, P) IMPLIES
     length[list[D]](LL2) < length[list[D]](LL);

extrae_modelos_e(P, LL): RECURSIVE list[list[D]] =
  CASES LL
    OF null: null,
       cons(L, LL2):
         IF es_modelo_e(L, P) THEN cons(L, extrae_modelos_e(P, LL2))
         ELSE extrae_modelos_e(P, LL2)
         ENDIF
    ENDCASES
   MEASURE length(LL)

extrae_modelos_e_caract: LEMMA
  member(L, extrae_modelos_e(P, LL)) IMPLIES es_modelo_e(L, P)

extrae_modelos_e_cn1: LEMMA
  member(L, extrae_modelos_e(P, LL)) IMPLIES member(L, LL)

cs_extrae_modelos_e: LEMMA
  member(L, LL) AND es_modelo_e(L, P) IMPLIES
   member(L, extrae_modelos_e(P, LL))

conj_modelos_herbrand_e_TCC1: OBLIGATION
  FORALL (P: clausulas_l[D].programa):
    cons?[list[D]]
        (extrae_modelos_e(P,
                          powerset_ref_2[D](base_herbrand_l_p[T, D](P))));

conj_modelos_herbrand_e(P): (cons?[list[D]]) =
    extrae_modelos_e(P, powerset_ref_2(base_herbrand_l_p(P)))

conj_modelos_herbrand_e_refina_l1: LEMMA
  conj_modelos_herbrand(transf_programa(P)) =
   extend[set[atomo[T]], interpretacion_finita[T], bool, FALSE]
       (c(transf_interp)(conj_modelos_herbrand_e(P)))

menor_modelo_herbrand_e(P): list[D] = Inter(conj_modelos_herbrand_e(P))

menor_modelo_herbrand_e_refinamiento: THEOREM
  es_refinamiento_op?[clausulas[T].programa, interpretacion_finita[T],
                      clausulas_l[D].programa, list[D], transf_programa,
                      transf_interp]
      (menor_modelo_herbrand, menor_modelo_herbrand_e)
```

```
 END menor_modelo_herbrand_l
```

D.2.7. Operador de consecuencia inmediata

```
consecuencia_i_l[T, D: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;
  ENDASSUMING

  REFINAMIENTO: LIBRARY = "../refinamiento"

  IMPORTING REFINAMIENTO@propiedades_operaciones_listas_via_ref[D]

  IMPORTING semantica_clausulas_l[D, T]

  IMPORTING consecuencia_i[T]

  x: VAR D

  C, C1, C2: VAR cl_definida

  P, P1, P2: VAR clausulas_l[D].programa

  I, I1, I2, J: VAR list[D]

  CH: VAR cl_horn

  consecuencia_i_TCC1: OBLIGATION
    FORALL (C: cl_definida[D], P2: list[cl_definida[D]], I: list[D],
            P: clausulas_l[D].programa):
      P = cons(C, P2) AND sublista?(cuerpo(C), I) IMPLIES
       length[cl_definida[D]](P2) < length[cl_definida[D]](P);

  consecuencia_i_TCC2: OBLIGATION
    FORALL (C: cl_definida[D], P2: list[cl_definida[D]], I: list[D],
            P: clausulas_l[D].programa):
      P = cons(C, P2) AND NOT sublista?(cuerpo(C), I) IMPLIES
       length[cl_definida[D]](P2) < length[cl_definida[D]](P);

  consecuencia_i(P)(I): RECURSIVE list[D] =
    CASES P
      OF null: null,
         cons(C, P2):
           IF sublista?(cuerpo(C), I)
             THEN cons(car(cabeza(C)), consecuencia_i(P2)(I))
           ELSE consecuencia_i(P2)(I)
           ENDIF
```

```
     ENDCASES
    MEASURE length(P)

  consecuencia_i_caract: LEMMA
    member(x, consecuencia_i(P)(I)) IFF
     (EXISTS C:
        member(C, P) AND sublista?(cuerpo(C), I) AND x = car(cabeza(C)))

  consecuencia_i_caract_2_TCC1: OBLIGATION
    FORALL (I: list[D], P: clausulas_l[D].programa, CH):
      es_cl_definida(CH) AND member(CH, P) AND sublista?(cuerpo(CH), I)
       IMPLIES cons?[D](cabeza[D](CH));

  consecuencia_i_caract_2: LEMMA
    member(x, consecuencia_i(P)(I)) IFF
     (EXISTS CH:
        es_cl_definida(CH) AND
         member(CH, P) AND sublista?(cuerpo(CH), I) AND x = car(cabeza(CH)))

  consecuencia_i_correcto_l1_p: LEMMA
    subset?(transf_interp(consecuencia_i(P)(I)),
            c_i(transf_programa(P))(transf_interp(I)))

  consecuencia_i_correcto_l2_p: LEMMA
    subset?(c_i(transf_programa(P))(transf_interp(I)),
            transf_interp(consecuencia_i(P)(I)))

  consecuencia_i_correcto_p: THEOREM
    transf_interp(consecuencia_i(P)(I)) =
     c_i(transf_programa(P))(transf_interp(I))

  consecuencia_i_correcto_p_f: COROLLARY
    transf_interp(consecuencia_i(P)(I)) =
     c_i_f(transf_programa(P))(transf_interp(I))

  consecuencia_i_es_refinamiento_p: THEOREM
    es_refinamiento_op?[interpretacion_finita[T], interpretacion_finita[T],
                        list[D], list[D], transf_interp, transf_interp]
       (c_i_f(transf_programa(P)), consecuencia_i(P))

  consecuencia_i_monotona: LEMMA
    sublista?(I, J) IMPLIES
     sublista?(consecuencia_i(P)(I), consecuencia_i(P)(J))

  consecuencia_i_igual_l: COROLLARY
    igual_l?(I, J) IMPLIES
     igual_l?(consecuencia_i(P)(I), consecuencia_i(P)(J))
END consecuencia_i_l
```

D.2.8. Cálculo del menor punto fijo

```
menor_punto_fijo_l_iterado[T, D: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

   D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;
  ENDASSUMING

  IMPORTING menor_modelo_herbrand_l[T, D]

  IMPORTING consecuencia_i_l[T, D]

  IMPORTING semantica_p_f[T]

  IMPORTING REFINAMIENTO@refinamiento_iteracion

  x: VAR D

  CH, CH1, CH2: VAR cl_horn

  LCH: VAR list[cl_horn]

  C, C1, C2: VAR cl_definida

  P, P1, P2: VAR clausulas_l[D].programa

  menor_punto_fijo_i(P): list[D] =
      iterate(consecuencia_i(P), cardinal_l(base_herbrand_l(P)))(null)

  transf_interp_nula: CLAIM transf_interp(null) = emptyset

  menor_punto_fijo_i_correcto: THEOREM
    transf_interp(menor_punto_fijo_i(P)) = mpf(transf_programa(P))

  menor_punto_fijo_es_refinamiento_mpf_i_op_TCC1: OBLIGATION
    FORALL (x1: programa[T]): is_finite[atomo[T]](mpf[T](x1));

  menor_punto_fijo_es_refinamiento_mpf_i_op: COROLLARY
    es_refinamiento_op?[clausulas[T].programa, interpretacion_finita[T],
                        clausulas_l[D].programa, list[D], transf_programa,
                        transf_interp]
        (mpf, menor_punto_fijo_i)

  mmh_igual_mpf: COROLLARY
    igual_l?(menor_modelo_herbrand_e(P), menor_punto_fijo_i(P))
 END menor_punto_fijo_l_iterado


menor_punto_fijo_l[T, D: TYPE+]: THEORY
 BEGIN
  ASSUMING
```

```
 T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;

 D_TCC1: ASSUMPTION EXISTS (x: D): TRUE;
ENDASSUMING

IMPORTING consecuencia_i_l[T, D]

IMPORTING base_herbrand_l[T, D]

x: VAR D

C, C1, C2: VAR cl_definida

P, P1, P2: VAR clausulas_l[D].programa

I, I1, I2, J: VAR list[D]

IC: VAR interpretacion[T]

interpretacion_l_principal(P): TYPE =
{I: list[D] | sublista?(I, base_herbrand_l(P))}

consecuencia_i_subconjunto_base_herbrand_l: LEMMA
  sublista?(consecuencia_i(P)(I), base_herbrand_l(P))

medida_p_f_TCC1: OBLIGATION
  FORALL (P: clausulas_l[D].programa, I: interpretacion_l_principal(P)):
    cardinal_l[D](base_herbrand_l[T, D](P)) - cardinal_l[D](I) >= 0;

medida_p_f(P)(I: interpretacion_l_principal(P)): nat =
    cardinal_l(base_herbrand_l(P)) - cardinal_l(I)

cn_interpretacion_principal: CLAIM
  FORALL (I: interpretacion_l_principal(P)):
    sublista?(append(consecuencia_i(P)(I), I), base_herbrand_l(P))

suc_no_colapsa_cardinal_l: CLAIM
  FORALL (I1: list[D], P: clausulas_l[D].programa,
          I: interpretacion_l_principal(P)):
    I1 = append(consecuencia_i(P)(I), I) AND NOT igual_l?(I1, I) IMPLIES
     cardinal_l(I) < cardinal_l(I1)

medida_p_f_paso_TCC1: OBLIGATION
  FORALL (I1: list[D], P: clausulas_l[D].programa,
          I: interpretacion_l_principal(P)):
    NOT igual_l?(I1, I) AND I1 = append(consecuencia_i(P)(I), I) IMPLIES
     sublista?[D](I1, base_herbrand_l[T, D](P));

medida_p_f_paso: LEMMA
  FORALL (I1: list[D], P: clausulas_l[D].programa,
          I: interpretacion_l_principal(P)):
    I1 = append(consecuencia_i(P)(I), I) AND NOT igual_l?(I1, I) IMPLIES
```

```
     medida_p_f(P)(I1) < medida_p_f(P)(I);

menor_punto_fijo_aux_TCC1: OBLIGATION
  FORALL (I1: list[D], P: clausulas_l[D].programa,
          I: interpretacion_l_principal(P)):
    I1 = append(consecuencia_i(P)(I), I) AND NOT igual_l?(I1, I) IMPLIES
     medida_p_f(P)(I1) < medida_p_f(P)(I);

menor_punto_fijo_aux(P)(I: interpretacion_l_principal(P)): RECURSIVE
list[D] =
  LET I1 = append(consecuencia_i(P)(I), I) IN
    IF igual_l?(I1, I) THEN I ELSE menor_punto_fijo_aux(P)(I1) ENDIF
   MEASURE medida_p_f(P)(I)

menor_punto_fijo_TCC1: OBLIGATION
  FORALL (P: clausulas_l[D].programa):
    sublista?[D](null[D], base_herbrand_l[T, D](P));

menor_punto_fijo(P): list[D] = menor_punto_fijo_aux(P)(null)

cs_menor_punto_fijo_aux_construye_monotona: CLAIM
  FORALL (I: interpretacion_l_principal(P)):
    sublista?(I, consecuencia_i(P)(I)) IMPLIES
     sublista?(append(consecuencia_i(P)(I), I),
               consecuencia_i(P)(append(consecuencia_i(P)(I), I)))

menor_punto_fijo_aux_es_punto_fijo: LEMMA
  FORALL (I: interpretacion_l_principal(P)):
    sublista?(I, consecuencia_i(P)(I)) AND menor_punto_fijo_aux(P)(I) = J
     IMPLIES igual_l?(consecuencia_i(P)(J), J)

menor_punto_fijo_aux_contenido_puntos_fijos: LEMMA
  FORALL (I: interpretacion_l_principal(P)):
    menor_punto_fijo_aux(P)(I) = J AND sublista?(I, consecuencia_i(P)(I))
     IMPLIES
     (FORALL I1:
        igual_l?(consecuencia_i(P)(I1), I1) AND sublista?(I, I1) IMPLIES
         sublista?(J, I1))

menor_punto_fijo_correcto: THEOREM
  menor_punto_fijo(P) = J IMPLIES
   igual_l?(consecuencia_i(P)(J), J) AND
    (FORALL I1:
       igual_l?(consecuencia_i(P)(I1), I1) IMPLIES sublista?(J, I1))

IMPORTING semantica_p_f[T]

IMPORTING REFINAMIENTO@refinamiento_conj_finitos[atomo[T], D] AS re_atomo

conserva_puntos_fijos: LEMMA
  igual_l?[D](consecuencia_i(P)(I), I) IMPLIES
   fixpoint?(c_i(transf_programa(P)))(transf_interp(I))
```

```
  conserva_menor_punto_fijo: COROLLARY
    fixpoint?(c_i(transf_programa(P)))(transf_interp(menor_punto_fijo(P)))

  transf_menor_punto_fijo_es_el_menor_l1: LEMMA
    fixpoint?(c_i(transf_programa(P)))(transf_interp(I)) IMPLIES
     subset?(transf_interp(menor_punto_fijo(P)), transf_interp(I))

  IMPORTING REFINAMIENTO@refinamiento_conj_finitos

  transf_menor_punto_fijo_es_el_menor: THEOREM
    fixpoint?(c_i(transf_programa(P)))(IC) IMPLIES
     subset?(transf_interp(menor_punto_fijo(P)), IC)

  transf_menor_punto_fijo_es_menor_punto_fijo: COROLLARY
    least_fixpoint?(c_i(transf_programa(P)))
                   (transf_interp(menor_punto_fijo(P)))

  menor_punto_fijo_es_refinamiento_mpf: THEOREM
    transf_interp(menor_punto_fijo(P)) = mpf(transf_programa(P))

  menor_punto_fijo_es_refinamiento_op_TCC1: OBLIGATION
    FORALL (x1: programa[T]): is_finite[atomo[T]](mpf[T](x1));

  menor_punto_fijo_es_refinamiento_op: COROLLARY
    es_refinamiento_op?[clausulas[T].programa, interpretacion_finita[T],
                        clausulas_l[D].programa, list[D], transf_programa,
                        transf_interp]
        (mpf, menor_punto_fijo)
 END menor_punto_fijo_l
```

D.2.9. Diversos algoritmos de resolución SLD

```
sld_resolucion_l[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_clausulas_l[T, T]

  C, C1, C2: VAR cl_definida

  G, G1, G2: VAR objetivo_definido

  P, P1, P2: VAR clausulas_l[T].programa

  medida(P, G): nat

  medida_ax_TCC1: OBLIGATION
```

```
  FORALL (G: objetivo_definido[T]):
    NOT es_vacio(G) IMPLIES
     (FORALL (C: cl_definida[T]): cons?[T](cuerpo[T](G)));

medida_ax_TCC2: OBLIGATION
  FORALL (G: objetivo_definido[T]):
    NOT es_vacio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = car(cuerpo(G)) IMPLIES es_objetivo_propio[T](G));

medida_ax: AXIOM
  FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa):
    NOT es_vacio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = car(cuerpo(G)) IMPLIES
         medida(P, resolvente(G, C)) < medida(P, G))

demostrable_por_sld_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa):
    NOT es_vacio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = car(cuerpo(G)) IMPLIES
         medida(P, resolvente[T](G, C)) < medida(P, G));

demostrable_por_sld(P, G): RECURSIVE bool =
  IF es_vacio(G) THEN TRUE
  ELSE some(LAMBDA (C):
              car(cabeza(C)) = car(cuerpo(G)) AND
               demostrable_por_sld(P, resolvente(G, C)))
          (P)
  ENDIF
   MEASURE medida(P, G)

medida_c(P, G, P1): nat

medida_c_ax1_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
    NOT es_vacio(G) AND NOT null?(P1) IMPLIES cons?[cl_definida[T]](P1);

medida_c_ax1_TCC2: OBLIGATION
  FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
    NOT es_vacio(G) AND NOT null?(P1) IMPLIES cons?[T](cuerpo[T](G));

medida_c_ax1_TCC3: OBLIGATION
  FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
    car(cabeza(car(P1))) = car(cuerpo(G)) AND
     NOT null?(P1) AND NOT es_vacio(G)
     IMPLIES es_objetivo_propio[T](G);

medida_c_ax1: AXIOM
  FORALL (G: objetivo_definido[T], P, P1):
    NOT es_vacio(G) AND
```

```
      NOT null?(P1) AND car(cabeza(car(P1))) = car(cuerpo(G))
      IMPLIES
      medida_c(P, resolvente(G, car[cl_definida[T]](P1)), P) <
       medida_c(P, G, P1)

  medida_c_ax2_TCC1: OBLIGATION
    FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
      NOT null?(P1) IMPLIES cons?[cl_definida[T]](P1);

  medida_c_ax2: AXIOM
    FORALL (G: objetivo_definido[T], P, P1):
      NOT null?(P1) IMPLIES
       medida_c(P, G, cdr[cl_definida[T]](P1)) < medida_c(P, G, P1)

  prueba_por_sld_aux_b_TCC1: OBLIGATION
    FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa,
            P1: clausulas_l[T].programa):
      NOT es_vacio(G) AND
       NOT null?(P1) AND car(cabeza(car(P1))) = car(cuerpo(G))
       IMPLIES
       medida_c(P, resolvente[T](G, car[cl_definida[T]](P1)), P) <
        medida_c(P, G, P1);

  prueba_por_sld_aux_b_TCC2: OBLIGATION
    FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa,
            P1: clausulas_l[T].programa,
            v:
              [{z:
                  [clausulas_l[T].programa, objetivo_definido[T],
                   clausulas_l[T].programa] |
                        medida_c(z`1, z`2, z`3) < medida_c(P, G, P1)} ->
                 [bool, list[par_de]]]):
      NOT es_vacio(G) AND
       NOT null?(P1) AND car(cabeza(car(P1))) = car(cuerpo(G))
       IMPLIES
       (FORALL (temp: [bool, list[par_de[T, T]]]):
          temp = v(P, resolvente(G, car(P1)), P) AND NOT PROJ_1(temp)
           IMPLIES
           medida_c(P, G, cdr[cl_definida[T]](P1)) < medida_c(P, G, P1));

  prueba_por_sld_aux_b_TCC3: OBLIGATION
    FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa,
            P1: clausulas_l[T].programa):
      NOT es_vacio(G) AND
       NOT null?(P1) AND NOT car(cabeza(car(P1))) = car(cuerpo(G))
       IMPLIES medida_c(P, G, cdr[cl_definida[T]](P1)) < medida_c(P, G, P1);

  prueba_por_sld_aux_b(P, G, P1): RECURSIVE [bool, list[par_de]] =
    IF es_vacio(G) THEN (TRUE, null)
    ELSIF null?(P1) THEN (FALSE, null)
    ELSIF car(cabeza(car(P1))) = car(cuerpo(G))
      THEN LET temp = prueba_por_sld_aux_b(P, resolvente(G, car(P1)), P) IN
```

```
            IF PROJ_1(temp) THEN (TRUE, cons((G, car(P1)), PROJ_2(temp)))
            ELSE prueba_por_sld_aux_b(P, G, cdr(P1))
            ENDIF
    ELSE prueba_por_sld_aux_b(P, G, cdr(P1))
    ENDIF
     MEASURE medida_c(P, G, P1)

  prueba_por_sld_b(P, G): list[par_de] =
      PROJ_2(prueba_por_sld_aux_b(P, G, P))

  medida_b(P, G, P1): ordinal = lex2(medida(P, G), length(P1))
 END sld_resolucion_l

sld_resolucion_l_3[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_clausulas_l[T, T]

  C, C1, C2: VAR cl_definida

  G, G1, G2: VAR objetivo_definido

  P, P1, P2, LC: VAR clausulas_l[T].programa

  LG: VAR list[objetivo_definido]

  lista_resolventes_TCC1: OBLIGATION
    FORALL (C1: cl_definida[T], P1: list[cl_definida[T]],
            (G: (es_objetivo_propio[T])), P: clausulas_l[T].programa):
      P = cons(C1, P1) IMPLIES cons?[T](cuerpo[T](G));

  lista_resolventes_TCC2: OBLIGATION
    FORALL (C1: cl_definida[T], P1: list[cl_definida[T]],
            (G: (es_objetivo_propio[T])), P: clausulas_l[T].programa):
      P = cons(C1, P1) AND car(cabeza(C1)) = car(cuerpo(G)) IMPLIES
       length[cl_definida[T]](P1) < length[cl_definida[T]](P);

  lista_resolventes_TCC3: OBLIGATION
    FORALL (C1: cl_definida[T], P1: list[cl_definida[T]],
            (G: (es_objetivo_propio[T])), P: clausulas_l[T].programa):
      P = cons(C1, P1) AND NOT car(cabeza(C1)) = car(cuerpo(G)) IMPLIES
       length[cl_definida[T]](P1) < length[cl_definida[T]](P);

  lista_resolventes(P, (G: (es_objetivo_propio))): RECURSIVE
  list[objetivo_definido] =
    CASES P
      OF null: null,
         cons(C1, P1):
           IF car(cabeza(C1)) = car(cuerpo(G))
```

```
          THEN cons(resolvente(G, C1), lista_resolventes(P1, G))
        ELSE lista_resolventes(P1, G)
        ENDIF
    ENDCASES
   MEASURE length(P)

medida_res_3(P, G, LG): nat

medida_res_3_ax_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T]):
    NOT es_vacio(G) IMPLIES es_objetivo_propio[T](G);

medida_res_3_ax_TCC2: OBLIGATION
  FORALL (G: objetivo_definido[T], LG: list[objetivo_definido[T]],
          P: clausulas_1[T].programa):
    NOT es_vacio(G) IMPLIES
     (FORALL (L: list[objetivo_definido[T]]):
        L = lista_resolventes(P, G) IMPLIES
         (FORALL (NLG: list[objetivo_definido[T]]):
            NOT null?(NLG) AND NLG = append(LG, L) IMPLIES
             cons?[objetivo_definido[T]](NLG)));

medida_res_3_ax: AXIOM
  FORALL (G: objetivo_definido[T], LG: list[objetivo_definido[T]],
          P: clausulas_1[T].programa):
    NOT es_vacio(G) IMPLIES
     (FORALL (L: list[objetivo_definido[T]]):
        L = lista_resolventes(P, G) IMPLIES
         (FORALL (NLG: list[objetivo_definido[T]]):
            NLG = append(LG, L) AND NOT null?(NLG) IMPLIES
             medida_res_3(P, car[objetivo_definido[T]](NLG),
                          cdr[objetivo_definido[T]](NLG))
              < medida_res_3(P, G, LG)))

demostrable_por_sld_3_aux_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T], LG: list[objetivo_definido[T]],
          P: clausulas_1[T].programa):
    NOT es_vacio(G) IMPLIES
     (FORALL (L: list[objetivo_definido[T]]):
        L = lista_resolventes(P, G) IMPLIES
         (FORALL (NLG: list[objetivo_definido[T]]):
            NLG = append(LG, L) AND NOT null?(NLG) IMPLIES
             medida_res_3(P, car[objetivo_definido[T]](NLG),
                          cdr[objetivo_definido[T]](NLG))
              < medida_res_3(P, G, LG)));

demostrable_por_sld_3_aux(P, G, LG): RECURSIVE bool =
  IF es_vacio(G) THEN TRUE
  ELSE LET L = lista_resolventes(P, G), NLG = append(LG, L) IN
         IF null?(NLG) THEN FALSE
         ELSE demostrable_por_sld_3_aux(P, car(NLG), cdr(NLG))
         ENDIF
```

```
    ENDIF
     MEASURE medida_res_3(P, G, LG)

  demostrable_por_sld_3(P, G): bool = demostrable_por_sld_3_aux(P, G, null)
 END sld_resolucion_l_3

sld_resolucion_l_via_r_2[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_clausulas_l[T, T]

  C, C1, C2: VAR cl_definida

  G, G1, G2: VAR objetivo_definido

  P, P1, P2: VAR clausulas_l[T].programa

  n: VAR nat

  R: VAR [(es_objetivo_propio) -> T]

  es_regla_computacion(R): bool =
      FORALL (G: (es_objetivo_propio)):
        list_props[T].member(R(G), cuerpo(G))

  tiene_resolvente_l(G, C): bool = member(car(cabeza(C)), cuerpo(G))

  PD_l: TYPE =
  {par: [(es_objetivo_propio), cl_definida] |
           LET (G, C) = par IN tiene_resolvente_l(G, C)}

  resolvente_g_TCC1: OBLIGATION
    FORALL (par: PD_l):
      es_cl_horn[T]
          (LET (G, C) = par IN
             (null[T],
              append[T]
                  (elimina[T](car[T](cabeza[T](C)), cuerpo[T](G)),
                   cuerpo[T](C))))
       AND
       es_objetivo_definido[T]
           (LET (G, C) = par IN
              (null[T],
               append[T]
                   (elimina[T](car[T](cabeza[T](C)), cuerpo[T](G)),
                    cuerpo[T](C))));

  resolvente_g(par: PD_l): objetivo_definido[T] =
      LET (G, C) = par IN
```

```
      (null, append(elimina(car(cabeza(C)), cuerpo(G)), cuerpo(C)))


demostrable_por_sld_long_via_r_2_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T], R: (es_regla_computacion), n: nat):
    NOT n = 0 AND es_objetivo_propio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = R(G) IMPLIES tiene_resolvente_l(G, C));

demostrable_por_sld_long_via_r_2_TCC2: OBLIGATION
  FORALL (G: objetivo_definido[T], R: (es_regla_computacion), n: nat):
    NOT n = 0 AND es_objetivo_propio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = R(G) IMPLIES n - 1 >= 0);

demostrable_por_sld_long_via_r_2_TCC3: OBLIGATION
  FORALL (G: objetivo_definido[T], R: (es_regla_computacion), n: nat):
    NOT n = 0 AND es_objetivo_propio(G) IMPLIES
     (FORALL (C: cl_definida[T]):
        car(cabeza(C)) = R(G) IMPLIES n - 1 < n);

demostrable_por_sld_long_via_r_2(R: (es_regla_computacion))(P, G, n):
RECURSIVE bool =
  IF n = 0 THEN es_vacio(G)
  ELSE es_objetivo_propio(G) AND
       some(LAMBDA (C):
              car(cabeza(C)) = R(G) AND
               demostrable_por_sld_long_via_r_2(R)
                                              (P,
                                               resolvente_g(G, C),
                                               n - 1))
            (P)
  ENDIF
   MEASURE n

medida_aux((R: (es_regla_computacion)), P, G, n): nat

medida_aux_ax: AXIOM
  FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa,
          (R: (es_regla_computacion)), n: nat):
    NOT demostrable_por_sld_long_via_r_2(R)(P, G, n) IMPLIES
     medida_aux(R, P, G, n + 1) < medida_aux(R, P, G, n)

comprueba_demostrable_aux_TCC1: OBLIGATION
  FORALL (G: objetivo_definido[T], P: clausulas_l[T].programa,
          (R: (es_regla_computacion)), n: nat):
    NOT demostrable_por_sld_long_via_r_2(R)(P, G, n) IMPLIES
     medida_aux(R, P, G, n + 1) < medida_aux(R, P, G, n);

comprueba_demostrable_aux((R: (es_regla_computacion)), P, G, n):
RECURSIVE bool =
  IF demostrable_por_sld_long_via_r_2(R)(P, G, n) THEN TRUE
```

```
    ELSE comprueba_demostrable_aux(R, P, G, n + 1)
    ENDIF
     MEASURE medida_aux(R, P, G, n)

  demostrable_por_sld_via_r_2(R: (es_regla_computacion))(P, G): bool =
      IF es_vacio(G) THEN TRUE
      ELSE comprueba_demostrable_aux(R, P, G, 1)
      ENDIF
 END sld_resolucion_l_via_r_2
```

D.2.10. Corrección de un algoritmo de demostrabilidad SLD

```
sld_resolucion_l_correccion[T: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T_TCC1: ASSUMPTION EXISTS (x: T): TRUE;
  ENDASSUMING

  IMPORTING semantica_clausulas_l[T, T]

  IMPORTING sld_resolucion_l[T]

  x, x1, x2: VAR T

  CH: VAR cl_horn[T]

  C: VAR cl_definida[T]

  P, P1, P2: VAR clausulas_l[T].programa

  I, I1, I2: VAR interpretacion[T]

  Int, Int1: VAR set[T]

  FSC: VAR finite_set[cl_horn[T]]

  LSC: VAR list[cl_horn[T]]

  G: VAR objetivo_definido[T]

  es_insatisfacible_rep: LEMMA
    es_insatisfacible(cons(G, P)) IFF
     es_insatisfacible(add(transf_clausula_horn(G),
                           transf_lista_cl_horn(P)))

  modelo_resolvente_lista_l_TCC1: OBLIGATION
    FORALL (C: cl_definida[T], G: objetivo_definido[T], Int: set[T]):
      es_modelo(Int, C) AND es_modelo(Int, G) AND es_objetivo_propio(G)
       IMPLIES cons?[T](cuerpo[T](G));
```

```
modelo_resolvente_lista_l: LEMMA
  es_modelo(Int, C) AND
   es_modelo(Int, G) AND
    es_objetivo_propio(G) AND car(cabeza(C)) = car(cuerpo(G))
   IMPLIES es_modelo(Int, resolvente(G, C))

resolvente_insatisfacible_lista_TCC1: OBLIGATION
  FORALL (C: cl_definida[T], P: clausulas_l[T].programa,
          G: (es_objetivo_propio[T])):
    member[cl_horn[T]](C, P) AND
     es_insatisfacible(cons(resolvente(G, C), P))
     IMPLIES cons?[T](cuerpo[T](G));

resolvente_insatisfacible_lista: LEMMA
  FORALL (G: (es_objetivo_propio)):
    member[cl_horn[T]](C, P) AND
     es_insatisfacible(cons(resolvente(G, C), P)) AND
      car(cabeza(C)) = car(cuerpo(G))
     IMPLIES es_insatisfacible(cons(G, P))

resolvente_insatisfacible_lista_p_TCC1: OBLIGATION
  FORALL (C: cl_definida[T], P: clausulas_l[T].programa,
          G: (es_objetivo_propio[T])):
    member(C, P) AND es_insatisfacible(cons(resolvente(G, C), P)) IMPLIES
     cons?[T](cuerpo[T](G));

resolvente_insatisfacible_lista_p: LEMMA
  FORALL (G: (es_objetivo_propio)):
    member(C, P) AND
     es_insatisfacible(cons(resolvente(G, C), P)) AND
      car(cabeza(C)) = car(cuerpo(G))
     IMPLIES es_insatisfacible(cons(G, P))

insatisfacible_vacio: LEMMA
  FORALL (G: objetivo_definido[T]):
    es_vacio(G) IMPLIES es_insatisfacible(cons(G, P))

cn_every_cl_horn: CLAIM
  FORALL (LS: list[clausula[T]]):
    every[clausula[T]]
        (LAMBDA (x: clausula[T]): es_cl_horn(x) AND es_cl_definida(x))
        (LS)
     IMPLIES every[clausula[T]](es_cl_horn)(LS)

cn_objetivo_no_vacío: CLAIM NOT es_vacio(G) IMPLIES es_objetivo_propio(G)

demostrable_por_sld_correcto: THEOREM
  FORALL (G: objetivo_definido[T]):
    demostrable_por_sld(P, G) IMPLIES es_insatisfacible(cons(G, P))

cs_car_cl_horn_cl_def_TCC1: OBLIGATION
```

```
    FORALL (LS: list[clausula[T]]):
      every[clausula[T]]
          (LAMBDA (x: clausula[T]): es_cl_horn(x) AND es_cl_definida(x))
          (LS)
       AND NOT null?(LS)
       IMPLIES cons?[clausula[T]](LS);

  cs_car_cl_horn_cl_def: CLAIM
    FORALL (LS: list[clausula[T]]):
      NOT null?(LS) AND
       every[clausula[T]]
           (LAMBDA (x: clausula[T]): es_cl_horn(x) AND es_cl_definida(x))
           (LS)
       IMPLIES es_cl_horn(car(LS)) AND es_cl_definida(car(LS))

  cn1_medida_b: CLAIM
    cons?(P1) IMPLIES medida_b(P, G, cdr(P1)) < medida_b(P, G, P1)

  cn2_medida_b_TCC1: OBLIGATION
    FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
      NOT null?(P1) AND es_objetivo_propio(G) IMPLIES
       cons?[cl_definida[T]](P1);

  cn2_medida_b_TCC2: OBLIGATION
    FORALL (G: objetivo_definido[T], P1: clausulas_l[T].programa):
      NOT null?(P1) AND es_objetivo_propio(G) IMPLIES
       cons?[T](cuerpo[T](G));

  cn2_medida_b: CLAIM
    NOT null?(P1) AND
     es_objetivo_propio(G) AND car(cabeza(car(P1))) = car(cuerpo(G))
     IMPLIES medida_b(P, resolvente(G, car(P1)), P) < medida_b(P, G, P1)

  prueba_por_sld_aux_b_correcto: THEOREM
    (FORALL C: member(C, P1) IMPLIES member(C, P)) IMPLIES
     (PROJ_1(prueba_por_sld_aux_b(P, G, P1)) IMPLIES
       es_refutacion(PROJ_2(prueba_por_sld_aux_b(P, G, P1)), P, G))

  prueba_por_sld_aux_correcto_l1: LEMMA
    es_objetivo_propio(G) AND cons?(PROJ_2(prueba_por_sld_aux_b(P, G, P1)))
     IMPLIES PROJ_1(prueba_por_sld_aux_b(P, G, P1))

  prueba_por_sld_b_correcto_vacio: THEOREM
    es_vacio(G) IMPLIES es_refutacion(prueba_por_sld_b(P, G), P, G)

  prueba_por_sld_b_correcto: THEOREM
    es_objetivo_propio(G) AND cons?(prueba_por_sld_b(P, G)) IMPLIES
     es_refutacion(prueba_por_sld_b(P, G), P, G)
END sld_resolucion_l_correccion
```

Apéndice E

Análisis formal de conceptos

E.1. Formalización usando conjuntos finitos

E.1.1. Contextos formales finitos: retículo de los conceptos

```
contextos_formales[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  TC: LIBRARY = "../auxiliares/conjuntos"

  IMPORTING TC@operaciones_conj_finitos, TC@propiedades_conjuntos

  d: VAR T1

  a, a1, a2: VAR T2

  X, X1, X2: VAR finite_set[T1]

  Y, Y1, Y2: VAR finite_set[T2]

  Ob: VAR set[T1]

  A: VAR set[T2]

  I: VAR set[[T1, T2]]

  FFCT: TYPE =
  [# obj: finite_set[T1],
     atrib: non_empty_finite_set[T2],
     relacion: finite_set[[T1, T2]] #]
```

```
FFC: TYPE =
{C: FFCT |
         LET Re = relacion(C) IN
           FORALL (par: (Re)): obj(C)(par`1) AND atrib(C)(par`2)}

C, C1, C2: VAR FFC

intencion_TCC1: OBLIGATION
  FORALL (C: FFC, X: finite_set[T1]):
    NOT empty?(X) IMPLIES
     is_finite[T2]({a: T2 | FORALL (d: (X)): relacion(C)(d, a)});

intencion(C)(X): finite_set[T2] =
    IF empty?(X) THEN atrib(C)
    ELSE {a: T2 | FORALL (d: (X)): relacion(C)(d, a)}
    ENDIF

intencion_subset_atrib: LEMMA subset?(intencion(C)(X), atrib(C))

intencion_r_TCC1: OBLIGATION
  FORALL (C: FFC, X: (powerset[T1](obj(C)))): is_finite[T1](X);

intencion_r_TCC2: OBLIGATION
  FORALL (C: FFC, X: (powerset[T1](obj(C)))):
    powerset[T2](atrib(C))(intencion(C)(X));

intencion_r(C)(X: (powerset(obj(C)))): (powerset(atrib(C))) =
    intencion(C)(X)

extension_l1: CLAIM
  nonempty?(Y) IMPLIES
   subset?({d: T1 | FORALL (a: (Y)): relacion(C)(d, a)}, obj(C))

extension_TCC1: OBLIGATION
  FORALL (C: FFC, Y: finite_set[T2]):
    NOT empty?(Y) IMPLIES
     is_finite[T1]({d: T1 | FORALL (a: (Y)): relacion(C)(d, a)});

extension(C)(Y: finite_set[T2]): finite_set[T1] =
    IF empty?(Y) THEN obj(C)
    ELSE {d: T1 | FORALL (a: (Y)): relacion(C)(d, a)}
    ENDIF

extension_subset_obj: LEMMA subset?(extension(C)(Y), obj(C))

extension_r_TCC1: OBLIGATION
  FORALL (C: FFC, Y: (powerset[T2](atrib(C)))): is_finite[T2](Y);

extension_r_TCC2: OBLIGATION
  FORALL (C: FFC, Y: (powerset[T2](atrib(C)))):
    powerset[T1](obj(C))(extension(C)(Y));
```

```
extension_r(C)(Y: (powerset(atrib(C)))): (powerset(obj(C))) =
    extension(C)(Y)

intencion_bd: LEMMA X1 = X2 IMPLIES intencion(C)(X1) = intencion(C)(X2)

extension_bd: LEMMA Y1 = Y2 IMPLIES extension(C)(Y1) = extension(C)(Y2)

clausura_o(C)(X): finite_set[T1] = extension(C)(intencion(C)(X))

clausura_a(C)(Y): finite_set[T2] = intencion(C)(extension(C)(Y))

clausura_o_r_TCC1: OBLIGATION
  FORALL (C: FFC, X: (powerset[T1](obj(C)))):
    powerset[T1](obj(C))(clausura_o(C)(X));

clausura_o_r(C)(X: (powerset(obj(C)))): (powerset(obj(C))) =
    clausura_o(C)(X)

clausura_a_r_TCC1: OBLIGATION
  FORALL (C: FFC, Y: (powerset[T2](atrib(C)))):
    powerset[T2](atrib(C))(clausura_a(C)(Y));

clausura_a_r(C)(Y: (powerset(atrib(C)))): (powerset(atrib(C))) =
    clausura_a(C)(Y)

intencion_subset: LEMMA
  subset?(X1, X2) IMPLIES subset?(intencion(C)(X2), intencion(C)(X1))

extension_subset: LEMMA
  subset?(Y1, Y2) IMPLIES subset?(extension(C)(Y2), extension(C)(Y1))

int_ext_subset: LEMMA
  subset?(X, obj(C)) IMPLIES subset?(X, extension(C)(intencion(C)(X)))

subset_clausura_o: COROLLARY
  subset?(X, obj(C)) IMPLIES subset?(X, clausura_o(C)(X))

ext_int_subset: LEMMA
  subset?(Y, atrib(C)) IMPLIES subset?(Y, intencion(C)(extension(C)(Y)))

subset_clausura_a: LEMMA
  subset?(Y, atrib(C)) IMPLIES subset?(Y, clausura_a(C)(Y))

int_es_int_ext_int: LEMMA
  subset?(X, obj(C)) IMPLIES
   intencion(C)(extension(C)(intencion(C)(X))) = intencion(C)(X)

int_clausura_o: LEMMA
  subset?(X, obj(C)) IMPLIES
   intencion(C)(clausura_o(C)(X)) = intencion(C)(X)
```

```
clausura_a_int: LEMMA
  subset?(X, obj(C)) IMPLIES
   clausura_a(C)(intencion(C)(X)) = intencion(C)(X)

ext_es_ext_int_ext: LEMMA
  subset?(Y, atrib(C)) IMPLIES
   extension(C)(intencion(C)(extension(C)(Y))) = extension(C)(Y)

ext_clausura_a: LEMMA
  subset?(Y, atrib(C)) IMPLIES
   extension(C)(clausura_a(C)(Y)) = extension(C)(Y)

equivalente_subset_1: LEMMA
  subset?(X, obj(C)) AND subset?(Y, atrib(C)) IMPLIES
   (subset?(X, extension(C)(Y)) IFF subset?(Y, intencion(C)(X)))

equivalente_subset_2: LEMMA
  subset?(X, obj(C)) AND subset?(Y, atrib(C)) IMPLIES
   (subset?(X, extension(C)(Y)) IFF
     (FORALL (d: (X), a: (Y)): relacion(C)(d, a)))

es_concepto_c?(C)((X), (Y)): bool =
    subset?(X, obj(C)) AND
     subset?(Y, atrib(C)) AND intencion(C)(X) = Y AND extension(C)(Y) = X

concepto_c_no_vacio: LEMMA
  es_concepto_c?(C)
               (extension(C)(atrib(C)),
                intencion(C)(extension(C)(atrib(C))))

concepto_c_no_vacio_2: LEMMA
  es_concepto_c?(C)(extension(C)(atrib(C)), clausura_a(C)(atrib(C)))

concepto_c_TCC1: OBLIGATION
  FORALL (C: FFC):
    EXISTS (x:
              {x: [finite_set[T1], finite_set[T2]] |
                       es_concepto_c?(C)(x)}):
      TRUE;

concepto_c(C): TYPE+ = (es_concepto_c?(C))

concepto_c_no_vacio_corol: COROLLARY EXISTS (x: concepto_c(C)): TRUE

concepto_s_subtipo: JUDGEMENT concepto_c(C) SUBTYPE_OF
    [finite_set[T1], finite_set[T2]]

extension_concepto(C)(par: concepto_c(C)): finite_set[T1] = PROJ_1(par)

intencion_concepto(C)(par: concepto_c(C)): finite_set[T2] = PROJ_2(par)

es_concepto?_c_bd: LEMMA
```

```
  X1 = X2 AND Y1 = Y2 AND es_concepto_c?(C)(X1, Y1) IMPLIES
   es_concepto_c?(C)(X2, Y2)

subconcepto_c?(C)(par1, par2: concepto_c(C)): bool =
    subset?(extension_concepto(C)(par1), extension_concepto(C)(par2))

cond_equiv_subconcepto_c: LEMMA
  FORALL (par1, par2: concepto_c(C)):
    subconcepto_c?(C)(par1, par2) IFF
     subset?(intencion_concepto(C)(par2), intencion_concepto(C)(par1))

subconcepto_c_es_reflexiva: LEMMA
  reflexive?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_transitiva: LEMMA
  transitive?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_preorden: LEMMA
  preorder?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_antisimetrica: LEMMA
  antisymmetric?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c_es_orden_parcial: LEMMA
  partial_order?[concepto_c(C)](subconcepto_c?(C))

subconcepto_c?_TCC1: OBLIGATION
  FORALL (C): partial_order?[concepto_c(C)](subconcepto_c?(C));

JUDGEMENT subconcepto_c?(C) HAS_TYPE (partial_order?[concepto_c(C)])

F: VAR finite_set[finite_set[T1]]

G: VAR finite_set[finite_set[T2]]

conj_intenciones_c_TCC1: OBLIGATION
  FORALL (C: FFC, F: finite_set[finite_set[T1]]):
    is_finite[finite_set[T2]]
        (image[finite_set[T1], finite_set[T2]](intencion(C))(F));

conj_intenciones_c(C)(F): finite_set[finite_set[T2]] =
    image(intencion(C))(F)

conj_extensiones_c_TCC1: OBLIGATION
  FORALL (C: FFC, G: finite_set[finite_set[T2]]):
    is_finite[finite_set[T1]]
        (image[finite_set[T2], finite_set[T1]](extension(C))(G));

conj_extensiones_c(C)(G): finite_set[finite_set[T1]] =
    image(extension(C))(G)

conj_intenciones_union_vacia_TCC1: OBLIGATION
```

```
  FORALL (C: FFC, F: finite_set[finite_set[T1]]):
    empty?(union_f(F)) AND nonempty?(F) IMPLIES
     NOT empty?[finite_set[T2]](conj_intenciones_c(C)(F));

conj_intenciones_union_vacia: CLAIM
  nonempty?(F) AND empty?(union_f(F)) IMPLIES
   intersection_f(conj_intenciones_c(C)(F)) = atrib(C)

intencion_union_gen_TCC1: OBLIGATION
  FORALL (C: FFC, F: finite_set[finite_set[T1]]):
    nonempty?(F) IMPLIES
     NOT empty?[finite_set[T2]](conj_intenciones_c(C)(F));

intencion_union_gen: LEMMA
  nonempty?(F) IMPLIES
   intencion(C)(union_f(F)) = intersection_f(conj_intenciones_c(C)(F))

conj_extensiones_union_vacia_TCC1: OBLIGATION
  FORALL (C: FFC, G: finite_set[finite_set[T2]]):
    empty?(union_f(G)) AND nonempty?(G) IMPLIES
     NOT empty?[finite_set[T1]](conj_extensiones_c(C)(G));

conj_extensiones_union_vacia: CLAIM
  nonempty?(G) AND empty?(union_f(G)) IMPLIES
   intersection_f(conj_extensiones_c(C)(G)) = obj(C)

extension_union_gen_TCC1: OBLIGATION
  FORALL (C: FFC, G: finite_set[finite_set[T2]]):
    nonempty?(G) IMPLIES
     NOT empty?[finite_set[T1]](conj_extensiones_c(C)(G));

extension_union_gen: LEMMA
  nonempty?(G) IMPLIES
   extension(C)(union_f(G)) = intersection_f(conj_extensiones_c(C)(G))

intenciones_conj_conceptos_TCC1: OBLIGATION
  FORALL (C: FFC, S: finite_set[concepto_c(C)]):
    is_finite[finite_set[T2]]
        (image[concepto_c(C), finite_set[T2]](intencion_concepto(C))(S));

intenciones_conj_conceptos(C)(S: finite_set[concepto_c(C)]):
finite_set[finite_set[T2]] = image(intencion_concepto(C))(S)

extensiones_conj_conceptos_TCC1: OBLIGATION
  FORALL (C: FFC, S: finite_set[concepto_c(C)]):
    is_finite[finite_set[T1]]
        (image[concepto_c(C), finite_set[T1]](extension_concepto(C))(S));

extensiones_conj_conceptos(C)(S: finite_set[concepto_c(C)]):
finite_set[finite_set[T1]] = image(extension_concepto(C))(S)

composicion_e_i_c: LEMMA
```

```
  extension(C) o intencion_concepto(C) = extension_concepto(C)

composicion_i_e_c: LEMMA
  intencion(C) o extension_concepto(C) = intencion_concepto(C)

conj_ext_conceptos_es_conj_extensiones: LEMMA
  FORALL (S: finite_set[concepto_c(C)]):
    extensiones_conj_conceptos(C)(S) =
     conj_extensiones_c(C)(intenciones_conj_conceptos(C)(S))

intenciones_conj_conceptos_no_vacio: OBLIGATION
  FORALL (C, S: non_empty_finite_set[concepto_c(C)]):
    NOT empty?[finite_set[T2]](intenciones_conj_conceptos(C)(S));

intenciones_conj_conceptos_no_vacio: JUDGEMENT
  intenciones_conj_conceptos(C)(S: non_empty_finite_set[concepto_c(C)])
    HAS_TYPE non_empty_finite_set[finite_set[T2]]

conj_int_conceptos_es_conj_intenciones: LEMMA
  FORALL (S: finite_set[concepto_c(C)]):
    intenciones_conj_conceptos(C)(S) =
     conj_intenciones_c(C)(extensiones_conj_conceptos(C)(S))

infimo_lema_1_TCC1: OBLIGATION
  FORALL (C: FFC, S: non_empty_finite_set[concepto_c(C)]):
    NOT empty?[finite_set[T1]](extensiones_conj_conceptos(C)(S));

infimo_lema_1: LEMMA
  FORALL (S: non_empty_finite_set[concepto_c(C)]):
    es_concepto_c?(C)
                (intersection_f(extensiones_conj_conceptos(C)(S)),
                 intencion(C)
                          (extension(C)
                                    (union_f
                                     (intenciones_conj_conceptos
                                      (C)(S)))))

infimo_lema_2: LEMMA
  FORALL (S: non_empty_finite_set[concepto_c(C)]):
    es_concepto_c?(C)
                (intersection_f(extensiones_conj_conceptos(C)(S)),
                 clausura_a(C)
                           (union_f(intenciones_conj_conceptos(C)(S))))

infimo_TCC1: OBLIGATION
  FORALL (C: FFC, S: non_empty_finite_set[concepto_c(C)]):
    es_concepto_c?(C)
                (intersection_f[T1](extensiones_conj_conceptos(C)(S)),
                 clausura_a(C)
                           (union_f[T2]
                                (intenciones_conj_conceptos(C)(S))));
```

```
  infimo(C)(S: non_empty_finite_set[concepto_c(C)]): concepto_c(C) =
      (intersection_f(extensiones_conj_conceptos(C)(S)),
       clausura_a(C)(union_f(intenciones_conj_conceptos(C)(S))))

  infimo_es_infimo_p: THEOREM
    FORALL (S: non_empty_finite_set[concepto_c(C)]):
      greatest_lower_bound?(subconcepto_c?(C))(infimo(C)(S), S)

  extensiones_conj_conceptos_no_vacio: OBLIGATION
    FORALL (C, S: non_empty_finite_set[concepto_c(C)]):
      NOT empty?[finite_set[T1]](extensiones_conj_conceptos(C)(S));

  extensiones_conj_conceptos_no_vacio: JUDGEMENT
    extensiones_conj_conceptos(C)(S: non_empty_finite_set[concepto_c(C)])
      HAS_TYPE non_empty_finite_set[finite_set[T1]]

  supremo_lema_1: LEMMA
    FORALL (S: non_empty_finite_set[concepto_c(C)]):
      es_concepto_c?(C)
                    (extension(C)
                              (intencion(C)
                                        (union_f
                                         (extensiones_conj_conceptos
                                          (C)(S)))),
                     intersection_f(intenciones_conj_conceptos(C)(S)))

  supremo_lema_2: LEMMA
    FORALL (S: non_empty_finite_set[concepto_c(C)]):
      es_concepto_c?(C)
                    (clausura_o(C)
                               (union_f(extensiones_conj_conceptos(C)(S))),
                     intersection_f(intenciones_conj_conceptos(C)(S)))

  supremo_TCC1: OBLIGATION
    FORALL (C: FFC, S: non_empty_finite_set[concepto_c(C)]):
      es_concepto_c?(C)
                    (clausura_o(C)
                               (union_f[T1]
                                    (extensiones_conj_conceptos(C)(S))),
                     intersection_f[T2](intenciones_conj_conceptos(C)(S)));

  supremo(C)(S: non_empty_finite_set[concepto_c(C)]): concepto_c(C) =
      (clausura_o(C)(union_f(extensiones_conj_conceptos(C)(S))),
       intersection_f(intenciones_conj_conceptos(C)(S)))

  supremo_es_supremo_p: THEOREM
    FORALL (S: non_empty_finite_set[concepto_c(C)]):
      least_upper_bound?(subconcepto_c?(C))(supremo(C)(S), S)
END contextos_formales
```

E.1.2. Generación del conjunto de los conceptos de un contexto

```
conceptos_contexto[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING contextos_formales[T1, T2]

  IMPORTING finite_sets@finite_sets_inductions

  d: VAR T1

  a: VAR T2

  C: VAR FFC

  forma_de_concepto_f_obj_1: LEMMA
    FORALL (X: finite_set[T1]):
      subset?(X, obj(C)) IMPLIES
       es_concepto_c?(C)(extension(C)(intencion(C)(X)), intencion(C)(X))

  forma_de_concepto_f_obj_1_cl: LEMMA
    FORALL (X: finite_set[T1]):
      subset?(X, obj(C)) IMPLIES
       es_concepto_c?(C)(clausura_o(C)(X), intencion(C)(X))

  forma_de_concepto_f_obj_2: LEMMA
    FORALL (par: concepto_c(C)):
      EXISTS (X: finite_set[T1]):
        subset?(X, obj(C)) AND
         par = ((extension(C)(intencion(C)(X))), intencion(C)(X))

  forma_de_concepto_f_obj_2_cl: LEMMA
    FORALL (par: concepto_c(C)):
      EXISTS (X: finite_set[T1]):
        subset?(X, obj(C)) AND par = ((clausura_o(C)(X)), intencion(C)(X))

  CONCEPTOS(C): set[[finite_set[T1], finite_set[T2]]] =
      {par: [finite_set[T1], finite_set[T2]] | es_concepto_c?(C)(par)}

  forma_de_conceptos_obj: LEMMA
    CONCEPTOS(C) =
     ({par: [finite_set[T1], finite_set[T2]] |
         EXISTS (X: finite_set[T1]):
           subset?(X, obj(C)) AND
            par = ((clausura_o(C)(X)), intencion(C)(X))})
```

```
forma_de_concepto_f_atrib_1: LEMMA
  FORALL (Y: finite_set[T2]):
    subset?(Y, atrib(C)) IMPLIES
     es_concepto_c?(C)(extension(C)(Y), intencion(C)(extension(C)(Y)))

forma_de_concepto_f_atrib_1_cl: LEMMA
  FORALL (Y: finite_set[T2]):
    subset?(Y, atrib(C)) IMPLIES
     es_concepto_c?(C)(extension(C)(Y), clausura_a(C)(Y))

forma_de_concepto_f_atrib_2: LEMMA
  FORALL (par: concepto_c(C)):
    EXISTS (Y: finite_set[T2]):
      subset?(Y, atrib(C)) AND
       par = (extension(C)(Y), intencion(C)(extension(C)(Y)))

forma_de_concepto_f_atrib_2_cl: LEMMA
  FORALL (par: concepto_c(C)):
    EXISTS (Y: finite_set[T2]):
      subset?(Y, atrib(C)) AND par = (extension(C)(Y), clausura_a(C)(Y))

forma_de_conceptos_atrib: LEMMA
  CONCEPTOS(C) =
   ({par: [finite_set[T1], finite_set[T2]] |
       EXISTS (Y: finite_set[T2]):
         subset?(Y, atrib(C)) AND
          par = (extension(C)(Y), clausura_a(C)(Y))})

agrega_extension_elto_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], S: finite_set[finite_set[T1]], a: T2):
    is_finite[finite_set[T1]]
        (LET fun =
               LAMBDA (D: finite_set[T1]):
                 intersection[T1]
                     (D, extension[T1, T2](C)(singleton_f[T2](a)))
           IN
           union[finite_set[T1]]
               (S,
                restrict[set[T1], finite_set[T1], boolean]
                    (image[finite_set[T1], set[T1]](fun)(S))));

agrega_extension_elto(C)(a: T2, S: finite_set[finite_set[T1]]):
finite_set[finite_set[T1]] =
    LET fun =
          LAMBDA (D: finite_set[T1]):
            intersection(D, extension(C)(singleton_f[T2](a)))
      IN
      union(S, restrict[set[T1], finite_set[T1], boolean](image(fun)(S)))

genera_extensiones_aux_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    NOT empty?(Y) IMPLIES nonempty?[T2](Y);
```

```
genera_extensiones_aux_TCC2: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    NOT empty?(Y) IMPLIES card[T2](rest[T2](Y)) < card[T2](Y);

genera_extensiones_aux(C)
                      (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
RECURSIVE finite_set[finite_set[T1]] =
  IF empty?(Y) THEN S
  ELSE genera_extensiones_aux(C)
                             (rest(Y),
                              agrega_extension_elto(C)(choose(Y), S))
  ENDIF
   MEASURE card(Y)

genera_extensiones(C): finite_set[finite_set[T1]] =
    genera_extensiones_aux(C)(atrib(C), singleton[finite_set[T1]](obj(C)))

genera_conceptos(C): set[[finite_set[T1], finite_set[T2]]] =
    LET fun = LAMBDA (X: finite_set[T1]): (X, intencion(C)(X)) IN
      image(fun)(genera_extensiones(C))

genera_conceptos_finito: THEOREM is_finite(genera_conceptos(C))

es_conj_extensiones_f?(C)(S: finite_set[finite_set[T1]]): bool =
    FORALL (X: (S)): EXISTS (Y: finite_set[T2]): X = extension(C)(Y)

es_conj_extensiones_relativas_f?(C)(S: finite_set[finite_set[T1]]):
bool =
    FORALL (X: (S)):
      EXISTS (Y: finite_set[T2]):
        subset?(Y, atrib(C)) AND X = extension(C)(Y)

extension_union_2: LEMMA
  FORALL (Y, Z: finite_set[T2]):
    intersection(extension(C)(Y), extension(C)(Z)) =
     extension(C)(union(Y, Z))

agrega_extension_elto_correcto: LEMMA
  FORALL (a: T2, S: finite_set[finite_set[T1]]):
    es_conj_extensiones_f?(C)(S) IMPLIES
     es_conj_extensiones_f?(C)(agrega_extension_elto(C)(a, S))

agrega_extension_elto_correcto_relativo: LEMMA
  FORALL (a: T2, S: finite_set[finite_set[T1]]):
    es_conj_extensiones_relativas_f?(C)(S) AND atrib(C)(a) IMPLIES
     es_conj_extensiones_relativas_f?(C)(agrega_extension_elto(C)(a, S))

genera_extensiones_aux_correcto: LEMMA
  FORALL (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
    es_conj_extensiones_f?(C)(S) IMPLIES
     es_conj_extensiones_f?(C)(genera_extensiones_aux(C)(Y, S))
```

```
genera_extensiones_aux_correcto_relativo: LEMMA
  FORALL (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
    es_conj_extensiones_relativas_f?(C)(S) AND subset?(Y, atrib(C))
     IMPLIES
     es_conj_extensiones_relativas_f?(C)(genera_extensiones_aux(C)(Y, S))

genera_extensiones_correcto: THEOREM
  es_conj_extensiones_f?(C)(genera_extensiones(C))

genera_extensiones_correcto_relativo: THEOREM
  es_conj_extensiones_relativas_f?(C)(genera_extensiones(C))

genera_conceptos_correcto_l1: LEMMA
  FORALL (X: finite_set[T1]):
    member(X, genera_extensiones(C)) IMPLIES
     es_concepto_c?(C)(X, intencion(C)(X))

genera_conceptos_correcto_b: LEMMA
  FORALL (par: [finite_set[T1], finite_set[T2]]):
    member(par, genera_conceptos(C)) IMPLIES es_concepto_c?(C)(par)

extension_vacio: LEMMA extension(C)(emptyset[T2]) = obj(C)

genera_extensiones_aux_completo_l1: LEMMA
  FORALL (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
    subset?(S, genera_extensiones_aux(C)(Y, S))

agrega_extension_elto_completo_l1: LEMMA
  FORALL (a: T2, S: finite_set[finite_set[T1]]):
    FORALL (Z: (S)):
      member(intersection(Z, extension(C)(singleton(a))),
             agrega_extension_elto(C)(a, S))

extension_add: LEMMA
  FORALL (Z: finite_set[T2], a: T2):
    extension(C)(add(a, Z)) =
     intersection(extension(C)(singleton(a)), extension(C)(Z))

genera_extensiones_aux_completo_l2: LEMMA
  FORALL (Y: finite_set[T2], S: finite_set[finite_set[T1]]):
    FORALL (Z: finite_set[T2], D: finite_set[T1]):
      subset?(Z, Y) AND subset?(D, obj(C)) AND member(D, S) IMPLIES
       member(intersection(D, extension(C)(Z)),
              genera_extensiones_aux(C)(Y, S))

genera_extensiones_completo: THEOREM
  FORALL (Y: finite_set[T2]):
    subset?(Y, atrib(C)) IMPLIES
     member(extension(C)(Y), genera_extensiones(C))

genera_conceptos_completo: LEMMA
```

```
    FORALL (par: concepto_c(C)): member(par, genera_conceptos(C))

  correccion_genera_conceptos_c: THEOREM genera_conceptos(C) = CONCEPTOS(C)

  CONCEPTOS_TCC1: OBLIGATION
    FORALL (C): is_finite[[finite_set[T1], finite_set[T2]]](CONCEPTOS(C));

  JUDGEMENT CONCEPTOS(C) HAS_TYPE
      finite_set[[finite_set[T1], finite_set[T2]]]

  correccion_genera_conceptos_alt: THEOREM
    FORALL (par: [finite_set[T1], finite_set[T2]]):
      es_concepto_c?(C)(par) IFF member(par, genera_conceptos(C))
 END conceptos_contexto
```

E.1.3. Implicaciones entre atributos de un contexto. Semántica

```
implicaciones_atrib[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING conceptos_contexto[T1, T2]

  C: VAR FFC[T1, T2]

  implicacion_gen: TYPE =
  [# antecedente: finite_set[T2], consecuente: finite_set[T2] #]

  implicacion(C): TYPE =
  {imp: implicacion_gen |
           subset?(antecedente(imp), atrib(C)) AND
            subset?(consecuente(imp), atrib(C))}

  respeta_imp?(C)(Z: finite_set[T2], imp: implicacion(C)): bool =
      NOT subset?(antecedente(imp), Z) OR subset?(consecuente(imp), Z)

  respeta_imp_equiv: LEMMA
    FORALL (Z: finite_set[T2], imp: implicacion(C)):
      respeta_imp?(C)(Z, imp) IFF
       (subset?(antecedente(imp), Z) IMPLIES subset?(consecuente(imp), Z))

  es_valida?(C)(imp: implicacion(C)): bool =
      FORALL (ob: T1):
        member(ob, obj(C)) IMPLIES
         respeta_imp?(C)(intencion(C)(singleton(ob)), imp)
```

```
valida_prop_1_l1: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IMPLIES
     subset?(consecuente(imp),
             intencion(C)(extension(C)(antecedente(imp))))

valida_prop_1_l2: LEMMA
  FORALL (imp: implicacion(C)):
    subset?(consecuente(imp),
            intencion(C)(extension(C)(antecedente(imp))))
     IMPLIES es_valida?(C)(imp)

valida_prop_1: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IFF
     subset?(consecuente(imp),
             intencion(C)(extension(C)(antecedente(imp))))

valida_prop_1_cl: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IFF
     subset?(consecuente(imp), clausura_a(C)(antecedente(imp)))

valida_prop_2_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], a: T2, imp: implicacion(C)):
    member(a, consecuente(imp)) IMPLIES
     subset?[T2](antecedente(imp), atrib(C)) AND
      subset?[T2](singleton[T2](a), atrib(C));

valida_prop_2: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IFF
     (FORALL (a: T2):
        member(a, consecuente(imp)) IMPLIES
         es_valida?(C)
                   ((# antecedente := antecedente(imp),
                       consecuente := singleton(a) #)))

valid_prop_3_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], a: T2, imp: implicacion(C)):
    member(a, consecuente(imp)) IMPLIES
     es_concepto_c?[T1, T2]
         (C)
         (extension[T1, T2](C)(antecedente(imp)),
          intencion[T1, T2](C)(extension[T1, T2](C)(antecedente(imp))));

valid_prop_3_TCC2: OBLIGATION
  FORALL (C: FFC[T1, T2], a: T2, imp: implicacion(C)):
    member(a, consecuente(imp)) IMPLIES
     es_concepto_c?[T1, T2]
         (C)
```

```
          (extension[T1, T2](C)(singleton[T2](a)),
           intencion[T1, T2](C)(extension[T1, T2](C)(singleton[T2](a))));

valid_prop_3: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IFF
     (FORALL (a: T2):
        member(a, consecuente(imp)) IMPLIES
         subconcepto_c?(C)
                       ((extension(C)(antecedente(imp)),
                         intencion(C)(extension(C)(antecedente(imp)))),
                        (extension(C)(singleton(a)),
                         intencion(C)(extension(C)(singleton(a))))))

valid_prop_3_cl_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], a: T2, imp: implicacion(C)):
    member(a, consecuente(imp)) IMPLIES
     es_concepto_c?[T1, T2]
         (C)
         (extension[T1, T2](C)(antecedente(imp)),
          clausura_a[T1, T2](C)(antecedente(imp)));

valid_prop_3_cl_TCC2: OBLIGATION
  FORALL (C: FFC[T1, T2], a: T2, imp: implicacion(C)):
    member(a, consecuente(imp)) IMPLIES
     es_concepto_c?[T1, T2]
         (C)
         (extension[T1, T2](C)(singleton[T2](a)),
          clausura_a[T1, T2](C)(singleton[T2](a)));

valid_prop_3_cl: LEMMA
  FORALL (imp: implicacion(C)):
    es_valida?(C)(imp) IFF
     (FORALL (a: T2):
        member(a, consecuente(imp)) IMPLIES
         subconcepto_c?(C)
                       ((extension(C)(antecedente(imp)),
                         clausura_a(C)(antecedente(imp))),
                        (extension(C)(singleton(a)),
                         clausura_a(C)(singleton(a)))))

respeta_imp?(C)(Z: finite_set[T2], S: set[implicacion(C)]): bool =
    FORALL (imp: (S)): respeta_imp?(C)(Z, imp)

es_consecuencia(C)(imp: implicacion(C), S: set[implicacion(C)]): bool =
    FORALL (Y: finite_set[T2]):
      subset?(Y, atrib(C)) AND respeta_imp?(C)(Y, S) IMPLIES
       respeta_imp?(C)(Y, imp)

VALIDAS(C): set[implicacion(C)] =
    {imp: implicacion(C) | es_valida?(C)(imp)}
```

```
  MODELOS(C)(S: set[implicacion(C)]): set[finite_set[T2]] =
      {Z: finite_set[T2] | subset?(Z, atrib(C)) AND respeta_imp?(C)(Z, S)}

  modelos_validas_l1: LEMMA
    subset?(MODELOS(C)(VALIDAS(C)),
            intenciones_conj_conceptos(C)
                                    (restrict[[finite_set[T1],
                                               finite_set[T2]],
                                              concepto_c[T1, T2](C),
                                              boolean]
                                     (CONCEPTOS(C))))

  modelos_validas_l2: LEMMA
    subset?(intenciones_conj_conceptos(C)
                                    (restrict[[finite_set[T1],
                                               finite_set[T2]],
                                              concepto_c[T1, T2](C),
                                              boolean]
                                     (CONCEPTOS(C))),
            MODELOS(C)(VALIDAS(C)))

  modelos_validas: THEOREM
    MODELOS(C)(VALIDAS(C)) =
     intenciones_conj_conceptos(C)
                               (restrict[[finite_set[T1], finite_set[T2]],
                                         concepto_c[T1, T2](C),
                                         boolean]
                                    (CONCEPTOS(C)))
 END implicaciones_atrib
```

E.1.4. Base de implicaciones: base de Duquenne–Guigues

```
base_DG[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING implicaciones_atrib[T1, T2]

  IMPORTING finite_sets@finite_sets_card_eq

  d: VAR T1

  a: VAR T2

  Y, Y1, Y2: VAR finite_set[T2]
```

```
G, G1: VAR finite_set[finite_set[T2]]

C: VAR FFC

imp: VAR implicacion_gen

k: VAR nat

es_adecuado(C)(L: set[implicacion(C)]): bool =
    FORALL (imp: (L)): es_valida?(C)(imp)

es_completo(C)(L: set[implicacion(C)]): bool =
    FORALL (imp: implicacion(C)):
      es_valida?(C)(imp) IMPLIES es_consecuencia(C)(imp, L)

es_no_redundante(C)(L: set[implicacion(C)]): bool =
    FORALL (imp: (L)): NOT es_consecuencia(C)(imp, remove(imp, L))

es_cerrado(C)(L: set[implicacion(C)]): bool =
    FORALL (imp: implicacion(C)):
      es_consecuencia(C)(imp, L) IMPLIES member(imp, L)

conjunto_objetos_finito: LEMMA is_finite(powerset(obj(C)))

conjunto_atributos_finito: LEMMA is_finite(powerset(atrib(C)))

es_pseudo_intencion?_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    Y /= clausura_a(C)(Y) IMPLIES
     (FORALL (R: finite_set[T2]):
        strict_subset?(R, Y) IMPLIES card[T2](R) < card[T2](Y));

es_pseudo_intencion?(C)(Y): RECURSIVE bool =
  Y /= clausura_a(C)(Y) AND
   (FORALL (R: finite_set[T2]):
      strict_subset?(R, Y) AND es_pseudo_intencion?(C)(R) IMPLIES
       subset?(clausura_a(C)(R), Y))
   MEASURE card(Y)

pseudo_int(C): TYPE = (es_pseudo_intencion?(C))

PSEUDOINT(C): set[finite_set[T2]] =
    {Y | subset?(Y, atrib(C)) AND es_pseudo_intencion?(C)(Y)}

cns_member_PSEUDOINT_cl: LEMMA
  member(Y, PSEUDOINT(C)) IFF
   member(Y, powerset(atrib(C))) AND
    (Y /= clausura_a(C)(Y) AND
      (FORALL (R: finite_set[T2]):
         member(R, powerset(atrib(C))) AND
          strict_subset?(R, Y) AND es_pseudo_intencion?(C)(R)
          IMPLIES subset?(clausura_a(C)(R), Y)))
```

```
IMPS(C): set[implicacion(C)] =
    restrict[[# antecedente: finite_set[T2],
                consecuente: finite_set[T2] #],
             implicacion[T1, T2](C), boolean]
        (image(LAMBDA (Y):
                 (# antecedente := Y, consecuente := clausura_a(C)(Y) #))
               (PSEUDOINT(C)))

cn_pseudo_intencion_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    subset?(Y, atrib(C)) AND es_pseudo_intencion?(C)(Y) IMPLIES
     subset?[T2](Y, atrib(C)) AND
      subset?[T2]
          (intencion[T1, T2](C)(extension[T1, T2](C)(Y)), atrib(C));

cn_pseudo_intencion: LEMMA
  es_pseudo_intencion?(C)(Y) AND subset?(Y, atrib(C)) IMPLIES
   NOT respeta_imp?(C)
                   (Y,
                    (# antecedente := Y,
                       consecuente := intencion(C)(extension(C)(Y)) #))

cn_pseudo_intencion_cl_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    subset?(Y, atrib(C)) AND es_pseudo_intencion?(C)(Y) IMPLIES
     subset?[T2](Y, atrib(C)) AND
      subset?[T2](clausura_a[T1, T2](C)(Y), atrib(C));

cn_pseudo_intencion_cl: LEMMA
  es_pseudo_intencion?(C)(Y) AND subset?(Y, atrib(C)) IMPLIES
   NOT respeta_imp?(C)
                   (Y,
                    (# antecedente := Y,
                       consecuente := clausura_a(C)(Y) #))

IMPS_es_adecuado: LEMMA es_adecuado(C)(IMPS(C))

antecedente_imp: CLAIM
  imp = (# antecedente := Y1, consecuente := Y2 #) IMPLIES
   antecedente(imp) = Y1

consecuente_imp: CLAIM
  imp = (# antecedente := Y1, consecuente := Y2 #) IMPLIES
   consecuente(imp) = Y2

IMPS_es_no_redundante: LEMMA es_no_redundante(C)(IMPS(C))

IMPS_es_completo: LEMMA es_completo(C)(IMPS(C))

es_pseudo_int_rest?(C)(Y: finite_set[T2], G: finite_set[finite_set[T2]]):
bool =
```

```
    NOT subset?(clausura_a(C)(Y), Y) AND
     (FORALL (Y1: (G)):
        strict_subset?(Y1, Y) IMPLIES subset?(clausura_a(C)(Y1), Y))

pseudo_restringidas_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], G, G1: finite_set[finite_set[T2]]):
    is_finite[finite_set[T2]]
        ({Y: finite_set[T2] |
            member[finite_set[T2]](Y, G) AND
             es_pseudo_int_rest?(C)(Y, G1)});

pseudo_restringidas(C)(G, G1: finite_set[finite_set[T2]]):
finite_set[finite_set[T2]] =
    {Y: finite_set[T2] | member(Y, G) AND es_pseudo_int_rest?(C)(Y, G1)}

gen_s_pasos_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2], k: upto(card[T2](Y))):
    card[T2](Y) - k >= 0;

gen_s_pasos_TCC2: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2], k: upto(card[T2](Y))):
    is_finite[finite_set[T2]](subconjuntos[T2](Y, k));

gen_s_pasos_TCC3: OBLIGATION
  FORALL (C: FFC[T1, T2], NS: finite_set[finite_set[T2]],
          S: finite_set[finite_set[T2]], Y: finite_set[T2],
          k: upto(card[T2](Y))):
    NS = pseudo_restringidas(C)(subconjuntos(Y, k), S) AND
     NOT k = card(Y)
     IMPLIES k + 1 <= card[T2](Y);

gen_s_pasos_TCC4: OBLIGATION
  FORALL (C: FFC[T1, T2], NS: finite_set[finite_set[T2]],
          S: finite_set[finite_set[T2]], Y: finite_set[T2],
          k: upto(card[T2](Y))):
    NS = pseudo_restringidas(C)(subconjuntos(Y, k), S) AND
     NOT k = card(Y)
     IMPLIES card[T2](Y) - (k + 1) < card[T2](Y) - k;

gen_s_pasos(C)
           (Y: finite_set[T2], k: upto(card(Y)),
            S: finite_set[finite_set[T2]]):
RECURSIVE set[finite_set[T2]] =
  LET NS = pseudo_restringidas(C)(subconjuntos(Y, k), S) IN
    IF k = card(Y) THEN union(S, NS)
    ELSE gen_s_pasos(C)(Y, k + 1, union(S, NS))
    ENDIF
   MEASURE card(Y) - k

gen_s_TCC1: OBLIGATION FORALL (C: FFC[T1, T2]): 0 <= card[T2](atrib(C));

gen_s(C): set[finite_set[T2]] = gen_s_pasos(C)(atrib(C), 0, emptyset)
```

```
gen_s_pasos_subset_lema1: LEMMA
  FORALL (A: finite_set[T2], k: nat):
    is_finite[finite_set[T2]]
        ({X: finite_set[T2] |
            subset?[T2](X, atrib(C)) AND
             es_pseudo_intencion?(C)(X) AND
              subset?[T2](X, A) AND card[T2](X) < k})

pseudo_card_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2], (k: nat)):
    is_finite[finite_set[T2]]
        ({X: finite_set[T2] |
            subset?[T2](X, atrib(C)) AND
             es_pseudo_intencion?(C)(X) AND
              subset?[T2](X, Y) AND card[T2](X) < k});

pseudo_card(C, Y, (k: nat)): finite_set[finite_set[T2]] =
    {X: finite_set[T2] |
       subset?[T2](X, atrib(C)) AND
        es_pseudo_intencion?(C)(X) AND
         subset?[T2](X, Y) AND card[T2](X) < k}

JUDGEMENT pseudo_card(C, Y, k) HAS_TYPE finite_set[finite_set[T2]]

pseudo_card_0: CLAIM pseudo_card(C, Y, 0) = emptyset

pseudo_restringidas_card: LEMMA
  FORALL (A: finite_set[T2], k: upto(card(A))):
    subset?(A, atrib(C)) AND k /= card(A) IMPLIES
     union(pseudo_card(C, A, k),
           pseudo_restringidas(C)
                               (subconjuntos(A, k), pseudo_card(C, A, k)))
      = pseudo_card(C, A, 1 + k)

S_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], (k: nat)):
    is_finite[finite_set[T2]]
        ({X: finite_set[T2] |
            subset?[T2](X, atrib(C)) AND
             es_pseudo_intencion?(C)(X) AND card[T2](X) <= k});

S(C: FFC, (k: nat)): finite_set[finite_set[T2]] =
    {X: finite_set[T2] |
       subset?[T2](X, atrib(C)) AND
        es_pseudo_intencion?(C)(X) AND card[T2](X) <= k}

S_pseudo_card: LEMMA
  FORALL (C: FFC, k: upto(card(atrib(C)))):
    S(C, k) = pseudo_card(C, atrib(C), 1 + k)

S_pseudo_restringidas_TCC1: OBLIGATION
```

```
  FORALL (C: FFC[T1, T2], k: below(card[T2](atrib(C)))):
    1 + k /= card(atrib(C)) IMPLIES
     is_finite[finite_set[T2]](subconjuntos[T2](atrib(C), 1 + k));

S_pseudo_restringidas: LEMMA
  FORALL (C: FFC, k: below(card(atrib(C)))):
    1 + k /= card(atrib(C)) IMPLIES
     union(S(C, k),
           pseudo_restringidas(C)
                               (subconjuntos(atrib(C), 1 + k), S(C, k)))
      = S(C, 1 + k)

gen_s_pasos_correcto_l0: LEMMA
  FORALL (Y: finite_set[T2], k: upto(card(Y)),
          S: finite_set[finite_set[T2]], Z: finite_set[T2]):
    subset?(Y, atrib(C)) AND
     S = pseudo_card(C, Y, k) AND member(Z, gen_s_pasos(C)(Y, k, S))
     IMPLIES subset?(Z, Y) AND es_pseudo_intencion?(C)(Z)

gen_s_correcto: LEMMA
  FORALL (X: finite_set[T2]):
    member(X, gen_s(C)) IMPLIES es_pseudo_intencion?(C)(X)

gen_s_correcto_b: LEMMA
  FORALL (X: finite_set[T2]):
    member(X, gen_s(C)) IMPLIES subset?(X, atrib(C))

gen_s_pasos_completo_l4: LEMMA
  FORALL (Y: finite_set[T2], k: upto(card(Y)),
          S: finite_set[finite_set[T2]], Z: finite_set[T2]):
         subset?(Y, atrib(C)) AND subset?(Z, atrib(C))
     AND S = pseudo_card(C, Y, k) AND subset?(Z, Y) AND card(Z) <= k
     AND NOT member(Z, gen_s_pasos(C)(Y, k, S))
     IMPLIES NOT es_pseudo_intencion?(C)(Z)

gen_s_pasos_subset_k_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2], k: upto(card[T2](Y))):
    k /= card(Y) AND subset?(Y, atrib(C)) IMPLIES k + 1 <= card[T2](Y);

gen_s_pasos_subset_k: LEMMA
  FORALL (Y: finite_set[T2], k: upto(card(Y))):
    subset?(Y, atrib(C)) AND k /= card(Y) IMPLIES
     subset?(gen_s_pasos(C)(Y, k + 1, pseudo_card(C, Y, k + 1)),
             gen_s_pasos(C)(Y, k, pseudo_card(C, Y, k)))

gen_s_pasos_subset_TCC1: OBLIGATION
  FORALL (C: FFC[T1, T2], Y: finite_set[T2]):
    card(Y) /= 0 AND subset?(Y, atrib(C)) IMPLIES 0 <= card[T2](Y);

gen_s_pasos_subset: LEMMA
  FORALL (Y: finite_set[T2], k: upto(card(Y))):
    subset?(Y, atrib(C)) AND card(Y) /= 0 IMPLIES
```

```
      subset?(gen_s_pasos(C)(Y, k, pseudo_card(C, Y, k)),
              gen_s_pasos(C)(Y, 0, emptyset))

  gen_s_completo: LEMMA
    FORALL (X: finite_set[T2]):
      subset?(X, atrib(C)) AND es_pseudo_intencion?(C)(X) IMPLIES
       member(X, gen_s(C))

  correccion_gen_s: THEOREM gen_s(C) = PSEUDOINT(C)

  genera_imps_g(C): set[implicacion_gen] =
      LET fun =
            LAMBDA (Y: finite_set[T2]):
              (# antecedente := Y, consecuente := clausura_a(C)(Y) #)
        IN image(fun)(gen_s(C))

  genera_imps(C): set[implicacion(C)] =
      restrict[[# antecedente: finite_set[T2],
                  consecuente: finite_set[T2] #],
               implicacion[T1, T2](C), boolean]
          (LET fun =
                 LAMBDA (Y: finite_set[T2]):
                   (# antecedente := Y, consecuente := clausura_a(C)(Y) #)
             IN image(fun)(gen_s(C)))

  genera_imps_igual_genera_imps_g: LEMMA
    genera_imps(C) =
     restrict[implicacion_gen[T1, T2], implicacion[T1, T2](C), boolean]
         (genera_imps_g(C))

  genera_imp_correcto: LEMMA subset?(genera_imps(C), IMPS(C))

  genera_imp_completo: LEMMA subset?(IMPS(C), genera_imps(C))

  correccion_genera_imp: THEOREM genera_imps(C) = IMPS(C)

  correccion_genera_imp_g: COROLLARY
    IMPS(C) =
     restrict[implicacion_gen[T1, T2], implicacion[T1, T2](C), boolean]
         (genera_imps_g(C))
 END base_DG
```

E.2. Formalización evaluable, usando listas

E.2.1. Contextos formales finitos

```
contextos_formales_ref[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
```

```
 T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

 T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
ENDASSUMING

REFINAMIENTO: LIBRARY = "../refinamiento"

IMPORTING REFINAMIENTO@refinamiento_proyecciones

IMPORTING REFINAMIENTO@refinamiento_conj_finitos_gen

IMPORTING REFINAMIENTO@refinamiento_comp_oper

IMPORTING REFINAMIENTO@propiedades_operaciones_listas_via_ref

IMPORTING contextos_formales

IMPORTING TL@listas_prop_1

FFCT_rep: TYPE =
[# obj_rep: list[T1],
   atrib_rep: (cons?[T2]),
   relacion_rep: list[[T1, T2]] #]

FFC_rep: TYPE =
{R: FFCT_rep |
         LET Re_rep = relacion_rep(R) IN
           every(LAMBDA (par: [T1, T2]):
                   member(par`1, obj_rep(R)) AND
                    member(par`2, atrib_rep(R)))
                (Re_rep)}

JUDGEMENT FFC_rep SUBTYPE_OF FFCT_rep

trans(R: FFCT_rep): FFCT[T1, T2] =
    (# obj := c(id[T1])(obj_rep(R)),
       atrib := c(id[T2])(atrib_rep(R)),
       relacion := c(id[[T1, T2]])(relacion_rep(R)) #)

d, d1, d2: VAR T1

a, a1, a2: VAR T2

C, C1, C2: VAR FFC[T1, T2]

R, R1, R2: VAR FFC_rep

trans_es_ref_FFCT: LEMMA es_refinamiento?[FFCT[T1, T2], FFCT_rep](trans)

trans_TCC1: OBLIGATION
  FORALL (R: FFC_rep), (par: (relacion(trans(R)))):
    obj(trans(R))(par`1) AND atrib(trans(R))(par`2);
```

```
JUDGEMENT trans(R: FFC_rep) HAS_TYPE FFC[T1, T2]

trans_TCC2: OBLIGATION es_refinamiento?[FFCT[T1, T2], FFCT_rep](trans);

JUDGEMENT trans HAS_TYPE (es_refinamiento?[FFCT[T1, T2], FFCT_rep])

trans_es_ref_FFC: LEMMA
  es_refinamiento?[FFC[T1, T2], FFC_rep]
      (restrict[FFCT_rep, FFC_rep, FFCT[T1, T2]](trans))

trans_f(R: FFC_rep): FFC[T1, T2] = trans(R)

trans_f_es_ref_FFC: LEMMA es_refinamiento?[FFC[T1, T2], FFC_rep](trans_f)

trans_f_TCC1: OBLIGATION es_refinamiento?[FFC[T1, T2], FFC_rep](trans_f);

JUDGEMENT trans_f HAS_TYPE (es_refinamiento?[FFC[T1, T2], FFC_rep])

atrib_trans_f: LEMMA c(id[T2])(atrib_rep(R)) = atrib(trans_f(R))

obj_trans_f: LEMMA c(id[T1])(obj_rep(R)) = obj(trans_f(R))

intencion_rep_aux_TCC1: OBLIGATION
  FORALL (a: T2, ls: list[T2], R: FFC_rep, l1: list[T1], l2: list[T2]):
    l2 = cons(a, ls) AND
     every(LAMBDA (d: T1): member((d, a), relacion_rep(R)))(l1)
     IMPLIES length[T2](ls) < length[T2](l2);

intencion_rep_aux_TCC2: OBLIGATION
  FORALL (a: T2, ls: list[T2], R: FFC_rep, l1: list[T1], l2: list[T2]):
    l2 = cons(a, ls) AND
     NOT every(LAMBDA (d: T1): member((d, a), relacion_rep(R)))(l1)
     IMPLIES length[T2](ls) < length[T2](l2);

intencion_rep_aux(R)(l1: list[T1], l2: list[T2]): RECURSIVE list[T2] =
  CASES l2
    OF null: null,
       cons(a, ls):
         IF every(LAMBDA (d: T1): member((d, a), relacion_rep(R)))(l1)
           THEN cons(a, intencion_rep_aux(R)(l1, ls))
         ELSE intencion_rep_aux(R)(l1, ls)
         ENDIF
    ENDCASES
   MEASURE length(l2)

intencion_rep_aux_caract_1: CLAIM
  FORALL (l1: list[T1], l2: list[T2]):
    member(a, intencion_rep_aux(R)(l1, l2)) IMPLIES member(a, l2)

intencion_rep_aux_null: CLAIM
  FORALL (l2: list[T2]): intencion_rep_aux(R)(null, l2) = l2
```

```
intencion_rep(R)(X: list[T1]): list[T2] =
    intencion_rep_aux(R)(X, atrib_rep(R))

intencion_rep_null: LEMMA intencion_rep(R)(null) = atrib_rep(R)

intencion_rep_aux_caract_2: LEMMA
  FORALL (l1: list[T1], l2: list[T2]):
    member(a, intencion_rep_aux(R)(l1, l2)) IMPLIES
     every(LAMBDA d: member((d, a), relacion_rep(R)))(l1)

intencion_rep_aux_caract_3: LEMMA
  FORALL (l1: list[T1], l2: list[T2]):
    every(LAMBDA d: member((d, a), relacion_rep(R)))(l1) AND
     member(a, l2)
     IMPLIES member(a, intencion_rep_aux(R)(l1, l2))

intencion_rep_refinamiento: THEOREM
  es_refinamiento_op?[finite_set[T1], finite_set[T2], list[T1], list[T2],
                      c(id[T1]), c(id[T2])]
      (intencion(trans_f(R)), intencion_rep(R))

intencion_rep_refinamiento_corol: COROLLARY
  FORALL (ls: list[T1]):
    intencion[T1, T2](trans_f(R))(c(id[T1])(ls)) =
     c(id[T2])(intencion_rep(R)(ls))

extension_rep_aux_TCC1: OBLIGATION
  FORALL (d: T1, ls: list[T1], R: FFC_rep, l1: list[T1], l2: list[T2]):
    l1 = cons(d, ls) AND
     every(LAMBDA (a: T2): member((d, a), relacion_rep(R)))(l2)
     IMPLIES length[T1](ls) < length[T1](l1);

extension_rep_aux_TCC2: OBLIGATION
  FORALL (d: T1, ls: list[T1], R: FFC_rep, l1: list[T1], l2: list[T2]):
    l1 = cons(d, ls) AND
     NOT every(LAMBDA (a: T2): member((d, a), relacion_rep(R)))(l2)
     IMPLIES length[T1](ls) < length[T1](l1);

extension_rep_aux(R)(l1: list[T1], l2: list[T2]): RECURSIVE list[T1] =
  CASES l1
    OF null: null,
       cons(d, ls):
         IF every(LAMBDA (a: T2): member((d, a), relacion_rep(R)))(l2)
           THEN cons(d, extension_rep_aux(R)(ls, l2))
         ELSE extension_rep_aux(R)(ls, l2)
         ENDIF
    ENDCASES
   MEASURE length(l1)

extension_rep_aux_caract_1: CLAIM
  FORALL (l1: list[T1], l2: list[T2]):
```

```
    member(d, extension_rep_aux(R)(l1, l2)) IMPLIES member(d, l1)

extension_rep_aux_null: LEMMA
  FORALL (l1: list[T1]): extension_rep_aux(R)(l1, null) = l1

extension_rep(R)(Y: list[T2]): list[T1] =
    extension_rep_aux(R)(obj_rep(R), Y)

extension_rep_null: LEMMA extension_rep(R)(null) = obj_rep(R)

extension_rep_aux_caract_2: LEMMA
  FORALL (l1: list[T1], l2: list[T2]):
    member(d, extension_rep_aux(R)(l1, l2)) IMPLIES
     every(LAMBDA a: member((d, a), relacion_rep(R)))(l2)

extension_rep_aux_caract_3: LEMMA
  FORALL (l1: list[T1], l2: list[T2]):
    every(LAMBDA a: member((d, a), relacion_rep(R)))(l2) AND
     member(d, l1)
     IMPLIES member(d, extension_rep_aux(R)(l1, l2))

extension_rep_refinamiento: THEOREM
  es_refinamiento_op?[finite_set[T2], finite_set[T1], list[T2], list[T1],
                      c(id[T2]), c(id[T1])]
      (extension(trans_f(R)), extension_rep(R))

extension_rep_refinamiento_corol: COROLLARY
  FORALL (ls: list[T2]):
    extension[T1, T2](trans_f(R))(c(id[T2])(ls)) =
     c(id[T1])(extension_rep(R)(ls))

clausura_o_rep(R)(X: list[T1]): list[T1] =
    extension_rep(R)(intencion_rep(R)(X))

clausura_a_rep(R)(Y: list[T2]): list[T2] =
    intencion_rep(R)(extension_rep(R)(Y))

clausura_o_rep_refinamiento: THEOREM
  es_refinamiento_op?[finite_set[T1], finite_set[T1], list[T1], list[T1],
                      c(id[T1]), c(id[T1])]
      (clausura_o(trans_f(R)), clausura_o_rep(R))

clausura_a_rep_refinamiento: THEOREM
  es_refinamiento_op?[finite_set[T2], finite_set[T2], list[T2], list[T2],
                      c(id[T2]), c(id[T2])]
      (clausura_a(trans_f(R)), clausura_a_rep(R))

es_concepto_c_rep?(R)(par: [list[T1], list[T2]]): bool =
    LET (l1, l2) = par IN
      sublista?[T1](l1, obj_rep(R)) AND
       sublista?[T2](l2, atrib_rep(R)) AND
        igual_l?[T2](intencion_rep(R)(l1), l2) AND
```

```
          igual_l?[T1](extension_rep(R)(l2), l1)

  es_concepto_c_rep?_es_refinamiento: LEMMA
    es_refinamiento_op?[[finite_set[T1], finite_set[T2]], bool,
                        [list[T1], list[T2]], bool,
                        prod_cart(c(id[T1]), c(id[T2])), id[bool]]
        (es_concepto_c?(trans_f(R)), es_concepto_c_rep?(R))

  es_concepto_c_rep?_es_refinamiento_corol: COROLLARY
    FORALL (par: [list[T1], list[T2]]):
      es_concepto_c_rep?(R)(par) IFF
       es_concepto_c?(trans_f(R))(prod_cart(c(id[T1]), c(id[T2]))(par))

  concepto_c_no_vacio_rep: LEMMA
    es_concepto_c_rep?(R)
                      (extension_rep(R)(atrib_rep(R)),
                       intencion_rep(R)(extension_rep(R)(atrib_rep(R))))

  concepto_c_no_vacio_rep_2: LEMMA
    es_concepto_c_rep?(R)
                      (extension_rep(R)(atrib_rep(R)),
                       clausura_a_rep(R)(atrib_rep(R)))

  concepto_c_rep_TCC1: OBLIGATION
    FORALL (R: FFC_rep):
      EXISTS (x: {x: [list[T1], list[T2]] | es_concepto_c_rep?(R)(x)}):
        TRUE;

  concepto_c_rep(R): TYPE+ = (es_concepto_c_rep?(R))

  concepto_c_no_vacio_rep_coro: COROLLARY
    EXISTS (x: concepto_c_rep(R)): TRUE

  concepto_c_rep_refinamiento_tipo_TCC1: OBLIGATION
    FORALL (R: FFC_rep, x1: concepto_c_rep(R)):
      es_concepto_c?[T1, T2]
          (trans_f(R))
          (prod_cart[finite_set[T1], finite_set[T2], list[T1], list[T2]]
               (c[T1, T1](id[T1]), c[T2, T2](id[T2]))(x1));

  concepto_c_rep_refinamiento_tipo: LEMMA
    es_refinamiento?[concepto_c(trans_f(R)), concepto_c_rep(R)]
        (restrict[[list[T1], list[T2]], concepto_c_rep(R),
                  [finite_set[T1], finite_set[T2]]]
            (prod_cart(c(id[T1]), c(id[T2]))))

  trans_concepto(R)(par: concepto_c_rep(R)): concepto_c(trans_f(R)) =
      prod_cart(c(id[T1]), c(id[T2]))(par)

  concepto_c_rep_refinamiento_tipo_2: LEMMA
    es_refinamiento?[concepto_c(trans_f(R)), concepto_c_rep(R)]
        (trans_concepto(R))
```

```
igual_concepto_rep_e(R)(par1, par2: [list[T1], list[T2]]): bool =
    igual_l?(PROJ_1(par1), PROJ_1(par2)) AND
     igual_l?(PROJ_2(par1), PROJ_2(par2))

igual_concepto_rep_e_igual?: LEMMA
  igual_concepto_rep_e(R) = igual?(prod_cart(c(id[T1]), c(id[T2])))

intencion_concepto_rep(R)(par: concepto_c_rep(R)): list[T2] = PROJ_2(par)

extension_concepto_rep(R)(par: concepto_c_rep(R)): list[T1] = PROJ_1(par)

intencion_concepto_rep_es_refinamiento_2_TCC1: OBLIGATION
  FORALL (R: FFC_rep):
    es_refinamiento?[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
        (trans_concepto(R));

intencion_concepto_rep_es_refinamiento_2: LEMMA
  es_refinamiento_op?[concepto_c(trans_f(R)), finite_set[T2],
                      concepto_c_rep(R), list[T2], trans_concepto(R),
                      c[T2, T2](id[T2])]
      (intencion_concepto(trans_f(R)), intencion_concepto_rep(R))

intencion_concepto_rep_es_refinamiento_corol: COROLLARY
  FORALL (par: concepto_c_rep(R)):
    c(id[T2])(intencion_concepto_rep(R)(par)) =
     intencion_concepto(trans_f(R))(trans_concepto(R)(par))

extension_concepto_rep_es_refinamiento_2: LEMMA
  es_refinamiento_op?[concepto_c(trans_f(R)), finite_set[T1],
                      concepto_c_rep(R), list[T1], trans_concepto(R),
                      c[T1, T1](id[T1])]
      (extension_concepto(trans_f(R)), extension_concepto_rep(R))

extension_concepto_rep_es_refinamiento_corol: COROLLARY
  FORALL (par: concepto_c_rep(R)):
    c(id[T1])(extension_concepto_rep(R)(par)) =
     extension_concepto(trans_f(R))(trans_concepto(R)(par))

subconcepto_c_rep?(R)(par1, par2: concepto_c_rep(R)): bool =
    sublista?[T1]
        (extension_concepto_rep(R)(par1), extension_concepto_rep(R)(par2))

subconcepto_c_rep_es_refinamiento_TCC1: OBLIGATION
  FORALL (R: FFC_rep):
    es_refinamiento?[[concepto_c[T1, T2](trans_f(R)),
                      concepto_c[T1, T2](trans_f(R))],
                     [concepto_c_rep(R), concepto_c_rep(R)]]
        (prod_cart[concepto_c[T1, T2](trans_f(R)),
                   concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R),
                   concepto_c_rep(R)]
             (trans_concepto(R), trans_concepto(R)));
```

```
subconcepto_c_rep_es_refinamiento: LEMMA
  es_refinamiento_op?[[concepto_c(trans_f(R)), concepto_c(trans_f(R))],
                      bool, [concepto_c_rep(R), concepto_c_rep(R)], bool,
                      prod_cart(trans_concepto(R), trans_concepto(R)),
                      id[bool]]
      (subconcepto_c?(trans_f(R)), subconcepto_c_rep?(R))

subconcepto_c_rep_es_refinamiento_corol: COROLLARY
  FORALL (par1, par2: concepto_c_rep(R)):
    subconcepto_c_rep?(R)(par1, par2) IFF
     subconcepto_c?(trans_f(R))
                   (trans_concepto(R)(par1), trans_concepto(R)(par2))

intenciones_conj_conceptos_rep(R)(LL: list[concepto_c_rep(R)]):
list[list[T2]] = map(intencion_concepto_rep(R))(LL)

intenciones_conj_conceptos_rep_es_refinamiento_2_TCC1: OBLIGATION
  FORALL (R: FFC_rep):
    es_refinamiento?[finite_set[concepto_c[T1, T2](trans_f(R))],
                     list[concepto_c_rep(R)]]
        (c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
             (trans_concepto(R)));

intenciones_conj_conceptos_rep_es_refinamiento_2_TCC2: OBLIGATION
  es_refinamiento?[finite_set[finite_set[T2]], list[list[T2]]]
      (c[finite_set[T2], list[T2]](c[T2, T2](id[T2])));

intenciones_conj_conceptos_rep_es_refinamiento_2: LEMMA
  es_refinamiento_op?[finite_set[concepto_c(trans_f(R))],
                      finite_set[finite_set[T2]],
                      list[concepto_c_rep(R)], list[list[T2]],
                      c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                          (trans_concepto(R)),
                      c(c[T2, T2](id[T2]))]
      (intenciones_conj_conceptos(trans_f(R)),
       intenciones_conj_conceptos_rep(R))

intenciones_conj_conceptos_rep_es_refinamiento_2_corol: COROLLARY
  FORALL (LL: list[concepto_c_rep(R)]):
    c(c[T2, T2](id[T2]))(intenciones_conj_conceptos_rep(R)(LL)) =
     intenciones_conj_conceptos(trans_f(R))
                                (c[concepto_c[T1, T2](trans_f(R)),
                                   concepto_c_rep(R)]
                                     (trans_concepto(R))(LL))

extensiones_conj_conceptos_rep(R)(LL: list[concepto_c_rep(R)]):
list[list[T1]] = map(extension_concepto_rep(R))(LL)

extensiones_conj_conceptos_rep_es_refinamiento_2_TCC1: OBLIGATION
  es_refinamiento?[finite_set[finite_set[T1]], list[list[T1]]]
      (c[finite_set[T1], list[T1]](c[T1, T1](id[T1])));
```

```
extensiones_conj_conceptos_rep_es_refinamiento_2: LEMMA
  es_refinamiento_op?[finite_set[concepto_c(trans_f(R))],
                      finite_set[finite_set[T1]],
                      list[concepto_c_rep(R)], list[list[T1]],
                      c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                          (trans_concepto(R)),
                      c(c[T1, T1](id[T1]))]
      (extensiones_conj_conceptos(trans_f(R)),
       extensiones_conj_conceptos_rep(R))

extensiones_conj_conceptos_rep_es_refinamiento_2_corol: COROLLARY
  FORALL (LL: list[concepto_c_rep(R)]):
    c(c[T1, T1](id[T1]))(extensiones_conj_conceptos_rep(R)(LL)) =
     extensiones_conj_conceptos(trans_f(R))
                                (c[concepto_c[T1, T2](trans_f(R)),
                                   concepto_c_rep(R)]
                                  (trans_concepto(R))(LL))

infimo_rep_l1_TCC1: OBLIGATION
  FORALL (R: FFC_rep, LL: (cons?[concepto_c_rep(R)])):
    cons?[list[T1]](extensiones_conj_conceptos_rep(R)(LL));

infimo_rep_l1: LEMMA
  FORALL (LL: (cons?[concepto_c_rep(R)])):
    es_concepto_c_rep?(R)
                      (Inter(extensiones_conj_conceptos_rep(R)(LL)),
                       intencion_rep(R)
                                    (extension_rep
                                     (R)
                                     (Append[T2]
                                      (intenciones_conj_conceptos_rep
                                       (R)(LL)))))

infimo_rep_l2: LEMMA
  FORALL (LL: (cons?[concepto_c_rep(R)])):
    es_concepto_c_rep?(R)
                      (Inter(extensiones_conj_conceptos_rep(R)(LL)),
                       clausura_a_rep(R)
                                     (Append[T2]
                                      (intenciones_conj_conceptos_rep
                                       (R)(LL))))

infimo_rep_TCC1: OBLIGATION
  FORALL (R: FFC_rep, LL: (cons?[concepto_c_rep(R)])):
    es_concepto_c_rep?(R)
                      (Inter[T1](extensiones_conj_conceptos_rep(R)(LL)),
                       clausura_a_rep(R)
                                     (Append[T2]
                                      (intenciones_conj_conceptos_rep
                                       (R)(LL))));
```

```
infimo_rep(R)(LL: (cons?[concepto_c_rep(R)])): concepto_c_rep(R) =
    (Inter(extensiones_conj_conceptos_rep(R)(LL)),
     clausura_a_rep(R)(Append(intenciones_conj_conceptos_rep(R)(LL))))

infimo_comp_l1_TCC1: OBLIGATION
  FORALL (R: FFC_rep,
          x1: non_empty_finite_set[concepto_c[T1, T2](trans_f(R))]):
    NOT empty?[finite_set[T1]]
            (extensiones_conj_conceptos[T1, T2](trans_f(R))(x1));

infimo_comp_l1_TCC2: OBLIGATION
  FORALL (R: FFC_rep):
    (FORALL (x1: ((cons?[concepto_c_rep(R)]))):
       NOT empty?[concepto_c[T1, T2](trans_f(R))]
               (c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                    (trans_concepto(R))(x1)))
     AND
     es_refinamiento?[non_empty_finite_set[concepto_c[T1,
                                                      T2](trans_f(R))],
                      ((cons?[concepto_c_rep(R)]))]
         (restrict[list[concepto_c_rep(R)], ((cons?[concepto_c_rep(R)])),
                   finite_set[concepto_c[T1, T2](trans_f(R))]]
              (c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                   (trans_concepto(R))));

infimo_comp_l1: LEMMA
  es_refinamiento_op?[non_empty_finite_set[concepto_c(trans_f(R))],
                      finite_set[T1], (cons?[concepto_c_rep(R)]),
                      list[T1],
                      restrict[list[concepto_c_rep(R)],
                               ((cons?[concepto_c_rep(R)])),
                               finite_set[concepto_c[T1, T2](trans_f(R))]]
                          (c[concepto_c[T1, T2](trans_f(R)),
                             concepto_c_rep(R)]
                               (trans_concepto(R))),
                      c(id[T1])]
      (o[non_empty_finite_set[concepto_c(trans_f(R))],
         non_empty_finite_set[finite_set[T1]], finite_set[T1]]
           (intersection_f,
            restrict[finite_set[concepto_c[T1, T2](trans_f(R))],
                     non_empty_finite_set[concepto_c(trans_f(R))],
                     finite_set[finite_set[T1]]]
                (extensiones_conj_conceptos(trans_f(R)))),
       o[(cons?[concepto_c_rep(R)]), (cons?[list[T1]]), list[T1]]
           (Inter,
            restrict[list[concepto_c_rep(R)],
                     ((cons?[concepto_c_rep(R)])), list[list[T1]]]
                (extensiones_conj_conceptos_rep(R))))

infimo_comp_l1_corol_TCC1: OBLIGATION
  FORALL (R: FFC_rep, LL: (cons?[concepto_c_rep(R)])):
    NOT empty?[finite_set[T1]]
```

```
            (extensiones_conj_conceptos[T1, T2]
                 (trans_f(R))
                 (c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                      (trans_concepto(R))(LL)));

infimo_comp_l1_corol: COROLLARY
  FORALL (LL: (cons?[concepto_c_rep(R)])):
    intersection_f(extensiones_conj_conceptos(trans_f(R))
                                            (c[concepto_c[T1, T2]
                                               (trans_f(R)),
                                               concepto_c_rep(R)]
                                             (trans_concepto(R))(LL)))
     = c(id[T1])(Inter(extensiones_conj_conceptos_rep(R)(LL)))

infimo_comp_l2: LEMMA
  es_refinamiento_op?[finite_set[concepto_c(trans_f(R))], finite_set[T2],
                      list[concepto_c_rep(R)], list[T2],
                      c[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
                          (trans_concepto(R)),
                      c(id[T2])]
      (o[finite_set[concepto_c(trans_f(R))], finite_set[finite_set[T2]],
         finite_set[T2]]
           (union_f, intenciones_conj_conceptos(trans_f(R))),
       o[list[concepto_c_rep(R)], list[list[T2]], list[T2]]
           (Append, intenciones_conj_conceptos_rep(R)))

infimo_comp_l2_corol: COROLLARY
  FORALL (LL: list[concepto_c_rep(R)]):
    union_f(intenciones_conj_conceptos(trans_f(R))
                                      (c[concepto_c[T1, T2](trans_f(R)),
                                         concepto_c_rep(R)]
                                       (trans_concepto(R))(LL)))
     = c(id[T2])(Append(intenciones_conj_conceptos_rep(R)(LL)))

infimo_rep_es_refinamiento: LEMMA
  es_refinamiento_op?[non_empty_finite_set[concepto_c[T1, T2]
                                           (trans_f(R))],
                      concepto_c[T1, T2](trans_f(R)),
                      (cons?[concepto_c_rep(R)]), concepto_c_rep(R),
                      restrict[list[concepto_c_rep(R)],
                               ((cons?[concepto_c_rep(R)])),
                               finite_set[concepto_c[T1, T2](trans_f(R))]]
                          (c[concepto_c[T1, T2](trans_f(R)),
                             concepto_c_rep(R)]
                               (trans_concepto(R))),
                      trans_concepto(R)]
      (infimo(trans_f(R)), infimo_rep(R))

supremo_rep_lema_1_TCC1: OBLIGATION
  FORALL (R: FFC_rep, LL: (cons?[concepto_c_rep(R)])):
    cons?[list[T2]](intenciones_conj_conceptos_rep(R)(LL));
```

```
  supremo_rep_lema_1: LEMMA
    FORALL (LL: (cons?[concepto_c_rep(R)])):
      es_concepto_c_rep?(R)
                        (extension_rep(R)
                                      (intencion_rep
                                       (R)
                                       (Append
                                        (extensiones_conj_conceptos_rep
                                         (R)(LL)))),
                         Inter(intenciones_conj_conceptos_rep(R)(LL)))

  supremo_rep_lema_2: LEMMA
    FORALL (LL: (cons?[concepto_c_rep(R)])):
      es_concepto_c_rep?(R)
                        (clausura_o_rep(R)
                                       (Append
                                        (extensiones_conj_conceptos_rep
                                         (R)(LL))),
                         Inter(intenciones_conj_conceptos_rep(R)(LL)))

  supremo_rep_TCC1: OBLIGATION
    FORALL (R: FFC_rep, LL: (cons?[concepto_c_rep(R)])):
      es_concepto_c_rep?(R)
                        (clausura_o_rep(R)
                                       (Append[T1]
                                        (extensiones_conj_conceptos_rep
                                         (R)(LL))),
                         Inter[T2](intenciones_conj_conceptos_rep(R)(LL)));

  supremo_rep(R)(LL: (cons?[concepto_c_rep(R)])): concepto_c_rep(R) =
      (clausura_o_rep(R)(Append(extensiones_conj_conceptos_rep(R)(LL))),
       Inter(intenciones_conj_conceptos_rep(R)(LL)))

  supremo_rep_es_refinamiento_TCC1: OBLIGATION
    FORALL (R: FFC_rep):
      es_refinamiento?[non_empty_finite_set[concepto_c[T1,
                                                       T2](trans_f(R))],
                       ((cons?[concepto_c_rep(R)]))]
          (cnv[concepto_c[T1, T2](trans_f(R)), concepto_c_rep(R)]
               (trans_concepto(R)));

  supremo_rep_es_refinamiento: THEOREM
    es_refinamiento_op?[non_empty_finite_set[concepto_c[T1, T2]
                                             (trans_f(R))],
                        concepto_c[T1, T2](trans_f(R)),
                        (cons?[concepto_c_rep(R)]), concepto_c_rep(R),
                        cnv[concepto_c[T1, T2](trans_f(R)),
                            concepto_c_rep(R)]
                            (trans_concepto(R)),
                        trans_concepto(R)]
        (supremo(trans_f(R)), supremo_rep(R))
END contextos_formales_ref
```

E.2.2. Cálculo de los conceptos de un contexto

```
conceptos_contexto_ref[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING contextos_formales_ref

  IMPORTING conceptos_contexto

  IMPORTING finite_sets@finite_sets_eq

  IMPORTING REFINAMIENTO@propiedades_operaciones_listas_via_ref

  C, C1, C2: VAR FFC[T1, T2]

  R, R1, R2: VAR FFC_rep[T1, T2]

  d, d1, d2: VAR T1

  a, a1, a2: VAR T2

  LL: VAR list[[list[T1], list[T2]]]

  agrega_extension_elto_rep(R)(a: T2, LL: list[list[T1]]): list[list[T1]] =
      LET fun = LAMBDA (ls: list[T1]): inter(ls, extension_rep(R)((: a :)))
        IN append(LL, map(fun)(LL))

  genera_extensiones_aux_rep_TCC1: OBLIGATION
    FORALL (a: T2, l1: list[T2], R: FFC_rep[T1, T2], l2: list[T2]):
      l2 = cons(a, l1) IMPLIES cons?[T2](l2);

  genera_extensiones_aux_rep_TCC2: OBLIGATION
    FORALL (a: T2, l1: list[T2], R: FFC_rep[T1, T2], l2: list[T2]):
      l2 = cons(a, l1) IMPLIES length[T2](rest_l[T2](l2)) < length[T2](l2);

  genera_extensiones_aux_rep(R)(l2: list[T2], LL: list[list[T1]]):
  RECURSIVE list[list[T1]] =
    CASES l2
      OF null: LL,
         cons(a, l1):
           genera_extensiones_aux_rep(R)
                                     (rest_l[T2](l2),
                                      agrega_extension_elto_rep
                                      (R)(car(l2), LL))
      ENDCASES
     MEASURE length(l2)
```

```
genera_extensiones_rep(R): list[list[T1]] =
    genera_extensiones_aux_rep(R)(atrib_rep(R), (: obj_rep(R) :))

genera_conceptos_rep(R): list[[list[T1], list[T2]]] =
    LET fun = LAMBDA (ls: list[T1]): (ls, intencion_rep(R)(ls)) IN
      map(fun)(genera_extensiones_rep(R))

es_conj_todos_conceptos_e_rep?(R: FFC_rep[T1, T2],
                               LL: list[[list[T1], list[T2]]]):
bool =
    FORALL (par1: [list[T1], list[T2]]):
      es_concepto_c_rep?(R)(par1) IFF
       (EXISTS (par2: [list[T1], list[T2]]):
          member(par2, LL) AND
           prod_cart(c(id[T1]), c(id[T2]))(par1) =
            prod_cart(c(id[T1]), c(id[T2]))(par2))

es_conj_todos_conceptos_e?(R: FFC_rep[T1, T2],
                           LL: list[[list[T1], list[T2]]]):
bool =
    FORALL (par1: [list[T1], list[T2]]):
      es_concepto_c_rep?(R)(par1) IFF
       (EXISTS (par2: [list[T1], list[T2]]):
          member(par2, LL) AND igual_concepto_rep_e(R)(par1, par2))

es_conj_todos_conceptos_e_igual: LEMMA
  es_conj_todos_conceptos_e?(R, LL) IFF
   es_conj_todos_conceptos_e_rep?(R, LL)

agrega_extension_elto_rep_es_refinamiento_TCC1: OBLIGATION
  es_refinamiento?[[T2, finite_set[finite_set[T1]]], [T2, list[list[T1]]]]
      (prod_cart[T2, finite_set[finite_set[T1]], T2, list[list[T1]]]
           (id[T2], c[finite_set[T1], list[T1]](c[T1, T1](id[T1]))));

agrega_extension_elto_rep_es_refinamiento_TCC2: OBLIGATION
  es_refinamiento?[finite_set[finite_set[T1]], list[list[T1]]]
      (c[finite_set[T1], list[T1]](c[T1, T1](id[T1])));

agrega_extension_elto_rep_es_refinamiento: LEMMA
  es_refinamiento_op?[[T2, finite_set[finite_set[T1]]],
                      finite_set[finite_set[T1]], [T2, list[list[T1]]],
                      list[list[T1]], prod_cart(id[T2], c(c(id[T1]))),
                      c(c(id[T1]))]
      (agrega_extension_elto(trans_f(R)), agrega_extension_elto_rep(R))

agrega_extension_elto_rep_es_refinamiento_corol: COROLLARY
  FORALL (a: T2, LL: list[list[T1]]):
    c(c(id[T1]))(agrega_extension_elto_rep(R)(a, LL)) =
     agrega_extension_elto(trans_f(R))(a, c(c(id[T1]))(LL))

genera_extensiones_aux_rep_null_1: CLAIM
  FORALL (ls: list[T2], LL: list[list[T1]]):
```

```
    null?(ls) IMPLIES genera_extensiones_aux_rep(R)(ls, LL) = LL

  genera_extensiones_aux_rep_es_refinamiento_l1: LEMMA
    FORALL (ls: list[T2], LL: list[list[T1]]):
      genera_extensiones_aux(trans_f(R))(c(id[T2])(ls), c(c(id[T1]))(LL)) =
       c(c(id[T1]))(genera_extensiones_aux_rep(R)(ls, LL))

  genera_extensiones_aux_rep_es_refinamiento_TCC1: OBLIGATION
    es_refinamiento?[[finite_set[T2], finite_set[finite_set[T1]]],
                     [list[T2], list[list[T1]]]]
        (prod_cart[finite_set[T2], finite_set[finite_set[T1]], list[T2],
                   list[list[T1]]]
             (c[T2, T2](id[T2]),
              c[finite_set[T1], list[T1]](c[T1, T1](id[T1]))));

  genera_extensiones_aux_rep_es_refinamiento: LEMMA
    es_refinamiento_op?[[finite_set[T2], finite_set[finite_set[T1]]],
                        finite_set[finite_set[T1]],
                        [list[T2], list[list[T1]]], list[list[T1]],
                        prod_cart(c(id[T2]), c(c(id[T1]))), c(c(id[T1]))]
        (genera_extensiones_aux(trans_f(R)), genera_extensiones_aux_rep(R))

  singleton_obj_trans_f: CLAIM
    c(c(id[T1]))((: obj_rep(R) :)) = singleton(obj(trans_f(R)))

  genera_extensiones_rep_es_refinamiento: LEMMA
    es_refinamiento_op?[FFC[T1, T2], finite_set[finite_set[T1]],
                        FFC_rep[T1, T2], list[list[T1]], trans_f[T1, T2],
                        c(c(id[T1]))]
        (genera_extensiones, genera_extensiones_rep)

  genera_conceptos_rep_es_refinamiento_TCC1: OBLIGATION
    FORALL (x1: FFC[T1, T2]):
      is_finite[[finite_set[T1], finite_set[T2]]]
          (genera_conceptos[T1, T2](x1));

  genera_conceptos_rep_es_refinamiento_TCC2: OBLIGATION
    es_refinamiento?[finite_set[[finite_set[T1], finite_set[T2]]],
                     list[[list[T1], list[T2]]]]
        (c[[finite_set[T1], finite_set[T2]], [list[T1], list[T2]]]
             (prod_cart[finite_set[T1], finite_set[T2], list[T1], list[T2]]
                  (c[T1, T1](id[T1]), c[T2, T2](id[T2]))));

  genera_conceptos_rep_es_refinamiento: THEOREM
    es_refinamiento_op?[FFC[T1, T2],
                        finite_set[[finite_set[T1], finite_set[T2]]],
                        FFC_rep[T1, T2], list[[list[T1], list[T2]]],
                        trans_f[T1, T2], c(prod_cart(c(id[T1]), c(id[T2])))]
        (genera_conceptos, genera_conceptos_rep)

  genera_conceptos_rep_es_refinamiento_corol: COROLLARY
    c(prod_cart(c(id[T1]), c(id[T2])))(genera_conceptos_rep(R)) =
```

```
    genera_conceptos(trans_f(R))

  correccion_genera_conceptos_e_se_preserva: THEOREM
    es_conj_todos_conceptos_e_rep?(R, genera_conceptos_rep(R))

  correccion_genera_conceptos_rep_alt: THEOREM
    FORALL (par: [list[T1], list[T2]]):
      es_concepto_c_rep?(R)(par) IFF
       member(igual?(prod_cart(c(id[T1]), c(id[T2]))))
             (par, genera_conceptos_rep(R))

  correccion_genera_conceptos_rep_alt_b_TCC1: OBLIGATION
    FORALL (R: FFC_rep[T1, T2]):
      equivalence?[[list[T1], list[T2]]](igual_concepto_rep_e[T1, T2](R));

  correccion_genera_conceptos_rep_alt_b: THEOREM
    FORALL (par: [list[T1], list[T2]]):
      es_concepto_c_rep?(R)(par) IFF
       member(igual_concepto_rep_e(R))(par, genera_conceptos_rep(R))

  genera_conceptos_rep_sd(R): list[[list[T1], list[T2]]] =
      elimina_duplicados(igual_concepto_rep_e(R))(genera_conceptos_rep(R))

  elimina_duplicados_genera_conceptos_rep_igual: LEMMA
    igual_l?(igual_concepto_rep_e(R))
            (genera_conceptos_rep_sd(R), genera_conceptos_rep(R))
 END conceptos_contexto_ref
```

E.2.3. Implicaciones entre atributos

```
implicaciones_atrib_ref[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING implicaciones_atrib

  IMPORTING contextos_formales_ref

  C, C1, C2: VAR FFC[T1, T2]

  R, R1, R2: VAR FFC_rep[T1, T2]

  implicacion_gen_ref: TYPE =
  [# antecedente_ref: list[T2], consecuente_ref: list[T2] #]

  transf_imp_gen(imp_g_r: implicacion_gen_ref): implicacion_gen[T1, T2] =
```

```
    (# antecedente := c(id[T2])(antecedente_ref(imp_g_r)),
       consecuente := c(id[T2])(consecuente_ref(imp_g_r)) #)

implicacion_gen_ref_es_refinamiento: LEMMA
  es_refinamiento?[implicacion_gen[T1, T2], implicacion_gen_ref]
      (transf_imp_gen)

implicacion_ref(R): TYPE =
{imp_r: implicacion_gen_ref |
         sublista?[T2](antecedente_ref(imp_r), atrib_rep(R)) AND
          sublista?[T2](consecuente_ref(imp_r), atrib_rep(R))}

JUDGEMENT implicacion_ref(R) SUBTYPE_OF implicacion_gen_ref

transf_imp_gen_TCC1: OBLIGATION
  FORALL (R: FFC_rep[T1, T2], imp_r: implicacion_ref(R)):
    subset?[T2]
        (antecedente(transf_imp_gen(imp_r)), atrib(trans_f[T1, T2](R)))
     AND
     subset?[T2]
         (consecuente(transf_imp_gen(imp_r)), atrib(trans_f[T1, T2](R)));

transf_imp_gen(R)(imp_r: implicacion_ref(R)):
implicacion[T1, T2](trans_f(R)) = transf_imp_gen(imp_r)

implicacion_ref_es_refinamiento: LEMMA
  es_refinamiento?[implicacion[T1, T2](trans_f(R)), implicacion_ref(R)]
      (transf_imp_gen(R))

transf_imp_gen_TCC2: OBLIGATION
  FORALL (R):
    es_refinamiento?[implicacion[T1, T2](trans_f[T1, T2](R)),
                     implicacion_ref(R)]
        (transf_imp_gen(R));

JUDGEMENT transf_imp_gen(R) HAS_TYPE
    (es_refinamiento?[implicacion[T1, T2](trans_f(R)),
                      implicacion_ref(R)])

respeta_imp_r?(R)(ls: list[T2], imp_r: implicacion_ref(R)): bool =
    NOT sublista?[T2](antecedente_ref(imp_r), ls) OR
     sublista?[T2](consecuente_ref(imp_r), ls)

respeta_imp_r_refinamiento: LEMMA
  es_refinamiento_op?[[finite_set[T2], implicacion[T1, T2](trans_f(R))],
                      bool, [list[T2], implicacion_ref(R)], bool,
                      prod_cart(c(id[T2]), transf_imp_gen(R)), id[bool]]
      (respeta_imp?(trans_f(R)), respeta_imp_r?(R))

respeta_imp_r_refinamiento_corol: COROLLARY
  FORALL (ls: list[T2], imp_r: implicacion_ref(R)):
    respeta_imp?(trans_f(R))(c(id[T2])(ls), transf_imp_gen(R)(imp_r)) IFF
```

```
      respeta_imp_r?(R)(ls, imp_r)

  es_valida_r?(R)(imp_r: implicacion_ref(R)): bool =
      every(LAMBDA (x: T1):
              respeta_imp_r?(R)(intencion_rep(R)((: x :)), imp_r))
          (obj_rep(R))

  es_valida_r_refinamiento: LEMMA
    es_refinamiento_op?[implicacion[T1, T2](trans_f(R)), bool,
                        implicacion_ref(R), bool, transf_imp_gen(R),
                        id[bool]]
       (es_valida?(trans_f(R)), es_valida_r?(R))

  respeta_imp_r?(R)(l1: list[T2], l2: list[implicacion_ref(R)]): bool =
      every(LAMBDA (imp_r: implicacion_ref(R)):
              respeta_imp_r?(R)(l1, imp_r))
          (l2)
 END implicaciones_atrib_ref
```

E.2.4. Cálculo de la base de Duquenne–Guigues

```
base_DG_ref[T1, T2: TYPE+]: THEORY
 BEGIN
  ASSUMING
   T1_TCC1: ASSUMPTION EXISTS (x: T1): TRUE;

   T2_TCC1: ASSUMPTION EXISTS (x: T2): TRUE;
  ENDASSUMING

  IMPORTING base_DG

  IMPORTING implicaciones_atrib_ref

  IMPORTING REFINAMIENTO@refinamiento_subconjuntos_card

  IMPORTING REFINAMIENTO@refinamiento_powerset

  d: VAR T1

  a: VAR T2

  C, C1, C2: VAR FFC[T1, T2]

  R, R1, R2: VAR FFC_rep[T1, T2]

  LL: VAR list[list[T2]]

  es_pseudo_int_rest_ref?(R)(ls: list[T2], LL: list[list[T2]]): bool =
      NOT sublista?[T2](clausura_a_rep(R)(ls), ls) AND
       every(LAMBDA (l: list[T2]):
```

```
            IF sublista_estricto?[T2](l, ls)
              THEN sublista?[T2](clausura_a_rep(R)(l), ls)
            ELSE TRUE
            ENDIF)
          (LL)

es_pseudo_int_rest_ref?_es_refinamiento_corol: LEMMA
  FORALL (ls: list[T2], LL: list[list[T2]]):
    es_pseudo_int_rest?(trans_f(R))(c(id[T2])(ls), c(c(id[T2]))(LL)) IFF
     es_pseudo_int_rest_ref?(R)(ls, LL)

pseudo_restringidas_ref_TCC1: OBLIGATION
  FORALL (LL2: list[list[T2]], ls: list[T2], LL, R: FFC_rep[T1, T2],
          S: list[list[T2]]):
    LL = cons(ls, LL2) AND es_pseudo_int_rest_ref?(R)(ls, S) IMPLIES
     length[list[T2]](LL2) < length[list[T2]](LL);

pseudo_restringidas_ref_TCC2: OBLIGATION
  FORALL (LL2: list[list[T2]], ls: list[T2], LL, R: FFC_rep[T1, T2],
          S: list[list[T2]]):
    LL = cons(ls, LL2) AND NOT es_pseudo_int_rest_ref?(R)(ls, S) IMPLIES
     length[list[T2]](LL2) < length[list[T2]](LL);

pseudo_restringidas_ref(R)(LL, S: list[list[T2]]): RECURSIVE
list[list[T2]] =
  CASES LL
    OF null: null,
       cons(ls, LL2):
         IF es_pseudo_int_rest_ref?(R)(ls, S)
           THEN cons(ls, pseudo_restringidas_ref(R)(LL2, S))
         ELSE pseudo_restringidas_ref(R)(LL2, S)
         ENDIF
    ENDCASES
   MEASURE length(LL)

pseudo_restringidas_ref_cn1: LEMMA
  FORALL (LL, S: list[list[T2]], ls: list[T2]):
    member(ls, pseudo_restringidas_ref(R)(LL, S)) IMPLIES member(ls, LL)

pseudo_restringidas_ref_cn2: LEMMA
  FORALL (LL, S: list[list[T2]], ls: list[T2]):
    member(ls, pseudo_restringidas_ref(R)(LL, S)) IMPLIES
     es_pseudo_int_rest_ref?(R)(ls, S)

pseudo_restringidas_ref_cs1: LEMMA
  FORALL (LL, S: list[list[T2]], ls: list[T2]):
    es_pseudo_int_rest_ref?(R)(ls, S) AND member(ls, LL) IMPLIES
     member(ls, pseudo_restringidas_ref(R)(LL, S))

pseudo_restringidas_ref_es_refinamiento_corol: LEMMA
  FORALL (LL, S: list[list[T2]]):
    pseudo_restringidas(trans_f(R))(c(c(id[T2]))(LL), c(c(id[T2]))(S)) =
```

```
     c(c(id[T2]))(pseudo_restringidas_ref(R)(LL, S))

gen_s_pasos_ref_TCC1: OBLIGATION
  FORALL (R: FFC_rep[T1, T2], ls: list[T2], k: upto(cardinal_l[T2](ls))):
    cardinal_l[T2](ls) - k >= 0;

gen_s_pasos_ref_TCC2: OBLIGATION
  FORALL (NS: list[list[T2]], R: FFC_rep[T1, T2], S: list[list[T2]],
          ls: list[T2], k: upto(cardinal_l[T2](ls))):
    NS = pseudo_restringidas_ref(R)(sublistas_card[T2](ls, k), S) AND
     NOT k = cardinal_l[T2](ls)
     IMPLIES k + 1 <= cardinal_l[T2](ls);

gen_s_pasos_ref_TCC3: OBLIGATION
  FORALL (NS: list[list[T2]], R: FFC_rep[T1, T2], S: list[list[T2]],
          ls: list[T2], k: upto(cardinal_l[T2](ls))):
    NS = pseudo_restringidas_ref(R)(sublistas_card[T2](ls, k), S) AND
     NOT k = cardinal_l[T2](ls)
     IMPLIES cardinal_l[T2](ls) - (k + 1) < cardinal_l[T2](ls) - k;

gen_s_pasos_ref(R)
               (ls: list[T2], k: upto(cardinal_l[T2](ls)),
                S: list[list[T2]]):
RECURSIVE list[list[T2]] =
  LET NS = pseudo_restringidas_ref(R)(sublistas_card[T2](ls, k), S) IN
    IF k = cardinal_l[T2](ls) THEN append(S, NS)
    ELSE gen_s_pasos_ref(R)(ls, k + 1, append(S, NS))
    ENDIF
   MEASURE cardinal_l[T2](ls) - k

gen_s_pasos_ref_es_refinamiento_corol_TCC1: OBLIGATION
  FORALL (ls: list[T2], k: upto(cardinal_l[T2](ls))):
    k <= card[T2](c[T2, T2](id[T2])(ls));

gen_s_pasos_ref_es_refinamiento_corol: LEMMA
  FORALL (ls: list[T2], k: upto(cardinal_l[T2](ls)), S: list[list[T2]]):
    gen_s_pasos(trans_f[T1, T2](R))(c(id[T2])(ls), k, c(c(id[T2]))(S)) =
     c(c(id[T2]))(gen_s_pasos_ref(R)(ls, k, S))

gen_s_ref_TCC1: OBLIGATION
  FORALL (R: FFC_rep[T1, T2]): 0 <= cardinal_l[T2](atrib_rep(R));

gen_s_ref(R): list[list[T2]] = gen_s_pasos_ref(R)(atrib_rep(R), 0, null)

gen_s_ref_es_refinamiento_TCC1: OBLIGATION
  FORALL (x1: FFC[T1, T2]): is_finite[finite_set[T2]](gen_s[T1, T2](x1));

gen_s_ref_es_refinamiento_TCC2: OBLIGATION
  es_refinamiento?[finite_set[finite_set[T2]], list[list[T2]]]
      (c[finite_set[T2], list[T2]](c[T2, T2](id[T2])));

gen_s_ref_es_refinamiento: LEMMA
```

```
  es_refinamiento_op?[FFC[T1, T2], finite_set[finite_set[T2]],
                      FFC_rep[T1, T2], list[list[T2]], trans_f[T1, T2],
                      c(c(id[T2]))]
      (gen_s, gen_s_ref)

gen_s_ref_es_refinamiento_corol: COROLLARY
  c(c(id[T2]))(gen_s_ref(R)) = gen_s(trans_f[T1, T2](R))

genera_imps_ref(R): list[implicacion_gen_ref[T1, T2]] =
    LET fun =
          LAMBDA (Y: list[T2]):
            (# antecedente_ref := Y,
               consecuente_ref := clausura_a_rep(R)(Y) #)
      IN map(fun)(gen_s_ref(R))

genera_imps_ref_es_refinamiento_TCC1: OBLIGATION
  FORALL (x1: FFC[T1, T2]):
    is_finite[implicacion_gen[T1, T2]](genera_imps_g[T1, T2](x1));

genera_imps_ref_es_refinamiento_TCC2: OBLIGATION
  es_refinamiento?[implicacion_gen[T1, T2], implicacion_gen_ref[T1, T2]]
      (transf_imp_gen[T1, T2]);

genera_imps_ref_es_refinamiento_TCC3: OBLIGATION
  es_refinamiento?[finite_set[implicacion_gen[T1, T2]],
                   list[implicacion_gen_ref[T1, T2]]]
      (c[implicacion_gen[T1, T2], implicacion_gen_ref[T1, T2]]
           (transf_imp_gen[T1, T2]));

genera_imps_ref_es_refinamiento: THEOREM
  es_refinamiento_op?[FFC[T1, T2], finite_set[implicacion_gen[T1, T2]],
                      FFC_rep[T1, T2], list[implicacion_gen_ref[T1, T2]],
                      trans_f[T1, T2], c(transf_imp_gen[T1, T2])]
      (genera_imps_g, genera_imps_ref)

genera_imps_ref_es_refinamiento_corol_1: COROLLARY
  c(transf_imp_gen[T1, T2])(genera_imps_ref(R)) =
   genera_imps_g(trans_f[T1, T2](R))

genera_imps_ref_es_refinamiento_corol_2: COROLLARY
  IMPS(trans_f[T1, T2](R)) =
   restrict[implicacion_gen[T1, T2],
            implicacion[T1, T2](trans_f[T1, T2](R)), boolean]
       (c(transf_imp_gen[T1, T2])(genera_imps_ref(R)))

genera_imps_ref_b_TCC1: OBLIGATION
  FORALL (R: FFC_rep[T1, T2],
          fun:
            [list[T2] ->
               [# antecedente_ref: list[T2],
                  consecuente_ref: list[T2] #]]):
    (fun =
```

```
       (LAMBDA (Y: list[T2]):
          (# antecedente_ref := Y,
             consecuente_ref := clausura_a_rep(R)(Y) #)))
      IMPLIES equivalence?[list[T2]](igual_l?[T2]);

 genera_imps_ref_b(R): list[implicacion_gen_ref[T1, T2]] =
     LET fun =
           LAMBDA (Y: list[T2]):
             (# antecedente_ref := Y,
                consecuente_ref := clausura_a_rep(R)(Y) #)
       IN
       map(fun)(elimina_duplicados[list[T2]](igual_l?[T2])(gen_s_ref(R)))

 elimina_duplicados_gen_s_ref_igual_TCC1: OBLIGATION
   equivalence?[list[T2]](igual_l?[T2]);

 elimina_duplicados_gen_s_ref_igual: LEMMA
   igual_l?(igual?(c(id[T2])))
           (gen_s_ref(R),
            elimina_duplicados[list[T2]](igual_l?[T2])(gen_s_ref(R)))

 IMPORTING REFINAMIENTO@propiedades_map_via_ref

 genera_imps_ref_b_igual: LEMMA
   igual_l?(igual?(transf_imp_gen[T1, T2]))
           (genera_imps_ref(R), genera_imps_ref_b(R))

 genera_imps_ref_b_es_refinamiento_corol_2: COROLLARY
   IMPS(trans_f[T1, T2](R)) =
    restrict[implicacion_gen[T1, T2],
             implicacion[T1, T2](trans_f[T1, T2](R)), boolean]
        (c(transf_imp_gen[T1, T2])(genera_imps_ref_b(R)))
END base_DG_ref
```