



DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
E INTELIGENCIA ARTIFICIAL

**Demostración de teoremas
basada en lean^{TAP}.
Posibilidades abductivas.**

Memoria presentada por
Fernando Soler Toscano
como trabajo de investigación
en el Programa de Doctorado
*Lógica, Computación e
Inteligencia Artificial*

Fernando Soler Toscano

V. B. Director

José Antonio Alonso Jiménez

Sevilla, 29 de junio de 2004

Índice general

1. Introducción	1
2. El demostrador lean^{TAP}	7
2.1. El lenguaje de lean^{TAP}	7
2.2. Forma normal negada	8
2.3. Tableros con variables libres	12
2.4. El código de lean^{TAP}	14
2.4.1. Versión simple de lean^{TAP}	14
2.4.2. Uso de variables universales	16
2.5. Corrección y completud	20
2.5.1. Corrección	21
2.5.2. Completud	21
3. Modificaciones de lean^{TAP}	25
3.1. Devolviendo las ramas abiertas	25
3.2. Listas fijas de variables libres	27
3.2.1. Corrección	28
3.2.2. Completud	28
3.2.3. Construcción de tableros	28
3.3. Tableros como grafos	29
3.4. Tableros como BDDs	31
3.5. Compilación de la búsqueda	34
3.6. Variantes proposicionales	37
4. Comparaciones entre demostradores	41
4.1. Comparación entre demostradores de tableros	41
4.1.1. Descripción del test	41
4.1.2. Conclusiones	43
4.2. Comparación entre sistemas Prolog	45
4.2.1. Descripción del test	45
4.2.2. Conclusiones	46
4.3. Comparación con fórmulas proposicionales	48
4.3.1. Descripción del test	48

4.3.2. Conclusiones	51
4.4. Comparación con otros sistemas	53
4.4.1. Descripción del test	53
4.4.2. Conclusiones	54
4.5. Conclusiones globales	56
5. Idea de aplicación a la abducción	59
5.1. Planteamiento del problema	59
5.2. La concepción falibilista del progreso científico	61
5.3. Sistema abductivo en primer orden	64
5.3.1. Aspectos lógicos	65
5.3.2. Sistema de revisión de teorías	66
5.4. Trabajo futuro	67
5.4.1. Sistema de abducción en primer orden	67
5.4.2. Elección de la mejor representación	68
5.4.3. Ampliaciones	68
5.4.4. Abducción en otras lógicas	69
5.4.5. Aplicaciones de los sistemas propuestos	69
Bibliografía	71
A. Forma normal negada	75
B. Algoritmo de unificación de Stickel	77
C. Código completo de lean^{AP}	81
D. lean^{AP} con el unificador correcto de Prolog	85
E. Versiones que devuelven las ramas abiertas	89
E.1. Versiones con <code>unify/2</code>	89
E.2. Versiones con el unificador correcto de Prolog	92
F. Versiones con listas fijas de variables libres	97
F.1. Versiones sin variables universales	97
F.2. Versiones con variables universales	101
G. Tableros como grafos, BDDs y compilación	107
H. Adaptación proposicional	115
H.1. Transformación a NNF	115
H.2. Adaptación de las pruebas por tableros	116
H.3. Adaptación de otros métodos	118

I. Comparación con fórmulas de primer orden	125
I.1. Fórmulas empleadas	125
I.2. Fichero para realizar el test (SWI-Prolog)	126
I.3. Resultados en cada sistema	129
I.3.1. SWI-Prolog 5.2.13	129
I.3.2. SWI-Prolog 5.3.10	134
I.3.3. Yap Prolog 4.4.4	140
I.3.4. Yap Prolog 4.5.2	145
I.3.5. GNU Prolog 1.2.16	151
J. Comparación con fórmulas proposicionales	159
J.1. Fórmulas empleadas	159
J.2. Sencillo demostrador proposicional	166
J.3. Transformación a formato DIMACS	168
J.4. Comparación entre todos los métodos y Otter	171
J.5. Comparación con Mace, zChaff y Anldp	219

Capítulo 1

Introducción

La demostración automática no es, en el presente, ni una rama de la ingeniería ni de la matemática pura. La clave para aplicarla con éxito es el análisis cuidadoso y la experimentación. Ambos factores son igual de importantes y dependen el uno del otro.

Así es como Posegga y Schmitt subrayan el carácter teórico-práctico de la demostración automática, según el cual hemos pretendido organizar esta memoria¹, combinando el análisis de ciertos demostradores basados en tableros con la implementación, experimentación y comparación de eficiencia entre los mismos.

Ciertamente, en la historia de la demostración automática se combinan siempre el análisis y la experimentación². La primera implementación de un demostrador automático en una máquina electrónica es *The logic Theory Machine* (Newel, Simon y Shaw, 1957), que llega a demostrar con éxito hasta cuarenta teoremas de *Principia Mathematica*. También Martin Davis había diseñado en 1954 un programa que seguía el algoritmo que Presburger había propuesto en 1929 para decidir la verdad de fórmulas dentro de la teoría de la adición en la aritmética de primer orden. Sin embargo, estas dos implementaciones no tuvieron demasiado éxito, pues empleaban técnicas más bien *ad hoc* que hacía que fracasaran fuera de los estrechos marcos para los que se habían diseñado. De hecho, Newel *et al.* sostenían que su programa, más que *algorítmico*, era *heurístico* —entendiendo por tal más o menos que no hay garantías de que el proceso vaya a funcionar siempre, dado suficiente espacio y tiempo—. También, Davis declaraba en 1983 que el mayor triunfo de su programa había sido demostrar que la suma de dos números pares es un número par. De todos modos, estos programas sí que sirvieron como paradigma y motivación para los demostradores que vendrían posteriormente.

Una importante fuente teórica para el desarrollo de demostradores automáticos fue el empleo de las funciones de Skolem y los universos de Herbrand, según

¹Este trabajo ha contado con la subvención de la Consejería de Educación y Ciencia de la Junta de Andalucía, a través de una ayuda para la Formación de Doctores.

²Para escribir esta introducción hemos manejado las siguientes fuentes: [Dav01], [Lus92], [PS99], [RN96] y [AM].

una sugerencia hecha por Abraham Robinson, de los Laboratorios Argonne –de gran relevancia en la historia de la demostración automática–, en 1954, durante un congreso en la Universidad de Cornell. Así Gilmore, en 1958, emplea formas normales disyuntivas y los universos de Herbrand para construir un demostrador capaz ya de probar teoremas no triviales de la lógica de primer orden sin identidad.

Pero hay ciertos teoremas no demasiado complejos donde el programa de Gilmore falla. Martin Davis y Hilary Putnam encontraron que la causa de estos errores estaba en que el programa de Gilmore dependía de la transformación a forma normal disyuntiva, sin que hubiese métodos eficientes para ello, con lo que fallaba ante fórmulas moderadamente grandes. Davis y Putnam decidieron dedicar el verano de 1958 a la búsqueda de un método efectivo para la comprobación de la satisfactibilidad de las fórmulas proposicionales, que pudiese servir de base a los demostradores de primer orden. En 1960 publican un trabajo donde dan cuentas de un procedimiento, conocido como Davis-Putnam, que introduce, entre otras cosas, el empleo de la forma normal conjuntiva. Desarrollaron también un programa que busca refutaciones mediante la introducción de variables para producir cláusulas base, para después buscar inconsistencias proposicionales entre las mismas.

En 1960 aparece el primer demostrador de primer orden basado en tableros, desarrollado por Dag Prawitz, Hakan Prawitz y Neri Voghera. Implementaba un cálculo similar a las versiones usadas hoy día. Sin embargo, no usaba variables libres.

Tras los primeros diez años de demostración automática, en 1965, Robinson vuelve a aportar una idea que se convertiría en un nuevo impulso: la combinación de resolución y unificación. Con ello se reduce considerablemente el número de términos del universo de Herbrand, que ahora sólo se generan cuando es necesario para buscar contradicciones. Esta idea ya había sido sugerida por Prawitz, que propuso que lo importante en las demostraciones fuese el proceso de búsqueda de las contradicciones, y que en función de ello se introdujesen los términos del universo de Herbrand en la medida en que fuesen necesarios.

Basándose en la resolución binaria, Larry Wos crea el demostrador P1 -“Programa 1”-. La experimentación con este programa le lleva a desarrollar la estrategia del *conjunto soporte* –presente por ejemplo en Otter, uno de los demostradores que emplearemos en esta memoria– y la paramodulación. La paramodulación surge para eliminar los axiomas de la identidad, combinando las sustituciones de identidades con la resolución. A partir de esta idea surgen los sistemas de reescritura de términos, como el algoritmo de Knuth-Bendix.

También es Larry Wos quien en 1980 propone el uso del término *razonamiento automatizado* para referirse a esta disciplina, en vez de *demostración automática de teoremas*, ya que la búsqueda de pruebas había dejado de ser ciega y rutinaria –como habían sido por ejemplo las estrategias *ah hoc* de los primeros programas de Newel *et al.* y Davis– para convertirse en procesos que incluyen planificación,

estrategias, etc.

En la década de los 80, el centro de investigación de IBM en Heidelberg (Alemania) jugó un papel fundamental dentro del desarrollo de demostradores basados en tableros. Wolfgang Schonfel desarrolló un demostrador dentro de un proyecto sobre razonamiento legal, basándose en el cálculo de tableros con variables libres, y usando unificación para cerrar las ramas. Unos años más tarde, Peter Schmitt desarrolló el demostrador THOT, que también implementaba tableros con variables libres, como parte de un proyecto sobre procesamiento del lenguaje natural. Ambas implementaciones estaban hechas en Prolog, que había aparecido en los años 70 de la mano de Colmerauer y Kowalski, sistematizando trabajos anteriores como los de Boyer y Moore, quienes desarrollaron algoritmos de unificación según las ideas de Robinson.

Alrededor de 1990, en la Universidad de Karlsruhe, también financiado por IBM, P. Schmitt y Reiner Hahnle desarrollaron ${}_3TAP$, sobre la base de la experiencia con THOT. También ${}_3TAP$ fue programado en Prolog, e implementaba un cálculo de tableros con variables libres, tanto para lógica clásica de primer orden con identidad, como para lógicas multivaluadas. Este programa es el antecesor directo de $leanTAP$.

Por otro lado, Oppacher y Suen publicaron en 1988 su demostrador HARP, programado en LISP, y probablemente el demostrador basado en tableros más conocido. También, *The Helsinki Logic Machine* es otro programa Prolog que implementa unos 60 cálculos diferentes, entre ellos algunos de tableros semánticos para lógica clásica de primer orden, lógica no-monótona, lógica dinámica y lógica autoepistémica. A partir de 1990 el interés por los tableros semánticos crece, y son muchos los sistemas que se desarrollan, entre ellos $leanTAP$, nuestro punto de partida en la presente memoria.

En los capítulos que siguen realizamos un estudio comparativo entre diversas variantes de demostradores basados en tableros con variables libres para lógica clásica de primer orden sin identidad. El objetivo del capítulo 2 es, en primer lugar, presentar los elementos comunes de todos los demostradores basados en $leanTAP$ con los que trabajaremos (lenguaje, transformación de las fórmulas a Forma Normal Negada Skolemizada, el cálculo de tableros con variables libres, la necesidad de la unificación correcta, etc.). Además, estudiamos las dos versiones originales de $leanTAP$, propuestas por Bernhard Beckert y Joachim Posegga: la minimal, que los autores presentan como el demostrador correcto y completo para lógica de primer orden con menos líneas de código, y la versión que emplea variables universales. Describimos ambas versiones, comentando con detenimiento las partes fundamentales del código.

El objetivo que nos proponemos en el capítulo 3 es introducir diversas modificaciones en $leanTAP$ que pretenden hacerlo más eficiente para ciertas clases de fórmulas. Nos hemos basado, en parte, en el capítulo que Posegga y Schmitt prepararon para el *Handbook of Tableau-based Methods in Automated De-*

duction [PS99], donde llevan a cabo esta tarea, tanto proponiendo modificaciones de $\text{lean}^{\mathcal{AP}}$ que emplean representaciones de los tableros diferentes a los árboles (grafos y BDDs) como desarrollando un compilador que transforma la búsqueda de tableros cerrados en programas Prolog procedimentalmente equivalentes. También introducimos otras variantes de $\text{lean}^{\mathcal{AP}}$. En primer lugar, creamos una variante que devuelve los literales que se han encontrado en las ramas abiertas, en caso de no cerrarse el tablero. El objetivo de esta variante es que pueda servirnos en un futuro para implementar el razonamiento abductivo en primer orden, basándonos en tableros semánticos, tal como aparece en los acercamientos de Cialdea y Pirri [CP93] o Aliseda [Ali97]. También, realizamos otra variante de $\text{lean}^{\mathcal{AP}}$ que modifica la forma en que se entiende el límite de variables libres que pueden introducirse por rama del tablero, basándonos en los trabajos de E. Díaz [D93] y G. Boolos [Boo84]. Con el objeto de tratar problemas de lógica proposicional con mayor eficiencia que la versión originaria de $\text{lean}^{\mathcal{AP}}$, al final de este capítulo discutimos qué modificaciones serían convenientes en el código, y proponemos varias versiones proposicionales de $\text{lean}^{\mathcal{AP}}$.

Todas las versiones de $\text{lean}^{\mathcal{AP}}$ que se discuten en los capítulos 2 y 3 se encuentran implementadas, al final de la memoria, a modo de apéndices. El sistema elegido ha sido SWI-Prolog, versión 5.2.13, que en el momento de escribir este trabajo era la última estable. Las razones de esta elección son dos. En primer lugar, que se trata de un compilador libre, como GNU Prolog y Yap-Prolog, a los que también hemos adaptado casi todo el código, comprobando la portabilidad de los demostradores desarrollados. Pero además, hemos preferido que el desarrollo original de los programas fuese con SWI-Prolog por la mayor documentación y los mejores recursos que ofrece para la programación. Por tanto, el código de los apéndices puede compilarse sin problemas con SWI-Prolog. Para Yap-Prolog y GNU Prolog hemos indicado las modificaciones principales que deben hacerse.

En el capítulo 4 llevamos a cabo diferentes tests con distintos objetivos. En primer lugar, comparamos los demostradores entre sí. En segundo lugar comparamos los tres sistemas Prolog empleados: SWI-Prolog (versión 5.2.13, última estable al escribir esta memoria, y 5.3.10, en desarrollo), Yap-Prolog (versión 4.4.4, última estable al escribir esta memoria, y 4.5.2, en desarrollo) y GNU Prolog (versión 1.2.16). Para estos dos primeros tests hemos empleado algunas de las fórmulas que Pelletier [Pel86] propone para evaluar la eficiencia de los demostradores. En el apéndice I mostramos los resultados de estos tests. Tras cada uno de estos tests comentamos las conclusiones sobre el rendimiento de cada demostrador y cada sistema, teniendo en cuenta tanto el tiempo como el espacio requeridos para demostrar cada teorema.

Un tercer test se realiza con fórmulas proposicionales. Para ello, hemos tomado las familias de problemas que Roy Dyckhoff [Dyc97] propone para evaluar demostradores proposicionales intuicionistas, aunque tomando las fórmulas más interesantes para lógica proposicional clásica. También hemos añadido algunas fa-

milias que hemos creado con la intención de que resultaran especialmente difíciles (o fáciles) para los demostradores basados en tableros. En este test han participado tanto demostradores de primer orden como proposicionales, concluyendo que las versiones proposicionales de lean^{TAP} que desarrollamos al término del capítulo 3 son más apropiadas que la versión de primer orden de lean^{TAP} . Igualmente, se decide cuál es el demostrador más apropiado para cada familia de fórmulas y se comprueba con el rendimiento de Otter. Otter, uno de los sistemas desarrollados por los Laboratorios Argonne, resultará fácilmente superable en eficiencia cuando se trata de trabajar con fórmulas proposicionales. También hemos empleado un demostrador basado en tablas de verdad, tomado de [AB02], comprobando que pese a ser un sistema aparentemente peor que los tableros semánticos, hay ocasiones donde resulta mucho más adecuado.

Por último, para comprobar los resultados de nuestro trabajo con el estado actual de la cuestión, un último test compara el rendimiento de los demostradores proposicionales basados en lean^{TAP} con sistemas de reconocida eficiencia en la resolución del problema SAT. Así, emplearemos zChaff, desarrollado por la Universidad de Princeton y Anldp, también de los Laboratorios Argonne. Ambos demostradores trabajan con entradas en formato DIMACS, por lo que tendremos que transformar nuestros problemas a este formato. También empleamos Mace4, buscador de modelos en primer orden (otro sistema de los Laboratorios Argonne), para comprobar cómo su eficiencia no llega a la de los sistemas especializados en lógica proposicional, siendo en muchas ocasiones peor que los demostradores en Prolog desarrollados en este trabajo.

Tras estos tests se realizan unas conclusiones globales, donde se tratan de valorar qué aspectos son más influyentes en la eficiencia de un demostrador. Podemos resumir tales conclusiones en unas líneas de Posegga y Schmitt [PS99]:

Un punto importante en demostración automática: no hay panaceas. Para casi toda heurística o modificación en un cálculo hay un contraejemplo donde las cosas se vuelven peor que antes. El lenguaje de primer orden no es decidible, y es muy improbable que esto cambie alguna vez. Esto pone las cosas difíciles, pero también interesantes: nunca nos quedaremos sin problemas.

Para terminar, el capítulo 5 recoge las líneas de trabajo futuro que nos quedan abiertas. Como comentamos más arriba, nuestro objetivo final es profundizar en la resolución de problemas abductivos mediante el método de tableros, tal como hacen Cialdea y Pirri [CP93] o Aliseda [Ali97]. Comenzamos el capítulo con un breve panorama del devenir de la Filosofía de la Ciencia a lo largo del siglo veinte, determinando las posturas a las que nos sentimos más cercanos. Entonces nos planteamos la caracterización de las nociones de *problema abductivo* y *solución abductiva* dentro de tal marco, así como los requisitos que esto impone a los sistemas formales que quieran emplearse. Entonces damos unas breves pinceladas

sobre cómo podría realizarse esto con los demostradores que hemos desarrollado en esta memoria, así como de posibles adaptaciones que permitirían resolver problemas abductivos en lenguajes diferentes a la lógica de primer orden.

Capítulo 2

El demostrador lean^{AP}

El programa que tomamos como punto de partida en esta memoria es lean^{AP} , creado por Bernhard Beckert y Joachim Posegga. Está escrito en Prolog e implementa un demostrador de teoremas completo y correcto para lógica clásica de primer orden sin identidad, basándose en el cálculo de tableros semánticos con variables libres. La peculiaridad de lean^{AP} es que probablemente sea el demostrador más pequeño existente. Para una descripción detallada del programa recomendamos [BP94a], [BP94b], [BP94c], [BP95], y [Pos93]. Tanto el código de lean^{AP} como los artículos citados pueden encontrarse en la web de lean^{AP} : <http://i12www.ira.uka.de/leantap/>.

2.1. El lenguaje de lean^{AP}

En cuanto al lenguaje de lean^{AP} , se trata de la lógica de primer orden sin identidad, tal como hemos comentado. Como en lo sucesivo comentaremos ciertas partes del código de lean^{AP} y sus variantes¹, veamos la sintaxis que emplearemos en Prolog. En cuanto a los operadores lógicos –por orden de precedencia, de menor a mayor– son:

- “-” para la negación,
- “&” para la conjunción,
- “v” para la disyunción,
- “=>” para la implicación, y
- “<=>” para la equivalencia.

¹Al final de esta memoria, en los apéndices, puede encontrarse el código completo de cada una de las versiones de lean^{AP} que vamos a comentar.

En cuanto a los cuantificadores, debemos tener en cuenta que las variables cuantificadas son siempre variables Prolog, lo que nos servirá para buscar fácilmente unificaciones. De este modo, usamos:

- “ $\text{all}(X, F)$ ” para cuantificar universalmente la variable X en la fórmula F .
- “ $\text{ex}(X, F)$ ” para cuantificar existencialmente la variable X en la fórmula F .

Los predicados y funtores serán siempre funtores Prolog. Las definiciones de átomo, literal, literal sin variables, etc, son las habituales.

Aunque en el código aparezca este lenguaje, en las explicaciones emplearemos una sintaxis más habitual, tomando para los operadores lógicos \neg , \wedge , \vee , \rightarrow y \leftrightarrow –la precedencia será la misma que en Prolog– y para los cuantificadores \forall y \exists .

2.2. Forma normal negada

La filosofía de $\text{lean}^{\mathcal{AP}}$ es lo que se conoce como *lean deduction*, que consiste en lograr la mayor eficiencia posible a partir de códigos minimales. Por ello mismo, la entrada de $\text{lean}^{\mathcal{AP}}$ no es una fórmula de primer orden según la sintaxis anteriormente comentada, sino una fórmula en *forma normal negativa skolemizada* –en adelante NNF–. Una fórmula de primer orden α está en NNF si y sólo si –en adelante syss– se cumple que:

1. Los únicos operadores lógicos que contiene α son la negación, la conjunción y la disyunción.
2. Todos los negadores que aparecen en α afectan sólo a fórmulas atómicas.
3. α no contiene cuantificadores existenciales.

Para cada fórmula ϕ de primer orden puede encontrarse otra fórmula en NNF, que llamaremos $\text{NNF}(\phi)$, tal que ϕ es satisfacible syss $\text{NNF}(\phi)$ también lo es. La complejidad del proceso de conversión a NNF, para fórmulas que no contienen equivalencias, es lineal con respecto al grado lógico de la fórmula. Para las equivalencias, puede hacerse con complejidad cuadrática. Sin embargo, por simplicidad, los autores de $\text{lean}^{\mathcal{AP}}$ prefieren emplear un algoritmo de conversión que en el peor caso es exponencial para las equivalencias.

El lenguaje en que quedarán las fórmulas resultantes de la transformación a NNF, sobre el que $\text{lean}^{\mathcal{AP}}$ trabajará, es algo diferente del lenguaje de primer orden expuesto con anterioridad. Los únicos operadores que hay son la coma “,” para la conjunción, el punto y coma “;” para la disyunción, y el guión “-” para la negación. Con esto, la sintaxis resulta conocida a Prolog. La cuantificación universal sigue siendo igual, así como los predicados y funtores.

A continuación comentamos el procedimiento que $\text{lean}^{\mathcal{AP}}$ sigue para convertir las fórmulas a NNF, a la vez que presentamos las cláusulas que componen el

predicado `nnf(+Fml, +FreeV, -NNF, -Paths)`, donde `Fml` es la fórmula que se quiere transformar, `FreeV` es la lista de las variables libres que ocurren en `Fml`, `NNF` es la NNF de `Fml`, y `Paths` es el número de caminos disyuntivos de `NNF`, es decir, el número de ramas que como máximo puede producir al expandirse su tablero. El uso de este argumento es para implementar una heurística que hace que las NNF resultantes sean tales que al construir el tablero se traten antes las fórmulas que van a dar lugar a un menor número de ramas, lo que permite ahorrar espacio.

El primer paso de la transformación a NNF es introducir los negadores y eliminar las implicaciones y equivalencias. Esto se lleva a cabo mediante la siguiente cláusula:

```
nnf(Fml,FreeV,NNF,Paths) :-
    (Fml = -(-A)      -> Fml1 = A;
     Fml = -all(X,F)  -> Fml1 = ex(X,-F);
     Fml = -ex(X,F)   -> Fml1 = all(X,-F);
     Fml = -(A v B)   -> Fml1 = -A & -B;
     Fml = -(A & B)   -> Fml1 = -A v -B;
     Fml = (A => B)   -> Fml1 = -A v B;
     Fml = -(A => B)  -> Fml1 = A & -B;
     Fml = (A <=> B)  -> Fml1 = (A & B) v (-A & -B);
     Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B)),!,
    nnf(Fml1,FreeV,NNF,Paths).
```

Mientras que `Fml` tenga alguna de las formas que aparecen a la izquierda de las reglas, se aplica esta cláusula. Si éste no fuera el caso, entonces `Fml` puede ser una fórmula cuantificada, o bien una conjunción o una disyunción. En los últimos casos, el operador –conjuntor o disyuntor– se cambia por el correspondiente –coma o punto y coma, respectivamente– y se aplica el procedimiento recursivamente. En las dos cláusulas que siguen puede verse cómo se realiza esto. Además, se ve la función de `Paths`, y como el orden de los términos en las conjunciones (disyunciones) depende del número de caminos disyuntivos que tenga cada uno. El número de caminos disyuntivos de una conjunción es el producto del número de caminos disyuntivos de cada uno de sus términos. Para una disyunción, es la suma.

```
nnf(A & B,FreeV,NNF,Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 * Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2,NNF1);
     NNF = (NNF1,NNF2)).

nnf(A v B,FreeV,NNF,Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 + Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2;NNF1);
     NNF = (NNF1;NNF2)).
```

La NNF de una fórmula $\text{all}(X, F)$ cuantificada universalmente es la cuantificación universal de la NNF de F . Obsérvese cómo para buscar la NNF de F se ha añadido X a la lista de variables libres, al quedar esta variable libre en F .

```
nnf(all(X,F),FreeV,all(X,NNF),Paths) :- !,
    nnf(F,[X|FreeV],NNF,Paths).
```

Pero lo más interesante del proceso es el tratamiento de la cuantificación existencial. La skolemización que emplea lean^{AP} se basa en la regla δ^{++} que en [BHS93] Beckert, Haehnle y Schmitt introducen en el cálculo de tableros con variables libres. Allí se demuestra que a efectos de preservar la corrección y completud del cálculo basta con introducir como término de Skolem un functor que contenga sólo las variables libres que ocurren en la fórmula cuantificada existencialmente –y no todas las variables libres de la rama, tal como en la regla δ originaria–. Pero es más, la peculiaridad de la regla δ^{++} es que basta con que tal functor sea único para la clase de fórmulas tipo- δ que comparten cierta estructura, con lo que no tiene que introducirse un nuevo functor a cada uso de la regla δ . Por tanto, el número de functores diferentes que pueden aparecer en un tablero –aunque sea infinito tal tablero– es siempre igual al número de subfórmulas tipo- δ diferentes contenidas en la raíz del tablero. El uso de la regla δ^{++} produce tableros que llegan a ser exponencialmente menores –y nunca mayores– que si el functor fuera diferente en cada uso de la regla δ , lo que hace el cálculo más eficiente. El uso que lean^{AP} hace de la regla δ^{++} no es al construir el tablero, sino como parte de la skolemización llevada a cabo durante la transformación a NNF. En cuanto a la elección del término que sustituirá a la variable cuantificada existencialmente, como debe ser propio de las fórmulas tipo- δ que comparten la misma estructura, y contener todas las variables libres de la fórmula, los autores consideran una buena opción emplear la propia fórmula como término. Esto es lo que se lleva a cabo mediante dos llamadas a `copy_term/2`.

```
nnf(ex(X,Fml),FreeV,NNF,Paths) :- !,
    copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
    copy_term((X,Fml1,FreeV),(ex,Fml2,FreeV)),
    nnf(Fml2,FreeV,NNF,Paths).
```

Con la primera llamada a `copy_term/2` se ha creado `Fml1`, una copia de `Fml`, donde ninguna de las variables libres que ocurren en `Fml` se ha renombrado (pues están en `FreeV`). Simplemente `X` se ha cambiado por `Fml`, al ser un término que contiene todas las variables libres que deben ocurrir. La segunda llamada a `copy_term/2` simplemente cambia la variable `X` por `ex`, de modo que no tengamos una nueva variable libre.

Por último, la NNF de un literal es el propio literal. Además, sólo contiene un camino disyuntivo:

```
nnf(Lit,_,Lit,1).
```

Veamos un ejemplo de transformación a NNF. Sea la fórmula $\forall x \exists y \neg(r(x) \vee p(y))$. Para transformarla, `nnf/4` sigue los siguientes pasos:

- `nnf(all(X,ex(Y,-(r(X) v p(Y))))), [], NNF, Paths)`, que es el objetivo inicial, con una lista vacía de variables libres encontradas, y las variables `NNF` y `Paths` que recogerán, respectivamente, la NNF y el número de caminos disyuntivos de `NNF`. Por ser el primer argumento una fórmula cuantificada universalmente, llega a la cláusula para tratar generalizaciones, lo que produce el siguiente objetivo (véase que ahora `NNF = all(X,NNF2)`):
- `nnf(ex(Y,-(r(X) v p(Y))), [X], NNF2, Paths)`, que por ser una cuantificación existencial, llega a la correspondiente cláusula, que en este caso, instanciando las variables, queda:

```
nnf(ex(Y,-(r(X) v p(Y))), [X], NNF2, Paths):- !,
    copy_term((Y,-(r(X) v p(Y)), [X]),
              (-(r(X) v p(Y)), -(r(X) v p(-(r(X) v p(Y))))), [X])),
    copy_term((Y,-(r(X) v p(-(r(X) v p(Y))))), [X]),
              (ex,-(r(X) v p(-(r(X) v p(ex))))), [X])),
    nnf(-(r(X) v p(-(r(X) v p(ex))))), [X], NNF2, Paths).
```

Como se puede observar, las dos llamadas a `copy_term/2` han creado `-(r(X) v p(ex))` como término de Skolem, con las características que comentamos de la regla- δ^{++} , el ser único para `ex(Y,-(r(X) v p(Y)))`, y el que sólo ocurra libre `X`, la única variable libre por el momento. Ahora, el nuevo objetivo es:

- `nnf(-(r(X) v p(-(r(X) v p(ex))))), [X], NNF2, Paths)`, que es como acababa la cláusula anterior. Por ser el primer argumento la negación de una disyunción, se aplica la correspondiente regla, resultando:
- `nnf(-r(X) & -p(-(r(X) v p(ex))), [X], NNF2, Paths)`, que es una conjunción. Se aplica la cláusula correspondiente, que mostramos ya casi por completo instanciada:

```
nnf(-r(X) & -p(-(r(X) v p(ex))), [X], NNF2, Paths) :-
    nnf(-r(X), [X], -r(X), 1), % Literal
    nnf(-p(-(r(X) v p(ex))), [X], -p(-(r(X) v p(ex))), 1) % Literal
    Paths is 1*1,
    (1 > 1 -> NNF2 = (-p(-(r(X) v p(ex))), -r(X));
     NNF2 = (-r(X), -p(-(r(X) v p(ex))))).
```

Dado que `-r(X)` y `-p(-(r(X) v p(ex)))` son ambos literales, no necesitan transformarse más. `Paths` es el producto de los caminos disyuntivos de cada término de la disyunción, en este caso 1. Como puede verse, el valor resultante para `NNF2` es `(-r(X), -p(-(r(X) v p(ex))))`. Como la NNF de la fórmula original era `all(X,NNF2)`, resulta que `NNF = all(X, -r(X), -p(-(r(X) v p(ex))))`.

En el apéndice A puede encontrarse el código completo del fichero `nnf.pl` de la distribución de lean^{AP} .

2.3. Tableros con variables libres

Pasemos a ver el cálculo subyacente a lean^{AP} . Se trata del cálculo de tableros con variables libres. Este método permite superar el problema que aparece al construir tableros de tipo Beth-Smullyan [Bet69] cuando hay que aplicar la regla γ :

$$\frac{\forall x \gamma(x)}{\gamma(a)} \quad (2.1)$$

En la regla 2.1 debemos averiguar qué término a será el mejor para que el tablero se cierre. Mientras el tablero tenga ramas abiertas con fórmulas tipo γ , siempre habrá que probar nuevas sustituciones. En la práctica, la regla anterior se acaba convirtiendo en:

$$\frac{\forall x \gamma(x)}{\begin{array}{c} \gamma(t_1) \\ \gamma(t_2) \\ \vdots \\ \gamma(t_n) \end{array}} \quad (2.2)$$

donde t_1, t_2, \dots, t_n son todos los términos que ocurren en la rama.

Fácilmente se observa que la regla 2.2 introduce muchas fórmulas que no son necesarias para el cierre del tablero, aumentando mucho la complejidad de las pruebas.

El método de tableros con variables libres permite posponer la elección de las sustituciones hasta el momento en que estemos buscando cierres. Como veremos, se trata de un método que emplea unificación, lo que lo hace sumamente interesante al implementar demostradores en Prolog. En vez de sustituir las variables por términos introducimos *variables libres*.

Las reglas proposicionales son las mismas que en los tableros tipo Beth-Smullyan. En cuanto a las reglas para cuantificadores, para las fórmulas tipo γ ,

$$\frac{\forall x \gamma(x)}{\gamma(u)} \quad (2.3)$$

donde u es una variable libre nueva. Y para las fórmulas tipo δ ,

$$\frac{\exists x \delta(x)}{\delta(f(x_1, x_2, \dots, x_n))} \quad (2.4)$$

donde $\{x_1, x_2, \dots, x_n\}$ es el conjunto de variables libres que ocurren en $\exists x \delta(x)$, y f un símbolo de función nuevo en el tablero. En el caso de lean^{AP} , al usar la regla δ^{++} , f será el mismo para todas las fórmulas tipo- δ que comparten la misma estructura, tal como se ha comentado más arriba.

Por último, para cerrar ramas se emplea la unificación. No se trata ahora de encontrar dos literales complementarios, sino que una rama que contenga las fórmulas A y B se considera cerrada,

$$\frac{A}{\frac{\neg B}{\otimes}}, \quad \sigma(A) = \sigma(B) \quad (2.5)$$

si hay una sustitución σ tal que $\sigma(A) = \sigma(B)$. La sustitución de cierre σ debe ser la misma para todas las ramas del tablero, de modo que a cada variable libre, σ asigne un único término. Por tanto, conviene que σ sea el unificador más general de A y B , ya que será el menos restrictivo.

Puede observarse la facilidad de implementación de este cálculo en Prolog, lo que ha provocado que a partir de los años 80 los demostradores basados en tableros hayan empleado mayoritariamente tableros con variables libres.

Como ejemplo, veamos el tablero con variables libres de $\{\forall xPx \vee \forall y\neg Ry, \neg Pa, Rb\}$, que aparece en la figura 2.1. Este sencillo ejemplo muestra el ahorro que supone usar la regla 2.3 en vez de 2.2, que requiere introducir en cada rama dos instancias de cada una de las fórmulas cuantificadas, al aparecer los dos términos a y b en cada una de las ramas.

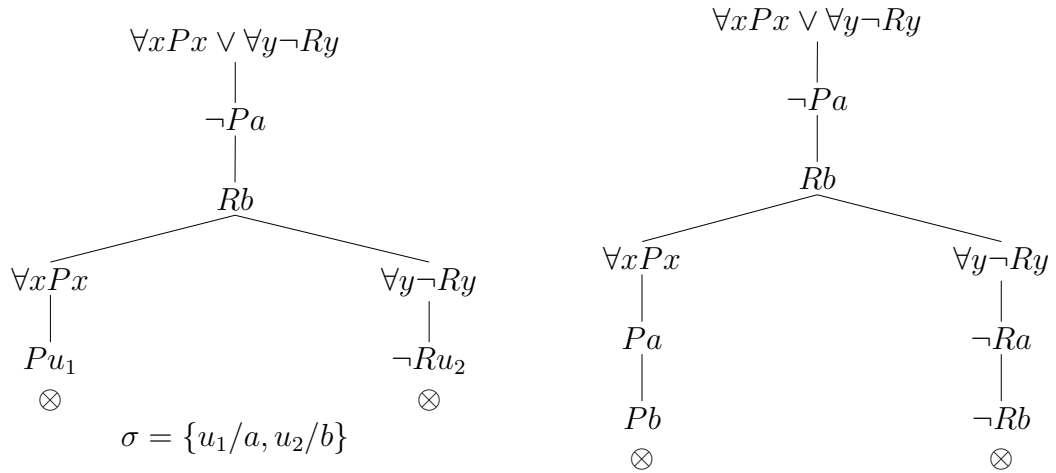


Figura 2.1: Tablero con la regla 2.3 (izquierda) y con la regla 2.2 (derecha)

El cálculo de $\text{lean}T^A\mathcal{P}$ será este mismo, aunque al trabajar con fórmulas en NNF hacen falta menos reglas, que serán, con la notación habitual:

Conjunción (regla α):

$$\frac{A \wedge B}{\frac{A}{B}} \quad (2.6)$$

Disyunción (regla β):

$$\frac{A \vee B}{A | B} \quad (2.7)$$

Cuantificación universal (regla γ):

$$\frac{\forall x \gamma(x)}{\gamma(u)} \quad (2.8)$$

siendo u una nueva variable libre.

A estas reglas tenemos que añadir 2.5 para los cierres.

Además, lean^{AP} contará con un sistema para limitar la longitud de las pruebas, ya que al no ser decidible la lógica de primer orden no puede determinarse a priori cuándo parar la construcción de un tablero. Lo que lean^{AP} hace es limitar el número de veces que en una misma rama puede emplearse la regla 2.8, con lo que el programa para –fallando– si no puede encontrar un tablero cerrado con un número de usos de la regla 2.8 menor o igual al máximo permitido. Además, los autores emplean un sistema de búsqueda en profundidad iterativa de forma que si no es posible encontrar una prueba con cierto número de usos de la regla 2.8, el sistema continúa construyendo otro tablero en que se permite un uso más, así hasta que se encuentra un tablero cerrado o bien el usuario o los recursos físicos de la máquina en que se ejecuta lean^{AP} detienen la búsqueda.

2.4. El código de lean^{AP}

En adelante, no comentaremos más que las partes más importantes del código de cada demostrador. Sin embargo, por ser lean^{AP} el punto de partida, vamos a comentar detenidamente su código. El fichero `leantap.pl`, de la distribución original de lean^{AP} , puede encontrarse en el apéndice C.

2.4.1. Versión simple de lean^{AP}

El predicado principal de lean^{AP} es `prove(+Fml, +UnExp, +Lits, +FreeV, +VarLim)`, donde:

Fml es una fórmula de primer orden sin identidad en NNF, según se ha explicado.

Se trata de la fórmula con la que lean^{AP} se encuentra trabajando.

UnExp es la lista de fórmulas que aún están por tratar en la rama con la que lean^{AP} trabaja (rama actual, en adelante).

Lits es la lista de los literales que han aparecido en la rama actual.

FreeV es la lista de las variables libres que ocurren en la rama actual.

VarLim Es un número natural que limita el número máximo de variables libres que pueden ocurrir en cada rama.

La ejecución $\text{prove}(+Fm1, [], [], [], +VarLim)$ tiene éxito si existe un tablero cerrado para la fórmula $Fm1$ con a lo sumo $VarLim$ variables libres por rama. Las cinco cláusulas que definen $\text{prove}/5$ son:

```

prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,           % 1
  prove(A, [B|UnExp],Lits,FreeV,VarLim).
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,          % 2
  prove(A,UnExp,Lits,FreeV,VarLim),
  prove(B,UnExp,Lits,FreeV,VarLim).
prove(all(X,Fm1),UnExp,Lits,FreeV,VarLim) :- !,     % 3
  \+ length(FreeV,VarLim),
  copy_term((X,Fm1,FreeV), (X1,Fm11,FreeV)),
  append(UnExp, [all(X,Fm1)],UnExp1),
  prove(Fm11,UnExp1,Lits, [X1|FreeV],VarLim).
prove(Lit,_,[L|Lits],_,_) :-                         % 4
  (Lit = -Neg; -Lit = Neg) ->
  (unify(Neg,L); prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-        % 5
  prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).

```

Comentamos brevemente cada cláusula. La numeración se corresponde con los números introducidos en los comentarios de la definición de $\text{prove}/5$:

1. Cuando lean^{TAP} encuentra una conjunción, pasa a trabajar con el primer miembro, guardando el segundo en la lista de fórmulas sin tratar. Como se observa, no hay nuevas variables libres, ni literales. Esta cláusula se corresponde con la regla- α .
2. Cuando lean^{TAP} encuentra una disyunción, se abren dos ramas, debiéndose cerrar ambas, por lo que deben hacerse dos llamadas a $\text{prove}/5$, una por cada rama. Esta cláusula se corresponde con la regla- β .
3. En el caso de la cuantificación universal, $\text{all}(X, Fm1)$, lean^{TAP} sólo continúa si se cumple que la longitud de la lista $FreeV$ es diferente a $VarLim$, lo que significa que aún no se ha llegado al límite de las variables libres permitidas. Entonces hace una llamada a $\text{copy_term}/2$, que tiene como objetivo crear $Fm11$, una copia de $Fm1$ donde se ha renombrado la variable antes cuantificada universalmente –de X pasa a $X1$ –.

A continuación, incluye la fórmula cuantificada en una nueva lista de fórmulas no tratadas, $UnExp1$. Se coloca al final para que se alterne la aplicación de la regla- γ entre todas las fórmulas cuantificadas universalmente que van apareciendo. De otro modo, el programa sería incompleto.

Finalmente, realiza una llamada a $\text{prove}/5$ con la nueva fórmula $Fm11$, la nueva lista de fórmulas sin tratar $UnExp1$ y una nueva lista de variables libres, $[X1|FreeV]$, resultado de añadir a $FreeV$ la nueva variable libre $X1$.

Así se implementa la regla- γ .

4. Si lean^{AP} llega a esta cláusula, entonces es que se encuentra ante un literal. En este caso lo primero que intenta es cerrar la rama. Busca Neg , literal complementario de Lit , y ve si unifica con el primero de los demás literales de la rama $\text{--unify}(\text{Neg}, \text{L})\text{--}$ o con alguno de los otros $\text{--prove}(\text{Lit}, [], \text{Lits}, _, _)\text{--}$. En caso de tener éxito, significa que la rama se ha cerrado. El predicado $\text{unify}/2$ implementa un algoritmo de unificación correcta tomado de Mark E. Stickel, que puede encontrarse en el apéndice B de esta memoria. En el apéndice D incluimos el código de lean^{AP} empleando el unificador correcto de Prolog $\text{unify_with_occurs_check}/2$.
5. Si la rama no se cerró, lean^{AP} añade el literal a la lista de literales de la rama, y pasa a trabajar con la primera de las fórmulas que estaba sin tratar.

2.4.2. Uso de variables universales

Como hemos visto, el límite en la construcción del tablero se fija en el número de variables libres que pueden introducirse por cada rama. Estas variables no están implícitamente cuantificadas universalmente, como ocurre en otros cálculos. Las variables libres, en este caso, son *rígidas*, lo cuál significa que a cada variable se debe aplicar la misma sustitución en todas las ramas del tablero.

Sin embargo, es posible distinguir una clase especial de variables libres, las *variables universales*, que no tienen por qué unificar con el mismo término en todas las ramas. El uso de variables universales permite en ciertas ocasiones reducir el número de variables libres necesarias para cerrar un tablero. En lo que sigue explicaremos la variación de lean^{AP} que emplea variables universales. Pero antes, daremos ciertas consideraciones teóricas sobre este método, siguiendo las explicaciones de [BP95].

El uso de variables libres *rígidas* se corresponde con la *relación de consecuencia fuerte*:

Definición 2.1 (Relaciones de consecuencia.) Sean ϕ, ψ fórmulas de primer orden. Decimos que ψ es consecuencia lógica débil de ϕ , lo que se escribe

$$\phi \models \psi,$$

si para todas las interpretaciones I se cumple:

$$\text{si } \text{val}_I(\phi) = \text{verdad} \text{ entonces } \text{val}_I(\psi) = \text{verdad}$$

considerando las variables libres universalmente cuantificadas.

Decimos que ψ es consecuencia lógica fuerte de ϕ , lo que se escribe

$$\phi \models^\circ \psi,$$

si para todas las interpretaciones I y para toda asignación de variables σ :

si $\text{val}_{I,\sigma}(\phi) = \text{verdad}$ entonces $\text{val}_{I,\sigma}(\psi) = \text{verdad}$.

Obsérvese que por ejemplo $p(x) \models \forall x p(x)$ pero sin embargo $p(x) \not\models \forall x p(x)$.

Estudiemos el ejemplo que aparece en la figura 2.2. Se trata de un tablero que no puede ser cerrado inmediatamente, al no haber una única sustitución que cierre todas sus ramas. Para encontrar la prueba, debemos emplear la regla γ de nuevo y crear otra instancia de $p(x)$.

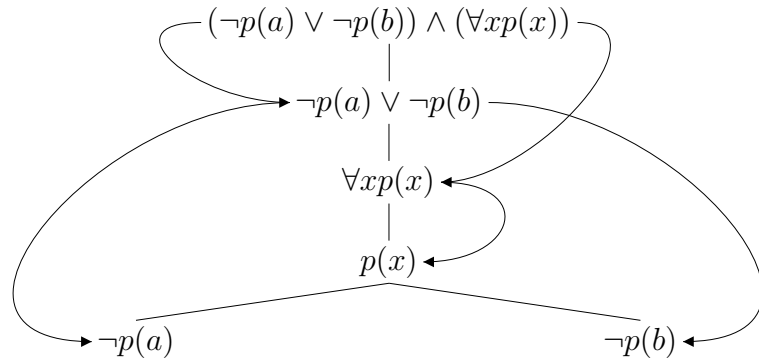


Figura 2.2: Ejemplo del uso de variables universales

Sin embargo puede ocurrir, como en la figura 2.2, que para ramas particulares R se cumpla que $R \models \forall x \phi(x)$. Entonces, en tales ramas podemos usar diferentes sustituciones de x que en el resto del tablero. En el ejemplo de la figura 2.2, el tablero cierra así inmediatamente. Reconocer las situaciones en que esto ocurre nos permite obtener pruebas más cortas, pues son necesarias menos aplicaciones de la regla γ .

Esta es la idea que emplea la variación de lean^{TAP} que ahora vamos a presentar. Se trata de un método que reconoce las *variables universales* que ocurren en cada rama. En realidad, no reconocerá todas las variables universales, lo cual es un problema indecidible, sino sólo las de un cierto tipo. Veamos qué entendemos por *variable universal*:

Definición 2.2 (Fórmula universal.) Supongamos que ϕ es una fórmula en cierta rama R . Se dice que ϕ es universal en R con respecto a la variable x si:

$$R \models \forall x \phi$$

Cuando esto ocurra, nos referiremos a la fórmula universal ϕ y a la variable universal x , siempre que el contexto no sea ambiguo.

Ahora podemos adaptar la regla que cierra las ramas de un tablero a la siguiente definición:

Definición 2.3 (Tablero cerrado.) Un tablero consistente en k ramas R_i ($1 \leq i \leq k$) es cerrado si existen, para cada i :

1. Una sustitución σ (la misma para todas las ramas),
2. Literales $l_i, \bar{l}_i \in R_i$, y
3. Sustituciones σ_i , tales que
 - a) $l_i\sigma_i$ y $\bar{l}_i\sigma_i$ son complementarios, y
 - b) si $\sigma_i(x) \neq \sigma(x)$ entonces ambos literales l_i y \bar{l}_i son universales en R con respecto a x .

Con esta definición de tablero cerrado es posible que se encuentren pruebas con menos reglas de expansión que en el cálculo normal de tableros con variables libres.

El problema de determinar qué variables son universales es indecidible en general. Sin embargo, una importante clase de variables universales puede reconocerse con gran facilidad: supongamos que hay una serie de aplicaciones de reglas que no producen nuevas ramas. Todas las fórmulas que se generan por esta secuencia de reglas son universales con respecto a las variables introducidas por tal secuencia de reglas. Pueden ignorarse las sustituciones para tales variables, ya que las reglas que produjeron las fórmulas que las contienen pueden aplicarse un número arbitrario de veces para generar nuevas instancias de las variables universales (sin que se generen más ramas). En términos más formales enunciaremos el siguiente lema, de prueba inmediata:

Lema 2.4 *Una fórmula ϕ en una rama R es universal con respecto a x si ϕ fue añadida a R por:*

1. una aplicación de una regla γ y x es la variable libre introducida por la aplicación de la regla, o bien por
2. aplicaciones de reglas que no crean nuevas ramas a una fórmula $\psi \in R$, siendo ψ universal en R con respecto a x .

Para reconocer con $\text{lean}^{\mathcal{AP}}$ las variables universales recogidas en el lema anterior debemos mantener para cada fórmula una lista con sus variables universales. Esta información se usa para renombrar las variables universales que ocurren en los literales, de modo que su instanciación no afecte al resto del tablero. Este proceso se hace al expandir las disyunciones, y simula la cuantificación universal de las variables que se renombran.

En cuanto al código, la aridad del predicado principal se extiende. Ahora es `prove(+Fml, +UnExp, +Lits, +DisV, +FreeV, +UnivV, +VarLim)`. El uso de los argumentos permanece igual que en `prove/5`, excepto `UnivV` y `DisV`, que son respectivamente la lista de las variables universales en `Fml` y un término `Prolog` que contiene todas las variables de la rama actual que no son universales en alguna de las fórmulas –las llamamos *variables disyuntivas*–. Cada fórmula sin expandir

en `UnExp` tendrá asociada la lista de sus variables universales, lo cual se hace con el functor “.”. El demostrador se inicia con `prove(+Fml, [], [], [], [], [], VarLim)` para probar la inconsistencia de `Fml`. A continuación explicamos el programa, mostrando las diferencias con la versión anterior.

La primera cláusula trata las conjunciones, considerando que todas las variables universales de una conjunción lo son también de cada una de sus componentes:

```
prove((A,B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
  prove(A,[(UnivV:B)|UnExp],Lits,DisV,FreeV,UnivV,VarLim).
```

La disyunción destruye la universalidad. Las variables universales de una disyunción no lo son de sus componentes. El tablero se divide, y las variables universales pasan a ser disyuntivas en cada una de las dos ramas resultantes. Por tanto las unimos a `DisV` formando un nuevo término². Las variables universales que hay en los literales se renombran con `copy_term/2` de modo que puedan instanciarse independientemente en cada una de las dos nuevas ramas.

```
prove((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
  copy_term((Lits,DisV),(Lits1,DisV)),
  prove(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
  prove(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).
```

Cuando se introduce una nueva variable libre al usar la regla γ , esta variable pasa a ser universal para la fórmula resultante (puede dejar de serlo si posteriormente se usa una regla que crea nuevas ramas, como vimos más arriba).

```
prove(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,VLim) :- !,
  \+ length(FreeV,VarLim),
  copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
  append(UnExp,[(UnivV:all(X,Fml))],UnExp1),
  prove(Fml1,UnExp1,Lits,DisV,[X1|FreeV],[X1|UnivV],VLim).
```

La siguiente cláusula no cambia, excepto en que hay dos nuevos argumentos:

```
prove(Lit,_,[L|Lits],_,_,_,_) :-
  (Lit = -Neg; -Lit = Neg) ->
  (unify(Neg,L); prove(Lit,[],Lits,_,_,_,_)).
```

Como el sexto argumento de `prove/7` contiene las variables universales de la fórmula actual, y no de la rama, cuando se extiende la rama debemos cambiar este argumento:

```
prove(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_,VarLim) :-
  prove(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV,VarLim).
```

²Podría crearse una lista, pero es más eficiente hacer un término con “+”

2.5. Corrección y completud

Una de las ventajas de lean^{AP} es que siendo tan compacto es posible probar formalmente su corrección y completud. En esta sección presentamos un esquema de tales pruebas, siguiendo la explicación que aparece en [BP95]. Demostraciones más simples pueden encontrarse en [Fit98] donde Fitting, a través de la definición de un original cálculo de secuentes, permite simplificar las pruebas de corrección y completud de lean^{AP} .

Para la prueba partimos de los teoremas de corrección y completud del cálculo de tableros con variables libres. Además, asumimos la corrección del compilador Prolog usado, así como de los predicados predefinidos.

Comenzamos con algunas definiciones útiles:

Definición 2.5 *Un tablero es un multiconjunto (finito) de ramas, donde cada rama es un multiconjunto (finito) de fórmulas de primer orden.*

Entendemos que las ramas de un tablero están implícitamente conectadas de modo disyuntivo, mientras que las fórmulas de una misma rama lo están de modo conjuntivo. Debemos definir las reglas de formación de tableros:

Definición 2.6 *Sea T un tablero, $R \in T$ una rama del mismo, y $\phi \in R$ una fórmula de R . Las reglas por las que T puede modificarse son:*

1. **Reglas de expansión:** *Puede derivarse un tablero a partir de T quitando la rama R a T y cambiándola por una o dos nuevas ramas, resultando:*

$$\begin{array}{ll}
 (R - \{\phi\}) \cup \{\psi_1, \psi_2\} & \text{si } \phi = \psi_1 \wedge \psi_2 \quad (\alpha) \\
 (R - \{\phi\}) \cup \{\psi_1\} \text{ y } (R - \{\phi\}) \cup \{\psi_2\} & \text{si } \phi = \psi_1 \vee \psi_2 \quad (\beta) \\
 R \cup \{\psi(y)\} \text{ (siendo } y \text{ una variable libre nueva)} & \text{si } \phi = \forall x \psi(x) \quad (\gamma)
 \end{array}$$

2. **Regla de cierre:** *Si R es cerrada (es decir, existen literales complementarios $l, \bar{l} \in R$) entonces $T - \{R\}$ puede derivarse de T .*
3. **Regla de sustitución:** *El tablero $T\sigma$ se puede derivar de T si σ es una sustitución (incluyendo la sustitución vacía) que no instancia las variables ligadas en T .*

Durante una prueba, cada llamada a `prove/5` junto a los objetivos de `prove/5` que están en la pila de Prolog esperando ser llamados, representa el tablero que se ha conseguido hasta el momento. Cada objetivo `prove(Fml, UnExp, Lits, FreeV, VarLim)` se corresponde con una rama abierta del tablero, consistente en las fórmulas de `Fml`, `UnExp` y `Lits`. Las diferentes cláusulas que definen `prove/5` van extendiendo el tablero en el sentido de la definición precedente.

2.5.1. Corrección

El teorema de corrección que debe probarse es:

Teorema 2.7 *Sea $Fm1$ una fórmula cerrada de primer orden ϕ en NNF. Si el objetivo $prove(Fm1, [], [], [], VarLim)$ tiene éxito, entonces ϕ es inconsistente.*

La prueba se basa en la corrección del cálculo de tableros con variables libres, recogido en la siguiente proposición:

Proposición 2.8 *Si ϕ es una fórmula cerrada de primer orden en forma normal negada y el tablero vacío (conjunto vacío) puede derivarse a partir del tablero inicial $\{\{\phi\}\}$, aplicando una secuencia finita de reglas de la definición 2.6, entonces ϕ es inconsistente.*

Usando la proposición 2.8 se puede demostrar lo siguiente para probar el teorema 2.7:

- Si $Fm1$ es ϕ , entonces el objetivo inicial $prove(Fm1, [], [], [], VarLim)$ representa el tablero inicial $\{\{\phi\}\}$.
- Cada vez que $lean^{TAP}$ cambia el conjunto de objetivos $prove/5$ en la pila de objetivos de Prolog, derivando un nuevo tablero, esto corresponde a la aplicación de una de las reglas de formación de tableros de la definición 2.6.
- $lean^{TAP}$ sólo termina con éxito (es decir, se vacía la pila de objetivos) cuando se deriva el tablero vacío.

2.5.2. Completud

En este caso, el teorema que debe probarse es:

Teorema 2.9 *Si ϕ es una fórmula inconsistente de primer orden en NNF, entonces existe un $n \geq 0$ tal que, si $Fm1$ representa ϕ y $VarLim$ con n , entonces el objetivo $prove(Fm1, [], [], [], VarLim)$ tiene éxito.*

La noción central para la prueba de este teorema es la de *tablero totalmente expandido*, y la de *secuencia de tableros* que termina en un tablero totalmente expandido:

Definición 2.10 *Una secuencia T_0, \dots, T_n de tableros es una secuencia de tableros totalmente expandida con respecto a los límites p y q ($p, q \geq 0$), si T_{i+1} ha sido derivado a partir de T_i usando una de las reglas de la definición 2.6 ($1 \leq i < n$), y:*

1. *En la secuencia sólo se han usado reglas de expansión.*

2. Sólo hay literales y fórmulas γ en T_n .
3. Si en una rama $R \in T_i$ ($1 \leq i \leq n$) aparece ϕ , una fórmula de tipo- γ , y ϕ que es una de las primeras q fórmulas que se han añadido a R (o estaban inicialmente presentes en R), entonces la regla γ se ha aplicado al menos p veces a esta ocurrencia de ϕ .

Las fórmulas tipo α y β se eliminan del tablero una vez que su correspondiente regla se les ha aplicado. Por tanto, la segunda condición de la definición 2.10 es equivalente a: “no hay fórmulas tipo α o β en el tablero a las que no se les haya aplicado la regla correspondiente”.

Empleando la noción de *secuencia de tableros totalmente expandida* el teorema de completud para el cálculo de tableros con variables libres puede ser formulado del siguiente modo:

Teorema 2.11 *Si la fórmula de primer orden ϕ , en NNF, es inconsistente, entonces hay límites p y q ($p, q \geq 0$) tales que para cada secuencia T_0, \dots, T_n de tableros que comienza con el tablero inicial $T_0 = \{\{\phi\}\}$ y que es una secuencia totalmente expandida con respecto a p y q , hay una sustitución σ tal que cada rama del tablero $T_n\sigma$ es cerrada.*

Es decir, existen:

- literales l_i y \bar{l}_i en cada rama $R_i \in T_n$ ($1 \leq i \leq m$) de T_n , y
- sustituciones μ_i que son más generales que σ , tales que $l_i\mu_i$ y $\bar{l}_i\mu_i$ son complementarios.

Las secuencias de tableros que hace lean^{AP} no están totalmente expandidas, porque lean^{AP} cierra las ramas en cuanto aparecen dos literales complementarios. Sin embargo, podemos evitar esto con $\text{lean}^{AP'}$, una versión de lean^{AP} que es idéntica a la presentada excepto que:

- Se omite la cuarta cláusula, que cierra ramas.
- Hay una cláusula adicional, que se añade al final:

```
prove(Fml,UnExp,Lits,_,_) :-
  write(['La rama consistente en ', Fml, UnExp, Lits,
        ' es parte del tablero totalmente expandido']).
```

Ahora, si Fml es la fórmula inconsistente ϕ en forma normal negativa, $VarLim$ es suficientemente alto, y $\text{lean}^{AP'}$ se inicia con el objetivo $\text{prove}(Fml, [], [], [], VarLim)$, entonces construye una secuencia totalmente expandida T'_0, \dots, T'_n con respecto a ciertos límites $p, q \geq 0$, en particular con respecto a los límites que existen de acuerdo con el teorema 2.11. Entonces el tablero T'_n es cerrado sólo si $VarLim$ es suficientemente alto. La prueba entonces tendría que pasar por los siguientes pasos:

- $\text{lean}T^{\mathcal{AP}}$, no aplica sustituciones, y no cierra ni quita ramas (definición 2.10, condición 1).
- Las reglas α y β se aplican tantas veces como sea posible (definición 2.10, condición 2).
- La lista **UnExp** implementa un criterio de prioridad, lo que hace que la regla γ se aplique un número de veces arbitrario a cada fórmula tipo γ sólo si **VarLim** es suficientemente alto (definición 2.10, condición 3).
- La computación de las ramas del tablero termina, pues, con cada paso o bien las fórmulas de la rama se hacen menos complejas, la longitud de **FreeV** aumenta, o el número de fórmulas en **UnExp** disminuye.

Quedaría por demostrarse que la versión original de $\text{lean}T^{\mathcal{AP}}$ construye también un tablero cerrado, y que de hecho cierra las ramas. Para hacer esto cambiamos, en dos pasos, la secuencia totalmente expandida $T'_0, \dots, T'_{n'}$ construida por $\text{lean}T^{\mathcal{AP}}$, tal que la secuencia resultante sea construida por $\text{lean}T^{\mathcal{AP}}$ y además acabe en el tablero vacío:

- Primero, todas las expansiones de ramas que ya contengan el par l_i, \bar{l}_i de literales de cierre se eliminan de la secuencia, ya que $\text{lean}T^{\mathcal{AP}}$ no expande tales ramas. Es fácil comprobar que el último tablero de la secuencia es cerrado al igual que $T'_{n'}$, y usando los mismos pares de literales y sustituciones.
- En un segundo paso, las aplicaciones de las reglas de sustitución y cierre se añaden a la secuencia. En el momento en que los literales de cierre l_i, \bar{l}_i ocurran en la rama $R \in T'_i$, la sustitución μ_i se aplica a T'_i y la rama cerrada $R\mu_i$ se elimina, usando la regla de cierre.

Obviamente, la secuencia de tableros resultante, T_0, \dots, T_n ($n \leq n'$), acaba con el tablero vacío $T_n = \emptyset$. Por inducción sobre i se prueba que después de un número finito de pasos de $\text{lean}T^{\mathcal{AP}}$ (y posiblemente después del backtracking, si hay puntos de elección), los objetivos `prove/5` que hay en la pila de Prolog representan exactamente el tablero T_i . Para $i = n$ ello implica que $\text{lean}T^{\mathcal{AP}}$ deriva el tablero cerrado, es decir, que acaba con éxito.

Para la prueba inductiva hay que validar que el aplicar sustituciones de cierre y el eliminar ramas cerradas no afecta a la expansión del resto de las ramas que aún no han sido cerradas. El orden en que se eligen las fórmulas para expandir permanece igual.

Capítulo 3

Modificaciones de lean^{TAP}

3.1. Devolviendo las ramas abiertas

Esta primera modificación es una variante nuestra de lean^{TAP} que devuelve, para cada rama del tablero que no consigue cerrarse, la lista de los literales aparecidos en la misma. Posteriormente comentaremos cómo puede hacerse un uso abductivo de esta variante.

Los programas creados con esta variante permiten trabajar en dos modos: “ref” y “abd”, como se verá. El primer modo funciona igual a lean^{TAP} , y sirve para buscar pruebas, es decir, tableros cerrados, fallando cuando no consigue este objetivo. El modo “abd”, por contra, permite obtener los literales de cada rama abierta si falla la construcción del tablero. La razón por la que se distingue entre ambos modos es que para buscar pruebas resulta mucho más eficiente usar “ref”, ya que se impide el uso de ciertas cláusulas que se emplean en modo “abd”.

La primera diferencia entre esta variante y lean^{TAP} es que ahora el predicado principal tiene un argumento más: `prove_abd(+Form, +UnExp, +Lits, +FreeV, +VarLim, +I:?A1-?A2)`. Los cinco primeros argumentos son conocidos. En cuanto al sexto:

- `I` es un indicador del modo en que el programa trabaja. Será “ref” si queremos que el tablero sea cerrado, y “abd” si queremos que en caso de ser abierto nos devuelva los literales encontrados en cada rama abierta.
- `A1-A2` es una lista de diferencia que representa el tablero que se va construyendo, como un multiconjunto de multiconjuntos de literales.

Veamos cómo se comportan los nuevos argumentos. Para ello comentaremos las cláusulas que difieren de la versión primera de lean^{TAP} , ya que el código de todas las versiones que emplean esta modificación puede encontrarse en el apéndice E. El uso de las listas de diferencia se puede ver al tratar las disyunciones:

```
prove_abd((A;B),UnExp,Lits,FreeV,VarLim,I:A1-A3) :- !,
```

```

prove_abd(A, UnExp, Lits, FreeV, VarLim, I: A1-A2),
prove_abd(B, UnExp, Lits, FreeV, VarLim, I: A2-A3).

```

Si `Form` es una disyunción $(A;B)$ hace dos ramas, usando A en una de ellas y B en la otra, al igual que en lean^{AP} . El tablero devuelto será la unión de los tableros resultantes de cada rama. En la cláusula de cierre vemos qué pasa con las ramas cerradas:

```

prove_abd(Lit, _, [L|Lits], _, _, I: A-A) :-
  (Lit = -Neg; -Lit = Neg) ->
  (unify(Neg, L); prove_abd(Lit, [], Lits, _, _, ref: A-A)).

```

Si tiene éxito esta cláusula, entonces la lista de diferencia que representa la rama correspondiente tablero unifica con $A-A$, la lista vacía. Además, en la llamada recursiva que se hace a `prove_abd/6`, vemos que el indicador se hace `ref`. Eso es así para evitar que la búsqueda de los cierres puedan tener éxito debido a la siguiente cláusula:

```

prove_abd(L, _, Ls, _, _, abd: [[L|Ls]|A]-A).

```

Se trata de la cláusula que, si falló el intento de cierre y si además se trabaja en el modo “abd” –pues el indicador debe ser “abd”– añade a la lista de diferencia que representa el tablero todos los literales encontrados en la rama.

El uso de `prove_abd/6` para cerrar tableros es `prove_abd(+Fml, [], [], [], +VarLim, ref:a-a)`, comportándose igual que lean^{AP} . Si queremos que devuelva los literales encontrados en las ramas abiertas, la llamada debe ser `prove_abd(+Fml, [], [], [], +VarLim, abd:?E-[])`, devolviendo en E los literales de cada rama abierta.

Un predicado interesante que puede definirse con esta variante es `tab(+Fml, +Lim, ?Tab)`, que sirve para hacer tableros, y tiene éxito cuando `Tab` es la representación de un tablero de `Fml` en que se emplean a lo sumo `Lim` variables libres en cada rama. La representación de `Tab` es la de un multiconjunto de multiconjuntos. Sólo debe tomarse la primera solución, pues si se aplica backtracking pueden devolverse soluciones inconsistentes. La definición es la siguiente:

```

tab(Fml, Lim, Tab) :-
  prove_abd(Fml, [], [], [], Lim, abd: X-[]), !, X=Tab.

```

Realmente, `Tab` no se corresponde siempre con la representación del tablero, pues sólo se devuelven los literales que han aparecido, así que no están las fórmulas tipo γ que pueda haber en `UnExp`. Igualmente, si el valor elegido para `Lim` no es lo suficientemente alto, puede que se queden literales en `UnExp`, detrás de fórmulas tipo γ que no pudieron ser usadas, al alcanzarse el límite `Lim`. Es fácil solventar estos problemas, de todas formas, tal como está, la definición es más eficiente, y para los propósitos que tenemos al definir estos predicados –poder

hacer posteriormente un uso abductivo— nos basta con los literales de cada rama. En cuanto al problema de que haya literales en `UnExp` detrás de fórmulas tipo γ aún no usadas, no es problema si se elige un valor para `Lim` lo suficientemente alto, lo cual no es difícil de calcular. Realmente si `Lim` es al menos igual al número de cuantificadores universales que hay en `Fml` ya se ha resuelto el problema.

3.2. Listas fijas de variables libres

Presentamos ahora una variación de `leanAP` que entiende de modo diferente el concepto de profundidad del tablero¹. Ahora ya no tendremos un argumento que sea un número natural que limite la profundidad del tablero —`VarLim` de los demostradores anteriores—, sino que el límite lo establecerá la lista de variables libres —`FreeV` en otros casos— que será fija desde el comienzo. De este modo, cuando se trata un cuantificador universal, se sustituye la variable cuantificada por todas las variables libres permitidas, y nos deshacemos de la fórmula cuantificada. La idea que da pie a esta versión es la de tratar de cerrar el tablero (respectivamente, crear modelos) empleando como máximo un número finito de términos nuevos (ya que cada variable libre puede unificar a lo sumo con un término).

Veamos las dos modificaciones principales que esta variante introduce en el código de `leanAP`:

- Si cada llamada en `leanAP` tiene la forma: `prove(+Fml, +UnExp, +Lits, +FreeV, +VarLim)`, ahora prescindimos de `VarLim`, y además `FreeV` es una lista de cardinalidad igual a la profundidad con la que se trabaja, a la que nunca se añade ni quita ningún elemento. De este modo, la llamada queda: `prove_var1(+Fml, +UnExp, +Lits, +FreeV)`.
- El tratamiento de las fórmulas tipo- γ consistirá en sustituir la variable cuantificada universalmente por todas las variables libres permitidas —o términos, más bien, ya que alguna variable ha podido unificar anteriormente con algún término— y añadir el resultado a la rama. Quedará por tanto:

```
prove_var1(all(X,Fml),UnExp,Lits,FreeV) :- !,
  findall(F,(member(Var,FreeV),
    copy_term((X,Fml,FreeV),(Var,F,FreeV))),
    [F1|Forms]),
  append(UnExp,Forms,UnExp1),
  prove_var1(F1,UnExp1,Lits,FreeV).
```

El resto de las cláusulas sólo cambian en que se omite el argumento `VarLim`. En el apéndice F se puede encontrar el código de todas las versiones de `leanAP` con esta variante.

¹Se trata de una modificación basada en los trabajos [Boo84], [D93] y [Nep02], en que se modifican los tableros de Beth [Bet69] de modo que se puedan obtener modelos mínimos.

Más adelante veremos cómo esta modificación aumenta la eficiencia de lean^{AP} para ciertas fórmulas, pues en muchos casos permite disminuir el número máximo de variables libres que deben introducirse en los tableros para cerrarse, lo que acorta las búsquedas en profundidad iterativa. Ahora nos detendremos en presentar esquemas de prueba de los teoremas de corrección y completud de estas variantes, para mostrar cómo las modificaciones introducidas no afectan a tales propiedades.

3.2.1. Corrección

Para probar la corrección de esta versión habría que demostrar que toda vez que encuentra un tablero cerrado, también lean^{AP} (o la versión correspondiente sin la modificación) lo habría encontrado.

Así, si dadas Fml y FreeV , $\text{prove_var1}(\text{Fml}, [], [], \text{FreeV})$ encuentra un tablero cerrado, y la lista FreeV tiene n elementos, significa que cada una de las m subfórmulas cuantificadas universalmente de Fml se ha instanciado n veces. Se puede demostrar que el objetivo $\text{prove}(\text{Fml}, [], [], [], \text{Lim})$, donde $\text{Lim} = m * n$ también tendría éxito, ya que extiende el tablero de Fml instanciando también n veces cada una de las m cuantificaciones universales que aparecen en Fml , esta vez con una variable diferente en cada una de las $m * n$ aplicaciones de la regla γ , por lo que es más fácil aún cerrar el tablero. De esta forma, si lean^{AP} ($\text{prove}/5$) es correcto, también lo es $\text{prove_var1}/4$.

3.2.2. Completud

Habría que probar que cada vez que lean^{AP} encuentra un tablero cerrado, también lo encuentra esta variante. Veamos un esquema de la prueba.

Supongamos que lean^{AP} encuentra un tablero cerrado, al tener éxito el objetivo $\text{prove}(\text{Fml}, [], [], [], \text{Lim})$, para la fórmula Fml y el número entero Lim . Eso significa que se han hecho Lim instancias con diferentes variables de las subfórmulas tipo γ que han aparecido.

Pues bien, si FreeV es una lista de longitud Lim , el objetivo $\text{prove_var1}(\text{Fml}, [], [], \text{FreeV})$ debe tener éxito también, pues para cada subfórmula tipo γ que aparezca en el tablero se realizan Lim instancias diferentes.

Si el tablero que hizo lean^{AP} se cerró con Lim instancias de entre todas las fórmulas tipo γ , también el tablero producido por $\text{prove_var1}/4$ debe cerrarse.

3.2.3. Construcción de tableros

También podrían obtenerse ciertos resultados interesantes respecto de los tableros que se pueden construir con versiones de lean^{AP} que integran las dos modificaciones hasta ahora propuestas, como es el caso de las versiones que aparecen en el apéndice F, con nombres que comienzan por prove_var1_abd . Todas

estas versiones permiten construir tableros si se usan en modo *abd*. En este caso, se puede probar que con que la lista de variables libres tenga al menos un elemento, las ramas que se devuelven contienen, para cada literal que aparecería en dicha rama en el tablero completamente expandido, al menos un literal sintácticamente similar –mismo predicado, mismas ligaduras entre variables y términos–, pues al instanciarse todas las fórmulas cuantificadas universalmente, no se da el problema de que se queden literales en *UnExp* detrás de fórmulas universales a las que no se pudo aplicar la regla- γ , y que por tanto no aparecen en el tablero –ver la discusión al final de la sección anterior–. Además, puede probarse que cuando se construyen estos tableros *UnExp* siempre se vacía.

Estas propiedades hacen que estos tableros sean aún más interesantes para su uso en problemas abductivos.

3.3. Tableros como grafos

Las modificaciones que siguen las hemos tomado de [PS99]. El código de todas las versiones de *lean^{TA}P* que comentamos en esta sección se puede encontrar en el apéndice G. La idea es buscar representaciones de los tableros más compactas que los árboles, de modo que la búsqueda se pueda llevar a cabo de una manera más eficiente. En este caso, se explora el uso de grafos acíclicos que permiten una representación de las ramas del tablero mucho más compacta que el uso de árboles, y que en casos extremos puede llegar a ser exponencialmente menor que la representación como un árbol.

Otra de las ideas es reducir aún más el trabajo que se realiza durante la prueba a cambio de un preprocesamiento mayor. En este caso, se construye un grafo que representa un tablero parcialmente extendido, en que las fórmulas tipo- α y tipo- β están completamente expandidas, por lo que no se considerarán ya durante la prueba.

En cuanto a la representación de los grafos, comencemos por la sintaxis usada. El grafo-tablero más simple consistirá en un solo nodo etiquetado con el átomo 1, que se empleará como un marcador de los finales de rama. Además, si F y G son grafos-tablero, $F \vee G$ será el grafo-tablero que tiene un nodo superior etiquetado con el operador \vee desde donde dos arcos conducen a los nodos superiores de los grafos F y G . También, $F \wedge G$ será el grafo-tablero obtenido desde G añadiéndole un nuevo nodo superior n^0 sobre el nodo superior original de G . El nodo n^0 se etiqueta con F . Así resulta posible representar grafos anidados, lo que se usará para tratar la cuantificación universal. En términos más formales, la siguiente función asigna para cada fórmula en NNF, un grafo-tablero que representa su tablero parcialmente extendido:

Definición 3.1 (Conversión de NNF en grafo-tablero) *Sea F una fórmula*

en NNF, y sea 1 la constante atómica verdadera:

$$\text{grafotab}(F) = \begin{cases} F \wedge 1 & \text{si } F \text{ es un literal} \\ \text{grafotab}(A) \left[\frac{1}{\text{grafotab}(B)} \right] & \text{si } F = (A \wedge B) \\ \text{grafotab}(A) \vee \text{grafotab}(B) & \text{si } F = (A \vee B) \\ (\forall x \text{ grafotab}(F')) \wedge 1 & \text{si } F = \forall x F' \end{cases}$$

donde

$$F \left[\frac{G}{G'} \right] = \begin{cases} A \wedge (B \left[\frac{G}{G'} \right]) & \text{si } F = A \wedge B \\ (A \left[\frac{G}{G'} \right]) \vee (B \left[\frac{G}{G'} \right]) & \text{si } F = A \vee B \\ G' & \text{si } F = G \end{cases}$$

Las fórmulas universalmente cuantificadas se representan construyendo una representación de un tablero completamente extendido del alcance del cuantificador y poniéndolo dentro de un nodo donde hay una referencia a la variable cuantificada. Así se produce un anidamiento de grafos. En el apéndice G puede encontrarse el programa que aparece en [PS99] para convertir fórmulas en NNF a grafos-tablero, implementando la definición anterior. Realiza un uso muy eficiente de las listas de diferencia para ahorrar espacio en las representaciones. La llamada a `tgraph(+Fm1, -G)` requiere sólo un esfuerzo lineal con respecto a la longitud de la fórmula `Fm1`, devolviendo `G`, un grafo-tablero que representa un tablero parcialmente expandido de `Fm1`.

Veamos un ejemplo de transformación a grafo-tablero de la fórmula proposicional $(p \wedge q) \vee r$:

$$\begin{aligned} \text{grafotab}((p \wedge q) \vee r) &= \text{grafotab}(p \wedge q) \vee \text{grafotab}(r) = \\ \left(\text{grafotab}(p) \left[\frac{1}{\text{grafotab}(q)} \right] \right) \vee (r \wedge 1) &= \left((p \wedge 1) \left[\frac{1}{q \wedge 1} \right] \right) \vee (r \wedge 1) = \\ \left(p \wedge \left(1 \left[\frac{1}{q \wedge 1} \right] \right) \right) \vee (r \wedge 1) &= (p \wedge q \wedge 1) \vee (r \wedge 1) \end{aligned}$$

El grafo-tablero en que se transforma $(p \wedge q) \vee r$ puede verse en la figura 3.1.

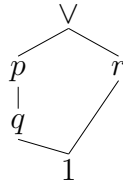


Figura 3.1: Grafo-tablero de $(p \wedge q) \vee r$

Para hacer demostraciones en grafos-tablero, probando que la fórmula original es inconsistente, debe comprobarse que cada camino que en el grafo-tablero conduce desde el nodo superior hasta 1 es inconsistente. Eso se hace recorriendo

recursivamente todos los caminos y guardando los literales que van apareciendo. Se considera cerrado un camino cuando existe una sustitución –que debe ser la misma para todos los caminos del grafo-tablero– que genera dos literales complementarios en dicho camino. La prueba tiene éxito cuando todos los caminos están cerrados. Si un camino se completa sin cerrarse, se debe hacer lo equivalente a aplicar una regla- γ en los tableros. En este caso, se vuelve a uno de los grafos anidados que representan fórmulas- γ por los que dicho camino pasa y simulamos la aplicación de la regla- γ añadiendo una copia del subgrafo al camino que estamos considerando y haciendo a continuación la sustitución de la variable cuantificada por una variable libre.

El demostrador que emplea los grafos-tablero es `gprove(+TGraph, +Gammas, +Lits, +FreeV, +VarLim)`, donde `TGraph` es un grafo-tablero producido por `tgraph/2`, `VarLim` limita el número de variables libres de cada rama durante la búsqueda de la prueba, `FreeV` contiene la lista de las variables libres aparecidas, `Lits` es la lista de literales aparecidos, y `Gammas` la lista de subgrafos universalmente cuantificados que se han encontrado en el camino que se está considerando. Comentemos dos de sus cláusulas:

Localización de cuantificadores universales. Vemos que cuando se localiza un subgrafo universalmente cuantificado, se continúa el camino fuera de tal subgrafo pero guardándolo en la lista `Gammas`:

```
gprove((all(X,Gr),Rest),Gammas,Lits,FreeV,VarLim):-!,
      gprove(Rest,[all(X,Gr)|Gammas],Lits,FreeV,VarLim).
```

Aplicación de cuantificadores universales. Cuando se llega a `true` –que en el programa representa el final de una rama– se toma el primero de los subgrafos universalmente cuantificados que se habían encontrado y sólo si no se ha llagado al límite de variables libres permitidas –la misma comprobación que en `leanAP`– se realiza una sustitución de la variable cuantificada por una nueva variable libre –que se guardará con las anteriores– y se visita el subgrafo. Además, el subgrafo cuantificado vuelve a guardarse al final de la lista `Gammas`:

```
gprove(true,[all(X,Gr)|Gammas],Lits,FreeV,VarLim):-!,
      \+ length(FreeV,VarLim),
      copy_term((X,Gr,FreeV),(X1,Gr1,FreeV)),
      append(Gammas,[all(X,Gr)],Gammas1),
      gprove(Gr1,Gammas1,Lits,[X1|FreeV],VarLim).
```

3.4. Tableros como BDDs

Otra representación muy compacta de los tableros se puede hacer a través de Diagramas Binarios de Decisión (BDD). Comentamos ahora brevemente la

implementación que aparece en [PS99], que puede encontrarse en el apéndice G. Mediante estos diagramas se representan las expresiones de tipo *if-then-else* agrupadas en el conjunto SH , que es el más pequeño² que verifica:

1. $\{0, 1\} \subset SH$
2. Si A es una fórmula atómica y $B_-, B_+ \in SH$ entonces $sh(A, B_-, B_+) \in SH$
3. Si $B, B_-, B_+ \in SH$ entonces $sh((\forall x B), B_-, B_+) \in SH$

La semántica de $sh(A, C, B)$ se define como *si A entonces B, en otro caso C*, es decir: $(A \rightarrow B) \wedge (\neg A \rightarrow C)$. Los elementos de SH constituyen una clase de fórmulas que pueden representarse gráficamente a través de BDDs. Además, para cada fórmula en NNF, existe una fórmula que pertenece a SH y que es equivalente a ella. Podemos obtenerla mediante la función definida a continuación.

Definición 3.2 Sea F una fórmula de primer orden en NNF. Entonces,

$$f2Sh(F) = \begin{cases} f2Sh(A) \left[\frac{1}{f2Sh(B)} \right] & \text{si } F = (A \wedge B) \\ f2Sh(A) \left[\frac{0}{f2Sh(B)} \right] & \text{si } F = (A \vee B) \\ sh((\forall x f2Sh(A)), 0, 1) & \text{si } F = \forall x A \\ sh(F, 0, 1) & \text{si } F \text{ es un literal positivo} \\ sh(F, 1, 0) & \text{si } F \text{ es un literal negativo} \end{cases}$$

donde

$$sh(A, B, C) \left[\frac{G}{G'} \right] = \begin{cases} sh(A, G', C) & \text{si } B = G \\ sh(A, B, G') & \text{si } C = G \end{cases}$$

Como ilustración, veamos la transformación de $A \vee (B \wedge C)$ en una expresión de SH :

$$\begin{aligned} f2Sh(A \vee (B \wedge C)) &= f2Sh(A) \left[\frac{0}{f2Sh(B \wedge C)} \right] = \\ sh(A, 0, 1) \left[\frac{0}{f2Sh(B \wedge C)} \right] &= sh(A, f2Sh(B \wedge C), 1) = \\ sh(A, f2Sh(B) \left[\frac{1}{f2Sh(C)} \right], 1) &= sh(A, sh(B, 0, 1) \left[\frac{1}{f2Sh(C)} \right], 1) = \\ sh(A, sh(B, 0, f2Sh(C)), 1) &= sh(A, sh(B, 0, sh(C, 0, 1)), 1) \end{aligned}$$

A partir de la conversión de una fórmula en NNF a una fórmula de SH , la transformación a BDD es inmediata. Sea ϕ un fórmula en NNF, y $f2Sh(\phi) =$

²El uso de las iniciales SH para representar este conjunto de expresiones es debido a que a estos diagramas, por su carácter binario, se les conoce también como grafos de Shannon.

$sh(A, B, C)$. El BDD de $sh(A, B, C)$ es un árbol que tiene como raíz un nodo etiquetado con A , del que salen dos arcos, uno etiquetado con “ $-$ ” que va hacia el BDD de B , y otro etiquetado con “ $+$ ” que va hacia el BDD de C . Además, el BDD de 0 es un solo nodo etiquetado con 0, y el BDD de 1 un solo nodo etiquetado con 1. En la figura 3.2 puede verse el BDD de la fórmula $A \vee (B \wedge C)$, construido a partir de su forma *if-then-else*, obtenida más arriba.

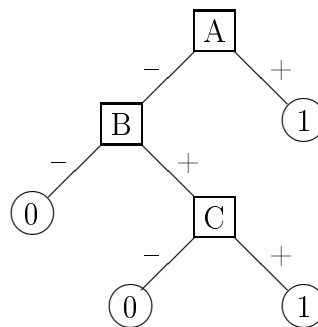


Figura 3.2: BDD de $A \vee (B \wedge C)$

Los BDDs representan tableros construidos con una modificación de la regla- β que usa lemas. Brevemente, el tratar una fórmula tipo- β , como $\eta_1 \vee \eta_2$, en vez de añadir a una rama η_1 y a la otra η_2 , con esta variante de la regla- β se añade a una rama η_1 y a la otra $\neg\eta_1 \wedge \eta_2$. Decimos que esta modificación de la regla- β emplea lemas porque el significado que tiene el que la rama que contiene η_1 se cierre, es que se ha probado $\neg\eta_1$, por lo que esto último puede añadirse a la otra rama sin pérdida de la corrección. Más detalles pueden encontrarse en el trabajo de Posegga y Schmitt [PS99].

En un BDD, los caminos que conducen a 1 se corresponden con las ramas de un tablero –construido con la modificación de la regla- β explicada– en que los literales que aparecen en nodos del BDD de los que el camino correspondiente sale por el arco etiquetado con “ $-$ ” aparecen negados y los literales que aparecen en nodos de los que el camino sale por el arco etiquetados con “ $+$ ” aparecen en forma positiva. Según esto, los dos caminos que conducen a hojas 1 en el BDD de $A \vee (B \wedge C)$ –véase de nuevo la figura 3.2– representan dos ramas con los conjuntos de literales $\{\neg A, B, C\}$ y $\{A\}$.

En los apéndices se puede encontrar el predicado `f2bdd(+Fml,+True,+False,-BDD)`, que implementa la definición anterior de tal forma que si `Fml` representa una fórmula en NNF y `True` y `False` son, respectivamente, las constantes 1 y 0 de la definición anterior, entonces `BDD` es el BDD que representa $f2Sh(Fml)$.

Una vez tenemos un BDD, como los caminos que conducen hacia 1 se corresponden con las ramas del tablero, la búsqueda de la prueba consiste en recorrer tales caminos tratando de encontrar en ellos literales complementarios que hagan inconsistente tal camino. En los apéndices puede encontrarse la definición del

predicado `bdd_prove(+BDD, +Gammas, +Lits, +FreeV, +VarLim)`, cuyos argumentos tienen el mismo significado que `gprove/5`, excepto en que la entrada, `BDD`, es ahora un BDD.

Veamos la cláusula principal de `bdd_prove/5`:

```
bdd_prove((A -> B; C), Gammas, Lits, FreeV, VarLim) :-
    (member_unify(-A, Lits);
     bdd_prove(B, Gammas, [A|Lits], FreeV, VarLim)),
    (member_unify(A, Lits);
     bdd_prove(C, Gammas, [-A|Lits], FreeV, VarLim)).
```

El modo elegido para representar en Prolog las expresiones $sh(A, C, B)$ es $(A \rightarrow B; C)$. Por tanto, lo que hace esta cláusula es tratar de cerrar los dos caminos que parten de $(A \rightarrow B; C)$. Los caminos que contienen A se pueden cerrar si ocurre que o bien $-A$ está entre los literales aparecidos –eso es lo que se comprueba mediante `member_unify(-A, Lits)`– o si se cierran todos los caminos desde B –segundo término de la primera disyunción–. Para los caminos que pasan por C ocurre lo contrario.

Además, todos los caminos que llegan a 0 se consideran cerrados, por corresponder a modelos que no satisfacen la fórmula:

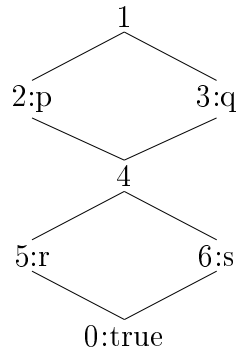
```
bdd_prove(0, _, _, _, _) :- !.
```

3.5. Compilación de la búsqueda

La otra modificación que Posegga y Schmitt presentan en [PS99] es la compilación de la búsqueda de la prueba. La idea, que parte de Stickel, es convertir fórmulas en programas que realizan la búsqueda de la prueba durante su ejecución. Si todas las versiones de lean^{AP} presentadas consistían en programar un metaintérprete que implementaba sistemas lógicos de demostración, el camino es ahora bien distinto. En este caso se trata de desarrollar, para cada fórmula, un programa que será procedimentalmente equivalente a la búsqueda de la prueba, pero no lógicamente. Para ello debe programarse un compilador que, dada una fórmula, genere el programa buscado.

El punto de partida que Posegga y Schmitt toman es la transformación de tableros en grafos que ya hemos presentado. Sin embargo, en este caso los grafos-tablero deben estar etiquetados, de modo que a cada nodo se asigne un número natural. Esto lo lleva a cabo el predicado `tgraph2(+Formula, -Graph)`, que dada una fórmula en NNF, construye un grafo-tablero equivalente y etiqueta sus nodos. Como ejemplo, la figura 3.3 muestra el grafo-tablero etiquetado para la fórmula $(p \vee q) \wedge (r \vee s)$.

A partir de estos grafos etiquetados, se realizará la compilación. Por ejemplo, para el grafo mostrado en la figura 3.3, el programa que se generará es:

Figura 3.3: Grafo-tablero para $(p \vee q) \wedge (r \vee s)$

```

cierra(Lit,[L|Lits]):-
    (Lit = -Neg; -Lit = Neg)->
    (unify_with_occurs_check(Neg,L);
     cierra(Lit,Lits)).

start(N) :- node(1,_,[],N,[]),!.

node(0,B,P,MaxVars,[Id|Gamma]):-
    MaxVars > 0, MaxVars1 is MaxVars - 1,
    append(Gamma,[Id],NewGamma),
    node(Id,B,P,MaxVars1,NewGamma).

node(1, A, B, C, D) :-
    node(2, A, B, C, D),
    node(3, A, B, C, D).
node(2, A, B, C, D) :-
    (cierra(p, B);
     node(4, A, [p|B], C, D)).
node(4, A, B, C, D) :-
    node(5, A, B, C, D),
    node(6, A, B, C, D).
node(5, A, B, C, D) :-
    (cierra(r, B);
     node(0, A, [r|B], C, D)).
node(6, A, B, C, D) :-
    (cierra(s, B);
     node(0, A, [s|B], C, D)).
node(3, A, B, C, D) :-
    (cierra(q, B);
     node(4, A, [q|B], C, D)).

```

Veamos cómo está estructurado el predicado `node/5` para poder comentar el programa anterior. Cada llamada tiene la forma:

```
node(+Id, +Ligaduras, +Camino, +MaxVars, +Gamma)
```

donde

Id es la etiqueta que identifica el nodo correspondiente en el grafo.

Ligaduras es una lista que contiene las ligaduras que van sufriendo las variables del grafo.

Camino es el camino que se va construyendo, que contiene los literales que han aparecido.

MaxVars es el número máximo de variables que se permite.

Gamma es una lista que contiene las etiquetas de los nodos que contienen subgrafos universalmente cuantificados.

Como el programa que hemos mostrado se ha generado para una fórmula proposicional, los dos únicos argumentos que entran en juego son **Id** y **Path**. Además, las definiciones de **cierra/2**, **start/1** y la primera cláusula de **node/5** son comunes a todos los programas generados. En concreto, **start(+N)** sirve para arrancar el programa permitiendo en la búsqueda un máximo de **N** variables libres por camino. La definición de **cierra/2** nos es habitual, se trata de una cláusula para buscar literales complementarios. La cláusula para los nodos etiquetados con 0, que como hemos comentado pertenece a todos los programas compilados, tiene un comportamiento que se corresponde con el de la cláusula que en la búsqueda de pruebas en grafos-tablero se encargaba de instanciar los subgrafos cuantificados universalmente. De hecho, sólo llegamos a esa cláusula al alcanzar el final de los caminos –etiquetados con 0–. Entonces se comprueba si todavía puede introducirse una nueva variable libre, aunque en este caso el procedimiento es diferente: el número de variables libres permitidas comienza con **N** al lanzar el programa con **start(+N)** y cada vez que se introduce una variable libre se resta una unidad, por lo que hay que comprobar simplemente si **MaxVars** > 0. Entonces el nodo **Id** que estaba pendiente de visitar, el primero de la lista de **Gammas**, se coloca ahora al final, y se acude a tal nodo.

En cuanto a las cláusulas específicas del programa compilado en nuestro ejemplo, encontramos diferencias entre las cláusulas que representan a los nodos 1 y 4 y las de los restantes nodos. La cláusula del nodo 4, por ejemplo, que en el grafo se ve que divide el camino en dos, introduce dos nuevos objetivos, que consisten en visitar los nodos 5 y 6, con el mismo **Path** –en este caso el argumento **B**– con el que se alcanzó el nodo 4. Sin embargo, al visitar el nodo 2, la correspondiente cláusula tiene otro comportamiento. Ahora, o bien *p* cierra con el **Path** acumulado hasta ese momento, o hay que visitar el nodo 4, añadiendo *p* al **Path**. En el caso de los nodos 5 y 6, si no se cierra el **Path**, hay que ir al nodo 0. Pero como en este caso la lista de **Gammas** será vacía la prueba fallará. De hecho, el programa falla al ejecutar **start(+N)** para todo valor de **N**, al tratarse de una fórmula satisfacible.

En el apéndice G se encuentra definido el predicado **comp/3**, encargado de compilar los grafos etiquetados para generar programas como el comentado.

3.6. Variantes proposicionales

En el apéndice H se pueden encontrar versiones de varios de los demostradores anteriormente presentados, adaptados para lógica proposicional. El motivo es que lean^{TAP} , al estar creado para trabajar con fórmulas de primer orden, no resulta un buen demostrador proposicional, pero sin embargo, con ciertas modificaciones, puede mejorar mucho su rendimiento, como más adelante veremos. Ya no son necesarios ni el límite que establecía la profundidad de los tableros, ni la lista de variables libres. Además, al no contener variables las fórmulas no hay que usar un unificador correcto, sino que basta con el unificador común de Prolog. Sin embargo, los dos cambios más profundos tienen que ver con las razones que exponemos a continuación.

Respecto a los cierres, lean^{TAP} cierra todas las ramas en que es posible una unificación correcta que produce dos literales complementarios. Sin embargo, lean^{TAP} permite un punto de elección en el momento de cerrar una rama, de tal forma que puedan contemplarse como posibles varios cierres en cada rama. Así, si se encuentra una rama abierta, antes de considerar que la prueba falla, lean^{TAP} vuelve sobre las ramas ya cerradas para intentar encontrar nuevas unificaciones que cierren tales ramas con otros pares de literales complementarios, lo que a veces, en primer orden, sirve para cerrar ramas que con las primeras unificaciones elegidas resultaban abiertas. Esto, que es necesario en primer orden para garantizar la completud, en lógica proposicional no tiene sentido. Por tanto, cuando una rama se cierre, se introducirá un corte que impedirá que Prolog pueda volver sobre las ramas ya cerradas. Con esto se mejorará mucho la eficiencia de los demostradores, sobre todo en los casos en que la fórmula que intenta probarse es satisfacible, ya que en el momento en que se detecta la primera rama completa abierta la prueba falla.

Además, cuando lean^{TAP} encuentra un literal, si no logra cerrar el tablero lo añade a la lista de literales encontrados. Así, el número de literales que hay en una rama no tiene un límite propio. Puede haber literales de tipo $p(\text{Var1})$ y $p(\text{Var2})$ que tiene sentido mantener, a pesar de ser unificables $\neg\text{Var1}$ y Var2 son variables libres-, ya que podrían dar lugar a cierres que emplean unificaciones diferentes. Sin embargo, en lógica proposicional el número máximo de literales diferentes que puede haber en una rama abierta es el número de átomos diferentes de la fórmula para la que se construye el tablero. No tiene sentido añadir literales repetidos a una rama, ya que aumentará innecesariamente el espacio consumido por la prueba. Por tanto, al aparecer un nuevo literal Lit , se comprobará no sólo si está ya su complementario NegLit –para cerrar la rama–, sino si ya está ese mismo literal Lit . En el momento en que se encuentre que el literal Lit ya está en la rama, puede detenerse la búsqueda de NegLit , pues seguro que no se encontrará, ya que en tal caso se habría detectado antes una contradicción. Por tanto, esto reduce el tamaño de las ramas –que ahora tendrán menos literales, como máximo el número de átomos diferentes de la fórmula de partida– y también el tiempo de

computación.

El predicado `busca(+L,+Lits,-R)` desempeña una función fundamental en estas versiones. Siendo `L` un literal y `Lits` una lista de literales, tiene éxito si `Lits` contiene o bien `L`, unificando `R` con `i` de *idéntico*, o su complementario, en cuyo caso `R` será `n` de *negación*. En caso de que `Lits` no contenga ni `L` ni su complementario, falla. Veamos la definición, pues se lleva a cabo con menos esfuerzo del que lean^{TAP} emplea para buscar simplemente si está el literal complementario, al no necesitarse para lógica proposicional el unificador correcto:

```
busca(L,[L|_],i):-!.
busca(-L,[L|_],n):-!.
busca(L,[-L|_],n):-!.
busca(L,[_|R],X):-
    busca(L,R,X).
```

Las modificaciones comentadas pueden verse, por ejemplo, en la cláusula tercera de `prove/3` (apéndice H):

```
prove(Lit,UE,Lits):-!,
    busca(Lit,Lits,R)-> (R=n;(UE=[U|E],prove(U,E,Lits)));
    (UE=[U|E],prove(U,E,[Lit|Lits])).
```

Se trata de la variante proposicional de la versión primera de lean^{TAP} . Si se encuentra el literal `Lit`, siendo `UE` la lista de fórmulas que esperan tratarse y `Lits` la lista de literales encontrados, entonces se introduce un corte, con lo que sólo se permite una posibilidad a partir de este momento. Una de las características de las adaptaciones proposicionales que hemos hecho es que son programas completamente deterministas, lo que se notará mucho en la eficiencia. Además, se busca `Lit` o su complementario en `Lits`. Entonces puede ocurrir:

- Que el complementario de `Lit` esté en `Lits`, si `R=n`. En tal caso la búsqueda de prueba termina y tiene éxito.
- Que `Lit` esté en `Lits`, lo que equivale a `R=i`, aunque en este caso no aparece tal comprobación ya que es la única otra opción posible si `busca(Lit, Lits, R)` tuvo éxito. Entonces, como la rama no se ha cerrado, la prueba sólo puede continuar si en la lista `UE` de fórmulas por tratar hay al menos un elemento. En otro caso la prueba falla, al encontrarse una rama abierta. Si `UE=[U|E]`, entonces continúa la prueba por `prove(U,E,Lits)`, es decir, con la primera de las fórmulas por tratar, y tomando la misma lista de literales que había.
- En caso de que `busca(Lit, Lits, R)` falle, ni `Lit` ni su complementario están en `Lits`. Así que en este caso tampoco puede seguir la prueba si en `UE` no hay fórmulas. Si la prueba sigue, ahora sí se añade `Lit` a la lista de literales encontrados, por no estar repetido.

Como se puede prever, esta modificación supondrá un ahorro considerable de tiempo y espacio en la búsqueda de pruebas.

De alguna manera, todas las adaptaciones proposicionales siguen la idea de la cláusula comentada. En cuanto a las versiones modificadas, son:

- `prove/3`, que modifica la versión más simple de `leanAP`.
- `prove_abd/4`, que modifica `prove_abd/6`. Sirve para devolver las ramas abiertas de los tableros. En este caso la representación de los tableros será completamente fiel.
- `gprove/2`, que es una versión proposicional de la búsqueda de pruebas en grafos-tablero.
- `comp/1`, para compilar la búsqueda de pruebas.
- `bdd_prove/2`, que busca pruebas en BDDs.

No se han hecho modificaciones de las demás versiones de `leanAP`, como la que emplea listas fijas de variables libres, o la que usa variables universales. Al ser versiones que inciden en aspectos propios de la lógica de primer orden, se comportan igual que `leanAP` en el tratamiento de fórmulas proposicionales. En el apéndice H se encuentra el código comentado de todas las versiones adaptadas. Además, se incluye una adaptación proposicional de la transformación a NNF.

Capítulo 4

Comparaciones entre demostradores

4.1. Comparación entre demostradores de tableros

4.1.1. Descripción del test

Para este primer test hemos usado todas las versiones de lean^{TAP} que representan los tableros como árboles. En la página 129 pueden encontrarse los cuadros correspondientes a los resultados de este test. Cada cuadro está titulado con el nombre de la fórmula que se empleó para tal experimento. Se trata de fórmulas que se corresponden con problemas propuestos por Pelletier [Pel86]. En la página 125 pueden verse las fórmulas que hemos tomado. Se trata de un fragmento del fichero `leantest.pl` de la distribución original de lean^{TAP} . Mediante `fml(?Nom, ?Lim, ?Fml)` se declara cada uno de tales experimentos, siendo `Nom` la etiqueta que identifica la fórmula, `Lim` el número de variables libres que lean^{TAP} necesita introducir por rama del tablero para poder probar la fórmula, y `Fml` la propia fórmula en primer orden, sin pasar aún a NNF. A continuación mostramos las ocho fórmulas de Pelletier empleadas en este test:

pel24: $\neg\exists x (p(x) \wedge r(x)) \wedge \neg\exists x1 (s(x1) \wedge q(x1)) \wedge \forall x4 (p(x4) \rightarrow (q(x4) \vee r(x4))) \wedge (\neg\exists x2 p(x2) \rightarrow \exists y q(y)) \wedge \forall x3 ((q(x3) \vee r(x3)) \rightarrow s(x3))$

pel25: $\neg\exists x (q(x) \wedge p(x)) \wedge \exists x1 p(x1) \wedge \forall x2 (f(x2) \rightarrow (\neg g(x2) \wedge r(x2))) \wedge \forall x3 (p(x3) \rightarrow (g(x3) \wedge f(x3))) \wedge (\forall x4 (p(x4) \rightarrow q(x4)) \vee \exists z (p(z) \wedge r(z)))$

pel34: $\neg((\exists x \forall y (p(x) \leftrightarrow p(y)) \leftrightarrow (\exists u q(u) \leftrightarrow \forall w (q(w)))) \leftrightarrow (\exists x1 \forall y1 (q(x1) \leftrightarrow q(y1)) \leftrightarrow (\exists u1 p(u1) \leftrightarrow \forall w1 p(w1))))$

pel36: $\neg\forall x \exists y h(x, y) \wedge \forall x1 \exists y2 f(x1, y2) \wedge \forall x2 \exists y1 g(x2, y1) \wedge \forall x3 \forall y3 ((f(x3, y3) \vee g(x3, y3)) \rightarrow \forall z3 ((f(y3, z3) \vee g(y3, z3)) \rightarrow h(x3, z3)))$

pel37: $\neg\forall x \exists y r(x, y) \wedge \forall z \exists w \forall x1 \exists y1 (p(x1, z) \rightarrow ((p(y1, w) \wedge p(y1, z)) \wedge (p(y1, w) \rightarrow \exists u q(u, w)))) \wedge \forall x2 \forall z2 (\neg p(x2, z2) \rightarrow \exists y2 q(y2, z2)) \wedge (\exists x3 \exists y3 q(x3, y3)) \rightarrow \forall z3 r(z3, z3)$

pel38: $\neg(\forall x ((p(a) \wedge (p(x) \rightarrow \exists y (p(y) \wedge r(x, y)))) \rightarrow \exists z \exists w ((p(z) \wedge r(x, w)) \wedge r(w, z))) \leftrightarrow \forall x1 (((\neg p(a) \vee p(x1)) \vee \exists z1 \exists w1 ((p(z1) \wedge r(x1, w1)) \wedge r(w1, z1))) \wedge ((\neg p(a) \vee (\neg \exists y1 (p(y1) \wedge r(x1, y1)))) \vee \exists z2 \exists w2 ((p(z2) \wedge r(x1, w2)) \wedge r(w2, z2))))))$

pel43: $\neg\forall x \forall y ((q(x, y) \rightarrow q(y, x)) \wedge (q(y, x) \rightarrow q(x, y))) \wedge \forall x1 \forall y1 (q(x1, y1) \rightarrow \forall z ((f(z, x1) \rightarrow f(z, y1)) \wedge (f(z, y1) \rightarrow f(z, x1)))) \wedge \forall x2 \forall y2 (\forall z2 ((f(z2, x2) \rightarrow f(z2, y2)) \wedge (f(z2, y2) \rightarrow f(z2, x2))) \rightarrow q(x2, y2))$

pel45: $\neg\exists x (f(x) \wedge \neg\exists y (g(y) \wedge h(x, y))) \wedge \forall x1 ((f(x1) \wedge \forall y ((g(y) \wedge h(x1, y)) \rightarrow j(x1, y))) \rightarrow \forall y1 ((g(y1) \wedge h(x1, y1)) \wedge k(y1))) \wedge \neg\exists y2 (l(y2) \wedge k(y2)) \wedge \exists x2 ((f(x2) \wedge \forall y3 (h(x2, y3) \rightarrow l(y3))) \wedge \forall y11 ((g(y11) \wedge h(x2, y11)) \rightarrow j(x2, y11)))$

En cada uno de los cuadros aparece una primera columna titulada MÉTODO. Debajo aparecen los demostradores que se han empleado para probar cada fórmula. Se trata de:

prove, la versión más simple de `leanAP`, que se define en el fichero `leantap.pl` de la distribución original de `leanAP`, apéndice C. El predicado para llamarlo es `prove/2`.

prove_var1, versión de `prove` que integra la variante explicada en la página 27. El fichero en que se encuentra es `variacion1.pl`, en la página 97. Se llama con `prove_var1/2`.

prove_uv, versión de `leanAP` con variables universales. Está definido en el fichero `leantap.pl` y se llama con `prove_uv/2`.

prove_var1_uv, que emplea variables universales y listas cerradas de variables libres. Está definido en `variacion1_uv.pl` –página 101– y se llama con `prove_var1_uv/2`.

prove_oc, **prove_var1_oc**, **prove_uv_oc**, **prove_var1_uv_oc**, son versiones de los cuatro métodos anteriores empleando el unificador correcto de Prolog `unify_with_occurs_check` en vez de la implementación del algoritmo de Stickel.

Versiones que construyen tableros, que son las que llevan `abd` en el nombre del método y que modifican las ocho versiones anteriores.

El test consiste en que cada método trata de probar cada fórmula mediante una búsqueda en profundidad iterativa de tal manera que el límite de variables libres que pueden introducirse por rama se incrementa hasta encontrar un tablero cerrado. Las columnas `INFERENCIAS`, `MILISEG`, `BYTES` y `LÍMITE` son, respectivamente, el número de inferencias lógicas que cuesta el proceso, los milisegundos que tarda –el tiempo de la conversión a NNF está incluido–, el espacio consumido en bytes y el número de variables libres que fue necesario introducir por rama del tablero para que se cerrara. La columna `ERROR` avisa si hay algún error durante el intento de prueba. El sistema empleado fue SWI-Prolog 5.2.13 en un PC con CPU Intel Pentium IV 1700 MHz, con sistema operativo Mandrake Linux 9.2. En la página 126 puede verse el fichero que empleamos para realizar este test. Con `trata_experimento(+Nom,+Tpo)` se trata de probar la fórmula correspondiente al experimento llamado `Nom` permitiendo un máximo de `Tpo` segundos a cada método de prueba. Mediante `test(+Tpo)` se tratan todos los experimentos con todos los métodos. La salida del programa son los cuadros que aparecen en la página 129, con las estadísticas de las pruebas de cada fórmula.

4.1.2. Conclusiones

Veamos cómo influye en la eficiencia de lean^{TAP} cada una de las modificaciones que hemos introducido. Un caso representativo de lo que ocurre lo vemos en el resultado del test para la fórmula `pe134`. Se observa lo siguiente:

- El uso del unificador correcto de SWI-Prolog, `unify_with_occurs_check/2`, reduce aproximadamente a un tercio el número de inferencias necesarias respecto al uso del unificador de Stickel `unify/2`. El tiempo se reduce a la mitad, y el gasto de memoria es el mismo. Esto es debido a que es una implementación a más bajo nivel que el algoritmo de Stickel. Por tanto, convienen las versiones que emplean el unificador correcto de SWI-Prolog (las versiones que llevan `oc` en el nombre de su método).
- El uso de variables universales llega a aumentar considerablemente el tiempo, debido al manejo de las variables universales, que requiere un mayor número de inferencias. La memoria, aunque en menor grado, también aumenta. Ya vimos al explicar esta modificación, en la página 16, que hay casos en que conviene el empleo de variables universales, pero entre las fórmulas de nuestro test no se encuentran. Los autores de lean^{TAP} muestran una estadística en [BP95] según la cual `pe134` se resuelve antes con variables universales que sin ellas –usando SICStus Prolog–, pero esto no concuerda con nuestros resultados. Con los experimentos hechos hasta ahora, consideramos prescindible esta modificación, al no encontrar ningún caso donde resulte ventajosa.
- Las versiones de lean^{TAP} que devuelven las ramas abiertas (las que llevan

abd en los nombres de los métodos que las usan), no incrementan el número de inferencias necesarias, aunque el tiempo es un poco mayor y la memoria, aunque aumenta, no lo hace demasiado. Como demostradores, estas versiones son, por tanto, menos eficaces que las que no construyen tableros, pero si necesitamos los literales que aparecen en cada rama, como será nuestro caso cuando construyamos sistemas abductivos, nos permiten obtener la información que necesitamos sin un gran coste adicional.

- Por último, el uso de listas cerradas de variables libres –métodos de prueba que llevan `var1` en su nombre– hace que la profundidad de las pruebas llegue a ser mucho menor que en las versiones sin esta modificación, y nunca es mayor. En `pe134` sólo se requieren, usando listas cerradas de variables libres, 2 variables libres por rama, mientras que el límite sin esta modificación debe ser al menos 5. Hay casos en que la diferencia es aún más llamativa. La clave radica en que la versión original de `leanAP` requiere una nueva variable libre por cada vez que trata una fórmula- γ , así que si para cerrar un tablero hay que instanciar n veces alguna fórmula- γ , el límite nunca será inferior a n . Sin embargo, a veces esas n variables unifican unas con otras, y al cerrar el tablero las variables libres diferentes que quedan son menos, pongamos m tal que $m < n$. En tales casos, empleando listas cerradas de variables libres basta con poner el límite en m , ya que así se instancia cada fórmula- γ con m variables libres diferentes, y el tablero se cierra. Por tanto, realizando búsquedas en profundidad iterativa, esta variante logra encontrar las pruebas antes que la versión originaria de `leanAP`. Esto implica que el número de inferencias requerido disminuya incluso hasta en veinte veces respecto a las versiones que no usan esta modificación. El tiempo también se ve reducido en una proporción similar, y aunque la memoria aumenta, el incremento no llega ni al 50% en promedio. También pueden encontrarse casos en que el uso de memoria es menor. El mayor gasto de memoria es debido a que al instanciar todas las fórmulas- γ con todas las variables libres, se introducen generalmente más fórmulas de las necesarias. A la vista de los resultados podemos concluir que el empleo de listas cerradas de variables libres demuestra ser enormemente ventajoso.

A partir de las observaciones anteriores puede decirse que:

- Para demostrar teoremas, el mejor de entre los métodos propuestos es `prove_var1_oc`.
- Para obtener los literales que aparecen en cada rama del tablero, el mejor método de los propuestos es `prove_var1_abd_oc`. Además, si se usa como demostrador la eficiencia no es mucho menor –por lo general– que la de `prove_var1_oc`.

4.2. Comparación entre sistemas Prolog

4.2.1. Descripción del test

Este test consiste en repetir el test anterior con distintos sistemas Prolog, y a su vez con diferentes versiones de cada uno de ellos, para comparar la eficiencia de los mismos. Los sistemas que hemos empleado son:

SWI-Prolog. Compilador de Prolog desarrollado en la Universidad de Amsterdam. Hemos empleado dos versiones. La versión 5.2.13 es la que en el momento de escribir esta memoria se ofrecía en la web de SWI-Prolog, <http://www.swi-prolog.org>, como la última versión estable. Los resultados del test pueden verse en la página 129. Para la versión 5.3.10, que al hacer los test era la última versión en desarrollo, pueden verse los resultados en la página 134.

Yap Prolog. Es un compilador de Prolog que destaca por su eficiencia. Se encargan de su desarrollo las Universidades de Oporto y Río de Janeiro. La primera versión que hemos empleado de Yap Prolog es la 4.4.4, que al escribir esta memoria era la que se ofrecía en la web: <http://www.ncc.up.pt/~vsc/Yap/>. Los resultados para esta versión aparecen en la página 140. Además, usamos la versión 4.5.2, que descargamos de: [http://sourceforge.net/projects/yap/-proyecto SourceForge-](http://sourceforge.net/projects/yap/-proyecto%20SourceForge-). Para esta última versión, los resultados aparecen en la página 145.

GNU Prolog. Se trata de un compilador desarrollado por Daniel Diaz que tiene la peculiaridad de resolver problemas con restricciones sobre dominios finitos. Para el test, hemos empleado la versión 1.2.16, descargable desde la web: <http://gnu-prolog.inria.fr/>. A pesar de que GNU Prolog permite compilar los programas, no empleamos esta opción, y realizamos los tests en las mismas condiciones que con los demás sistemas. Los resultados se encuentran en la página 151.

En Yap Prolog y GNU Prolog no ha sido posible obtener el número de inferencias lógicas. La máquina empleada para estos tests es un PC con CPU Intel Pentium IV 1700Mhz, con sistema operativo Mandrake Linux 9.2.

El código que aparece en los apéndices de esta memoria fue escrito en principio para SWI-Prolog. Sin embargo, para los tests con Yap Prolog fue necesario adaptar el código. En Yap Prolog, ciertos predicados como `member/2` y `append/3` deben importarse de la librería `lists` cuando se quieran emplear, mediante

```
:- use_module(library(lists), [append/3, member/2]).
```

Pero donde hubo que modificar más el código fue en las versiones que emplean `unify_with_occurs_check/2`, debido a que cuando Yap encuentra que un

término $p(A)$ contiene la variable A que debería unificar con él –caso en que sólo es posible una unificación incorrecta– unifica $p(A)$ con $p(\text{'FoundVar'})$. Veamos la traza:

```
?- unify_with_occurs_check(A,p(A)).
(1) call:unify_with_occurs_check(_172,p(_172)) ?
(1) fail:unify_with_occurs_check(_172,p('FoundVar')) ?
no
```

El problema es que la unificación de la variable encontrada con 'FoundVar' va más allá del alcance de `unify_with_occurs_check/2`, quedando el término ya siempre ligado a $p(\text{'FoundVar'})$, como vemos a continuación:

```
?- S=p(A), (unify_with_occurs_check(A,S) -> true; true).
S = p('FoundVar') ?
yes
```

Se trata de un comportamiento que no aparece recogido en el estándar de Prolog [DEDC96], y que pensamos que resulta indeseable. En nuestro caso, hace que deba modificarse el código en las versiones de lean^{AP} que emplean el unificador correcto de Prolog. La solución más eficiente que encontramos puede verse en la siguiente cláusula, que recoge la transformación de todas las versiones que usan `unify_with_occurs_check/2`:

```
prove_oc(Lit,_,[L|Lits],_,_,_,_) :-
  (Lit = -Neg; -Lit = Neg) ->
  ((copy_term((Neg,L),(Neg2,L2)),
   unify_with_occurs_check(Neg2,L2),
   copy_term((Neg2,L2),(Neg,L)));
   prove_oc(Lit,[],Lits,_,_,_,_)).
```

Ahora, `unify_with_occurs_check/2` se aplica a dos nuevos términos que se crean, con la misma distribución de variables que los originales, mediante una llamada a `copy_term/2`. Si la unificación tiene éxito, se traslada a los términos originales mediante otra llamada a `copy_term/2`.

4.2.2. Conclusiones

A la vista de los resultados de este test, podemos hacer las siguientes observaciones:

Yap Prolog. En promedio, la versión 4.5.2 consume un poco menos memoria que la 4.4.4, pero emplea más tiempo en las pruebas. En los ejemplos de estos tests, las fórmulas no son muy grandes y estas diferencias no pueden apreciarse con claridad, pero en fórmulas mayores resulta obvio que Yap 4.5.2 es más lento. Ocurre por ejemplo con `palomar_4` –ver fichero `benchmark.pl`, página 159–, al demostrarse con el método `proposic` –página 166–. El siguiente cuadro muestra los resultados para las dos versiones:

```

Experimento: palomar_4 (Yap 4.4.4)
+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| propositic | - | no disp. | 875770 | 112 | - |
+-----+-----+-----+-----+-----+-----+

Experimento: palomar_4 (Yap 4.5.2)
+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| propositic | - | no disp. | 1406560 | 88 | - |
+-----+-----+-----+-----+-----+-----+

```

Mientras que Yap 4.4.4 tarda 875 segundos, Yap 4.5.2 requiere casi el doble. El gasto de memoria es levemente menor.

Es curioso lo que ocurre en el caso de los métodos de prueba que emplean el unificador correcto de Prolog. La modificación que hubo que hacer en el código para Yap –comentada más arriba– hace que requiera más memoria usando `unify_with_occurs_check` que si emplea el unificador correcto de Stickel, debido a la duplicación de términos que debe hacer. Pero en la versión 4.5.2, el uso de `unify_with_occurs_check` requiere menos tiempo que el algoritmo de Stickel, mientras que en la versión 4.4.4 ocurre lo contrario. Por tanto, la versión 4.5.2, aunque en general sea peor que la 4.4.4, mejora la eficiencia del unificador correcto.

SWI-Prolog. En este caso, la versión 5.3.10 sí supone una clara mejora en cuanto a la eficiencia con respecto a la 5.2.13. Tanto la memoria como el tiempo de prueba se reducen considerablemente.

Si comparamos SWI-Prolog 5.3.10 con Yap Prolog 4.4.4 –las dos versiones más rápidas de cada sistema– observamos que Yap es casi cuatro veces más rápido que SWI, a pesar de ser el gasto de memoria ligeramente superior.

GNU Prolog. Si comparamos GNU Prolog con SWI-Prolog 5.3.10, vemos que resulta más lento. Es así por tratarse de código interpretado, ya que cuando se compila, GNU resulta alrededor del doble de rápido que SWI. En cuanto al uso de memoria, la mayoría de los métodos necesitan más espacio con GNU que con SWI. Sin embargo hay cuatro métodos, todos ellos con listas cerradas de variables libres, que requieren menos espacio con GNU que con SWI. Si, como hemos sugerido, el incremento de espacio que tales métodos suponen en SWI es debido al mayor número de términos –muchos innecesarios– que introducen, pensamos que el hecho de que en GNU consuman menos espacio debe ser porque este compilador consigue representar los términos de una forma más compacta.

Comparando GNU con Yap 4.4.4 observamos que GNU consume menos memoria, pero es mucho más lento.

En promedio, las observaciones anteriores nos conducen a concluir que Yap-4.4.4 es el compilador más eficiente de los que hemos probado. Para el resto de los tests emplearemos SWI-Prolog, por ser el compilador que dispone de mayor documentación. Preferiremos la versión 5.2.13 pues aunque es algo menos eficiente que la 5.3.10, es la versión estable al escribir esta memoria. Como hemos comprobado que los programas son portables a otros sistemas, cuando se desee podrán emplearse otros compiladores.

4.3. Comparación con fórmulas proposicionales

4.3.1. Descripción del test

4.3.1.1. Familias de fórmulas

Para este test se usaron las fórmulas proposicionales generadas con los programas que aparecen en la página 159, que en su mayoría provienen de [Dyc97]. Mostramos a continuación cada una de tales familias. Seguimos la misma notación que en [Dyc97], de modo que $\&\&_{\text{rango}}$ indica una conjunción múltiple, y $\vee\vee_{\text{rango}}$ una disyunción múltiple.

dB. Esta familia recibe su nombre por haberla propuesto de Brujin. Las instancias pares no son satisfacibles, mientras que las impares sí lo son. La forma que tienen es:

$$\text{dB}(n) =_{\text{def}} \neg(\text{LHS}(n) \Rightarrow \text{RHS}(n))$$

$$\text{RHS}(n) =_{\text{def}} \&\&_{\{i=1..n\}} p(i)$$

$$\text{LHS}(n) =_{\text{def}} \&\&_{\{i=1..n\}} ((p(i) \Leftrightarrow p(i+1)) \Rightarrow \text{RHS}(n))$$

donde la suma se computa módulo n . Como ejemplo, veamos $\text{dB}(3)$:

$$\neg(((p(1) \Leftrightarrow p(2)) \Rightarrow p(1) \&p(2) \&p(3)) \& ((p(2) \Leftrightarrow p(3)) \Rightarrow p(1) \&p(2) \&p(3)) \& ((p(3) \Leftrightarrow p(1)) \Rightarrow p(1) \&p(2) \&p(3)) \Rightarrow p(1) \&p(2) \&p(3))$$

Por su cantidad de equivalencias, que se expanden al transformar las fórmulas a NNF, esta familia producirá un gran número de ramas en el tablero.

palomar. Son las clásicas fórmulas del teorema de Ramsey o del palomar. Son fórmulas no satisfacibles con la forma:

$$\text{palomar}(n) =_{\text{def}} \neg(\text{izq}(n) \Rightarrow \text{der}(n))$$

$$\text{izq}(n) =_{\text{def}} \&\&_{\{p=1..n+1\}} (\vee\vee_{\{j=1..n\}} \text{occ}(p, j))$$

$$\text{der}(n) =_{\text{def}} \vee\vee_{\{h=1..n, p_1=1..n+1, p_2=\{p_1+1\}..n+1\}} s(p_1, p_2, h)$$

$$s(p_1, p_2, h) =_{\text{def}} \text{occ}(p_1, h) \& \text{occ}(p_2, h)$$

La cantidad de literales y disyuntores que intervienen en estas fórmulas, producirán igualmente gran cantidad de ramas. Como ejemplo, veamos $\text{palomar}(2)$:

$$\begin{aligned} & -((o(1,1) \vee o(1,2)) \& (o(2,1) \vee o(2,2)) \& (o(3,1) \vee o(3,2)) \Rightarrow \\ & o(1,1) \& o(2,1) \vee o(1,1) \& o(3,1) \vee o(2,1) \& o(3,1) \vee o(1,2) \& \\ & o(2,2) \vee o(1,2) \& o(3,2) \vee o(2,2) \& o(3,2)) \end{aligned}$$

franzén. Se trata de una familia de fórmulas no satisfacibles:

$$\text{franzén}(n) =_{def} -(((\text{IZQ}(n) \vee \text{DER}(n)) \Rightarrow f) \Rightarrow f)$$

$$\text{IZQ}(n) =_{def} \&\&_{\{i=1..n\}} p(i)$$

$$\text{DER}(n) =_{def} \vee\vee_{\{i=1..n\}} (p(i) \Rightarrow f)$$

El número de ramas que generan no es demasiado grande. Como ejemplo, veamos $\text{franzén}(2)$:

$$-((p(1)\&p(2) \vee ((p(1)\Rightarrow f) \vee (p(2)\Rightarrow f))\Rightarrow f)\Rightarrow f)$$

schwicht. Se trata de una clase de fórmulas no satisfacibles propuesta por Schwichtenberg tal que el tamaño de la demostración normal por deducción natural de la n -ésima fórmula es siempre exponencial en n . Tienen la siguiente forma:

$$\text{schwicht}(n) =_{def} -(\text{ant_sch}(n) \Rightarrow p(n))$$

$$\text{ant_sch}(n) =_{def} p(n) \& \&\&_{\{i=1..n\}} (p(i) \Rightarrow (p(i) \Rightarrow p(i-1)))$$

Como ejemplo, $\text{schwicht}(2)$:

$$-(p(2)\& (p(1)\Rightarrow p(1)\Rightarrow p(0))\& (p(2)\Rightarrow p(2)\Rightarrow p(1)\Rightarrow p(0)))$$

kk. Esta familia debe su nombre a que fue inicialmente propuesta por Korn y Kreitz. La eficiencia de los demostradores con estas fórmulas depende del orden en que se guarden los antecedentes de las implicaciones. Entonces, lo que se hace en [Dyc97] es generar dos versiones (cada una se prueba mejor guardando los antecedentes de cierta manera) y unir las conjuntamente. Debido a esto, estas fórmulas contendrán gran número de subfórmulas repetidas. La forma que tienen es:

$$\text{kk}(n) =_{def} -((\text{kk_izq}(n)\Rightarrow f) \& (\text{kk_der}(n)\Rightarrow f))$$

$$\text{kk_izq}(n) =_{def} ((a(0)\Rightarrow f)\& ((b(n)\Rightarrow b(0)) \Rightarrow a(n))) \&$$

$$\&\&\&_{\{i=1..n\}} ((b(i-1)\Rightarrow a(i)) \Rightarrow a(i-1))$$

$$\text{kk_der}(n) =_{def} \&\&\&_{\{i=n..1\}} ((b(i-1)\Rightarrow a(i)) \Rightarrow a(i-1)) \&$$

$$((a(0)\Rightarrow f)\& ((b(n)\Rightarrow b(0)) \Rightarrow a(n)))$$

Como ejemplo, $\text{kk}(2)$:

$$-(((a(0)\Rightarrow f)\& ((b(2)\Rightarrow b(0))\Rightarrow a(2))\& ((b(0)\Rightarrow a(1))\Rightarrow a(0))\&$$

$$((b(1)\Rightarrow a(2))\Rightarrow a(1))\Rightarrow f)\& (((b(1)\Rightarrow a(2))\Rightarrow a(1))\& (((b(0)\Rightarrow$$

$$a(1))\Rightarrow a(0))\& (((b(2)\Rightarrow b(0))\Rightarrow a(2))\& (a(0)\Rightarrow f)))\Rightarrow f))$$

equiv. Fórmulas no satisfacibles, basadas en la asociatividad de la equivalencia, que producirán en los tableros gran cantidad de ramas. Como ejemplo, $\text{equiv}(3)$:

$$-(((a(1)\Leftrightarrow a(2))\Leftrightarrow a(3))\Leftrightarrow (a(3)\Leftrightarrow (a(2)\Leftrightarrow a(1))))$$

tipo1. Fórmulas muy simples que producen tableros cerrados en un pequeño número de pasos. La forma que tienen es:

$$\begin{aligned} \text{equiv}(n) &=_{def} \text{conj}(n) \ \& \ \neg \text{conj}(n) \\ \text{conj}(n) &=_{def} \ \&\&_{i=1..n}\{p(i)\} \\ \text{Así, tipo1}(2): \\ p(2)\&p(1)\& \ \neg \ (p(2)\&p(1)) \end{aligned}$$

tipo2. Estas fórmulas no satisfacibles están basadas en la transitividad de la equivalencia. Tienen la forma:

$$\begin{aligned} \text{tipo2}(n) &=_{def} \ \neg (\text{tipo_2_izq}(n) \Rightarrow (p(1) \Leftrightarrow p(n))) \\ \text{tipo_2_izq}(n) &=_{def} \ \&\&_{i=1..(n-1)}\{p(i) \Leftrightarrow p(i+1)\} \\ \text{Así, tipo2}(3) \text{ resulta:} \\ \neg ((p(1) \Leftrightarrow p(2)) \ \& \ (p(2) \Leftrightarrow p(3))) \Rightarrow (p(1) \Leftrightarrow p(3)) \end{aligned}$$

tipo3. Fórmulas no satisfacibles muy similares a las anteriores, aunque en este caso emplean la implicación, por lo que en número de ramas a que darán lugar es menor. Tienen, pues, la forma:

$$\begin{aligned} \text{tipo3}(n) &=_{def} \ \neg (\text{tipo_2_izq}(n) \Rightarrow (p(1) \Rightarrow p(n))) \\ \text{tipo_3_izq}(n) &=_{def} \ \&\&_{i=1..(n-1)}\{p(i) \Rightarrow p(i+1)\} \\ \text{De modo que tipo3}(3) \text{ es:} \\ \neg ((p(1) \Rightarrow p(2)) \ \& \ (p(2) \Rightarrow p(3))) \Rightarrow (p(1) \Rightarrow p(3)) \end{aligned}$$

tipo4. Familia en que las instancias pares son satisfacibles, y las impares no. Su forma es: $\text{tipo4}(n) =_{def} \ \neg (\text{tipo_2_izq}(n) \Rightarrow (p(1) \Leftrightarrow p(n)))$

$$\begin{aligned} \text{tipo_4_izq}(n) &=_{def} \ \&\&_{i=1..(n-1)}\{\neg (p(i) \Leftrightarrow p(i+1))\} \\ \neg (\neg (p(1) \Leftrightarrow p(2)) \ \& \ \neg (p(2) \Leftrightarrow p(3))) \Rightarrow (p(1) \Leftrightarrow p(3)) \end{aligned}$$

tipo5. Familia de fórmulas no satisfacibles. Como ejemplo, $\text{tipo5}(2)$ tiene la forma:

$$(\neg p(1) \vee \neg p(2)) \ \& \ (\neg p(1) \vee p(2)) \ \& \ (p(1) \vee \neg p(2)) \ \& \ (p(1) \vee p(2))$$

Como se puede observar, son fórmulas en forma normal conjuntiva que contienen una disyunción elemental por cada posible valoración de las variables proposicionales. Producirán gran cantidad de ramas, cada una de ellas con muchos literales repetidos.

tipo6. Familia de fórmulas satisfacibles que resultan de quitar a las fórmulas tipo5 una de las disyunciones. Así, $\text{tipo6}(2)$ resulta ser:

$$(\neg p(1) \vee p(2)) \ \& \ (p(1) \vee \neg p(2)) \ \& \ (p(1) \vee p(2))$$

4.3.1.2. Demostradores

Los demostradores basados en lean^{TAP} que se emplearon para este test son:

prove_oc, prove_abd_oc, que ya formaban parte de los tests anteriores.

Otras variantes de lean^{TAP} no aparecen en este test porque las modificaciones que introducen sólo influyen al tratar fórmulas de primer orden –variables universales o listas fijas de variables–.

proposic, que es un método proposicional de evaluación de fórmulas mediante tablas de verdad, tomado de J. A. Alonso y J. Borrego [AB02]. Puede encontrarse este programa, con mínimas adaptaciones para las fórmulas de este test, en la página 166.

prueba_tgraph, **prueba_comp**, **prueba_bdd**, que son, respectivamente, las versiones de lean^{AP} que transforman los tableros en grafos, en programas Prolog, y en BDDs –página 107–.

prop_prove, **prop_prove_abd**, **prop_tgraph**, **prop_comp**, **prop_bdd**, versiones proposicionales de lean^{AP} que aparecen en el apéndice H.

Los resultados de este test pueden consultarse en la página 171. Al ser fórmulas proposicionales, los métodos de primer orden que se usaron no realizaron búsquedas en profundidad iterativa, sino que construyeron el tablero sólo para el límite de variables libres igual a cero. Además, cuando un método falla aparece en la columna **ERROR** el símbolo **Sat**, indicando que la fórmula es satisfacible. Las pruebas se realizaron con SWI-Prolog 5.2.13, siendo el tiempo máximo de prueba para cada método de 600 segundos, y el tamaño de las pilas *local*, *global* y *tail* de 20 megabytes cada una. Cuando algún método no puede demostrar alguna fórmula en el tiempo máximo aparece en mensaje de error **Tiempo**. Si se agota alguna pila, también se produce un error que lo indica. El tiempo que se muestra incluye el preprocesamiento que cada método tenga que hacer.

Debajo de cada cuadro aparecen los datos del intento de prueba de cada fórmula con Otter-3.3 (Organized Techniques for Theorem-proving and Effective Research), un demostrador para primer orden basado en resolución, programado en ANSI C en el Argonne National Laboratory de la Universidad de Chicago. Para cada fórmula se permitió un máximo de 600 segundos y 40 megabytes. Se empleó un PC con CPU Intel Pentium III 866 MHz, con sistema operativo Mandrake Linux 9.2.

4.3.2. Conclusiones

Lo primero que puede constatarse a la vista de los resultados del test es que los métodos optimizados para lógica proposicional resuelven los problemas siempre de modo más eficiente que las correspondientes versiones para primer orden. Al comentar las adaptaciones proposicionales ya vimos las razones que explican estos resultados, siendo la principal de ellas que los programas proposicionales son completamente deterministas, evitando puntos de elección inútiles que sin embargo los demostradores para primer orden deben mantener. Hay fórmulas en que los únicos métodos que tienen éxito son los proposicionales, rebasando los de primer orden las pilas o el tiempo máximo.

Dentro de los métodos proposicionales, encontramos que para cada tipo de fórmula hay siempre algún demostrador que destaca frente a los demás. Concretamente:

dB. Para las instancias impares, que no son satisfacibles, la búsqueda en el tablero debe pasar por todas las ramas, comprobando que todas son cerradas. Por ello, el mejor método en estos casos es `prop_bdd`, pues el mayor tiempo de preprocesamiento que emplea le resulta rentable al disponer de una representación del tablero donde puede realizar la búsqueda de forma más eficiente –sólo con una cláusula determinista–. Para las instancias pares, sin embargo, al ser satisfacibles basta con encontrar una rama abierta, por lo que no hay que recorrer todo el tablero. En este caso el mejor método es `prop_prove_abd`, que emplea menos tiempo en el preprocesamiento, aunque lo siguen de cerca `prop_comp` y `prop_bdd`. Resulta curioso que pese a parecer `prop_prove` y `prop_prove_abd` tan similares, en la fórmula `dB_8` los resultados sean tan diferentes, siendo despreciable el tiempo que `prop_prove_abd` tarda en descubrir que es satisfacible, y no pudiendo `prop_prove` por falta de tiempo, agotando los 10 minutos de que disponía.

palomar. También en este caso la mejor estrategia es la de `prop_bdd`, empleando un mayor esfuerzo en el preprocesamiento que en la prueba.

franzén. En este caso el mejor método es `prop_prove`, aunque `prop_prove_abd` le sigue a poca distancia. Al ser fórmulas que no generan gran número de ramas, las estrategias más directas son mejores, por requerir menor preprocesamiento. Si en las instancias pares de `dB` el método `prop_prove_abd` resultaba mucho más eficiente que `prop_prove` porque al estar diseñado para encontrar modelos trabaja mejor reconociendo fórmulas satisfacibles, ahora, ante fórmulas no satisfacibles como `franzén`, ocurre lo contrario.

schwicht. En este caso también el tablero contiene un gran número de ramas, con lo que nos encontramos ante otro éxito de `prop_bdd`.

kk. Esta es la única familia de fórmulas donde el uso de los grafos-tablero resulta la mejor estrategia. Las repeticiones de subfórmulas quizás expliquen que la representación de los grafos-tablero sea más compacta que la de los BDDs –en el caso de `kk_8` puede observarse esto último–. De todos modos, es probable que la construcción de BDDs minimales fuese aún mejor.

equiv. Dada la gran expansión en ramas que producen estas fórmulas, resulta que `proposic`, basado en tablas de verdad resulta más eficiente que cualquier método basado en `leanAP`.

tipo1. Al producir tableros cerrados en un número muy pequeño de pasos, los demostradores `prop_prove` y `prop_prove_abd` tienen mucho más éxito que

los demás métodos, al ser el preprocesamiento, en este caso, una pérdida de tiempo. Vemos que de los dos `prop_prove` es un poco más eficiente, al ser mejor para fórmulas no satisfacibles, como se vio más arriba.

tipo2. En este caso, todos los sistemas proposicionales basados en `leanTAP` presentan un rendimiento parecido, aunque `prop_bdd` se sitúa algo por encima.

tipo3. También en este caso el rendimiento de todos los métodos proposicionales basados en tableros es similar, aunque ahora es `prop_prove` quien obtiene resultados algo mejores.

tipo4. Para las instancias pares, que son satisfacibles, `prop_prove_abd` es el método más eficiente. Para las impares, no satisfacibles, es `prop_prove`.

tipo5, tipo6. Como ya resultará esperable, tratándose de fórmulas que generan gran cantidad de ramas, también ahora es `prop_bdd` es método que obtiene mejores resultados, al lograr una representación más compacta.

Llama la atención que la estrategia de compilar la búsqueda no aparezca como la mejor para ninguno de los tipos de fórmulas. Pensamos que ello se debe a que el tiempo de compilación –el uso de `assert/1` y `retract/1` consume un tiempo considerable– no merece la pena frente a metaintérpretes tan eficientes como son los otros programas. Seguramente, la estrategia de compilar sólo convenga si se compila a un lenguaje de más bajo nivel, o si los metaintérpretes no están demasiado optimizados.

Otra observación es que excepto para el **tipo6**, todas las fórmulas satisfacibles se prueban mejor con `prop_prove_abd`, la variante proposicional de `leanTAP` que crea modelos. Esto es una buena noticia de cara a un posterior uso abductivo de estos demostradores. El hecho de que para el **tipo6** no obtenga tan buenos resultados, nos sugiere que una nueva versión de `prop_bdd` que recoja las ramas abiertas, seguramente sería aún mejor que `prop_bdd` para detectar fórmulas satisfacibles cuyos tableros tengan un gran número de ramas, pudiendo además devolver las ramas abiertas, y por tanto los modelos.

Respecto a Otter, en la mitad de los tipos de fórmulas queda peor que los demostradores proposicionales. La razón es que al estar pensado para trabajar con fórmulas de primer orden, no emplea estrategias propias de la lógica proposicional.

4.4. Comparación con otros sistemas

4.4.1. Descripción del test

En este último test tomamos instancias grandes de las fórmulas del test anterior, y comprobamos la demostración que hace el sistema basado en `leanTAP` que mejor trata tal fórmula –según los resultados del test anterior– con otros sistemas especializados en la búsqueda de modelos, como son:

Mace4. Se trata de un buscador de modelos finitos para fórmulas de primer orden. Lo desarrolla el Argonne National Laboratory de la Universidad de Chicago. En nuestro test, limitamos el tiempo máximo a 600 segundos. La memoria no tuvo más límite que los 128 megabytes de la máquina. La entrada para cada prueba era directamente la fórmula en primer orden.

zChaff. Es un buscador de modelos especializado en lógica proposicional, escrito en C++ por Lintao Zhang en 2001 en la Universidad de Princeton. La versión que hemos usado¹, la 2004.5.13, ha sido desarrollada por Yogesh Mahajan y Zhaohui Fu, quien mantiene las versión oficial de zChaff. Este demostrador es conocido por su rapidez, al haber ganado dos premios (en las categorías de problemas industriales y problemas hechos a mano) al mejor demostrador en la competición SAT 2002. La entrada es una fórmula en formato DIMACS. En los resultados no se cuenta el tiempo empleado para la transformación de las fórmulas proposicionales a este formato, al no disponer de una implementación muy eficiente, pues usamos el programa que aparece en la página 168. No empleamos ningún límite de tiempo para las pruebas ni tampoco de memoria, más allá de los 128 megabytes de la máquina.

Anldp. Otro programa de búsqueda de modelos finitos desarrollado en el Argonne National Laboratory. Implementa la estrategia de búsqueda de Davis-Putnam. Como zChaff, también trabaja con entradas en formato DIMACS. Hemos empleado la versión 2.2. Como con zChaff, los ficheros de entrada se generaron con el programa que aparece en la página 168. Tampoco se contó en el test el tiempo necesario para la transformación a DIMACS. Igualmente, el único límite que tuvieron las pruebas fueron los 128 megabytes de memoria de la máquina.

Los resultados pueden consultarse a partir de la página 219. Los tiempos para Mace4, zChaff y Anldp aparecen en segundos. Para los demostradores tipo lean^{AP} se usó SWI-Prolog 5.2.13, con un tiempo máximo de 600 segundos para cada prueba, y 20 megabytes para cada pila *-local, global y tail-*. La máquina fue un PC con CPU Intel Pentium III 866 MHz, con sistema operativo Mandrake Linux 9.2.

4.4.2. Conclusiones

Antes que nada, digamos algo de los errores que aparecen en los resultados de este test. Respecto al que aparece etiquetado como **error**, en los métodos lean^{AP} , es debido a un problema que SWI-Prolog produce en la transformación

¹Esta versión apareció durante la redacción de este trabajo. En un comienzo empleamos la versión anterior, pero producía ciertos errores que hacían que considerase satisficibles fórmulas que no lo son.

a NNF, tanto en la versión original de `leanAP` como en la que hemos adaptado para lógica proposicional, cuando trabaja con fórmulas que producen un gran número de ramas, y por tanto los valores de `Paths`² son muy grandes. De hecho, el error es producido al manejar números mayores a cierto límite:

```
?- A is 7.98847e+307*2.
A = 1.59769e+308
Yes
?- A is 8.98847e+307*2.
ERROR: Arithmetic: evaluation error: 'float_overflow'
```

Ya hemos encontrado la solución a este error. Compilando SWI-Prolog con la librería `mp` de precisión arbitraria es posible manejar números mayores sin que se produzca este desbordamiento, siempre que usemos `mp_is/2` en lugar de `is/2`:

```
?- use_module(library(mp)).
% library(mp) compiled into mp 0.02 sec, 62,148 bytes
Yes
?- A mp_is 8.98847e+307*2.
A = '1.79769400000000004532E308'
Yes
```

Con `Anldp` también se producen algunos errores, debido a un bug de Mace, que produce la siguiente alarma:

```
+-----+
| SEGMENTATION FAULT!! This is probably caused by a bug |
| in MACE. Please send copy of the input file to      |
| otter@mcs.anl.gov, let us know what version of MACE you |
| are using, and send any other info that might be useful. |
+-----+
```

Para fórmulas demasiado grandes no ha sido posible la transformación a formato DIMACS, por rebasarse el tamaño de las pilas de Prolog, lo que está indicado en los casos correspondientes.

En cuanto a las observaciones que se derivan de los resultados, respecto a Mace4, en la mitad de los tipos de fórmulas es rendimiento es peor (a veces mucho peor) que con los mejores métodos proposicionales. Ocurre incluso que en un tipo de fórmula satisfacible, como son las instancias pares de `dB`, el rendimiento de `prop_prove_abd` –buscador de modelos optimizado para lógica proposicional– es mucho mejor que el de Mace4, que incluso llega a agotar la memoria de la máquina –128 megabytes– con `dB_400` cuando `prop_prove_abd` sólo necesitó 84 bytes para constatar que la fórmula es satisfacible.

Cada uno de los cinco métodos proposicionales que en el test anterior obtuvo mejores resultados tratando alguna de las familias de problemas, sigue siendo

²Ver página 75

mejor que Mace4 para alguna de tales familias. Con Mace ocurre lo mismo que con Otter, que al ser un buscador de modelos para fórmulas de primer orden el rendimiento en lógica proposicional es menor que el de los demostradores –o buscadores de modelos– optimizados para esta lógica. Eso es lo que vemos en los resultados de zChaff y Anldp, diseñados para trabajar con fórmulas proposicionales en formato DIMACS, una representación de la fórmula en forma normal conjuntiva. Estos dos últimos demostradores implementan estrategias proposicionales de búsqueda como la de Davis-Putnam, consiguiendo una eficiencia mucho mayor que la de nuestras versiones de *lean^{TP}*. De los dos, zChaff es con mucho el más rápido.

4.5. Conclusiones globales

A partir de los test realizados, podemos extraer algunas conclusiones generales, válidas para toda vez que volvamos a enfrentarnos al diseño de un demostrador de teoremas. Pensamos que resulta conveniente:

- Adaptar el demostrador a la lógica con la que se va a trabajar. Resulta atractiva la idea de diseñar demostradores que prueben teoremas de muy diversas lógicas, pero como se ha visto, los resultados no son demasiado eficientes. Aunque la expresividad de los lenguajes de primer orden englobe la de los lenguajes proposicionales, y por tanto todo demostrador desarrollado para lógica de primer orden es en teoría capaz de demostrar un teorema proposicional, en la práctica puede que los recursos que necesite para ello sean exponencialmente mayores que una versión más simple del demostrador optimizada para lógica proposicional. Por tanto, cuando sea posible, optimizaremos los demostradores para cada lenguaje en que se trabaje.
- Adaptar el demostrador a los problemas. Como hemos visto, un demostrador puede ser muy bueno para cierta clase de problemas proposicionales pero muy malo para otra. Lo ideal sería tratar cada problema con el mejor demostrador para tal problema, pero por lo general no resulta posible determinar a priori cuál es el mejor demostrador para cada teorema, sino sólo en casos muy especiales. Como en un futuro nos planteamos aplicar estos demostradores a la resolución de problemas abductivos, nos cabe hacer una pregunta: *¿son los problemas abductivos de algún tipo especial tal que permitan tratarse mejor con algún demostrador que con otros?*
- Importancia de la representación. También hemos observado que no es trivial la representación que se elija para los tableros: árboles, grafos y BDDs son diferentes representaciones con distinto rendimiento según los problemas. Aunque ciertas representaciones puedan requerir un mayor tiempo de preprocesamiento, como hemos visto puede resultar conveniente para ciertos problemas.

- Trabajar al más bajo nivel posible. El gran éxito de Anldp y zChaff es debido a esto. El empleo de estructuras más elementales para representar las fórmulas –DIMACS en vez de NNF– e incluso de lenguajes a más bajo nivel –C en vez de Prolog– mejora con mucho la eficiencia de los demostradores. En cierto sentido, también un grafo o un BDD representan el tablero con estructuras más simples que un árbol, razón del éxito de los métodos que usan estas estrategias.

Capítulo 5

Idea de aplicación a la abducción

5.1. Planteamiento del problema

La noción de *abducción* como modo de razonamiento fue introducida por el filósofo Peirce, al distinguir entre tres formas de razonar:

Deducción, que es un proceso analítico basado en la aplicación de reglas generales a casos particulares, infiriendo cierto resultado.

Inducción, es un razonamiento sintético que infiere una regla general a partir de casos particulares.

Abducción, que también es un razonamiento sintético, pero que a partir de la regla y el resultado busca una explicación.

Peirce caracteriza la abducción como la adopción de una hipótesis que sea explicación de ciertos hechos observados, de acuerdo con reglas conocidas. La define como un tipo de inferencia débil, porque no podemos asegurar la verdad de la explicación, sino sólo la posibilidad de que sea verdad. Dice Peirce,

Se observa un hecho sorprendente, C ; pero si A fuera verdadera, entonces C sería algo común; por lo tanto, hay razón para sospechar que A es verdadera.

Esta cita de Peirce caracteriza la inferencia abductiva como una inferencia basada en *sospechas*, en el sentido conjetural del término, pues se trata de encontrar la explicación A –siguiendo la cita–, que haría a C pasar de ser sorprendente a algo normal –explicado–.

La abducción es ampliamente utilizada tanto en el razonamiento de sentido común como en Inteligencia Artificial, en procesos como los de diagnóstico. También en Filosofía de la Ciencia es frecuentemente tomado como modelo de la explicación científica, que Popper [Pop74] define como “la explicación de lo conocido mediante lo desconocido”.

Si bien el razonamiento abductivo resulta de gran interés en diversas áreas, hay pocos tratamientos en lógicas de primer orden, debido a la indecidibilidad de estos sistemas. Los sistemas formales en que hemos encontrado acercamientos al razonamiento abductivo son mayoritariamente o bien sistemas proposicionales, como se hace en el estudio de Aliseda [Ali97], o lenguajes clausales de sintaxis restringida, como hacen Flach [Fla94], Poole [PMG98], o los acercamientos recogidos en el trabajo fundacional de Kakas, Kowalski y Toni [KKT98].

El problema de estos tratamientos es que al trabajar con lenguajes de expresividad muy reducida no puede representarse más que una pequeña clase de problemas abductivos, siempre muy lejos de los casos que realmente se dan en el razonamiento de sentido común o en la explicación científica. Si queremos realizar un tratamiento de la abducción que dé cuentas de estos fenómenos debemos abordar el razonamiento explicativo en primer orden, e incluso plantear la posibilidad de tratar lógicas de orden superior, ya que como se explica en [Her01], ciertas propuestas de modelización de un lenguaje empirista, como la de los *lenguajes-cosa* de Carnap, sólo son formalizables mediante sistemas lógicos de segundo orden.

Veamos cuál es el obstáculo que plantea la indecidibilidad de la lógica de primer orden al tratar problemas abductivos.

Sea Γ nuestra teoría, y α en hecho que queremos explicar, formalizados en un lenguaje de primer orden. Para que exista un problema abductivo, debe verificarse, en primer lugar, que el hecho encontrado no sea inconsistente con la teoría, es decir,

$$\Gamma, \alpha \not\perp \quad (5.1)$$

Y lo más importante, que el hecho no esté explicado ya por la teoría,

$$\Gamma \not\models \alpha \quad (5.2)$$

Además, para que una proposición γ se pueda considerar una explicación de α en la teoría Γ , debe cumplirse que

$$\Gamma, \gamma \models \alpha \quad (5.3)$$

Si además la explicación no debe ser autosuficiente, es decir, queremos que explique en hecho *dentro de la teoría*, y no por sí sola –lo que en [Ali97] se llama *abducción explicativa*–, debe verificarse que

$$\gamma \not\models \alpha \quad (5.4)$$

Y para asegurar la consistencia de la explicación debe verificarse

$$\Gamma, \gamma \not\perp \quad (5.5)$$

Pero como la relación “ \models ” no es decidible en primer orden, muchas de las condiciones anteriores no son verificables, lo que puede plantear ciertos problemas,

pudiéndonos llevar, entre otras cosas, a aceptar explicaciones inconsistentes con nuestra teoría.

Si queremos tratar el razonamiento abductivo en primer orden, debemos plantear un modo de entender la abducción diferente al que se ha presentado más arriba, de modo que la revisión de teorías no descansa en la decidibilidad de los sistemas formales empleados. Acabaremos este capítulo con un planteamiento que pretende dar una respuesta a este problema, pero antes veamos qué podemos aprender del desarrollo de la Filosofía de la Ciencia a lo largo del siglo XX.

5.2. La concepción falibilista del progreso científico

La abducción, en Filosofía de la Ciencia, se entiende como la búsqueda de hipótesis dentro del modelo deductivo de explicación científica [Nag68]. Sin embargo, a lo largo del siglo veinte se han propuesto diversas maneras de entender cuándo la ciencia explica algo, según las exigencias que se hayan impuesto a las teorías científicas.

Un primer acercamiento es el del positivismo lógico del Círculo de Viena. Hempel [Hem65] recoge el *requisito de verificabilidad completa en principio*, que es la primera versión del criterio empirista de significado:

Una oración tiene significado empírico si, y sólo si, no es analítica y se deduce lógicamente de una clase finita y lógicamente consistente de oraciones observacionales.

Con este requisito se exige la verificación (prueba) por medio de la observación. Pero se excluyen muchos de los enunciados que las ciencias necesitan, y de hecho emplean, como los enunciados universales que toda teoría contiene, al no ser posible su prueba a partir de enunciados particulares. Además, como más tarde se hizo evidente, los enunciados no se pueden derivar de las propias observaciones, sino de otros enunciados. Es posible formalizar la observación, pero ello carga de contenido teórico al propio dato observacional. Por tanto, el ideal de verificación a través de la experiencia queda muy lejos de ser posible en el conocimiento científico.

Con el requisito de verificabilidad completa sí necesitaríamos que los sistemas formales en que se formalizaran nuestras teorías fuesen decidibles, pero este criterio entró pronto en crisis. Aparecen nuevas versiones del criterio empirista de significado, dada la pérdida de credibilidad en la verificabilidad de los enunciados científicos, reforzada por los resultados que por otra parte se iban obteniendo sobre la indecidibilidad de los sistemas formales de primer orden o la incompletud de los de segundo orden. Toma fuerza la idea de que en las ciencias empíricas, a diferencia de lo que ocurre en las ciencias formales, las leyes son siempre conjeturales, abriéndose así una vía para que la indecidibilidad de los sistemas formales

deje de ser un obstáculo cuando tratemos de modelar lógicamente el progreso científico.

Uno de los autores que proponen una alternativa más elaborada al criterio empirista de significado es Popper. En la *Lógica de la investigación científica* [Pop62] realiza un análisis lógico del modo de proceder de las ciencias empíricas:

No exigiré que un sistema científico pueda ser seleccionado, de una vez para siempre, en un sentido positivo; pero sí que sea susceptible de selección en un sentido negativo por medio de contrastes o pruebas empíricas: *ha de ser posible refutar por la experiencia un sistema científico empírico.*

Así el enunciado “lloverá o no lloverá aquí mañana” no se considerará empírico, por el simple hecho de que no puede ser refutado; mientras que a este otro, “lloverá aquí mañana”, debe considerársele empírico.

Popper es el filósofo representante del falsacionismo o refutacionismo, corriente que no exige a las teorías científicas que sean verificadas por la experiencia –algo imposible, como se ha visto–; ahora lo que debe pedirse a una teoría es que sea refutable, es decir, que sea posible encontrar un hecho tal que resulte falseada la teoría, por resultar inconsistente con ella. Se trata, con una frase común entre los refutacionistas, de que *la teoría ofrezca su cuello.*

Mientras la teoría no se falsea, está corroborada por la experiencia, que es algo más débil que la verificación de los positivistas, al tratarse simplemente de que no se ha encontrado ningún hecho que la refute. Cuando la teoría es refutada, se reemplaza por otra mejor. Así entiende Popper en [Pop67] el progreso científico:

Cuando hablo del desarrollo del conocimiento científico, lo que tengo *in mente* no es la acumulación de observaciones, sino el repetido derrocamiento de teorías científicas y su reemplazo por otras mejores o más satisfactorias.

Podemos preguntarnos cuándo se falsea una teoría, si cuando *existe* una contradicción con algún dato observacional, o cuando *la encontramos*. Esto será de suma importancia cuando diseñemos un sistema de abducción sobre estas bases. Debemos interpretar que la teoría se falsea cuando *encontramos* la contradicción, quedando así libre del requisito de decidibilidad. Si se exigiera detectar toda contradicción que *exista*, estaríamos más cerca del verificacionismo que del falsacionismo, ya que se supondría que disponemos de los medios de probar todo lo que se derive de los datos con que contamos. Desde la perspectiva falsacionista, las teorías no se mantienen por *ser* verdaderas, como se ha visto, sino por *no haberse encontrado* ningún contraejemplo. Las teorías científicas tienen, por tanto, un carácter tentativo, son siempre hipotéticas. La verdad queda como un ideal regulativo de la ciencia.

En cuanto a la forma que deben tener las explicaciones, en [Pop74] leemos

La pregunta “¿Qué tipo de explicación puede considerarse satisfactorio?”, nos lleva a reponder: una explicación en términos de leyes universales contrastables y falsables junto con condiciones iniciales. Además, una explicación tal será tanto más satisfactoria, cuanto más altamente contrastables sean estas leyes y cuanto mejor contrastadas hayan sido. (Esto también sirve para las condiciones iniciales).

De este modo caben dos posibilidades: comprender la abducción como la propuesta de las leyes universales, o de las condiciones iniciales. En principio, no debemos descartar ninguna de las dos opciones.

Algo importante es que Popper considera que no todo puede ponerse en tela de juicio al mismo tiempo cuando aparece una contradicción, ya que así la ciencia sería el constante crecimiento del caos. Esta idea está muy bien tratada por Lakatos [Lak83]:

El falsacionista metodológico comprende que en las “técnicas experimentales” del científico hay implicadas teorías falibles con las que interpreta los hechos. A pesar de ello, “aplica” tales teorías; en el contexto dado, las considera no como teorías bajo contrastación, sino como *conocimiento fundamental carente de problemas* que aceptamos (tentativamente) como no problemático mientras estamos contrastando la teoría.

Así pues, las técnicas experimentales, por las que se obtienen los datos “observacionales” son teorías que decidimos no someter a contrastación. Hay, por tanto, convenciones institucionalizadas que toda la comunidad científica acepta. Pero no sólo las técnicas experimentales se consideran conocimiento no problemático, sino también gran parte de las teorías científicas:

Los experimentos pueden tener poder suficiente como para refutar a las teorías jóvenes, pero no para refutar a las teorías antiguas y asentadas: *conforme crece la ciencia disminuye el poder de la evidencia empírica.*

¿Qué teorías se consideran no problemáticas y cuáles están sometidas a la refutación? Lakatos considera que las teorías se agrupan en *programas de investigación*:

El programa consiste en reglas metodológicas: algunas nos dicen las rutas de investigación que deben ser evitadas (*heurística negativa*), y otras, los caminos que deben seguirse (*heurística positiva*).

La *heurística negativa* constituye el “centro firme” del programa de investigación, y contiene las teorías que no deben someterse a falsación. Lakatos dice que

“debemos utilizar nuestra inteligencia para incorporar e incluso inventar hipótesis auxiliares que formen un *cinturón protector* en torno a ese centro, y *contra ellas* debemos dirigir el *modus tollens*. El cinturón protector de hipótesis auxiliares debe recibir los impactos de las contrastaciones y para defender al centro firme, será ajustado y reajustado e incluso completamente sustituido”. Ese cinturón protector es la *heurística positiva* del programa de investigación, consistente en los ajustes que deben realizarse para mantener irrefutable el centro firme del programa.

Un ejemplo que ofrece Lakatos es el del programa de Newton. Allí, la heurística negativa está constituida por sus tres leyes de la dinámica y la ley de gravitación. Contra ellas no puede “dirigirse el *modus tollens*”. Las anomalías deben originar cambios sólo en el cinturón “protector” de hipótesis auxiliares que los defensores del programa de Newton van creando en torno a su “centro firme”.

Desde esta perspectiva, la abducción debe entenderse dentro de la heurística positiva de un programa de investigación, como ajuste de su cinturón protector de hipótesis auxiliares.

5.3. Sistema abductivo en primer orden

Veamos ahora ciertas notas sobre la propuesta de un sistema abductivo refutacionista. La idea con la que trabajaremos es la de aceptar las hipótesis propuestas mientras no hayan sido refutadas, y cuando encontremos un hecho que refute nuestras hipótesis, revisar las hipótesis asumidas. Por tanto, nuestro sistema abductivo será también un sistema de revisión de creencias, al ser indisociables en la práctica la tarea de proponer hipótesis y la de revisarlas.

En cuanto a los aspectos lógicos, debemos partir de que la lógica de primer orden es indecidible, lo cual significa que las exigencias 5.1–5.5 no pueden garantizarse. Ya que la relación de consecuencia lógica “ \models ” no es decidible, tenemos que conformarnos con algo más débil. Informalmente, podemos definir una *relación de consecuencia lógica condicionada* \models^c de la siguiente manera:

Definición 5.1 *Decimos que un fórmula α es consecuencia lógica condicionada de un conjunto de fórmulas Γ , y lo representamos $\Gamma \models^c \alpha$, si podemos comprobar que $\Gamma \models \alpha$ empleando a lo sumo los recursos que denote el parámetro condicionante c .*

El valor de c puede ser un tiempo máximo de prueba, un espacio máximo de memoria, o cualquier otro límite de recursos que permita el cálculo que empleemos.

La definición de la *consecuencia lógica condicionada* modela en cierto sentido el proceder que Popper atribuye a la investigación científica, ya que las conclusiones que se extraen no son las que lógicamente se derivan, lo cual sería la postura

del verificacionismo. Las consecuencias relevantes de una teoría son las que podemos hallar, las que encontremos con los medios finitos de que dispongamos. Es un criterio más débil, pero más realista que la consecuencia lógica clásica. Además, con que dispongamos de un cálculo correcto, aseguramos que toda consecuencia lógica condicionada será una consecuencia lógica en sentido clásico. Si es la corrección el único requisito formal que deberá exigirse a los cálculos para poder ser usados abductivamente, se abre la puerta a la abducción en lógicas indecidibles e incluso incompletas.

Lo importante ahora será el diseño de métodos de prueba eficientes, que con los recursos disponibles sean capaces de extraer más consecuencias de un mismo conjunto de fórmulas.

Puede parecer insatisfactoria la elección de un criterio tan pragmático como la posibilidad de deducir con los recursos disponibles, pero es que en la práctica la ciencia hace eso. Se trata de un intento de modelar lógicamente lo que de hecho se da. En lean^{AP} , c podría ser el número máximo de variables libres que pueden ser introducidas por rama del tablero, o bien el tiempo que damos para la prueba, o el tamaño de las pilas de Prolog.

5.3.1. Aspectos lógicos

Veamos qué requisitos debemos exigir ahora a las teorías, hechos y explicaciones que constituyen los problemas abductivos. De nuevo, sea Γ un conjunto de fórmulas, y α un hecho que quiere explicarse. El primer requisito, reescribiendo 5.1 con la nueva relación de consecuencia lógica condicionada, queda:

$$\Gamma, \alpha \not\vdash^c \perp \quad (5.6)$$

lo cual significa que *no encontramos* ninguna refutación a la teoría existente con el nuevo hecho. El siguiente requisito, queda:

$$\Gamma \not\vdash^c \alpha \quad (5.7)$$

que equivale a que *no encontramos* una explicación al hecho dentro de la teoría. Véase que, a efectos prácticos, no hay diferencia entre *no encontrar* una explicación y *que no la haya*.

Ahora, para que una proposición γ se pueda considerar una explicación de α en la teoría Γ , debe cumplirse que

$$\Gamma, \gamma \vdash^c \alpha \quad (5.8)$$

es decir, no basta cualquier explicación simplemente porque con ella y la teoría *se derive* el hecho que quiere explicarse. Debe ser una explicación tal que seamos capaces de derivar el hecho a partir de la misma.

El requisito de no autosuficiencia resulta:

$$\gamma \not\equiv^c \alpha \quad (5.9)$$

lo cual significa que no debemos poder inferir el hecho de la sola hipótesis.

Por último la consistencia de la explicación se transforma en el requisito:

$$\Gamma, \gamma \not\equiv^c \perp \quad (5.10)$$

que equivale a que no encontramos contradicción entre la teoría y la nueva hipótesis.

Puede objetarse que con los requisitos presentados más arriba pueden colarse en la teoría hipótesis que a la larga se descubran inconsistentes. Esto no es ninguna novedad en la ciencia, y como ya hemos visto, lo que hace válida una hipótesis no es que sea *verdad*, sino que sea *refutable*. Debemos ser, como Popper recomienda, *austeros al refutar*, y disponer de buenos sistemas de revisión de las hipótesis propuestas, de modo que, como anteriormente anunciamos, el sistema de abducción acaba dando paso a un sistema de revisión de creencias.

Otra razón que justifica la aceptación de hipótesis cuya consistencia no queda siempre probada la encontramos en teoría de agentes. Desde ciertas orientaciones se considera que un agente inteligente debe realizar de entre las opciones que tenga, la que demuestre ser mejor. Pero si no puede demostrar que ninguna acción posible sea buena, debe hacer alguna de la que al menos no pueda demostrar que sea mala, ya que generalmente se prefiere la acción –se les llama agentes porque actúan– a la inacción. También en ciencia preferimos el *progreso* al *estancamiento* de las teorías, y resulta preferible aceptar hipótesis que luego puedan ser refutadas, que dejar hechos sin explicar. Ya vimos anteriormente que el progreso científico es el continuo derrocamiento de teorías y su reemplazo por otras mejores. No podemos pretender que nuestros sistemas lógicos sean más puros que la realidad que intentan modelar¹.

5.3.2. Sistema de revisión de teorías

Veamos un esquema de cómo podría construirse un sistema de revisión de teorías siguiendo las ideas presentadas en este capítulo. Partimos de la distinción de Lakatos entre heurística positiva y heurística negativa. Llamemos N al “núcleo duro” de la teoría, que será un conjunto de fórmulas que modelen el contenido de la heurística negativa del programa de investigación. Sea H el conjunto de hipótesis auxiliares. α será el hecho nuevo que aparezca. Veamos cómo procederíamos:

1. Comprobamos si $N \cup H \models^c \alpha$. En caso de que así sea, el hecho ya está explicado. Si no:

¹Ya que partimos de una consideración *descriptiva* de la ciencia, de estudiar cómo es de hecho el progreso científico. Sólo desde consideraciones *prescriptivas* pueden exigirse criterios de decidibilidad. Pero entonces tendríamos que llegar a la conclusión de que la ciencia empírica es imposible.

2. Se comprueba si $N \cup H, \alpha \models^c \perp$. Si así es, debemos revisar la teoría en el estado existente (con N , H , y todo el conjunto $\{\alpha_1, \dots, \alpha_n\}$ de hechos que hasta ahora hayan aparecido). Si no se verifica entonces debemos encontrar una explicación γ tal que:
 - $N \cup H, \gamma \models^c \alpha$
 - $\gamma \not\models^c \alpha$
 - $N \cup H, \gamma \not\models^c \perp$

A continuación incluimos γ en H , alcanzando así un nuevo estado de la teoría.

En cuanto a la revisión de teorías cuando aparece una inconsistencia –anomalía– con las observaciones podemos hacer varias cosas, que se corresponden con distintas opciones teóricas:

- Se eliminan todas las hipótesis existentes –haciendo H vacía– y se buscan nuevas hipótesis que expliquen todos los hechos que hubiesen aparecido hasta el momento de la revisión.
- Se eliminan gradualmente las hipótesis según algún criterio de prioridad, hasta que no se encuentre ninguna inconsistencia. Entonces se añaden nuevas hipótesis hasta que se expliquen todos los hechos aparecidos hasta el momento de la revisión.

Una cuestión que queda por determinar es qué ocurre cuando se descubre una inconsistencia del núcleo duro de la teoría con los datos observacionales –los hechos–. Lo que entonces puede hacerse es considerar que el programa de investigación está agotado, que el núcleo duro ya no sirve, y por tanto, debemos proponer otra teoría.

5.4. Trabajo futuro

5.4.1. Sistema de abducción en primer orden

Una primera tarea es aplicar el material de esta memoria para crear un sistema de revisión de teorías en primer orden. Disponemos de demostradores que recogen los literales que aparecen en las ramas de los tableros abiertos, lo que permite abducir tal como M. Cialdea y F. Pirri explican en [CP93]². Los criterios que se usen en la relación de consecuencia condicionada pueden ser el tiempo máximo

²Precisamente estas autoras consideran, como una de las posibles vías de continuación del trabajo citado, el uso de cálculos con versiones de la regla- δ más eficientes que la que ellas adoptan. Pensamos que *leanTAP*, que usa la regla- δ^{++} [BHS93], es un buen candidato como implementación eficiente.

de prueba, el número máximo de variables libres que pueden introducirse por rama del tablero, o la memoria que puede emplearse durante la prueba, medidas habituales en las pruebas que anteriormente hemos comentado.

Puede implementarse un criterio de minimalidad de la explicación, por el que se prefieran las hipótesis con un menor número de literales. Puede también usarse anti-unificación para buscar hipótesis más generales –y minimales– a partir de los cierres de las ramas. Así, si $p(a)$ cierra una rama y $p(b)$ otra, la anti-unificación de ambos literales, $p(x)$ –entendiendo las variables libres esta vez como universales–, es una hipótesis más general que $p(a) \ \& \ p(b)$ y más pequeña.

5.4.2. Elección de la mejor representación

Entre las versiones de $lean^{AP}$ presentadas, algunas emplean sistemas de representación de los tableros mucho más eficientes que los árboles. Deberían explorarse tales representaciones en cuanto a su posible uso abductivo. Si bien el esfuerzo de precomputación es mayor, las teorías pueden transformarse –o compilarse– como grafos o BDDs de una vez por todas, haciendo mucho menor el esfuerzo durante la búsqueda de explicaciones que si cada vez que hubiese que explicar una nueva observación fuera necesario desarrollar todo el tablero de la teoría con la negación de la observación, como se hace en [Ali97] y [CP93].

Además, la cuestión sobre la mejor representación de las teorías resulta de gran interés en Filosofía de la Ciencia.

5.4.3. Ampliaciones

La primera ampliación que nos planteamos realizar es la introducción de la identidad en los tableros, lo cual permite el tratamiento de teorías mucho más expresivas. En [Bec97], Beckert estudia las dos formas en que esto puede llevarse a cabo. En primer lugar, podemos añadir una nueva regla tal que si en una rama se encuentran $t \approx s$ (o $s \approx t$) y $\phi(t)$, entonces podemos añadir $\phi(s)$. Además, si en una misma rama se encuentran $t \approx s$ y $\neg(t \approx s)$, puede cerrarse dicha rama. El problema de este tratamiento es que el espacio de búsqueda se hace excesivamente grande. Por ello, Beckert estudia también la posibilidad de emplear E -unificación, que resulta mucho más eficiente. Brevemente, un *problema de E -unificación* $\langle E, s, t \rangle$ consiste en un conjunto finito E de identidades de la forma $(\forall x_1) \dots (\forall x_n)(l \approx r)$ y términos s y t . Una sustitución σ es una *solución* del problema si

$$E\sigma \models_{\approx}^{\circ} (s\sigma \approx t\sigma).$$

siendo $\models_{\approx}^{\circ}$ una versión de la noción de *consecuencia lógica fuerte*, definida en la página 16, para tableros con identidad. La construcción de los tableros pasa por localizar los problemas de E -unificación de cada rama. Si se encuentra una sustitución σ que sea solución de un problema de tal rama, dicha rama será cerrada. Como puede imaginarse, los problemas de E -unificación de una rama

son aquellos en que σ produce la unificación de literales complementarios en dicha rama.

Por otra parte, hemos visto [Her01] que para formalizar las teorías científicas “reales” necesitamos una expresividad de segundo orden. Si la idea de sistema abductivo presentada se mostrara eficaz en primer orden, podría aplicarse a segundo orden también, al necesitarse cálculos que sean simplemente correctos, y no necesariamente completos.

5.4.4. Abducción en otras lógicas

También puede ser interesante estudiar cómo se comporta este tipo de sistema abductivo en lógicas no clásicas, ya sean extensiones (lógicas modales, temporales, epistémicas, etc.) o alternativas (lógicas paraconsistentes, lógicas por defecto, etc.).

Otra cosa conveniente sería adaptar los sistemas creados a nivel proposicional, y comparar la eficacia con la de otros sistemas de revisión de teorías proposicionales. Ya que disponemos de adaptaciones proposicionales de los demostradores que en problemas proposicionales se muestran mucho más eficientes que las versiones de lean^{TAP} para primer orden, resulta esperable el que ocurra algo similar a nivel abductivo.

5.4.5. Aplicaciones de los sistemas propuestos

Si queremos saber si nuestros sistemas formalizan el proceder del conocimiento científico, deberíamos aplicarlos a contextos prácticos, donde pudiésemos ver de qué forma se comportan.

Podrían realizarse experimentos probando diferentes valores para el parámetro c condicionante de “ \models^c ”, viendo que límite resulta más conveniente para cada tipo de problemas.

Bibliografía

- [AB02] J. A. ALONSO Y J. BORREGO. “Deducción automática (Vol. 1: Construcción lógica de sistemas lógicos)”. *Kronos* (2002).
- [Ali97] A. ALISEDA. “Seeking Explanations: Abduction in Logic, Philosophy of Science and Artificial Intelligence”. Dissertation Series. Institute for Logic, Language, and Computation, Holland (1997).
- [AM] JOSÉ ALFREDO AMOR Y RAYMUNDO MORADO. Demostración Automática. Disponible en <http://minerva.filosoficas.unam.mx/~morado/LogicaHoy/morado.html>.
- [Bec97] BERNHARD BECKERT. Semantic tableaux with equality. *Journal of Logic and Computation* **7**(1), 39–58 (1997).
- [Bet69] E. W. BETH. “Entrañamiento semántico y derivabilidad formal”. Número 18. Cuadernos Teorema (1969).
- [BHS93] BERNHARD BECKERT, REINER HAEHNLE Y PETER SCHMITT. “Computational logic and proof theory”, capítulo The even more liberalized delta-rule in free variable semantic tableaux, páginas 108–119. G. Gottlob (1993).
- [Boo84] G. BOOLOS. Trees and finite satisfiability. *Notre Dame Journal of Formal Logic* (25), 110–115 (1984).
- [BP94a] BERNHARD BECKERT Y JOACHIM POSEGGA. Lean theorem proving: Maximal efficiency from minimal means (position paper). En “Working Notes, AISB Workshop “Automated Reasoning: Closing the Gap between Theory and Practice”, Leeds, England”, páginas 7–8 (abril 1994).
- [BP94b] BERNHARD BECKERT Y JOACHIM POSEGGA. *leanTAP*: Lean tableau-based theorem proving. extended abstract. En A. BUNDY, editor, “Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France”, LNCS 814, páginas 793–797. Springer (1994).

- [BP94c] BERNHARD BECKERT Y JOACHIM POSEGGA. Logic programming as a basis for lean deduction: Achieving maximal efficiency from minimal means. En N. E. FUCHS Y G. GOTTLOB, editores, “Proceedings, 10th Logic Programming Workshop, Zürich, Switzerland”. Institut für Informatik der Universität Zürich (octubre 1994).
- [BP95] BERNHARD BECKERT Y JOACHIM POSEGGA. $leanTAP$: Lean tableau-based deduction. *Journal of Automated Reasoning* **15**(3), 339–358 (1995).
- [CP93] M. CIALDEA Y F. PIRRI. First order abduction via tableau and sequent calculi. *Bulletin of the IGPL* **1**, 99–117 (1993).
- [D93] E. DÍAZ. Árboles semánticos y modelos mínimos. En E. PÉREZ, editor, “Actas del I Congreso de la Sociedad de Lógica, Metodología y Filosofía de la Ciencia en España”, página 40. Universidad Complutense de Madrid (1993).
- [Dav01] MARTIN DAVIS. “Handbook of Automated Reasoning”, capítulo The Early History of Automated Deduction. Elsevier Science Publishers (2001).
- [DEDC96] P. DERANSART, A. ED-DBALI Y L. CERVONI. “Prolog, The Standard: Reference Manual”. Springer Verlag (1996).
- [Dyc97] ROY DYCKHOFF. Some benchmark formulae for intuitionistic propositional logic. Informe técnico, University of St Andrews (1997). <http://www.dcs.st-and.ac.uk/~rd/logic/marks.html>.
- [Fit98] M. FITTING. $leanTAP$ revisited. *Journal of Logic and Computation* (8), 33–47 (1998).
- [Fla94] PETER FLACH. “Simply Logical. Intelligent Reasoning by Example”. John Wiley & Sons (1994).
- [Hem65] C. G. HEMPEL. Problemas y cambios en el criterio empirista de significado. En A. J. AYER, editor, “El positivismo Lógico”, páginas 115–136. F. C. E. (1965).
- [Her01] C. HERNÁNDEZ. El análisis lógico del lenguaje científico. los lenguajes-cosa de Carnap. En A. NEPOMUCENO, J. F. QUESADA, F. J. SALGUERO, editor, “Información: Tratamiento y Representación”, páginas 217–229. Universidad de Sevilla, Sevilla (2001).
- [KKT98] A. KAKAS, R. KOWALSKI Y F. TONI. The role of abduction in logic programming (1998).

- [Lak83] I. LAKATOS. “La metodología de los programas de investigación científica”. Alianza Universidad, Madrid (1983).
- [Lus92] EWING E. LUSK. Controlling Redundancy in Large Search Spaces: Argonne-Style Theorem Proving Through the Years. En “LPAR’92”. Springer (1992).
- [Nag68] E. NAGEL. “La estructura de la ciencia”. Paidós, Buenos Aires (1968).
- [Nep02] A. NEPOMUCENO. Scientific explanation and modified semantic tableaux. En “Logical and Computational Aspects of Model-Based Reasoning”, Applied Logic Series, páginas 181–198. Kluwer Academic Publishers (2002).
- [Pel86] F. J. PELLETIER. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning* (2), 191–216 (1986).
- [PMG98] D. POOLE, A. MACKWORTH Y R. GOEBEL. “Computational Intelligence: A Logical Approach”. Oxford University Press, New York (1998).
- [Pop62] K. R. POPPER. “La lógica de la investigación científica”. Tecnos, Madrid (1962).
- [Pop67] K. R. POPPER. “El desarrollo del conocimiento científico”. Paidós, Buenos Aires (1967).
- [Pop74] K. R. POPPER. “Conocimiento objetivo”. Tecnos, Madrid (1974).
- [Pos93] JOACHIM POSEGGA. Compiling the proof search in semantic tableaux. En “Seventh International Symposium on Methodologies for Intelligent Systems”, LNAI, Trondheim, Norway (June 1993). Springer.
- [PS99] J. POSEGGA Y P. H. SCHMITT. “Handbook of Tableau Methods”, capítulo Implementing Semantic Tableaux, páginas 581–629. Kluwer Academic Publishers (1999).
- [RN96] S. RUSSELL Y P. NORVIG. “Inteligencia Artificial: un enfoque moderno”. Prentice Hall Hispanoamericana (1996).

Apéndice A

Forma normal negada

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fichero: nnf.pl
% Copyright (C) 1993: Bernhard Beckert & Joachim Posegga
%                               Universit"at Karlsruhe
%                               Email: {beckert|posegga}@ira.uka.de
%
% Purpose:      computes Skolemized negation normal form
%               for a first-order formula
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:-      module(nnf,[nnf/2]).
:-      op(400,fy,-),      % negation
        op(500,xfy,&),    % conjunction
        op(600,xfy,v),    % disjunction
        op(650,xfy,=>),  % implication
        op(700,xfy,<=>). % equivalence

% -----
% nnf(+Fml,?NNF)
%
% Fml is a first-order formula and NNF its Skolemized negation
% normal form.
%
% Syntax of Fml:
% negation: '-', disj: 'v', conj: '&', impl: '=>', eqv: '<=>',
% quant. 'all(X,<Formula>)', where 'X' is a prolog variable.
%
% Syntax of NNF: negation: '-', disj: ';', conj: ',', quant.:
% 'all(X,<Formula>)', where 'X' is a prolog variable.
%
% Example: nnf(ex(Y, all(X, (f(Y) => f(X))))),NNF).
%          NNF = all(_A,(-(f(all(X,f(ex)=>f(X)))));f(_A))) ?

nnf(Fml,NNF) :- nnf(Fml,[],NNF,_).

% -----
% nnf(+Fml,+FreeV,-NNF,-Paths)
```

```

%
% Fml,NNF:      See above.
% FreeV:        List of free variables in Fml.
% Paths:        Number of disjunctive paths in Fml.

nnf(Fml,FreeV,NNF,Paths) :-
    (Fml = ~(-A)      -> Fml1 = A;
     Fml = ~all(X,F)  -> Fml1 = ex(X,~F);
     Fml = ~ex(X,F)   -> Fml1 = all(X,~F);
     Fml = ~(A v B)   -> Fml1 = ~A & ~B;
     Fml = ~(A & B)   -> Fml1 = ~A v ~B;
     Fml = (A => B)   -> Fml1 = ~A v B;
     Fml = ~(A => B)  -> Fml1 = A & ~B;
     Fml = (A <=> B)  -> Fml1 = (A & B) v (~A & ~B);
     Fml = ~(A <=> B) -> Fml1 = (A & ~B) v (~A & B)),!,
    nnf(Fml1,FreeV,NNF,Paths).
nnf(all(X,F),FreeV,all(X,NNF),Paths) :- !,
    nnf(F,[X|FreeV],NNF,Paths).
nnf(ex(X,Fml),FreeV,NNF,Paths) :- !,
    copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
    copy_term((X,Fml1,FreeV),(ex,Fml2,FreeV)),
    nnf(Fml2,FreeV,NNF,Paths).
nnf(A & B,FreeV,NNF,Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 * Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2,NNF1);
     NNF = (NNF1,NNF2)).
nnf(A v B,FreeV,NNF,Paths) :- !,
    nnf(A,FreeV,NNF1,Paths1),
    nnf(B,FreeV,NNF2,Paths2),
    Paths is Paths1 + Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2;NNF1);
     NNF = (NNF1;NNF2)).
nnf(Lit,_,Lit,1).

```

Apéndice B

Algoritmo de unificación de Stickel

Ya que la unificación de Prolog no es correcta (para una explicación más detallada ver [Fla94]), `leanTAP` necesita definir un predicado de unificación que sí lo sea. Es lo que veremos en este apéndice, donde comentamos todo el código del fichero `unify.pl` de `leanTAP`, que contiene el algoritmo de unificación correcta de Stickel. El predicado principal es `unify(?TermA, ?TermB)`, que tiene éxito si hay una unificación correcta de los términos `TermA` y `TermB`.

Comenzamos declarando el fichero como un módulo donde el predicado que se exporta es `unify/2`.

```
:- module(unify,[unify/2]).
```

La primera cláusula es la principal, encargada de comprobar si existe una unificación correcta entre dos términos. Mostramos la cláusula y a continuación explicamos su funcionamiento:

```
unify(X,Y) :-
    var(X) -> % 1
        (var(Y) -> % 1a
            X = Y;
        %true -> % 1b
            functor(Y,_,N),
            (N = 0 -> % 1b-1
                true;
            N = 1 -> % 1b-2
                arg(1,Y,Y1), not_occurs_in(X,Y1);
            %true -> % 1b-3
                not_occurs_in_args(X,Y,N)),
            X = Y);
    var(Y) -> % 2
        functor(X,_,N),
        (N = 0 ->
            true;
        N = 1 ->
            arg(1,X,X1), not_occurs_in(Y,X1);
```

```

        %true ->
                not_occurs_in_args(Y,X,N)),
    X = Y;
%true ->                                % 3
    functor(X,F,N),
    functor(Y,F,N),
    (N = 0 ->                               % 3a
        true;
    N = 1 ->                               % 3b
        arg(1,X,X1), arg(1,Y,Y1), unify(X1,Y1);
%true ->                                % 3c
        unify_args(X,Y,N)).

```

Para que exista una unificación correcta entre los términos X e Y tiene que ocurrir uno de entre varios casos posibles. Los mostramos a continuación, según una numeración que coincide con los números anotados en los comentarios de la cláusula anterior:

1. Que X sea una variable. Entonces puede ocurrir:
 - 1a. Que también Y lo sea, en cuyo caso basta unificar ambas variables con la unificación común de Prolog.
 - 1b. Que Y no sea una variable, en cuyo caso será un término. La unificación se producirá sólo si X no ocurre en los argumentos de Y . Los diferentes casos de **1b** aseguran esta condición, ya que:
 - 1b-1. Si la aridad de Y es cero, la unificación está asegurada.
 - 1b-2. Si la aridad de Y es uno, sólo hay que comprobar que X no ocurra en el único argumento de Y .
 - 1b-3. En otro caso, si la aridad de Y es N , habrá que comprobar que X no ocurra en ninguno de los N argumentos de Y .
2. Que no siendo X una variable, lo sea Y . Este caso es análogo al **1b**, pero siendo X el término e Y la variable.
3. Que ni X ni Y sean variables, sino términos los dos. En este caso ambos términos deben tener el mismo functor y la misma aridad. Entonces,
 - 3a. Si la aridad es igual a cero, son el mismo término y unifican.
 - 3b. Si la aridad es igual a uno, los dos argumentos de X e Y deben tener una unificación correcta.
 - 3c. Si la aridad es mayor a uno, debe haber unificaciones correctas para los argumentos de X e Y .

En el caso **3c** anterior hemos usado el predicado `unify_args/3`, que toma por primer y segundo argumento los dos términos que pretenden unificarse, y como tercero la aridad de los mismos. Este predicado está definido por la siguiente cláusula, explicada a continuación:

```

unify_args(X,Y,N) :-
    N = 2 ->
        arg(2,X,X2), arg(2,Y,Y2), unify(X2,Y2),
        arg(1,X,X1), arg(1,Y,Y1), unify(X1,Y1);
    %true ->
        arg(N,X,Xn), arg(N,Y,Yn), unify(Xn,Yn),
        N1 is N - 1, unify_args(X,Y,N1).

```

Vimos que para llamar a este predicado, la aridad de los términos era mayor o igual a 2. Ahora pueden ocurrir los siguientes casos:

- Que la aridad sea 2. Entonces debe haber unificaciones correctas tanto para los primeros como para los segundos argumentos de **X** e **Y**.
- Que la aridad sea mayor a dos. Entonces debe haber una unificación correcta para los últimos argumentos de **X** e **Y**, y también unificaciones correctas para los argumentos anteriores, lo que se comprueba con la llamada recursiva a `unify_args/3`.

Otro de los predicados empleados es `not_occurs_in(+Var, +Term)`, donde **Var** es una variable y **Term** un término. Tiene éxito si la variable **Var** no ocurre en **Term**:

```

not_occurs_in(Var,Term) :-
    Var == Term -> % 1
        fail;
    var(Term) -> % 2
        true;
    %true -> % 3
        functor(Term,_,N),
        (N = 0 -> % 3a
            true;
        N = 1 -> % 3b
            arg(1,Term,Arg1), not_occurs_in(Var,Arg1);
    %true -> % 3c
        not_occurs_in_args(Var,Term,N)).

```

También aquí se reconocen distintos casos:

1. Que el término sea la misma variable. Entonces falla.
2. Que el término sea otra variable, entonces tiene éxito.
3. Que el término sea una functor de aridad **N**. Entonces:
 - 3a. Si la aridad es 0, tiene éxito, al ser una constante.
 - 3b. Si la aridad es uno, hay que comprobar que la variable no ocurra en el único argumento.

3c. En otro caso, hay que comprobar que la variable no ocurra en ninguno de los N argumentos.

Por último, `not_occurs_in_args(+Var,+Term,+N)` tiene éxito si la variable `Var` no ocurre en ninguno de los N argumentos de `Term`:

```
not_occurs_in_args(Var,Term,N) :-
    N = 2 ->
        arg(2,Term,Arg2), not_occurs_in(Var,Arg2),
        arg(1,Term,Arg1), not_occurs_in(Var,Arg1);
    %true ->
        arg(N,Term,ArgN), not_occurs_in(Var,ArgN),
        N1 is N - 1, not_occurs_in_args(Var,Term,N1).
```

En este caso si N , la aridad de `Term`, es 2, no debe ocurrir `Var` en ninguno de los dos argumentos de `Term`. En otro caso, la aridad es mayor a dos (pues siendo la aridad igual a uno nunca se llama a este predicado) y debe comprobarse que `Var` no ocurra en el último de los N argumentos, y luego llamar recursivamente el mismo predicado sobre los $N-1$ argumentos anteriores.

Los autores de `leanTP` recomiendan emplear implementaciones de la unificación correcta en lenguajes de más bajo nivel. Es lo que hemos hecho al construir versiones de `leanTP` que emplean el unificador correcto de Prolog, `unify_with_occurs_check/2`, comprobando que el número de inferencias, así como el tiempo, disminuyen.

Apéndice C

Código completo de lean^{TAP}

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fichero: leantap.pl %
% Copyright (C) 1993: Bernhard Beckert & Joachim Posegga %
%                               Universit"at Karlsruhe %
%                               Email: {beckert|posegga}@ira.uka.de %
%                               %
% Purpose: \LeanTaP: tableau-based theorem prover for NNF. %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- module(leantap,[prove/2,prove_uv/2]).
:- use_module(library(lists),[append/3]).
:- use_module(unify,[unify/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% BEGIN OF TOPLEVEL PREDICATES

% -----
% prove(+Fml,?VarLim)
% prove_uv(+Fml,?VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% prove_uv uses universal variables, prove does not.
%
% Iterative deepening is used when VarLim is unbound.
% Examples:
%
% | ?- prove((p(a) , -p(f(f(a))) , all(X,(-p(X) ; p(f(X))))), 1).
% no
% | ?- prove((p(a) , -p(f(f(a))) , all(X,(-p(X) ; p(f(X))))), 2).
% yes
%
prove(Fml,VarLim) :- nonvar(VarLim),!,prove(Fml,[],[],[],VarLim).
prove(Fml,Result) :-
    iterate(VarLim,0,prove(Fml,[],[],[],VarLim),Result).

prove_uv(Fml,VarLim) :- nonvar(VarLim),!,
    prove(Fml,[],[],[],[],VarLim).
```

```

prove_uv(Fml,Result) :-
    iterate(VarLim,0,prove(Fml, [], [], [], [], [], VarLim),Result).

iterate(Current,Current,Goal,Current) :- nl,
    write('Limit = '),
    write(Current),nl,
    Goal.
iterate(VarLim,Current,Goal,Result) :-
    Current1 is Current + 1,
    iterate(VarLim,Current1,Goal,Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END OF TOPLEVEL PREDICATES

% -----
% prove(+Fml,+UnExp,+Lits,+FreeV,+VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% Fml: inconsistent formula in skolemized negation normal form.
%   syntax: negation: '-', disj: ';', conj: ',',
%   quantifiers: 'all(X,<Formula>)', where 'X' is a prolog
%   variable.
%
% UnExp:  list of formula not yet expanded
% Lits:   list of literals on the current branch
% FreeV:  list of free variables on the current branch
% VarLim: max. number of free variables on each branch
%         (controlls when backtracking starts and alternate
%         substitutions for closing branches are considered)
%
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).
prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
prove(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_)).
prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).

% -----
% prove(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV,+VarLim)
%
% same as prove/5 above, but uses universal variables.
% additional parameters:
% DisV:  list of non-universal variables on branch

```

```

% UnivV: list of universal variables on branch
prove((A,B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    prove(A,[(UnivV:B)|UnExp],Lits,DisV,FreeV,UnivV,VarLim).
prove((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
    prove(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).
prove(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[(UnivV:all(X,Fml))],UnExp1),
    prove(Fml1,UnExp1,Lits,DisV,[X1|FreeV],[X1|UnivV],VarLim).
prove(Lit,_,[L|Lits],_,_,_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove(Lit,[],Lits,_,_,_,_)).
prove(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV,VarLim).

```


Apéndice D

lean^{TAP} con el unificador correcto de Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: leantap_oc.pl                                         %%
%% Versión de leantap con unify_with_occurs_check/2             %%
%% FERNANDO SOLER TOSCANO                                       %%
%% Versión para SWI-Prolog                                       10-oct-2003 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(leantap_oc, [prove_oc/2, prove_uv_oc/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% BEGIN OF TOPLEVEL PREDICATES
% -----
% prove_oc(+Fml,?VarLim)
% prove_uv_oc(+Fml,?VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% prove_uv_oc uses universal variables, prove_oc does not.
%
% Iterative deepening is used when VarLim is unbound.
% Examples:
%
% | ?- prove_oc((p(a) , -p(f(f(a))) , all(X,(-p(X); p(f(X))))), 1).
% no
% | ?- prove_oc((p(a) , -p(f(f(a))) , all(X,(-p(X); p(f(X))))), 2).
% yes
%
prove_oc(Fml,VarLim) :- nonvar(VarLim),!,
    prove_oc(Fml,[],[],[],VarLim).
prove_oc(Fml,Result) :-
    iterate(VarLim,0,prove_oc(Fml,[],[],[],VarLim),Result).
prove_uv_oc(Fml,VarLim) :- nonvar(VarLim),!,
    prove_oc(Fml,[],[],[],[],VarLim).
prove_uv_oc(Fml,Res) :-
    iterate(VarLim,0,prove_oc(Fml,[],[],[],[],VarLim),Res).
```

```

iterate(Current,Current,Goal,Current) :- %nl,
%   write('Limit = '),
%   write(Current),nl,
    Goal.

iterate(VarLim,Current,Goal,Result) :-
    Current1 is Current + 1,
    iterate(VarLim,Current1,Goal,Result).

%%%%%%%%%% END OF TOPLEVEL PREDICATES
% -----
% prove_oc(+Fml,+UnExp,+Lits,+FreeV,+VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% Fml: inconsistent formula in skolemized negation normal form.
%   syntax: negation: '-', disj: ';', conj: ',',
%   quantifiers: 'all(X,<Formula>)', where 'X' is a prolog
%   variable.
%
% UnExp:  list of formula not yet expanded
% Lits:   list of literals on the current branch
% FreeV:  list of free variables on the current branch
% VarLim: max. number of free variables on each branch
%         (controlls when backtracking starts and alternate
%         substitutions for closing branches are considered)
%
prove_oc((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove_oc(A,[B|UnExp],Lits,FreeV,VarLim).
prove_oc((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove_oc(A,UnExp,Lits,FreeV,VarLim),
    prove_oc(B,UnExp,Lits,FreeV,VarLim).
prove_oc(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove_oc(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
prove_oc(Lit,_,[L|Lits],_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg,L);
     prove_oc(Lit,[],Lits,_,_)).
prove_oc(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove_oc(Next,UnExp,[Lit|Lits],FreeV,VarLim).

% -----
% prove_oc(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV,+VarLim)
%
% same as prove_oc/5 above, but uses universal variables.
% additional parameters:
% DisV:  list of non-universal variables on branch
% UnivV: list of universal variables on branch

```

```

prove_oc((A,B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    prove_oc(A,[(UnivV:B)|UnExp],Lits,DisV,FreeV,UnivV,VarLim).
prove_oc((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove_oc(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
    prove_oc(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).
prove_oc(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,VLim) :- !,
    \+ length(FreeV,VLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[(UnivV:all(X,Fml))],UnExp1),
    prove_oc(Fml1,UnExp1,Lits,DisV,[X1|FreeV],[X1|UnivV],VLim).
prove_oc(Lit,_,[L|Lits],_,_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg,L);
     prove_oc(Lit,[],Lits,_,_,_)).
prove_oc(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_,VarLim) :-
    prove_oc(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV,VarLim).

```


Apéndice E

Versiones que devuelven las ramas abiertas

E.1. Versiones con unify/2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: leantab.pl                                     %%
%% Buscador de modelos de primer orden.                 %%
%% Basado en leanTaP (Bernhard Beckert & Joachim Posegga) %%
%% Fernando Soler Toscano    10-oct-2003.              %%
%% Versión para SWI-Prolog                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(leantab,[prove_abd/2,prove_abd/6,prove_abd_uv/2,
                prove_abd/8,tab/3,tab_uv/3]).
:- use_module(unify,[unify/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados de alto nivel                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd(+Fml,?VL)
% prove_abd_uv(+Fml,?VL)
%     tiene éxito si hay un tablero cerrado de Fml que tiene a lo
%     sumo VL variables libres en cada rama; prove_abd_uv/2 usa
%     variables universales. Se usa búsqueda en profundidad itera-
%     tiva cuando VL se deja sin especificar. Fml debe estar en
%     forma normal negativa skolemizada; con la sintaxis: negador:
%     '- ', disy: ';', conj: ','; cuantif.univ: 'all(X,<Formula>)'
%     donde 'X' es una variable prolog.
prove_abd(Fml,VL):- nonvar(VL),!,
    prove_abd(Fml,[],[],[],VL,ref:a-a).
prove_abd(Fml,Rs):-
    iterate(VL,0,prove_abd(Fml,[],[],[],VL,ref:a-a),Rs).
prove_abd_uv(Fml,VL):- nonvar(VL),!,
    prove_abd(Fml,[],[],[],[],VL,ref:a-a).
prove_abd_uv(Fml,Rs):-
```

```

        iterate(VL,0,prove_abd(Fml, [], [], [], [], [], VL, ref:a-a),Rs).
% iterate(+Variable,+Actual,+Objetivo,?Resultado)
%     tiene éxito cuando se cumple Objetivo dando a Variable el
%     valor Resultado, al que se llega incrementando Actual hasta
%     tener éxito.
iterate(Current,Current,Goal,Current) :- %nl,
%     write('Limit = '),
%     write(Current),nl,
%     Goal.
iterate(VL,Current,Goal,Result) :-
%     Current1 is Current + 1,
%     iterate(VL,Current1,Goal,Result).

% tab(+Fml,+Lim,?Tab)
% tab_uv(+Fml,+Lim,?Tab)
%     tienen éxito cuando Tab es la representación del tablero de
%     Fml que emplea a lo sumo Lim variables libres en cada rama.
%     - Fml debe estar en forma normal negativa skolemizada.
%     - Tab es un multiconjunto de multiconjuntos.
%     Sólo tomamos la primera solución, que es la buena.
%     La diferencia: tab_uv/2 usa variables universales, tab/2 no.
tab(Fml,Lim,Tab):-
%     prove_abd(Fml, [], [], [], Lim,abd:X-[]),!,X=Tab.
tab_uv(Fml,Lim,Tab):-
%     prove_abd(Fml, [], [], [], [], [], Lim,abd:X-[]),!,X=Tab.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prove_abd/6: demostrador y creador de modelos. %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd(+Form,+UnExp,+Lits,+FreeV,+VarLim,+I:?A1-?A2).
%     emplea los siguientes argumentos:
%     - Form: Fórmula de primer orden en forma normal negativa
%     skolemizada. Es con la que se trabaja.
%     - UnExp: Lista de fórmulas de primer orden en forma nor-
%     mal negativa skolemizada, que esperan a ser utilizadas.
%     - Lits: Pila de literales que se van encontrando, que
%     sirven para buscar cierres o construir modelos.
%     - FreeV: En cada rama, es la lista de sus variables
%     libres.
%     - VarLim: Es el máximo número de variables permitidas en
%     cada rama del tablero. Controla cuándo empieza el
%     backtracking y hace que se busquen cierres alternativos
%     a anteriores ramas.
%     - I: Indicador del modo en el que trabajamos. Será 'ref'
%     si buscamos un tablero cerrado, y 'abd' si queremos que
%     nos devuelva los modelos, caso de ser abierto.
%     - A1-A2: Lista de diferencia que representa el tablero
%     como multiconjunto de multiconjuntos de literales.
%     trabaja con Form, realizando lo siguiente:
%     1) Si Form es una conjunción (A,B) guarda B en UnExp y
%     sigue trabajando con A.
%     2) Si Form es una disyunción (A;B) hace dos ramas, usando

```

```

%           A en una de ellas y B en la otra. El tablero devuelto
%           será la unión de los tableros de cada rama.
%           3) Si Form es una cuantificación universal entonces, sólo
%           si hay en la rama menos variables libres que el límite
%           permitido (en otro caso falla), se toma una instancia
%           de la fórmula sin cuantificar, con una variable libre
%           que se guarda también en la lista de variables libres.
%           Form pasa al final de UnExp.
%           4) Si Form es un literal, trata de unificar su complemen-
%           tario con alguno de los literales ya encontrados. En
%           ese caso, el tablero devuelto en esa rama, es X-X,
%           vacío.
%           5) Cuando 4 no se da, el literal se añade a los demás li-
%           terales, y se pasa a trabajar con la primera fórmula
%           de UnExp.
%           6) Llega aquí si no hay ningún cierre en la rama, si no
%           quedan fórmulas por usar, y sólo si trabaja en modo
%           'abd'. Entonces construye el tablero de la rama con
%           todos los literales encontrados.
%           NOTA: Para que a 6 sólo llegue cuando se hayan realizado
%           todos los intentos posibles en 1-5, debe pedirse sólo
%           la primera solución para el modo 'abd'.
%           Ejemplos:
%           - Para pedir tablas:
%           ?- prove_abd((a,(-b;d)), [], [], [], 0,abd:E-[]).
%           E = [[-b, a], [d, a]]
%           - Para buscar tablas cerradas:
%           ?- prove_abd((a,-a), [], [], [], 0,ref:a-a).
%           Yes
prove_abd((A,B),UnExp,Lits,FreeV,VarLim,I:A1-A2) :- !,      % 1
    prove_abd(A,[B|UnExp],Lits,FreeV,VarLim,I:A1-A2).
prove_abd((A;B),UnExp,Lits,FreeV,VarLim,I:A1-A3) :- !,      % 2
    prove_abd(A,UnExp,Lits,FreeV,VarLim,I:A1-A2),
    prove_abd(B,UnExp,Lits,FreeV,VarLim,I:A2-A3).
prove_abd(all(X,Fml),UnExp,Lits,FreeV,VarLim,I:A1-A2) :- !, % 3
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove_abd(Fml1,UnExp1,Lits,[X1|FreeV],VarLim,I:A1-A2).
prove_abd(Lit,_,[L|Lits],_,_,_I:A-A) :-                      % 4
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove_abd(Lit,[],Lits,_,_,ref:A-A)).
prove_abd(Lit,[Next|UnExp],Lits,FreeV,VarLim,I:A1-A2) :-    % 5
    prove_abd(Next,UnExp,[Lit|Lits],FreeV,VarLim,I:A1-A2).
prove_abd(L,_,Ls,_,_,abd:[L|Ls]|A)-A)                       % 6

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prove_abd/8: añadiendo variables universales.              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd(+Form,+UnExp,+Lits,+DisV,+FreeV,+UnivV,+VarLim,
%           +I:?A1-?A2).
%           variante de prove_abd/6 para trabajar con variables univer-

```

```

%      sales. Se trata de la misma diferencia que hay entre prove/5
%      y prove/7.
%      Tiene como parámetros adicionales:
%      - DisV:  lista de variables no universales de la rama.
%      - UnivV: lista de variables universales de la rama.
prove_abd((A,B),UE,Lits,DisV,FreeV,UnivV,VL,I:A1-A2) :- !,
    prove_abd(A,[(UnivV:B)|UE],Lits,DisV,FreeV,UnivV,VL,
        I:A1-A2).
prove_abd((A;B),UE,Lits,DisV,FreeV,UnivV,VL,I:A1-A3) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove_abd(A,UE,Lits,(DisV+UnivV),FreeV,[],VL,I:A1-A2),
    prove_abd(B,UE,Lits1,(DisV+UnivV),FreeV,[],VL,I:A2-A3).
prove_abd(all(X,Fml),UE,Lits,DisV,FreeV,UnivV,VL,I:A1-A2) :- !,
    \+ length(FreeV,VL),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UE,[(UnivV:all(X,Fml))],UE1),
    prove_abd(Fml1,UE1,Lits,DisV,[X1|FreeV],[X1|UnivV],VL,
        I:A1-A2).
prove_abd(Lit,_,[L|Lits],_,_,_,_,I:A-A) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove_abd(Lit,[],Lits,_,_,_,ref:A-A)).
prove_abd(Lit,[(UnivV:Next)|UE],Lits,DisV,FreeV,_,VL,I:A1-A2) :-
    prove_abd(Next,UE,[Lit|Lits],DisV,FreeV,UnivV,VL,I:A1-A2).
prove_abd(L,_,Ls,_,_,_,_,abd:[[L|Ls]|A]-A).

```

E.2. Versiones con el unificador correcto de Prolog

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: leantab_oc.pl                                     %%
%% Variante de leantab:                                     %%
%% Sólo cambia unify/2 por unify_with_occurs_check/2      %%
%% Fernando Soler Toscano                                  10-oct-2003 %%
%% Versión para SWI-Prolog                                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(leantab_oc,[prove_abd_oc/2,prove_abd_oc/6,
    prove_abd_uv_oc/2,prove_abd_oc/8,tab_oc/3,tab_uv_oc/3]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados de alto nivel                                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd_oc(+Fml,?VL)
% prove_abd_uv_oc(+Fml,?VL)
% tiene éxito si hay un tablero cerrado de Fml que tiene a lo
% sumo VL variables libres en cada rama; prove_abd_uv_oc/2 usa
% variables universales. Se usa búsqueda en profundidad itera-
% tiva cuando VL se deja sin especificar. Fml debe estar en
% forma normal negativa skolemizada; con la sintaxis: negador:
% '-', disy: ','; conj: ','; cuantif.univ: 'all(X,<Formula>)'

```

```

%     donde 'X' es una variable prolog.
prove_abd_oc(Fml,VL):- nonvar(VL),!,
    prove_abd_oc(Fml,[],[],[],VL,ref:a-a).
prove_abd_oc(Fml,Rs) :-
    iterate(VL,0,prove_abd_oc(Fml,[],[],[],VL,ref:a-a),Rs).
prove_abd_uv_oc(Fml,VL) :- nonvar(VL),!,
    prove_abd_oc(Fml,[],[],[],[],[],VL,ref:a-a).
prove_abd_uv_oc(Fml,Rs) :-
    iterate(VL,0,prove_abd_oc(Fml,[],[],[],[],[],VL,ref:a-a),
        Rs).

% iterate(+Variable,+Actual,+Objetivo,?Resultado)
%     tiene éxito cuando se cumple Objetivo dando a Variable el
%     valor Resultado, al que se llega incrementando Actual hasta
%     tener éxito.
iterate(Current,Current,Goal,Current) :- %nl,
%     write('Limit = '),
%     write(Current),nl,
    Goal.
iterate(VL,Current,Goal,Result) :-
    Current1 is Current + 1,
    iterate(VL,Current1,Goal,Result).

% tab_oc(+Fml,+Lim,?Tab)
% tab_uv_oc(+Fml,+Lim,?Tab)
%     tienen éxito cuando Tab es la representación del tablero de
%     Fml que emplea a lo sumo Lim variables libres en cada rama.
%     - Fml debe estar en forma normal negativa skolemizada.
%     - Tab es un multiconjunto de multiconjuntos.
%     Sólo se toma la primera solución que es la buena.
%     la diferencia, que tab_uv_oc/2 usa variables universales
tab_oc(Fml,Lim,Tab):-
    prove_abd_oc(Fml,[],[],[],Lim,abd:X-[]),!,X=Tab.
tab_uv_oc(Fml,Lim,Tab):-
    prove_abd_oc(Fml,[],[],[],[],[],Lim,abd:X-[]),!,X=Tab.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%     prove_abd_oc/6: demostrador y creador de modelos.                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd_oc(+Form,+UnExp,+Lits,+FreeV,+VarLim,+I:?A1-?A2).
%     emplea los siguientes argumentos:
%     - Form: Fórmula de primer orden en forma normal negativa
%           skolemizada. Es con la que se trabaja.
%     - UnExp: Lista de fórmulas de primer orden en forma nor-
%           mal negativa skolemizada, que esperan a ser utilizadas.
%     - Lits: Pila de literales que se van encontrando, que
%           sirven para buscar cierres o construir modelos.
%     - FreeV: En cada rama, es la lista de sus variables li-
%           bres.
%     - VarLim: Es el máximo número de variables permitidas en
%           cada rama del tablero. Controla cuándo empieza el
%           backtracking y hace que se busquen cierres alternativos

```

```

%      a anteriores ramas.
%      - I: Indicador del modo en el que trabajamos. Será 'ref'
%      si buscamos un tablero cerrado, y 'abd' si queremos que
%      nos devuelva los modelos, caso de ser abierto.
%      - A1-A2: Lista de diferencia que representa el tablero
%      como multiconjunto de multiconjuntos de literales.
%      trabaja con Form, realizando lo siguiente:
%      1) Si Form es una conjunción (A,B) guarda B en UnExp y
%      sigue trabajando con A.
%      2) Si Form es una disyunción (A;B) hace dos ramas, usan-
%      do A en una de ellas y B en la otra. El tablero de-
%      vuelto será la unión de los tableros resultantes de
%      cada rama.
%      3) Si Form es una cuantificación universal entonces, sólo
%      si hay en la rama menos variables libres que el límite
%      permitido (en otro caso falla), se toma una instancia
%      de la fórmula sin cuantificar, con una variable libre
%      que se guarda también en la lista de variables libres.
%      Form pasa al final de UnExp.
%      4) Si Form es un literal, trata de unificar su complemen-
%      tario con alguno de los literales ya encontrados. En
%      ese caso, el tablero devuelto en esa rama, es X-X,
%      vacío.
%      5) Cuando 4 no se da, el literal se añade a los demás li-
%      terales, y se pasa a trabajar con la primera fórmula
%      de UnExp.
%      6) Llega aquí si no hay ningún cierre en la rama, si no
%      quedan fórmulas por usar, y sólo si trabaja en modo
%      'abd'. Entonces construye el tablero de la rama con
%      todos los literales encontrados.
%      NOTA: Para que a 6 sólo llegue cuando se hayan realizado
%      todos los intentos posibles en 1-5, debe pedirse sólo
%      la primera solución para el modo 'abd'.
%      Ejemplos:
%      - Para pedir tablas:
%      ?- prove_abd_oc((a,(-b;d)),[],[],[],0,abd:E-[]).
%      E = [[-b, a], [d, a]]
%      - Para buscar tablas cerradas:
%      ?- prove_abd_oc((a,-a),[],[],[],0,ref:a-a).
%      Yes
prove_abd_oc((A,B),UnExp,Lits,FreeV,VarLim,I:A1-A2) :- !,      % 1
    prove_abd_oc(A,[B|UnExp],Lits,FreeV,VarLim,I:A1-A2).
prove_abd_oc((A;B),UnExp,Lits,FreeV,VarLim,I:A1-A3) :- !,    % 2
    prove_abd_oc(A,UnExp,Lits,FreeV,VarLim,I:A1-A2),
    prove_abd_oc(B,UnExp,Lits,FreeV,VarLim,I:A2-A3).
prove_abd_oc(all(X,Fml),UnExp,Lits,FreeV,VarLim,I:A1-A2) :- !, % 3
    \+ length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove_abd_oc(Fml1,UnExp1,Lits,[X1|FreeV],VarLim,I:A1-A2).
prove_abd_oc(Lit,_,[L|Lits],_,_,I:A-A) :-                    % 4
    (Lit = -Neg; -Lit = Neg) ->

```

```

        (unify_with_occurs_check(Neg,L);
         prove_abd_oc(Lit, [],Lits,_,_,ref:A-A)).
prove_abd_oc(Lit, [Next|UnExp],Lits,FreeV,VarLim,I:A1-A2) :- % 5
    prove_abd_oc(Next,UnExp, [Lit|Lits],FreeV,VarLim,I:A1-A2).
prove_abd_oc(L,_,Ls,_,_,abd: [[L|Ls] | A]-A). % 6

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prove_abd_oc/8: añadiendo variables universales. %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_abd_oc(+For,+UExp,+Lts,+DisV,+FreeV,+UnivV,+VarLim,
%             +I:?A1-?A2).
% Variante de prove_abd_oc/6 que usa con variables universales
% Se trata de la misma diferencia que hay entre prove/5 y
% prove/7. Tiene como argumentos adicionales:
% - DisV: lista de variables no universales de la rama.
% - UnivV: lista de variables universales de la rama.
prove_abd_oc((A,B),UE,Lts,DisV,FreeV,UnivV,VL,I:A1-A2) :- !,
    prove_abd_oc(A, [(UnivV:B)|UE],Lts,DisV,FreeV,UnivV,VL,
        I:A1-A2).
prove_abd_oc((A;B),UE,Lits,DisV,FreeV,UnivV,VL,I:A1-A3) :- !,
    copy_term((Lits,DisV), (Lits1,DisV)),
    prove_abd_oc(A,UE,Lits, (DisV+UnivV),FreeV, [],VL,I:A1-A2),
    prove_abd_oc(B,UE,Lits1, (DisV+UnivV),FreeV, [],VL,I:A2-A3).
prove_abd_oc(all(X,F),UE,Lts,DisV,FreeV,UnivV,VL,I:A1-A2) :- !,
    \+ length(FreeV,VL),
    copy_term((X,F,FreeV), (X1,F1,FreeV)),
    append(UE, [(UnivV:all(X,F))],UE1),
    prove_abd_oc(F1,UE1,Lts,DisV, [X1|FreeV], [X1|UnivV],VL,
        I:A1-A2).
prove_abd_oc(Lit,_, [L|Lits],_,_,_,_,I:A-A) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg,L);
     prove_abd_oc(Lit, [],Lits,_,_,_,ref:A-A)).
prove_abd_oc(Lit, [(UnivV:Next)|UE],Lits,DisV,FreeV,_,VL,I:A1-A2) :-
    prove_abd_oc(Next,UE, [Lit|Lits],DisV,FreeV,UnivV,VL,
        I:A1-A2).
prove_abd_oc(L,_,Ls,_,_,_,_,abd: [[L|Ls] | A]-A).

```


Apéndice F

Versiones con listas fijas de variables libres

F.1. Versiones sin variables universales

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fichero: variacion1.pl
%% Versiones de:
%%   - prove/2           de leantap.pl
%%   - prove_oc/2       de leantap_oc
%%   - prove_abd/2      de leantab.pl
%%   - prove_abd_oc/2   de leantab_oc.pl
%% usando listas de variables libres de cardinalidad prefijada
%% FERNANDO SOLER TOSCANO 16-oct-2003.
%% Versión para SWI-Prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-module(variacion1,[prove_var1/2,prove_var1_oc/2,
                    prove_var1_abd/2,prove_var1_abd_oc/2,
                    tab_var1/3,tab_var1_oc/3]).

:-use_module(unify,[unify/2]).

% La variación que se introduce en este fichero sobre los métodos
% de prueba mencionados consiste en quitar de los argumentos el lí-
% mite y hacer la lista de variables libres fija desde el princi-
% pio. De esta forma, cuando se trata un cuantificador universal,
% se sustituye la variable por todas las variables libres permiti-
% das, y nos deshacemos de la fórmula cuantificada.

% Ilustramos la variación que se realiza en LeanTaP:
% - Si cada llamada en LeanTaP tiene la forma:
%
%     prove(+Fml,+UnExp,+Lits,+FreeV,+VarLim)
%
% ahora prescindimos de VarLim, y a cambio FreeV es una lista de
% cardinalidad igual a la profundidad con la que se trabaja, a
% la que nunca se añade ni quita ningún elemento. De este modo,
% nos queda:
```

```

%           prove_var1(+Fml,+UnExp,+Lits,+FreeV).
% - El tratamiento de los cuantificadores universales consistirá
%   en sustituir la variable cuantificada por todas las variables
%   libres permitidas y añadir el resultado a la rama. Queda:
%   prove_var1(all(X,Fml),UnExp,Lits,FreeV) :- !,
%       findall(F,(member(Var,FreeV),
%                   copy_term((X,Fml,FreeV),(Var,F,FreeV))),[F1|Forms]),
%       append(UnExp,Forms,UnExp1),
%       prove_var1(F1,UnExp1,Lits,FreeV).
% - El procedimiento de iteración incluirá la creación de la lista
%   de variables libres con la longitud igual a la profundidad que
%   en cada momento se considere.

```

```

% Mostramos a continuación una tabla que muestra los nombres de los
% procedimientos definidos, así como los procedimientos de los que
% son variante. Finalmente, se asigna a cada procedimiento el nom-
% bre del método que lo empleará en test.pl:

```

```

% +-----+-----+-----+
% |   PROCEDIMIENTO: | VARIANTE DE: |   MÉTODO: |
% +-----+-----+-----+
% |   prove_var1/2 |   prove/2 |   provevar1 |
% +-----+-----+-----+
% |   prove_var1_oc/2 |   prove_oc/2 |   provevar1_oc |
% +-----+-----+-----+
% |   prove_var1_abd/2 |   prove_abd/2 |   provevar1_abd |
% +-----+-----+-----+
% |   prove_var1_abd_oc/2 |   prove_abd_oc/2 |   provevar1_abd_oc |
% +-----+-----+-----+

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Procedimientos principales: %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Mostramos a continuación los procedimientos que se llamarán en
% las pruebas. Según la tabla anterior, cada procedimiento es una
% variante de un procedimiento ya conocido.

```

```

% prove_var1(+Fml,?VarLim)
%   mismo funcionamiento que prove/2 pero con la variante expli-
%   cada. Vemos que en cada caso se crea una lista de longitud
%   igual al límite y se hace la llamada a prove_var1/4.
prove_var1(Fml,VarLim) :- nonvar(VarLim),!,
    length(L,VarLim),
    prove_var1(Fml,[],[],L).
prove_var1(Fml,Result) :-
    iterate(VarLim,0,(length(L,VarLim),
        prove_var1(Fml,[],[],L)),Result).

```

```

% prove_var1_oc(+Fml,?VarLim)
%   como prove_oc/2 con la variante explicada.
prove_var1_oc(Fml,VarLim) :- nonvar(VarLim),!,
    length(L,VarLim),
    prove_var1_oc(Fml,[],[],L).

```

```

prove_var1_oc(Fml,Result) :-
    iterate(VarLim,0,(length(L,VarLim),
        prove_var1_oc(Fml,[],[],L)),Result).

% prove_var1_abd(+Fml,?VarLim)
%   como prove_abd/2 con la variante explicada.
prove_var1_abd(Fml,VL):- nonvar(VL),!,
    length(L,VL),
    prove_var1_abd(Fml,[],[],L,ref:a-a).
prove_var1_abd(Fml,Rs) :-
    iterate(VL,0,(length(L,VL),
        prove_var1_abd(Fml,[],[],L,ref:a-a)),Rs).

% prove_var1_abd_oc(+Fml,?VarLim)
%   como prove_abd_oc/2 con la variante explicada.
prove_var1_abd_oc(Fml,VL):- nonvar(VL),!,
    length(L,VL),
    prove_var1_abd_oc(Fml,[],[],L,ref:a-a).
prove_var1_abd_oc(Fml,Rs) :-
    iterate(VL,0,(length(L,VL),
        prove_var1_abd_oc(Fml,[],[],L,ref:a-a)),Rs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Iteración: iterate/4 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% iterate/4 está definido igual que en los demás ficheros.
iterate(Current,Current,Goal,Current) :- !,
%   write('Limit = '),
%   write(Current),nl,
    Goal.
iterate(VarLim,Current,Goal,Result) :-
    Current1 is Current + 1,
    iterate(VarLim,Current1,Goal,Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definición de los demostradores: %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_var1(+Fml,+UnExp,+Lits,+FreeV).
%   Variante de prove/5 con lista de variables libres prefijada
prove_var1((A,B),UnExp,Lits,FreeV) :- !,
    prove_var1(A,[B|UnExp],Lits,FreeV).
prove_var1((A;B),UnExp,Lits,FreeV) :- !,
    prove_var1(A,UnExp,Lits,FreeV),
    prove_var1(B,UnExp,Lits,FreeV).
prove_var1(all(X,Fml),UnExp,Lits,FreeV) :- !,
    findall(F,(member(Var,FreeV),
        copy_term((X,Fml,FreeV),(Var,F,FreeV))),[F1|Forms]),
    append(UnExp,Forms,UnExp1),
    prove_var1(F1,UnExp1,Lits,FreeV).
prove_var1(Lit,_,[L|Lits],_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L);

```

```

    prove_var1(Lit, [], Lits, _).
prove_var1(Lit, [Next|UnExp], Lits, FreeV) :-
    prove_var1(Next, UnExp, [Lit|Lits], FreeV).

% prove_var1_oc(+Fml,+UnExp,+Lits,+FreeV).
% Variante de prove_oc/5 con lista de variables libres prefijada
prove_var1_oc((A,B), UnExp, Lits, FreeV) :- !,
    prove_var1_oc(A, [B|UnExp], Lits, FreeV).
prove_var1_oc((A;B), UnExp, Lits, FreeV) :- !,
    prove_var1_oc(A, UnExp, Lits, FreeV),
    prove_var1_oc(B, UnExp, Lits, FreeV).
prove_var1_oc(all(X, Fml), UnExp, Lits, FreeV) :- !,
    findall(F, (member(Var, FreeV),
        copy_term((X, Fml, FreeV), (Var, F, FreeV))), [F1|Forms]),
    append(UnExp, Forms, UnExp1),
    prove_var1_oc(F1, UnExp1, Lits, FreeV).
prove_var1_oc(Lit, _, [L|Lits], _) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg, L);
        prove_var1_oc(Lit, [], Lits, _)).
prove_var1_oc(Lit, [Next|UnExp], Lits, FreeV) :-
    prove_var1_oc(Next, UnExp, [Lit|Lits], FreeV).

% prove_var1_abd(+Fml,+UnExp,+Lits,+FreeV,?I:?A1-?A2).
% La misma variante, para prove_abd/6.
prove_var1_abd((A,B), UnExp, Lits, FreeV, I:A1-A2) :- !,
    prove_var1_abd(A, [B|UnExp], Lits, FreeV, I:A1-A2).
prove_var1_abd((A;B), UnExp, Lits, FreeV, I:A1-A3) :- !,
    prove_var1_abd(A, UnExp, Lits, FreeV, I:A1-A2),
    prove_var1_abd(B, UnExp, Lits, FreeV, I:A2-A3).
prove_var1_abd(all(X, Fml), UnExp, Lits, FreeV, I:A1-A2) :- !,
    findall(F, (member(Var, FreeV),
        copy_term((X, Fml, FreeV), (Var, F, FreeV))), [F1|Forms]),
    append(UnExp, Forms, UnExp1),
    prove_var1_abd(F1, UnExp1, Lits, FreeV, I:A1-A2).
prove_var1_abd(Lit, _, [L|Lits], _, I:A-A) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg, L); prove_var1_abd(Lit, [], Lits, _, ref:A-A)).
prove_var1_abd(Lit, [Next|UnExp], Lits, FreeV, I:A1-A2) :-
    prove_var1_abd(Next, UnExp, [Lit|Lits], FreeV, I:A1-A2).
prove_var1_abd(L, _, Ls, _, abd: [[L|Ls] | A]-A).

% prove_var1_abd_oc(+Fml,+UnExp,+Lits,+FreeV,?I:?A1-?A2).
% La misma variante, para prove_abd_oc/6.
prove_var1_abd_oc((A,B), UnExp, Lits, FreeV, I:A1-A2) :- !,
    prove_var1_abd_oc(A, [B|UnExp], Lits, FreeV, I:A1-A2).
prove_var1_abd_oc((A;B), UnExp, Lits, FreeV, I:A1-A3) :- !,
    prove_var1_abd_oc(A, UnExp, Lits, FreeV, I:A1-A2),
    prove_var1_abd_oc(B, UnExp, Lits, FreeV, I:A2-A3).
prove_var1_abd_oc(all(X, Fml), UnExp, Lits, FreeV, I:A1-A2) :- !,
    findall(F, (member(Var, FreeV),
        copy_term((X, Fml, FreeV), (Var, F, FreeV))), [F1|Forms]),

```

```

    append(UnExp,Forms,UnExp1),
    prove_var1_abd_oc(F1,UnExp1,Lits,FreeV,I:A1-A2).
prove_var1_abd_oc(Lit,_,[L|Lits],_,_I:A-A) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg,L);
     prove_var1_abd_oc(Lit,[],Lits,_,ref:A-A)).
prove_var1_abd_oc(Lit,[Next|UnExp],Lits,FreeV,I:A1-A2) :-
    prove_var1_abd_oc(Next,UnExp,[Lit|Lits],FreeV,I:A1-A2).
prove_var1_abd_oc(L,_,Ls,_,abd:[[L|Ls]|A]-A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Construcción de tableros %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tab_var1(+Fml,+Lim,?Tab).
%     tiene éxito cuando Tab es un tablero de Fml construido con
%     prove_var1_abd/5, empleando Lim variables libres.
tab_var1(Fml,Lim,Tab):-
    length(L,Lim),
    prove_var1_abd(Fml,[],[],L,abd:X-[]),!,X=Tab.

% tab_var1_oc(+Fml,+Lim,?Tab).
%     tiene éxito cuando Tab es un tablero de Fml construido con
%     prove_var1_abd_oc/5, empleando Lim variables libres.
tab_var1_oc(Fml,Lim,Tab):-
    length(L,Lim),
    prove_var1_abd_oc(Fml,[],[],L,abd:X-[]),!,X=Tab.

```

F.2. Versiones con variables universales

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fichero: variacion1_uv.pl %%
%% Versiones de: %%
%%   - prove_uv/2           de leantap.pl %%
%%   - prove_uv_oc/2       de leantap_oc.pl %%
%%   - prove_abd_uv/2      de leantab.pl %%
%%   - prove_abd_uv_oc/2   de leantab_oc.pl %%
%%   con listas de variables libres de cardinalidad prefijada %%
%% FERNANDO SOLER TOSCANO 17-oct-2003. %%
%% Versión para SWI-Prolog %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-module(variacion1_uv,[prove_var1_uv/2,prove_var1_uv_oc/2,
    prove_var1_abd_uv/2,prove_var1_abd_uv_oc/2,
    tab_var1_uv/3,tab_var1_uv_oc/3]).

:-use_module(unify,[unify/2]).

% Se trata de la misma modificación que se hace en el fichero
% variacion1.pl, pero adaptada a las variables universales de cada
% uno de los métodos de prueba. Seguimos la misma presentación. En
% este caso:

```

```

% +-----+-----+-----+
% |      PROCEDIMIENTO: |      VARIANTE DE: |      MÉTODO: |
% +-----+-----+-----+
% |   prove_var1_uv/2 |   prove_uv/2 |   provevar1_uv |
% +-----+-----+-----+
% |   prove_var1_uv_oc/2 |   prove_uv_oc/2 |   provevar1_uv_oc |
% +-----+-----+-----+
% |   prove_var1_abd_uv/2 |   prove_abd_uv/2 |   provevar1_abd_uv |
% +-----+-----+-----+
% | prove_var1_abd_uv_oc/2 | prove_abd_uv_oc/2 | provevar1_abd_uv_oc |
% +-----+-----+-----+

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Procedimientos principales:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Mostramos a continuación los procedimientos que se llamarán en
% las pruebas. Según la tabla anterior, cada procedimiento es una
% variante de un procedimiento ya conocido.

% prove_var1_uv(+Fml,?VarLim)
%   mismo funcionamiento que prove_uv/2, con la variante expli-
%   cada.
prove_var1_uv(Fml,VarLim) :- nonvar(VarLim),!,
    length(L,VarLim),
    prove_var1_uv(Fml,[],[],[],L,[]).
prove_var1_uv(Fml,Result) :-
    iterate(VarLim,0,(length(L,VarLim),
        prove_var1_uv(Fml,[],[],[],L,[])),Result).

% prove_var1_uv_oc(+Fml,?VarLim)
%   Variante de prove_uv_oc/2.
prove_var1_uv_oc(Fml,VarLim) :- nonvar(VarLim),!,
    length(L,VarLim),
    prove_var1_uv_oc(Fml,[],[],[],L,[]).
prove_var1_uv_oc(Fml,Result) :-
    iterate(VarLim,0,(length(L,VarLim),
        prove_var1_uv_oc(Fml,[],[],[],L,[])),Result).

% prove_var1_abd_uv(+Fml,?VarLim)
%   Variante de prove_abd_uv/2.
prove_var1_abd_uv(Fml,VL):- nonvar(VL),!,
    length(L,VL),
    prove_var1_abd_uv(Fml,[],[],[],L,[],ref:a-a).
prove_var1_abd_uv(Fml,Rs) :-
    iterate(VL,0,(length(L,VL),
        prove_var1_abd_uv(Fml,[],[],[],L,[],ref:a-a)),Rs).

% prove_var1_abd_uv_oc(+Fml,?VarLim)
%   Variante de prove_abd_uv_oc/2.
prove_var1_abd_uv_oc(Fml,VL):- nonvar(VL),!,
    length(L,VL),

```

```

    prove_var1_abd_uv_oc(Fml, [], [], [], L, [], ref:a-a) .
prove_var1_abd_uv_oc(Fml, Rs) :-
    iterate(VL, 0, (length(L, VL),
        prove_var1_abd_uv_oc(Fml, [], [], [], L, [], ref:a-a)), Rs) .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Iteración: iterate/4 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% iterate/4 está definido igual que en los demás ficheros.
iterate(Current, Current, Goal, Current) :- !,
%     write('Limit = '),
%     write(Current), nl,
    Goal.
iterate(VarLim, Current, Goal, Result) :-
    Current1 is Current + 1,
    iterate(VarLim, Current1, Goal, Result) .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Definición de los demostradores: %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prove_var1_uv(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV) .
%     Variante de prove/7 con lista de variables libres fija.
prove_var1_uv((A,B), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    prove_var1_uv(A, [(UnivV:B)|UnExp], Lits, DisV, FreeV, UnivV) .
prove_var1_uv((A;B), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    copy_term((Lits, DisV), (Lits1, DisV)),
    prove_var1_uv(A, UnExp, Lits, (DisV+UnivV), FreeV, []),
    prove_var1_uv(B, UnExp, Lits1, (DisV+UnivV), FreeV, []) .
prove_var1_uv(all(X, Fml), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    findall(( [Var|UnivV]:F), (member(Var, FreeV),
        copy_term((X, Fml, FreeV), (Var, F, FreeV))),
        [(UnivV1:F1)|Forms]),
    append(UnExp, Forms, UnExp1),
    prove_var1_uv(F1, UnExp1, Lits, DisV, FreeV, UnivV1) .
prove_var1_uv(Lit, _, [L|Lits], _, _, _) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg, L); prove_var1_uv(Lit, [], Lits, _, _, _)) .
prove_var1_uv(Lit, [(UnivV:Next)|UnExp], Lits, DisV, FreeV, _) :-
    prove_var1_uv(Next, UnExp, [Lit|Lits], DisV, FreeV, UnivV) .

% prove_var1_uv_oc(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV) .
%     Variante de prove_oc/7 con lista de variables libres fija.
prove_var1_uv_oc((A,B), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    prove_var1_uv_oc(A, [(UnivV:B)|UnExp], Lits, DisV, FreeV, UnivV) .
prove_var1_uv_oc((A;B), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    copy_term((Lits, DisV), (Lits1, DisV)),
    prove_var1_uv_oc(A, UnExp, Lits, (DisV+UnivV), FreeV, []),
    prove_var1_uv_oc(B, UnExp, Lits1, (DisV+UnivV), FreeV, []) .
prove_var1_uv_oc(all(X, Fml), UnExp, Lits, DisV, FreeV, UnivV) :- !,
    findall(( [Var|UnivV]:F), (member(Var, FreeV),
        copy_term((X, Fml, FreeV), (Var, F, FreeV))),
        [(UnivV1:F1)|Forms]),

```

```

    append(UnExp,Forms,UnExp1),
    prove_var1_uv_oc(F1,UnExp1,Lits,DisV,FreeV,UnivV1).
prove_var1_uv_oc(Lit,_,[L|Lits],_,_,_) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify_with_occurs_check(Neg,L);
     prove_var1_uv_oc(Lit,[],Lits,_,_,_)).
prove_var1_uv_oc(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_) :-
    prove_var1_uv_oc(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV).

% prove_var1_abd_uv(+Fml,+UnExp,+Lits,+DisV,+FreeV,
%                 +UnivV,?I:?A1-?A2).
% Variante de prove_abd/8 con lista de variables libres fija.
prove_var1_abd_uv((A;B),UE,Ls,DisV,FreeV,UnivV,I:A1-A2) :- !,
    prove_var1_abd_uv(A,[(UnivV:B)|UE],Ls,DisV,FreeV,UnivV,
                      I:A1-A2).
prove_var1_abd_uv((A;B),UE,Lits,DisV,FreeV,UnivV,I:A1-A3) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove_var1_abd_uv(A,UE,Lits,(DisV+UnivV),FreeV,[],I:A1-A2),
    prove_var1_abd_uv(B,UE,Lits1,(DisV+UnivV),FreeV,[],I:A2-A3).
prove_var1_abd_uv(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,I:A1-A2):-
    !,findall(([Var|UnivV]:F),(member(Var,FreeV),
    copy_term((X,Fml,FreeV),(Var,F,FreeV))),
    [(UnivV1:F1)|Forms]),
    append(UnExp,Forms,UnExp1),
    prove_var1_abd_uv(F1,UnExp1,Lits,DisV,FreeV,UnivV1,I:A1-A2).
prove_var1_abd_uv(Lit,_,[L|Lits],_,_,_,I:A-A) :-
    (Lit = -Neg; -Lit = Neg) ->
    (unify(Neg,L); prove_var1_abd_uv(Lit,[],Lits,_,_,_,ref:A-A)).
prove_var1_abd_uv(Lit,[(UnivV:Next)|UE],Lits,DisV,FreeV,_,
                  I:A1-A2) :-
    prove_var1_abd_uv(Next,UE,[Lit|Lits],DisV,FreeV,UnivV,
                      I:A1-A2).
prove_var1_abd_uv(L,_,Ls,_,_,_,abd:[L|Ls]|A)-A).

% prove_var1_abd_uv_oc(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV,
%                    ?I:?A1-?A2).
% Variante de prove_abd_oc/8 con lista de variables libres fija.
prove_var1_abd_uv_oc((A;B),UE,Lits,DisV,FreeV,UnivV,I:A1-A2) :- !,
    prove_var1_abd_uv_oc(A,[(UnivV:B)|UE],Lits,DisV,FreeV,UnivV,
                         I:A1-A2).
prove_var1_abd_uv_oc((A;B),UE,Lits,DisV,FreeV,UnivV,I:A1-A3) :- !,
    copy_term((Lits,DisV),(Lits1,DisV)),
    prove_var1_abd_uv_oc(A,UE,Lits,(DisV+UnivV),FreeV,[],I:A1-A2),
    prove_var1_abd_uv_oc(B,UE,Lits1,(DisV+UnivV),FreeV,[],I:A2-A3).
prove_var1_abd_uv_oc(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,
                     I:A1-A2) :- !,
    findall(([Var|UnivV]:F),(member(Var,FreeV),
    copy_term((X,Fml,FreeV),(Var,F,FreeV))),
    [(UnivV1:F1)|Forms]),
    append(UnExp,Forms,UnExp1),
    prove_var1_abd_uv_oc(F1,UnExp1,Lits,DisV,FreeV,UnivV1,
                         I:A1-A2).

```

```

prove_var1_abd_uv_oc(Lit,_,[L|Lits],_,_,_,I:A-A) :-
    (Lit = -Neg; -Lit = Neg ) ->
    (unify_with_occurs_check(Neg,L);
     prove_var1_abd_uv_oc(Lit,[],Lits,_,_,_,ref:A-A)).
prove_var1_abd_uv_oc(Lit,[(UnivV:Next)|UE],Lits,DisV,FreeV,_,
    I:A1-A2) :-
    prove_var1_abd_uv_oc(Next,UE,[Lit|Lits],DisV,FreeV,UnivV,
        I:A1-A2).
prove_var1_abd_uv_oc(L,_,Ls,_,_,_,abd:[[L|Ls]|A]-A).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Construcción de tableros                                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% tab_var1_uv(+Fml,+Lim,?Tab).
%     tiene éxito cuando Tab es un tablero de Fml construido con
%     prove_var1_abd_uv/5, empleando Lim variables libres.
tab_var1_uv(Fml,Lim,Tab):-
    length(L,Lim),
    prove_var1_abd_uv(Fml,[],[],[],L,[],abd:X-[]),!,X=Tab.

% tab_var1_uv_oc(+Fml,+Lim,?Tab).
%     tiene éxito cuando Tab es un tablero de Fml construido con
%     prove_var1_abd_uv_oc/5, empleando Lim variables libres.
tab_var1_uv_oc(Fml,Lim,Tab):-
    length(L,Lim),
    prove_var1_abd_uv_oc(Fml,[],[],[],L,[],abd:X-[]),!,X=Tab.

```


Apéndice G

Tableros como grafos, BDDs y compilación

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fichero: variaciones.pl
%% Variantes de leanTAP recogidas en: 'Implementing Semant Tabl'
%% de Possega y Schmitt
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(variaciones,[prueba_tgraph/2,prueba_comp/2,
                      prueba_bdd/2]).

:-use_module(nnf,[nnf/2]).
:- dynamic node/5.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados de alto nivel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tableros como grafos:
% prueba_tgraph(+Fml,+Limit)
%   tiene éxito si para la fórmula de primer orden Fml, después de
%   transformarla a forma normal negada skolemizada y convertirla
%   en un grafo-tablero, se encuentra una prueba que introduce a
%   lo sumo Limit variables libres en cada camino del grafo.
%   Si Limit es una variable, comienza una búsqueda en profundidad
%   iterativa.
prueba_tgraph(Fml,Limit):-
    nnf(Fml,NFml),
    tgraph(NFml,Graph/true),
    gprove(Graph,Limit).
gprove(Graph,Limit):-
    nonvar(Limit),!,
    gprove(Graph,[],[],[],Limit).
gprove(Graph,Result):-
    iterate(VarLim,0,gprove(Graph,[],[],[],VarLim),Result).
```

```

% Compilación de la búsqueda:
% prueba_comp(+Fml,+Limit)
%   tiene éxito si para la fórmula de primer orden Fml, después de
%   transformarla a forma normal negada skolemizada, y luego
%   convertirla en un grafo-tablero etiquetado y compilar la búsqueda
%   de la prueba, tiene éxito la ejecución del programa compilado
%   equivalente a la prueba en el grafo-tablero que permite a lo sumo
%   Limit variables libres en cada camino.
%   Si Limit es una variable, comienza una búsqueda en profundidad
%   iterativa.
prueba_comp(Fml,Limit):-
    retractall(node(_,_,_,_)),
    nnf(Fml,NFml),
    tgraph2(NFml,Graph),
    comp(Graph,[],[]),
    start_comp(Limit).
start_comp(Limit):-
    nonvar(Limit),!,
    start(Limit).
start_comp(Result):-
    iterate(VarLim,0,start(VarLim),Result).

% BDDs:
% prueba_bdd(+Fml,+Limit)
%   tiene éxito si para la fórmula de primer orden Fml, después de
%   transformarla a forma normal negada skolemizada, y convertirla
%   en un BDD, se encuentra una prueba que introduce a lo sumo
%   Limit variables libres en cada camino del BDD.
%   Si Limit es una variable, comienza una búsqueda en profundidad
%   iterativa.
prueba_bdd(Fml,Limit):-
    nnf(Fml,NFml),
    f2bdd(NFml,1,0,BDD),
    bdd_prove(BDD,Limit).
bdd_prove(BDD,Limit):-
    nonvar(Limit),!,
    bdd_prove(BDD,[],[],Limit).
bdd_prove(BDD,Result):-
    iterate(VarLim,0,bdd_prove(BDD,[],[],VarLim),Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Tableros como grafos                                                    %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% tgraph(+Fml,?Grafo)
%   tiene éxito si dada la Fml, una fórmula de primer orden en
%   NNF, Grafo es su grafo-tablero equivalente. Se emplean listas
%   de diferencia.
tgraph((A,B),GraphA/GraphEnd):-!,           % Conjunción
    tgraph(A,GraphA/GraphB),
    tgraph(B,GraphB/GraphEnd).
tgraph((A;B),(GraphA;GraphB)/GraphEnd):-!, % Disyunción

```

```

        tgraph(A,GraphA/GraphEnd),
        tgraph(B,GraphB/GraphEnd).
tgraph(all(X,F),(all(X,GraphF),TEnd)/TEnd):-!,      % Cuant. Univ.
        tgraph(F,GraphF/true).
tgraph(Lit,(Lit,End)/End).                          % Literales.

% Ejemplo:
% ?- tgraph((all(X,(p(X);t(X))), (r;s)),Gr/true).
% X = _G168
% Gr = all(_G168, (p(_G168), true;t(_G168), true)),
%      (r, true;s, true)

% gprove(+Grafo,+Gammas,+Lits,+FreeV,+VarLim)
%   tiene éxito si se encuentra una unificación que cierra todos
%   los caminos de Grafo con a lo sumo VarLim variables libres por
%   camino. Lits es la lista de literales encontrados. Gammas la
%   lista de subgrafos cuantificados universalmente por los que se
%   ha pasado.
%   FreeV es la lista de las variables libres introducidas.
gprove((A;B),Gammas,Lits,FreeV,VarLim):-!,          % Disy.
        gprove(A,Gammas,Lits,FreeV,VarLim),
        gprove(B,Gammas,Lits,FreeV,VarLim).
gprove((all(X,Gr),Rest),Gammas,Lits,FreeV,VarLim):-!, % Localiza
        gprove(Rest,[all(X,Gr)|Gammas],Lits,FreeV,VarLim). % univ.
gprove(true,[all(X,Gr)|Gammas],Lits,FreeV,VarLim):-!, % Usa cuant.
        \+ length(FreeV,VarLim),                          % universal.
        copy_term((X,Gr,FreeV),(X1,Gr1,FreeV)),
        append(Gammas,[all(X,Gr)],Gammas1),
        gprove(Gr1,Gammas1,Lits,[X1|FreeV],VarLim).
gprove((Lit,Rest),Gammas,Lits,FreeV,VarLim):-      % Cierre.
        (Lit = -Neg; -Lit = Neg) -> (member_unify(Neg,Lits);
        gprove(Rest,Gammas,[Lit|Lits],FreeV,VarLim)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Compilando la búsqueda de la prueba                                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% tgraph2(+Fml,?Grafo)
%   tiene éxito si Grafo es el grafo-tablero etiquetado equivalen-
%   te a Fml, una fórmula de primer orden en NNF. Llama a
%   tgraph2/3.
tgraph2(Formula,Graph):-
        tgraph2(Formula,IDs/[],Graph/(0:true)),
        instantiate(1,IDs).

% Ejemplo:
% ?- tgraph2((all(X,(p(X);t(X))), (r;s)),Gr).
% Gr = 1: (all(_G168, 2: (3: (p(_G168), 0:true);
%      4: (t(_G168), 0:true))), 5: (6: (r, 0:true);
%      7: (s, 0:true)))

% tgraph2(+Fml,?IDs,?Grafo)

```

```

%   tiene éxito si para la Fml, una fórmula de primer orden en
%   NNF, Grafo es su grafo-tablero etiquetado, e IDs la lista de
%   las etiquetas de cada nodo. Al igual que tgraph/2 emplea lis-
%   tas de diferencia. La diferencia entre la definición de ambos
%   predicados, como puede verse, está en el uso de las etiquetas.
tgraph2((A,B),IDs/IDSTail,GrA/GrEnd):-!,           % Conj.
    tgraph2(A,IDs/IDsB,GrA/GrB),
    tgraph2(B,IDsB/IDSTail,GrB/GrEnd).
tgraph2((A;B),[N|IDs]/IDSTail,(N:(GrA;GrB))/GrEnd):-!, % Disy.
    tgraph2(A,IDs/IDsB,GrA/GrEnd),
    tgraph2(B,IDsB/IDSTail,GrB/GrEnd).
tgraph2(all(X,F),[N|IDs]/IDST,                       % Cuant.
    (N:(all(X,GrF),GrEnd))/GrEnd):-!,               % univ.
    tgraph2(F,IDs/IDST,GrF/(0:true))).
tgraph2(Literal,[N|IDs]/IDs,(N:(Literal,End))/End). % Lits.

% comp(+Grafo,+BindIn,+BindOut)
%   compila la búsqueda de la prueba en el grafo-tablero etiqueta-
%   do Grafo. BindIn y BindOut se hacen al principio la lista
%   vacía, y representan, respectivamente, las ligaduras de las va-
%   riables a la entrada a a la salida de cada nodo. El resultado
%   de compila un grafo es un programa en que el predicado princi-
%   pal es:
%       node(+Id,+Binding,+Path,+MaxVars,+Gamma)
%   donde,
%   - Id es la etiqueta del nodo al que representa la cláusula en el
%   grafo-tablero etiquetado.
%   - Binding recoge las ligaduras de las variables al entrar en la
%   cláusula.
%   - Path es la lista de los literales que ya se han encontrado
%   mientras se recorre en grafo.
%   - MaxVars es el máximo número de variables libres que aún pueden
%   introducirse.
%   - Gammas es una lista que recoge las etiquetas correspondientes
%   a subgrafos universalmente cuantificados que están en el cami-
%   no recorrido.
%   Las cláusulas de comp/3 son:
%   Los nodos etiquetados con 0 no se compilan, ya que existe una
%   cláusula predefinida para tales nodos:
comp(0:true,_,_):-!.
%   Si un nodo ya se ha compilado no vuelve a compilarse:
comp((Id:_),_,_):-
    clause(node(Id,_,_,_,_),_),!.
%   Disyunciones:
comp(Id:((LeftId:Left);(RightId:Right)),BindIn,BindOut):-!,
    append(BindIn,BTail,BI),
    append(BindOut,BTail,BO),
    assert((node(Id,BI,P,MaxVars,Gamma):-
        node(LeftId,BO,P,MaxVars,Gamma),
        node(RightId,BO,P,MaxVars,Gamma))),
    comp(LeftId:Left,BindOut,BindOut),
    comp(RightId:Right,BindOut,BindOut).

```

```

%   Cuantificación universal:
comp(Id: (all(X, (ScId:Scope)), SuccId:Succ), BindIn, BindOut):-!,
    append(BindIn, [], ScBindIn),
    append(BindOut, [X], ScBindOut),
    assert((node(Id, Bind, P, MaxVars, Gamma):-
        node(SuccId, Bind, P, MaxVars, [ScId|Gamma]))),
    comp(ScId:Scope, ScBindIn, ScBindOut),
    comp(SuccId:Succ, ScBindIn, ScBindOut).

%   Literales:
comp(Id: (Lit, SuccId:Succ), BindIn, BindOut):-!,
    append(BindIn, BTail, BI),
    append(BindOut, BTail, BO),
    assert((node(Id, BI, Path, MaxVars, Gamma):-
        cierra(Lit, Path);
        node(SuccId, BO, [Lit|Path], MaxVars, Gamma))),
    comp(SuccId:Succ, BindOut, BindOut).

%   Cláusulas comunes a todos los programas compilados:
% cierra(+Lit, +Lits)
%   tiene éxito si es posible unificar alguno de los literales que
%   pertenecen a la lista Lits con el literal complementario de
%   Lit.
cierra(Lit, [L|Lits]):-
    (Lit = -Neg; -Lit = Neg)->
    (unify_with_occurs_check(Neg, L); cierra(Lit, Lits)).

% start(+N)
%   Lanza el programa compilando a realizar una búsqueda equiva-
%   lente a una prueba en el grafo-tablero compilado donde sólo se
%   permite un máximo de N introducciones de variables libres por
%   rama.
start(N) :- node(1, _, [], N, []), !.
% node(0, +Binds, +Path, +MaxVars, +Gammas)
%   Esta es la cláusula que representa los finales de rama en los
%   programas compilados. Binds es la lista de las ligaduras a las
%   que se ha sometido a las variables libres introducidas. Path
%   es el camino que se ha ido creando con los literales encontra-
%   dos. MaxVars es el número de variables libres que aún pueden
%   introducirse. Gammas es la lista que contiene las etiquetas
%   correspondientes a nodos del grafo que se corresponden con
%   subgrafos universalmente cuantifica-
%   dos anidados. Si llega a esta cláusula la prueba tiene éxito
%   si:
%   - Hay al menos un subgrafo universal en el camino recorrido, y
%   se puede introducir al menos una variable libre más.
%   - Tiene éxito la prueba tras visitar el primer subgrafo univer-
%   sal. Ahora se permite introducir una variable libre menos, y
%   además en la lista Gammas, la Id del subgrafo visitado se ha
%   pasado al final.
node(0, B, P, MaxVars, [Id|Gamma]):-
    MaxVars > 0, MaxVars1 is MaxVars - 1,
    append(Gamma, [Id], NewGamma),
    node(Id, B, P, MaxVars1, NewGamma).

```

```

/* Ejemplo:
?- tgraph2((all(X,(p(X);t(X))), (r;s)), Gr), comp(Gr, [], []).
X = _G168
Gr = 1: (all(_G168, 2: (3: (p(_G168), 0:true);
      4: (t(_G168), 0:true))), 5: (6: (r, 0:true);7: (s, 0:true)))
Yes
?- listing(node/5).
:- dynamic node/5.
node(0, A, B, C, [D|E]) :-
    C>0,
    F is C-1,
    append(E, [D], G),
    node(D, A, B, F, G).
node(1, A, B, C, D) :-
    node(5, A, B, C, [2|D]).
node(2, [A|B], C, D, E) :-
    node(3, [F|B], C, D, E),
    node(4, [F|B], C, D, E).
node(3, [A|B], C, D, E) :-
    (cierra(p(A), C);
     node(0, [A|B], [p(A)|C], D, E)).
node(4, [A|B], C, D, E) :-
    (cierra(t(A), C);
     node(0, [A|B], [t(A)|C], D, E)).
node(5, [A|B], C, D, E) :-
    node(6, [F|B], C, D, E),
    node(7, [F|B], C, D, E).
node(6, [A|B], C, D, E) :-
    (cierra(r, C);
     node(0, [A|B], [r|C], D, E)).
node(7, [A|B], C, D, E) :-
    (cierra(s, C);
     node(0, [A|B], [s|C], D, E)).
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Pruebas mediante BDDs %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% f2bdd(+Fml,+True,+False,-BDD)
% Dada Fml, una formula de primer orden en NNF, devuelve su
% BDD, de forma que True es lo que aparece que tiene True en
% las hojas donde Fml es verdadera y False donde es falsa.
f2bdd((A,B), True_B, False, BDD_A):-!,
    f2bdd(A, BDD_B, False, BDD_A),
    f2bdd(B, True_B, False, BDD_B).
f2bdd((A;B), True, False_B, BDD_A):-!,
    f2bdd(A, True, BDD_B, BDD_A),
    f2bdd(B, True, False_B, BDD_B).
f2bdd(all(X,Fml), True, False,
    (all(X, BDD_Fml) -> True; False)):-!,
    f2bdd(Fml, 1, 0, BDD_Fml).

```

```

f2bdd(Literal, True, False, BDD):-
    (Literal = -Lit) -> BDD = (Lit -> False; True);
    BDD = (Literal -> True; False).

% Ejemplo:
% ?- f2bdd((all(X, (p(X);t(X))), (r;s)), 1, 0, BDD).
% X = _G168
% BDD = all(_G168, (p(_G168)->1;t(_G168)->1;0))-> (r->1;s->1;0);0

% bdd_prove(+BDD,+Gammas,+Lits,+FreeV,+VarLim)
%   Dado un BDD, tiene éxito si logra cerrar todas sus ramas que
%   terminan en 1, introduciendo un máximo de VarLim variables
%   libres en cada una. Gammas es la lista de subgrafos universa-
%   les anidados en el BDD; Lits es la lista de los literales en-
%   contrados, y FreeV la lista que recoge las variables libres
%   introducidas.
% - Las ramas que acaban en 'false' son cerradas, por lo que la
%   búsqueda para con éxito el llegar a ellas.
bdd_prove(0,_,_,_):-!.
% - Localización de cuantificadores universales. No se usan en el
%   mismo momento en que se encuentran, sino que se guardan para
%   usarlos si no se cierra algún camino.
bdd_prove((all(X,BDDFml)->True;False), Gammas, Lits, FreeV, VarLim):-
    !, bdd_prove(True, [all(X,BDDFml)|Gammas], Lits, FreeV, VarLim),
    bdd_prove(False, Gammas, Lits, FreeV, VarLim).
% - Aplicación de universales. Si se llega al final de un camino
%   sin cerrarse, se acude a un subgrafo universal.
bdd_prove(1, [all(X,BDD_Fml)|Gammas], Lits, FreeV, VarLim):-!,
    \+ length(FreeV, VarLim),
    copy_term((X,BDD_Fml,FreeV), (X1,BDD_Fml1,FreeV)),
    append(Gammas, [all(X,BDD_Fml)], Gammas1),
    bdd_prove(BDD_Fml1, Gammas1, Lits, [X1|FreeV], VarLim).
% - Cierre de ramas. Al llegar a un literal A, tiene éxito si
%   cierra los dos caminos que surgen, bien porque está entre los
%   literales el complementario del literal que se añade, o porque
%   se cierra el camino tras continuarlo.
bdd_prove((A -> B; C), Gammas, Lits, FreeV, VarLim):-
    (member_unify(-A, Lits);
     bdd_prove(B, Gammas, [A|Lits], FreeV, VarLim)),
    (member_unify(A, Lits);
     bdd_prove(C, Gammas, [-A|Lits], FreeV, VarLim)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados comunes %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% iterate/4
%   La definición de interate/4, para realizar búsquedas en pro-
%   fundidad iterativa, es común con los demás ficheros.
iterate(Current, Current, Goal, Current) :- !nl,
%   write('Limit = '),
%   write(Current), nl,

```

```
Goal.
iterate(VarLim,Current,Goal,Result) :-
    Current1 is Current + 1,
    iterate(VarLim,Current1,Goal,Result).

% member_unify(+E,+L)
%   tiene éxito si E tiene una unificación correcta con algún
%   elemento de la lista L.
member_unify(X,[H|T]):-
    unify_with_occurs_check(X,H);
    member_unify(X,T).

% instantiate(+N,+L)
%   instancia una lista L, originariamente de variables, a una
%   lista de enteros comenzando por N.
instantiate(_,[]).
instantiate(N,[N|Tail]):-
    N1 is N+1,
    instantiate(N1,Tail).
```

Apéndice H

Adaptación proposicional

H.1. Transformación a NNF

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: prop_nnf.pl
%% Transformación a NNF de fórmulas proposicionales.
%% Fernando Soler Toscano - Adaptación de nnf.pl de leanTAP.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-module(prop_nnf,[prop_nnf/2]).

:-      op(400,fy,-),    % negación
        op(500,xfy,&),   % conjunción
        op(600,xfy,v),  % disyunción
        op(650,xfy,=>), % implicación
        op(700,xfy,<=>). % equivalencia

% En este fichero se hace una simplificación proposicional de la
% transformación a NNF de nnf.pl. Al no haber cuantificadores, de
% saparece el problema de la skolemización, por lo que no inter-
% vienen variables libres.

% prop_nnf(+Fml,?NNF)
%   tiene éxito si dada una fórmula proposicional Fml, su forma
%   normal negada --ahora equivalente-- es NNF.
prop_nnf(Fml,NNF) :- nnf(Fml,NNF,_).

% nnf(+Fml,?NNF,?Paths)
%   dada la fórmula proposicional Fml, tiene éxito si NNF es su
%   forma normal negada, y Paths es el número de caminos disyun-
%   tivos de la misma.
nnf(Fml,NNF,Paths) :-
    (Fml = -(-A)      -> Fml1 = A;
     Fml = -all(X,F)  -> Fml1 = ex(X,-F);
     Fml = -ex(X,F)   -> Fml1 = all(X,-F);
     Fml = -(A v B)   -> Fml1 = -A & -B;
     Fml = -(A & B)   -> Fml1 = -A v -B;
```

```

    Fml = (A => B)   -> Fml1 = -A v B;
    Fml = -(A => B)  -> Fml1 = A & -B;
    Fml = (A <=> B)  -> Fml1 = (A & B) v (-A & -B);
    Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B),!,
    nnf(Fml1,NNF,Paths).
nnf(A & B,NNF,Paths) :- !,
    nnf(A,NNF1,Paths1),
    nnf(B,NNF2,Paths2),
    Paths is Paths1 * Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2,NNF1);
     NNF = (NNF1,NNF2)).

nnf(A v B,NNF,Paths) :- !,
    nnf(A,NNF1,Paths1),
    nnf(B,NNF2,Paths2),
    Paths is Paths1 + Paths2,
    (Paths1 > Paths2 -> NNF = (NNF2;NNF1);
     NNF = (NNF1;NNF2)).

nnf(Lit,Lit,1).

```

H.2. Adaptación de las pruebas por tableros

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: prop_leantap.pl                                     %%
%% Versión de leantap para lógica proposicional              %%
%% FERNANDO SOLER TOSCANO                                     %%
%% Versión para SWI-Prolog                                     20-may-2003 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(prop_leantap,[prop_prove/1,prop_prove_abd/1]).
:- use_module(prop_nnf,[prop_nnf/2]).

:- op(400,fy,-),           % negación.
   op(500,xfy,&),          % conjunción.
   op(600,xfy,v),         % disjunción.
   op(650,xfy,=>),        % implicación.
   op(700,xfy,<=>).       % equivalencia.

% Este fichero implementa una adaptación eficiente de 'prove/2'
% (de 'leantap.pl') y 'prove_abd' ('leantab.pl') para lógica propo-
% sicional. Ya no son necesarios ni el límite que establecía la
% profundidad de los tableros, ni la lista de variables libres.
% Además, al no contener variables las fórmulas no es necesario
% usar un unificador correcto, sino que basta con el unificador
% común.

% Predicados de alto nivel:
% prop_prove(+Fml) y prop_prove_abd(+Fml) tienen éxito cuando Fml
% es una fórmula proposicional inconsistente en NNF.
prop_prove(Fml) :- prop_nnf(Fml,N),prove(N,[],[]).
prop_prove_abd(Fml):-prop_nnf(Fml,N),prove_abd(N,[],[],a-a).

```

```

% Las modificaciones más importantes que se realizan son:
% - Cierres de las ramas. Cuando hay dos literales complementarios
%   en una rama, ésta se cierra, de modo que no se haga nunca
%   backtracking para tratar de buscar otros cierres posibles. Si
%   bien esto tiene sentido en lógica de primer orden, pues cierres
%   diferentes pueden suponer unificaciones distintas, en lógica
%   proposicional pierde sentido. Por tanto, cuando una rama se
%   cierre, se introducirá un corte. De esta forma, al encontrarse
%   una rama completa abierta, la prueba falla (o se añade tal rama
%   al tablero, si es el caso).
% - Como el número máximo de literales diferentes que puede haber
%   por rama abierta es el número de literales de la fórmula, y no
%   tiene sentido introducir literales repetidos, antes de añadir un
%   literal, se comprobará no sólo si está ya su complementario --
%   para cerrar la rama--, sino si está ese propio literal. Si se
%   encuentra que el literal ya está en la rama, puede detenerse la
%   búsqueda de la contradicción pues es seguro que no se
%   encontrará. Por tanto, esto reduce el tamaño de las ramas --
%   que ahora tendrán menos literales-- y también el tiempo de
%   computación.

% El predicado busca(+L,+Lits,-R) juega un papel fundamental en
% estas versiones. Siendo L un literal y Lits una lista de literales,
% tiene éxito si Lits contiene o bien L, unificando R con 'i' de
% 'idéntico', o su complementario, en cuyo caso R será 'n' de
% 'negación'. En caso de que Lits no contenga ni L ni su complementario,
% falla. Veamos la definición, pues se lleva a cabo con menos esfuerzo
% del que leanTAP emplea para buscar simplemente si está el literal
% complementario:
busca(L,[L|_],i):-!.
busca(-L,[L|_],n):-!.
busca(L,[-L|_],n):-!.
busca(L,[_|R],X):-
    busca(L,R,X).

% prove(+Fml,+UnExp,+Lits)
% tiene éxito si se logra cerrar la rama del tablero en que se
% está tratando la fórmula proposicional Fml, siendo UnExp la lista
% de fórmulas por tratar y Lits los literales encontrados.
% El funcionamiento es:
% 1.- Para las conjunciones, se trabaja con el primer miembro,
%     guardando el segundo en UnExp.
% 2.- En las disyunciones, se abren dos ramas que deben cerrar.
% 3.- Al encontrar un literal Lit:
%     a. Si su complementario está en Lits, la rama se cierra.
%     b. Si Lit está en Lits, se pasa a trabajar con la siguiente
%        fórmula, sin añadir Lit a Lits. Si no hay ninguna fórmula
%        en UnExp, fracasa la prueba.
%     c. En otro caso, se añade Lit a Lits a se pasa a trabajar
%        con la siguiente fórmula. También ahora, si no hay

```



```

:- module(variaciones_propos, [prop_tgraph/1, prop_comp/1,
                               prop_bdd/1]).
:- use_module(prop_nnf, [prop_nnf/2]).
:- dynamic node/2.

% Al igual que el fichero 'prop_leantap.pl' adapta para lógica pro-
% posicional algunas de las variantes de leanTaP que usan árboles
% para representar los tableros, este fichero realiza la adaptación
% proposicional de los métodos de prueba que aparecen en el fichero
% 'variaciones.pl'.

% Las ideas que subyacen a las modificaciones que a continuación
% presentaremos son las mismas que se usaron para crear los demos-
% tradores de 'prop_leantap.pl'. También en este caso la definición
% de busca/3 es igual:
% busca(+L,+Lits,-R)
%     siendo L un literal y Lits una lista de literales, tiene éxi-
%     to si Lits contiene o bien L, unificando R con 'i' de 'idén-
%     tico', o su complementario, en cuyo caso R será 'n' de 'nega-
%     ción'. En caso de que Lits no contenga ni L ni su complemen-
%     tario, falla.
busca(L, [L|_], i) :- !.
busca(-L, [L|_], n) :- !.
busca(L, [-L|_], n) :- !.
busca(L, [_|R], X) :-
    busca(L, R, X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Métodos de prueba
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Tableros como grafos:
% prop_tgraph(+Fml)
%     tiene éxito si trar convertir la fórmula proposicional Fml en
%     si forma normal negada NFml, y a su convertir ésta en un gra-
%     fo, todos los caminos de tal grafo son inconsistentes.
prop_tgraph(Fml) :-
    prop_nnf(Fml, NFml),
    tgraph(NFml, Graph/true),
    gprove(Graph, []).

% Compilación de la búsqueda:
% prop_comp(+Fml)
%     tiene éxito si dada la fórmula proposicional Fml, tras pasarla
%     a NFml y de ahí al grafo-tablero etiquetado Graph, el programa
%     que resulta de compilar la búsqueda de la prueba, tiene éxito,
%     con lo que Fml es inconsistente.
prop_comp(Fml) :-
    retractall(node(_, _)),
    prop_nnf(Fml, NFml),
    tgraph2(NFml, Graph),
    comp(Graph),

```



```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% instantiate(+N,+L)
%   instancia una lista L, originariamente de variables, a una
%   lista de enteros que comienzan por N.
instantiate(_, []).
instantiate(N, [N|Tail]):-
    N1 is N+1,
    instantiate(N1,Tail).
instantiate(_, []).
instantiate(N, [N|Tail]):-
    N1 is N+1,
    instantiate(N1,Tail).

% tgraph2(+Fml,?Grafo)
%   dada la fórmula proposicional en NNF, tiene éxito si Grafo es
%   el grafo-tablero etiquetado que la representa. Con respecto a
%   la versión de primer orden, lo único que se he hecho es qui-
%   tar la cláusula que trataba los cuantificadores universales.
%   Procede llamando a tgraph2/3, que difiere en tgraph/2 en
%   que asigna a cada nodo una etiqueta. Cuando ha creado el grafo
%   instancia las etiquetas enumerándolas.
tgraph2(Formula,Graph):-
    tgraph2(Formula,IDs/[],Graph/(0:true)),
    instantiate(1,IDs).
% Conjunción: además de unir los grafos, une las listas de etique-
% tas.
tgraph2((A,B),IDs/IDsTail,GrA/GrEnd):-!,
    tgraph2(A,IDs/IDsB,GrA/GrB),
    tgraph2(B,IDsB/IDsTail,GrB/GrEnd).
% Disyunción: Asigna una etiqueta N al nodo disyuntivo, que luego
% añade a la lista de etiquetas, también formada por las etiquetas
% de los grafos formados a partir de A y a partir de B.
tgraph2((A;B),[N|IDs]/IDsTail,(N:(GrA;GrB))/GrEnd):-!,
    tgraph2(A,IDs/IDsB,GrA/GrEnd),
    tgraph2(B,IDsB/IDsTail,GrB/GrEnd).
% Literales: asigna una nueva etiqueta al literal. A continuación
% hay otro nodo que servirá para reconocer el final del grafo.
tgraph2(Literal,[N|IDs]/IDs,(N:(Literal,End))/End).

% start/0. Como el lógica proposicional no hay que limitar la pro-
% fundidad de las pruebas, start/0 no necesita ningún argumento
% para poner en marcha los programas compilados.
start :- node(1,[]),!.

% Además, node(+Id,+Lits) sólo tiene dos argumentos, el primero que
% es la etiqueta del nodo correspondiente en el grafo-tablero eti-
% quetado, y el segundo que contiene la lista de literales recogidos
% por el camino. Si se llega a cierto nodo, significa que pre-
% viamente no se ha cerrado el camino, por eso cuando se llega al
% final de un camino, lo que se corresponde con node(0,_), la bús-
% queda de prueba falla.

```

```

node(0,_):-!,fail.

% comp(+Grafo)
% Siendo Grafo un grafo-tablero etiquetado, comp/1 compila la búsqueda de la prueba, generando cláusulas de node/2.
% Finales de caminos: No hace falta compilarlos. Como hemos visto,
% la búsqueda de la prueba falla al llegar a los nodos 0.
comp(0:true):-!.
comp((Id:_)):-
    clause(node(Id,_),_),!.
% Disyunciones: En la siguiente cláusula, Id se corresponde con
% un nodo disyuntivo, que tiene a su izquierda el nodo LeftId y
% a la derecha RightId. Por tanto, se crea una cláusula que hace
% que cuando la búsqueda llegue a Id, se abran dos caminos, uno
% que continúa por LeftId, y otro por RightId. Además, deben
% compilarse las dos partes de la disyunción.
comp(Id:((LeftId:Left);(RightId:Right))):-!,
    assert((node(Id,P):-
        node(LeftId,P),
        node(RightId,P))),
    comp(LeftId:Left),
    comp(RightId:Right).
% Literales: Los nodos no disyuntivos que tampoco son finales de
% rama contienen un literal. Es aquí donde aplicamos la adaptación
% proposicional. Se crea una cláusula tal que al llegar al nodo
% Id con los literales Path, si Lit (el literal que está en el nodo
% Id) es el complementario de alguno de los literales de Path (R=n)
% entonces la búsqueda acaba con éxito. Si Lit está en Path, entonces se continúa buscando en SuccId, el siguiente nodo, con el
% mismo Path. Si falla busca(Lit,Path,R) entonces no está en Path
% no Lit ni su complementario, con lo que la búsqueda continúa por
% SuccId, pero esta vez incluyendo Lit en el Path.
comp(Id:(Lit,SuccId:Succ)):-!,
    assert((node(Id,Path):-
        (busca(Lit,Path,R) -> (R = n;node(SuccId,Path));
        node(SuccId,[Lit|Path])),!)),
    comp(SuccId:Succ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Demostración con BDDs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% f2bdd(+Fml,+True,+False,?BDD)
% Dada Fml, una fórmula proposicional en NNF, y siendo True y
% False los símbolos que habrá, respectivamente, en las hojas
% verdaderas y falsas, BDD es el diagrama binario de decisión
% equivalente a Fml. En esta caso la única modificación con respecto
% a la versión de primer orden ha sido quitar la cláusula
% que trataba la cuantificación universal.
f2bdd((A,B),True_B,False,BDD_A):-!,
    f2bdd(A,BDD_B,False,BDD_A),
    f2bdd(B,True_B,False,BDD_B).

```

```

f2bdd((A;B),True,False_B,BDD_A):-!,
    f2bdd(A,True,BDD_B,BDD_A),
    f2bdd(B,True,False_B,BDD_B).
f2bdd(Literal,True,False,BDD):-
    (Literal = -Lit) -> BDD = (Lit -> False; True);
    BDD = (Literal -> True; False).

% bdd_prove(+BDD,+Lits)
%   tiene éxito si todos los caminos que conducen a 1 en BDD son
%   cerrados, siendo Lits la lista de los literales encontrados.
% Nodos 0: Si la búsqueda llega a un nodo 0, entonces acaba con
% éxito, al corresponderse con una valoreación que no satisface
% la fórmula. Vease que no hay cláusulas para los nodos 1, ya que
% si la búsqueda llegase a tal nodo, debe fallar, pues no se ha
% cerrado previamente.
bdd_prove(0,_):-!.
% Para los demás casos, el BDD tiene la forma (A -> B; C). Lo
% primero que se hace es buscar A en Lits. Puede ocurrir:
% - Que el complementario de A esté en Lits (R=n). Entonces
% la parte positiva, B, es cerrada, por lo que sólo debe conti-
% nuarse por C, con la misma lista de literales Lits, ya que si
% el complementario de A está en Lits, no hace falta introducirlo
% de nuevo.
% - Que A esté en Lits. Entonces la parte C del BDD, a la que en
% otro caso debería añadirse -A, es cerrada. Entonces se sigue por
% la parte B, pero sin añadir A de nuevo.
% - Si no están en Lits ni A ni -A, entonces se continúa por B
% añadiendo A a Lits, y por C añadiendo -A a Lits.
bdd_prove((A -> B; C),Lits):- !,
    busca(A,Lits,R) -> ((R=n,bdd_prove(C,Lits));
        (R=i,bdd_prove(B,Lits)));
    (bdd_prove(B,[A|Lits]),bdd_prove(C,[-A|Lits])).

```


Apéndice I

Comparación con fórmulas de primer orden

I.1. Fórmulas empleadas

```
fml(pe124,6,-ex(X, (p(X) & r(X))) & -ex(X1, (s(X1) & q(X1))) &
    all(X4, (p(X4) => (q(X4) v r(X4)))) &
    (-ex(X2, (p(X2))) => ex(Y, (q(Y)))) &
    all(X3, ((q(X3) v r(X3)) => s(X3)))).
fml(pe125,3,-ex(X, (q(X) & p(X))) & ex(X1, (p(X1))) &
    all(X2, (f(X2) => (-g(X2) & r(X2)))) & all(X3, (p(X3)
    => (g(X3) & f(X3)))) & (all(X4, (p(X4) => q(X4)))
    v ex(Z, (p(Z) & r(Z)))).
fml(pe134,5,-((ex(X, all(Y, (p(X) <=> p(Y)))) <=> (ex(U, (q(U)))
    <=> all(W, (q(W))))) <=> (ex(X1, all(Y1, (q(X1) <=>
    q(Y1)))) <=> (ex(U1, (p(U1))) <=> all(W1, (p(W1)))))).
fml(pe136,6,-all(X, ex(Y, (h(X,Y)))) & all(X1, ex(Y2, (f(X1,Y2))))
    & all(X2, ex(Y1, (g(X2,Y1)))) & all(X3, all(Y3,
    ((f(X3,Y3) v g(X3,Y3)) => all(Z3, ((f(Y3,Z3) v g(Y3,Z3))
    => h(X3,Z3)))).
fml(pe137,7,-all(X, ex(Y, (r(X,Y)))) & all(Z, ex(W, all(X1,
    ex(Y1, (p(X1,Z) => ((p(Y1,W) & p(Y1,Z)) & (p(Y1,W) =>
    ex(U, (q(U,W))))) & all(X2, all(Z2, (-p(X2,Z2) =>
    ex(Y2, (q(Y2,Z2))))) & (ex(X3, ex(Y3, (q(X3,Y3))) =>
    all(Z3, (r(Z3,Z3)))).
fml(pe138,4,-(all(X, ((p(a) & (p(X) => ex(Y, (p(Y) & r(X,Y)))) =>
    ex(Z, ex(W, ((p(Z) & r(X,W)) & r(W,Z)))))
    <=> all(X1, (((-p(a)) v p(X1)) v
    ex(Z1, ex(W1, ((p(Z1) & r(X1,W1)) & r(W1,Z1))))
    & (((-p(a)) v (-ex(Y1, (p(Y1) & r(X1,Y1))))
    v ex(Z2, ex(W2, ((p(Z2) & r(X1,W2)) & r(W2,Z2)))))))).
fml(pe143,5,-all(X,all(Y, ((q(X,Y) => q(Y,X)) & (q(Y,X) =>
    q(X,Y)))) & all(X1,all(Y1,(q(X1,Y1) =>
    all(Z, ((f(Z,X1) => f(Z,Y1)) & (f(Z,Y1) =>
    f(Z,X1))))) & all(X2,all(Y2,(all(Z2, ((f(Z2,X2) =>
    f(Z2,Y2)) & (f(Z2,Y2) => f(Z2,X2))) => q(X2,Y2)))).
```

```
fml(pel145,5,-ex(X, (f(X) & -ex(Y, (g(Y) & h(X,Y)))) &
  all(X1, ((f(X1) & all(Y, ((g(Y) & h(X1,Y)) => j(X1,Y)))) =>
  all(Y1, ((g(Y1) & h(X1,Y1)) & k(Y1)))) &
  -ex(Y2, (l(Y2) & k(Y2))) &
  ex(X2, ((f(X2) & all(Y3, (h(X2,Y3) => l(Y3))))
  & all(Y11, ((g(Y11) & h(X2,Y11)) => j(X2,Y11)))))).
```

I.2. Fichero para realizar el test (SWI-Prolog)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fichero para comparar rendimiento entre demostradores.           %%
%%                                                                    %%
%%                                FERNANDO SOLER TOSCANO             %%
%% Versión para SWI-Prolog                                           14-oct-2003 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Carga de ficheros y predicados necesarios.                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:-module(test,[test/1,perfil/4,perfil/2,trata_experimento/2]).
:-ensure_loaded(principal).
:-ensure_loaded(dibujar_tablas).
:-use_module(experimentos,[experimento/2]).
:-use_module(library(time)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Declaración de métodos de prueba.                                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% método(?Nombre,?Fml,?Lim,?Procedimiento).
% así se declaran los métodos de prueba que se van a comparar, en
% el orden en que se quiere que se traten.
% - Nombre, es el nombre del método de prueba.
% - Procedimiento, es el objetivo que debe cumplirse para pro-
% bar una fórmula. De ahí, la fórmula debe unificar con Fml
% y el límite con Lim.
método('prove',Fml,Lim,(nnf(Fml,NFml),prove(NFml,Lim))).
método('prove_var1',Fml,Lim,(nnf(Fml,NFml),prove_var1(NFml,Lim))).
método('prove_uv',Fml,Lim,(nnf(Fml,NFml),prove_uv(NFml,Lim))).
método('prove_var1_uv',Fml,Lim,(nnf(Fml,NFml),
  prove_var1_uv(NFml,Lim))).
método('prove_oc',Fml,Lim,(nnf(Fml,NFml),prove_oc(NFml,Lim))).
método('prove_var1_oc',Fml,Lim,(nnf(Fml,NFml),
  prove_var1_oc(NFml,Lim))).
método('prove_uv_oc',Fml,Lim,(nnf(Fml,NFml),prove_uv_oc(NFml,Lim))).
método('prove_var1_uv_oc',Fml,Lim,(nnf(Fml,NFml),
  prove_var1_uv_oc(NFml,Lim))).
método('prove_abd',Fml,Lim,(nnf(Fml,NFml),prove_abd(NFml,Lim))).
método('prove_var1_abd',Fml,Lim,(nnf(Fml,NFml),
  prove_var1_abd(NFml,Lim))).
método('prove_abd_uv',Fml,Lim,(nnf(Fml,NFml),prove_abd_uv(NFml,Lim))).
método('prove_var1_abd_uv',Fml,Lim,(nnf(Fml,NFml),
```

```

                                prove_var1_abd_uv(NFml,Lim))).
método('prove_abd_oc',Fml,Lim,(nnf(Fml,NFml),prove_abd_oc(NFml,Lim))).
método('prove_var1_abd_oc',Fml,Lim,(nnf(Fml,NFml),
                                prove_var1_abd_oc(NFml,Lim))).
método('prove_abd_uv_oc',Fml,Lim,(nnf(Fml,NFml),
                                prove_abd_uv_oc(NFml,Lim))).
método('prove_var1_abd_uv_oc',Fml,Lim,(nnf(Fml,NFml),
                                prove_var1_abd_uv_oc(NFml,Lim))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados principales. %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% test(+Tpo).
%     realiza las estadísticas de todas las fórmulas-experimento,
%     con el tiempo máximo de Tpo segundos.
test(Tpo) :-
    findall(E,trata_experimento(E,Tpo),_).

% trata_experimento(+Nom,+Tpo).
%     dado Nom, nombre de una fórmula-experimento, y Tpo, el máximo
%     tiempo que damos a cada método para probar una fórmula, el re-
%     sultado es una tabla con las estadísticas de cada método para
%     probar la fórmula correspondiente a Nom.
trata_experimento(Nom,Tpo) :-
    experimento(Nom,F),nl,
    format('Experimento: ~w',Nom),nl,
    findall(Res,prueba_con_tiempo(Tpo,F,Res),Conj),
    hacer_tabla([[ 'MÉTODO', 'ERROR',
                  'INFERENCIAS', 'MILISEG', 'BYTES', 'LÍMITE' ]|Conj]).

% perfil(+Met,+Nom,+Estilo,+N)
%     el resultado es el perfil de la ejecución de la fórmula corres-
%     pondiente al experimento Nom con el método Met, según el Estilo
%     ('plain' o 'cummulative') y el máximo número de predicados N.
perfil(Met,Nom,Estilo,N):-
    experimento(Nom,Fml),
    método(Met,Fml,_Lim,Goal),
    profile(Goal,Estilo,N).

% perfil(+Met,+Nom)
%     es como perfil/4, pero por defecto el estilo es plain y sólo
%     muestra los 10 predicados más llamados.
perfil(Met,Nom):-
    experimento(Nom,Fml),
    método(Met,Fml,_Lim,Goal),
    profile(Goal,plain,10).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Construcción del test. %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% prueba_con_tiempo(+Tpo,+Fml,-Resultado).
%     tiene éxito cuando Resultado es la información sobre los datos

```

```

%      estadísticos de la prueba de Fml en el tiempo máximo de Tpo
%      segundos. Resultado = [Met,Err,Inf,Seg,Byt,Lim], siendo:
%      - Met, el método de prueba (por backtracking, va empleando
%      todos los métodos declarados como tales).
%      - Err, un indicador de error, caso de que se produzca. En
%      otro caso, Err = '-'.
%      - Inf, el número de inferencias necesarias en la prueba. En
%      caso de no probar, Inf = '-'.
%      - Seg, el tiempo necesario para la prueba, en milisegundos.
%      En caso de no probar, Seg = '-'.
%      - Byt, la memoria requerida, en bytes. Si no se prueba,
%      Byt = '-'.
%      - Lim, el límite al que hizo falta llegar. Si no se prueba,
%      Lim = '-'.
prueba_con_tiempo(Tpo,Fml,[Met,Err,Inf,Seg,Byt,Lim]):-
    método(Met,Fml,Lim,Goal),
    catch((call_with_time_limit(Tpo,estad(Goal,Inf,Seg,Byt)),Err='-'),
        Fallo,
        (mensaje(Fallo,Err),Inf='- ',Seg='- ',Byt='- ',Lim='- ')).

%mensaje(+Error,?Mensaje).
%      tiene éxito cuando Mensaje es el indicador del fallo codificado
%      por Error. Para fallos aún no clasificados se usa el propio
%      Error.
mensaje(error(resource_error(stack), local),'Pila local'):-!.
mensaje(error(resource_error(stack), global),'Pila global'):-!.
mensaje(time_limit_exceeded,'Tiempo'):-!.
mensaje('$aborted','Ej. abort.'):-.
mensaje(Error,Error).

% estad(+Goal,-Infer,-Tiempo,-Espacio).
%      tiene éxito siempre que el objetivo Goal se cumple. Entonces:
%      - Infer es el número de inferencias necesarias para probar
%      Goal.
%      - Tiempo, los milisegundos necesarios.
%      - Espacio, los bytes requeridos.
estad(Goal,Infer,Tiempo,Espacio):-
    statistics(runtime,[_,_]),
    statistics(global_stack,[Espacio0,_]),
    statistics(inferences,Infer0),
    Goal, !,
    statistics(inferences,Infer1),
    statistics(runtime,[_,Tiempo]),
    statistics(global_stack,[Espacio1,_]),
    Infer is Infer1-Infer0,
    Espacio is Espacio1-Espacio0.

```

I.3. Resultados en cada sistema

I.3.1. SWI-Prolog 5.2.13

Experimento: pel24

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	572327	229	5980	6
prove_var1	-	122041	50	7264	2
prove_uv	-	577002	300	9400	6
prove_var1_uv	-	122890	61	11428	2
prove_oc	-	225819	140	5980	6
prove_var1_oc	-	43410	19	7264	2
prove_uv_oc	-	230494	200	9400	6
prove_var1_uv_oc	-	44259	40	11428	2
prove_abd	-	572327	250	10088	6
prove_var1_abd	-	121975	50	11276	2
prove_abd_uv	-	577002	341	13508	6
prove_var1_abd_uv	-	122824	59	15440	2
prove_abd_oc	-	225819	170	10088	6
prove_var1_abd_oc	-	43410	31	11276	2
prove_abd_uv_oc	-	230494	239	13508	6
prove_var1_abd_uv_oc	-	44259	40	15440	2

Experimento: pel25

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	975	0	2192	3
prove_var1	-	899	0	2368	1
prove_uv	-	982	0	2912	3
prove_var1_uv	-	903	0	3232	1

prove_oc	-	632	0	2192	3
prove_var1_oc	-	598	0	2368	1
prove_uv_oc	-	639	0	2912	3
prove_var1_uv_oc	-	602	0	3232	1
prove_abd	-	975	0	3012	3
prove_var1_abd	-	899	0	3188	1
prove_abd_uv	-	982	0	3732	3
prove_var1_abd_uv	-	903	11	4052	1
prove_abd_oc	-	632	9	3012	3
prove_var1_abd_oc	-	598	0	3188	1
prove_abd_uv_oc	-	639	0	3732	3
prove_var1_abd_uv_oc	-	602	0	4052	1

Experimento: pel34

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	10486036	3539	36260	5
prove_var1	-	574278	221	46820	2
prove_uv	-	37869029	15389	50664	5
prove_var1_uv	-	1689459	811	62836	2
prove_oc	-	3353275	1720	36284	5
prove_var1_oc	-	185971	109	46844	2
prove_uv_oc	-	12567906	8980	50676	5
prove_var1_uv_oc	-	569576	541	62836	2
prove_abd	-	10486036	3940	55368	5
prove_var1_abd	-	574278	229	67632	2
prove_abd_uv	-	37869029	17201	67552	5
prove_var1_abd_uv	-	1689459	890	79496	2

prove_abd_oc	-	3353275	2159	55368	5
prove_var1_abd_oc	-	185971	131	67632	2
prove_abd_uv_oc	-	12567906	10900	67552	5
prove_var1_abd_uv_oc	-	569576	610	79496	2

Experimento: pel36

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	1317	0	3104	6
prove_var1	-	758	0	3124	1
prove_uv	-	1241	10	3744	6
prove_var1_uv	-	760	0	4004	1
prove_oc	-	744	0	3104	6
prove_var1_oc	-	485	0	3124	1
prove_uv_oc	-	734	0	3744	6
prove_var1_uv_oc	-	487	0	4004	1
prove_abd	-	1317	0	3684	6
prove_var1_abd	-	758	0	3680	1
prove_abd_uv	-	1241	0	4300	6
prove_var1_abd_uv	-	760	0	4560	1
prove_abd_oc	-	744	0	3684	6
prove_var1_abd_oc	-	485	0	3680	1
prove_abd_uv_oc	-	734	0	4300	6
prove_var1_abd_uv_oc	-	487	0	4560	1

Experimento: pel37

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	3831	9	15024	7
prove_var1	-	3356	0	14320	1

prove_uv	-	3742	11	22776	7
prove_var1_uv	-	3257	9	22976	1
prove_oc	-	1189	0	15024	7
prove_var1_oc	-	896	0	14320	1
prove_uv_oc	-	1206	10	22776	7
prove_var1_uv_oc	-	903	0	22976	1
prove_abd	-	3831	11	16396	7
prove_var1_abd	-	3356	0	15692	1
prove_abd_uv	-	3742	9	24148	7
prove_var1_abd_uv	-	3257	0	24348	1
prove_abd_oc	-	1189	10	16396	7
prove_var1_abd_oc	-	896	0	15692	1
prove_abd_uv_oc	-	1206	0	24148	7
prove_var1_abd_uv_oc	-	903	11	24348	1

Experimento: pel38

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	97518	41	34384	4
prove_var1	-	65062	30	29208	1
prove_uv	-	97691	69	95260	4
prove_var1_uv	-	66197	41	79796	1
prove_oc	-	11069	19	34384	4
prove_var1_oc	-	7817	11	29208	1
prove_uv_oc	-	11242	50	95260	4
prove_var1_uv_oc	-	7927	19	79796	1
prove_abd	-	97518	50	40892	4
prove_var1_abd	-	65062	21	35716	1

prove_abd_uv	-	97691	69	101768	4
prove_var1_abd_uv	-	66197	41	86304	1
prove_abd_oc	-	11069	19	40892	4
prove_var1_abd_oc	-	7817	11	35716	1
prove_abd_uv_oc	-	11242	50	101768	4
prove_var1_abd_uv_oc	-	7927	30	86304	1

Experimento: pel43

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	44726	20	27416	5
prove_var1	-	39594	11	17848	1
prove_uv	-	44756	19	110148	5
prove_var1_uv	-	39612	21	94496	1
prove_oc	-	1395	9	27416	5
prove_var1_oc	-	1149	0	17848	1
prove_uv_oc	-	1425	11	110148	5
prove_var1_uv_oc	-	1167	9	94496	1
prove_abd	-	44726	21	29820	5
prove_var1_abd	-	39594	9	20252	1
prove_abd_uv	-	44756	21	112552	5
prove_var1_abd_uv	-	39612	21	96900	1
prove_abd_oc	-	1395	0	29820	5
prove_var1_abd_oc	-	1149	0	20252	1
prove_abd_uv_oc	-	1425	9	112552	5
prove_var1_abd_uv_oc	-	1167	11	96900	1

Experimento: pel45

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove	-	51474	20	12160	5
prove_var1	-	8767	10	9216	1
prove_uv	-	51572	31	36240	5
prove_var1_uv	-	8783	0	33428	1
prove_oc	-	7830	10	12160	5
prove_var1_oc	-	1622	0	9216	1
prove_uv_oc	-	7928	20	36240	5
prove_var1_uv_oc	-	1638	10	33428	1
prove_abd	-	51474	20	15908	5
prove_var1_abd	-	8767	0	12964	1
prove_abd_uv	-	51572	30	39988	5
prove_var1_abd_uv	-	8783	9	37176	1
prove_abd_oc	-	7830	10	15908	5
prove_var1_abd_oc	-	1622	0	12964	1
prove_abd_uv_oc	-	7928	20	39988	5
prove_var1_abd_uv_oc	-	1638	11	37176	1

I.3.2. SWI-Prolog 5.3.10

Experimento: pel24

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	572327	199	5980	6
prove_var1	-	122041	41	7264	2
prove_uv	-	577002	219	9400	6
prove_var1_uv	-	122890	51	11428	2
prove_oc	-	225819	90	5980	6
prove_var1_oc	-	43410	20	7264	2

prove_uv_oc	-	230494	120	9400	6
prove_var1_uv_oc	-	44259	30	11428	2
prove_abd	-	572327	219	10088	6
prove_var1_abd	-	121975	50	11276	2
prove_abd_uv	-	577002	240	13508	6
prove_var1_abd_uv	-	122824	50	15440	2
prove_abd_oc	-	225819	131	10088	6
prove_var1_abd_oc	-	43410	19	11276	2
prove_abd_uv_oc	-	230494	151	13508	6
prove_var1_abd_uv_oc	-	44259	30	15440	2

Experimento: pel25

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	975	0	2164	3
prove_var1	-	899	0	2340	1
prove_uv	-	982	0	2884	3
prove_var1_uv	-	903	0	3204	1
prove_oc	-	632	0	2164	3
prove_var1_oc	-	598	0	2340	1
prove_uv_oc	-	639	0	2884	3
prove_var1_uv_oc	-	602	0	3204	1
prove_abd	-	975	0	2984	3
prove_var1_abd	-	899	0	3160	1
prove_abd_uv	-	982	10	3704	3
prove_var1_abd_uv	-	903	0	4024	1
prove_abd_oc	-	632	0	2984	3
prove_var1_abd_oc	-	598	0	3160	1

prove_abd_uv_oc	-	639	0	3704	3
prove_var1_abd_uv_oc	-	602	0	4024	1
Experimento: pel34					
MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	10486036	3249	34456	5
prove_var1	-	574278	200	46820	2
prove_uv	-	37869029	12321	48948	5
prove_var1_uv	-	1689459	640	62836	2
prove_oc	-	3353275	1329	34480	5
prove_var1_oc	-	185971	90	46844	2
prove_uv_oc	-	12567906	5630	48960	5
prove_var1_uv_oc	-	569576	320	62836	2
prove_abd	-	10486036	3590	53564	5
prove_var1_abd	-	574278	180	67632	2
prove_abd_uv	-	37869029	14319	65836	5
prove_var1_abd_uv	-	1689459	691	79496	2
prove_abd_oc	-	3353275	1750	53564	5
prove_var1_abd_oc	-	185971	110	67632	2
prove_abd_uv_oc	-	12567906	7379	65836	5
prove_var1_abd_uv_oc	-	569576	390	79496	2
Experimento: pel36					
MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	1317	0	3080	6
prove_var1	-	758	0	3124	1
prove_uv	-	1241	0	3720	6
prove_var1_uv	-	760	0	4004	1

prove_oc	-	744	0	3080	6
prove_var1_oc	-	485	0	3124	1
prove_uv_oc	-	734	0	3720	6
prove_var1_uv_oc	-	487	0	4004	1
prove_abd	-	1317	0	3660	6
prove_var1_abd	-	758	0	3680	1
prove_abd_uv	-	1241	0	4276	6
prove_var1_abd_uv	-	760	0	4560	1
prove_abd_oc	-	744	0	3660	6
prove_var1_abd_oc	-	485	0	3680	1
prove_abd_uv_oc	-	734	0	4276	6
prove_var1_abd_uv_oc	-	487	0	4560	1

Experimento: pel37

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	3831	0	9696	7
prove_var1	-	3356	0	12064	1
prove_uv	-	3742	0	14088	7
prove_var1_uv	-	3257	10	20720	1
prove_oc	-	1189	0	9696	7
prove_var1_oc	-	896	0	12064	1
prove_uv_oc	-	1206	0	14088	7
prove_var1_uv_oc	-	903	9	20720	1
prove_abd	-	3831	0	11068	7
prove_var1_abd	-	3356	0	13436	1
prove_abd_uv	-	3742	0	15460	7
prove_var1_abd_uv	-	3257	0	22092	1

prove_abd_oc	-	1189	0	11068	7
prove_var1_abd_oc	-	896	0	13436	1
prove_abd_uv_oc	-	1206	0	15460	7
prove_var1_abd_uv_oc	-	903	11	22092	1

Experimento: pel38

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	97518	40	23164	4
prove_var1	-	65062	19	26220	1
prove_uv	-	97691	30	55604	4
prove_var1_uv	-	66197	31	58940	1
prove_oc	-	11069	0	23332	4
prove_var1_oc	-	7817	9	26220	1
prove_uv_oc	-	11242	10	56612	4
prove_var1_uv_oc	-	7927	0	59108	1
prove_abd	-	97518	30	29672	4
prove_var1_abd	-	65062	30	32728	1
prove_abd_uv	-	97691	31	62112	4
prove_var1_abd_uv	-	66197	30	65448	1
prove_abd_oc	-	11069	0	29840	4
prove_var1_abd_oc	-	7817	9	32728	1
prove_abd_uv_oc	-	11242	10	63120	4
prove_var1_abd_uv_oc	-	7927	11	65616	1

Experimento: pel43

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	44726	19	12176	5
prove_var1	-	39594	11	14800	1

prove_uv	-	44756	9	33932	5
prove_var1_uv	-	39612	10	35032	1
prove_oc	-	1395	0	12176	5
prove_var1_oc	-	1149	0	14800	1
prove_uv_oc	-	1425	0	33932	5
prove_var1_uv_oc	-	1167	0	35032	1
prove_abd	-	44726	10	14580	5
prove_var1_abd	-	39594	9	17204	1
prove_abd_uv	-	44756	11	36336	5
prove_var1_abd_uv	-	39612	10	37436	1
prove_abd_oc	-	1395	0	14580	5
prove_var1_abd_oc	-	1149	9	17204	1
prove_abd_uv_oc	-	1425	0	36336	5
prove_var1_abd_uv_oc	-	1167	0	37436	1

Experimento: pel45

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	51474	21	9632	5
prove_var1	-	8767	0	8624	1
prove_uv	-	51572	19	20504	5
prove_var1_uv	-	8783	0	19532	1
prove_oc	-	7830	11	9632	5
prove_var1_oc	-	1622	0	8624	1
prove_uv_oc	-	7928	10	20504	5
prove_var1_uv_oc	-	1638	0	19532	1
prove_abd	-	51474	9	13380	5
prove_var1_abd	-	8767	11	12372	1

prove_abd_uv	-	51572	10	24252	5
prove_var1_abd_uv	-	8783	9	23280	1
prove_abd_oc	-	7830	0	13380	5
prove_var1_abd_oc	-	1622	0	12372	1
prove_abd_uv_oc	-	7928	10	24252	5
prove_var1_abd_uv_oc	-	1638	0	23280	1

I.3.3. Yap Prolog 4.4.4

Experimento: pel24

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	70	7328	6
prove_var1	-	no disp.	10	7360	2
prove_uv	-	no disp.	80	15700	6
prove_var1_uv	-	no disp.	20	16324	2
prove_oc	-	no disp.	70	10924	6
prove_var1_oc	-	no disp.	10	10924	2
prove_uv_oc	-	no disp.	80	19296	6
prove_var1_uv_oc	-	no disp.	20	19888	2
prove_abd	-	no disp.	70	11436	6
prove_var1_abd	-	no disp.	20	11372	2
prove_abd_uv	-	no disp.	90	19808	6
prove_var1_abd_uv	-	no disp.	20	20336	2
prove_abd_oc	-	no disp.	70	15032	6
prove_var1_abd_oc	-	no disp.	20	14936	2
prove_abd_uv_oc	-	no disp.	90	23404	6
prove_var1_abd_uv_oc	-	no disp.	20	23900	2

Experimento: pel25

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2112	3
prove_var1	-	no disp.	0	2208	1
prove_uv	-	no disp.	0	2920	3
prove_var1_uv	-	no disp.	0	3128	1
prove_oc	-	no disp.	0	2684	3
prove_var1_oc	-	no disp.	0	2780	1
prove_uv_oc	-	no disp.	0	3492	3
prove_var1_uv_oc	-	no disp.	0	3700	1
prove_abd	-	no disp.	0	2932	3
prove_var1_abd	-	no disp.	0	3028	1
prove_abd_uv	-	no disp.	0	3740	3
prove_var1_abd_uv	-	no disp.	0	3948	1
prove_abd_oc	-	no disp.	0	3504	3
prove_var1_abd_oc	-	no disp.	0	3600	1
prove_abd_uv_oc	-	no disp.	0	4312	3
prove_var1_abd_uv_oc	-	no disp.	0	4520	1

Experimento: pel34

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	970	38708	5
prove_var1	-	no disp.	60	42168	2
prove_uv	-	no disp.	3980	63784	5
prove_var1_uv	-	no disp.	230	69204	2
prove_oc	-	no disp.	990	50876	5
prove_var1_oc	-	no disp.	60	55988	2

prove_uv_oc	-	no disp.	4330	75012	5
prove_var1_uv_oc	-	no disp.	210	80416	2
prove_abd	-	no disp.	1110	57792	5
prove_var1_abd	-	no disp.	70	62956	2
prove_abd_uv	-	no disp.	4610	80660	5
prove_var1_abd_uv	-	no disp.	250	85864	2
prove_abd_oc	-	no disp.	1170	69960	5
prove_var1_abd_oc	-	no disp.	60	76776	2
prove_abd_uv_oc	-	no disp.	4870	91888	5
prove_var1_abd_uv_oc	-	no disp.	260	97076	2

Experimento: pel36

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2888	6
prove_var1	-	no disp.	0	2896	1
prove_uv	-	no disp.	0	3508	6
prove_var1_uv	-	no disp.	0	3700	1
prove_oc	-	no disp.	0	3488	6
prove_var1_oc	-	no disp.	0	3496	1
prove_uv_oc	-	no disp.	0	4108	6
prove_var1_uv_oc	-	no disp.	0	4300	1
prove_abd	-	no disp.	0	3468	6
prove_var1_abd	-	no disp.	0	3452	1
prove_abd_uv	-	no disp.	0	4064	6
prove_var1_abd_uv	-	no disp.	0	4256	1
prove_abd_oc	-	no disp.	0	4068	6
prove_var1_abd_oc	-	no disp.	0	4052	1

prove_abd_uv_oc	-	no disp.	0	4664	6
prove_var1_abd_uv_oc	-	no disp.	0	4856	1

Experimento: pel37

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	16292	7
prove_var1	-	no disp.	0	15632	1
prove_uv	-	no disp.	0	23768	7
prove_var1_uv	-	no disp.	0	23860	1
prove_oc	-	no disp.	0	19448	7
prove_var1_oc	-	no disp.	0	18788	1
prove_uv_oc	-	no disp.	0	27004	7
prove_var1_uv_oc	-	no disp.	10	27096	1
prove_abd	-	no disp.	0	17664	7
prove_var1_abd	-	no disp.	0	17004	1
prove_abd_uv	-	no disp.	0	25140	7
prove_var1_abd_uv	-	no disp.	0	25232	1
prove_abd_oc	-	no disp.	0	20820	7
prove_var1_abd_oc	-	no disp.	0	20160	1
prove_abd_uv_oc	-	no disp.	0	28376	7
prove_var1_abd_uv_oc	-	no disp.	0	28468	1

Experimento: pel38

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	10	56572	4
prove_var1	-	no disp.	10	46808	1
prove_uv	-	no disp.	10	125048	4
prove_var1_uv	-	no disp.	10	103580	1

prove_oc	-	no disp.	10	86532	4
prove_var1_oc	-	no disp.	10	76744	1
prove_uv_oc	-	no disp.	10	155008	4
prove_var1_uv_oc	-	no disp.	10	133556	1
prove_abd	-	no disp.	10	63080	4
prove_var1_abd	-	no disp.	0	53316	1
prove_abd_uv	-	no disp.	20	131556	4
prove_var1_abd_uv	-	no disp.	0	110088	1
prove_abd_oc	-	no disp.	10	93040	4
prove_var1_abd_oc	-	no disp.	10	83252	1
prove_abd_uv_oc	-	no disp.	10	161516	4
prove_var1_abd_uv_oc	-	no disp.	0	140064	1

Experimento: pel43

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	65716	5
prove_var1	-	no disp.	0	55920	1
prove_uv	-	no disp.	10	148316	5
prove_var1_uv	-	no disp.	10	132284	1
prove_oc	-	no disp.	0	112484	5
prove_var1_oc	-	no disp.	0	102688	1
prove_uv_oc	-	no disp.	0	195084	5
prove_var1_uv_oc	-	no disp.	10	179052	1
prove_abd	-	no disp.	0	68120	5
prove_var1_abd	-	no disp.	0	58324	1
prove_abd_uv	-	no disp.	10	150720	5
prove_var1_abd_uv	-	no disp.	0	134688	1

prove_abd_oc	-	no disp.	0	114888	5
prove_var1_abd_oc	-	no disp.	10	105092	1
prove_abd_uv_oc	-	no disp.	0	197488	5
prove_var1_abd_uv_oc	-	no disp.	0	181456	1

Experimento: pel45

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	17704	5
prove_var1	-	no disp.	0	13916	1
prove_uv	-	no disp.	10	43072	5
prove_var1_uv	-	no disp.	0	39272	1
prove_oc	-	no disp.	0	27176	5
prove_var1_oc	-	no disp.	20	23472	1
prove_uv_oc	-	no disp.	0	52544	5
prove_var1_uv_oc	-	no disp.	0	48828	1
prove_abd	-	no disp.	10	21452	5
prove_var1_abd	-	no disp.	0	17664	1
prove_abd_uv	-	no disp.	10	46820	5
prove_var1_abd_uv	-	no disp.	0	43020	1
prove_abd_oc	-	no disp.	0	30924	5
prove_var1_abd_oc	-	no disp.	0	27220	1
prove_abd_uv_oc	-	no disp.	10	56292	5
prove_var1_abd_uv_oc	-	no disp.	0	52576	1

I.3.4. Yap Prolog 4.5.2

Experimento: pel24

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove	-	no disp.	70	7268	6
prove_var1	-	no disp.	30	7512	2
prove_uv	-	no disp.	80	15640	6
prove_var1_uv	-	no disp.	20	16476	2
prove_oc	-	no disp.	60	10864	6
prove_var1_oc	-	no disp.	20	11076	2
prove_uv_oc	-	no disp.	70	19236	6
prove_var1_uv_oc	-	no disp.	20	20040	2
prove_abd	-	no disp.	90	11376	6
prove_var1_abd	-	no disp.	20	11524	2
prove_abd_uv	-	no disp.	90	19748	6
prove_var1_abd_uv	-	no disp.	20	20488	2
prove_abd_oc	-	no disp.	80	14972	6
prove_var1_abd_oc	-	no disp.	20	15088	2
prove_abd_uv_oc	-	no disp.	90	23344	6
prove_var1_abd_uv_oc	-	no disp.	20	24052	2

Experimento: pel25

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2052	3
prove_var1	-	no disp.	0	2200	1
prove_uv	-	no disp.	0	2860	3
prove_var1_uv	-	no disp.	0	3120	1
prove_oc	-	no disp.	0	2624	3
prove_var1_oc	-	no disp.	0	2772	1
prove_uv_oc	-	no disp.	0	3432	3
prove_var1_uv_oc	-	no disp.	0	3692	1

prove_abd	-	no disp.	0	2872	3
prove_var1_abd	-	no disp.	0	3020	1
prove_abd_uv	-	no disp.	0	3680	3
prove_var1_abd_uv	-	no disp.	0	3940	1
prove_abd_oc	-	no disp.	0	3444	3
prove_var1_abd_oc	-	no disp.	0	3592	1
prove_abd_uv_oc	-	no disp.	0	4252	3
prove_var1_abd_uv_oc	-	no disp.	0	4512	1

Experimento: pel34

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	1080	38648	5
prove_var1	-	no disp.	80	43424	2
prove_uv	-	no disp.	4420	63724	5
prove_var1_uv	-	no disp.	260	70236	2
prove_oc	-	no disp.	940	50816	5
prove_var1_oc	-	no disp.	60	57244	2
prove_uv_oc	-	no disp.	3930	74952	5
prove_var1_uv_oc	-	no disp.	220	81448	2
prove_abd	-	no disp.	1250	57732	5
prove_var1_abd	-	no disp.	90	64212	2
prove_abd_uv	-	no disp.	5040	80600	5
prove_var1_abd_uv	-	no disp.	270	86896	2
prove_abd_oc	-	no disp.	1130	69900	5
prove_var1_abd_oc	-	no disp.	70	78032	2
prove_abd_uv_oc	-	no disp.	4710	91828	5
prove_var1_abd_uv_oc	-	no disp.	260	98108	2

Experimento: pel36

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2828	6
prove_var1	-	no disp.	0	2920	1
prove_uv	-	no disp.	0	3448	6
prove_var1_uv	-	no disp.	0	3724	1
prove_oc	-	no disp.	0	3428	6
prove_var1_oc	-	no disp.	0	3520	1
prove_uv_oc	-	no disp.	0	4048	6
prove_var1_uv_oc	-	no disp.	0	4324	1
prove_abd	-	no disp.	0	3408	6
prove_var1_abd	-	no disp.	0	3476	1
prove_abd_uv	-	no disp.	0	4004	6
prove_var1_abd_uv	-	no disp.	0	4280	1
prove_abd_oc	-	no disp.	0	4008	6
prove_var1_abd_oc	-	no disp.	0	4076	1
prove_abd_uv_oc	-	no disp.	0	4604	6
prove_var1_abd_uv_oc	-	no disp.	0	4880	1

Experimento: pel37

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	16232	7
prove_var1	-	no disp.	0	15816	1
prove_uv	-	no disp.	0	23708	7
prove_var1_uv	-	no disp.	0	24044	1
prove_oc	-	no disp.	0	19388	7
prove_var1_oc	-	no disp.	0	18972	1

prove_uv_oc	-	no disp.	0	26944	7
prove_var1_uv_oc	-	no disp.	0	27280	1
prove_abd	-	no disp.	0	17604	7
prove_var1_abd	-	no disp.	0	17188	1
prove_abd_uv	-	no disp.	0	25080	7
prove_var1_abd_uv	-	no disp.	0	25416	1
prove_abd_oc	-	no disp.	0	20760	7
prove_var1_abd_oc	-	no disp.	0	20344	1
prove_abd_uv_oc	-	no disp.	10	28316	7
prove_var1_abd_uv_oc	-	no disp.	0	28652	1

Experimento: pel38

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	10	56512	4
prove_var1	-	no disp.	0	47040	1
prove_uv	-	no disp.	10	124988	4
prove_var1_uv	-	no disp.	10	103812	1
prove_oc	-	no disp.	10	86472	4
prove_var1_oc	-	no disp.	0	76976	1
prove_uv_oc	-	no disp.	10	154948	4
prove_var1_uv_oc	-	no disp.	10	133788	1
prove_abd	-	no disp.	10	63020	4
prove_var1_abd	-	no disp.	10	53548	1
prove_abd_uv	-	no disp.	10	131496	4
prove_var1_abd_uv	-	no disp.	0	110320	1
prove_abd_oc	-	no disp.	10	92980	4
prove_var1_abd_oc	-	no disp.	10	83484	1

prove_abd_uv_oc	-	no disp.	10	161456	4
prove_var1_abd_uv_oc	-	no disp.	10	140296	1

Experimento: pel43

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	10	65656	5
prove_var1	-	no disp.	0	56104	1
prove_uv	-	no disp.	10	148256	5
prove_var1_uv	-	no disp.	0	132468	1
prove_oc	-	no disp.	0	112424	5
prove_var1_oc	-	no disp.	0	102872	1
prove_uv_oc	-	no disp.	10	195024	5
prove_var1_uv_oc	-	no disp.	0	179236	1
prove_abd	-	no disp.	0	68060	5
prove_var1_abd	-	no disp.	0	58508	1
prove_abd_uv	-	no disp.	10	150660	5
prove_var1_abd_uv	-	no disp.	0	134872	1
prove_abd_oc	-	no disp.	0	114828	5
prove_var1_abd_oc	-	no disp.	10	105276	1
prove_abd_uv_oc	-	no disp.	0	197428	5
prove_var1_abd_uv_oc	-	no disp.	0	181640	1

Experimento: pel45

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	17644	5
prove_var1	-	no disp.	0	14084	1
prove_uv	-	no disp.	10	43012	5
prove_var1_uv	-	no disp.	0	39440	1

prove_oc	-	no disp.	0	27116	5
prove_var1_oc	-	no disp.	0	23640	1
prove_uv_oc	-	no disp.	10	52484	5
prove_var1_uv_oc	-	no disp.	0	48996	1
prove_abd	-	no disp.	10	21392	5
prove_var1_abd	-	no disp.	0	17832	1
prove_abd_uv	-	no disp.	0	46760	5
prove_var1_abd_uv	-	no disp.	0	43188	1
prove_abd_oc	-	no disp.	10	30864	5
prove_var1_abd_oc	-	no disp.	0	27388	1
prove_abd_uv_oc	-	no disp.	10	56232	5
prove_var1_abd_uv_oc	-	no disp.	0	52744	1

I.3.5. GNU Prolog 1.2.16

Experimento: pel24

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	240	6416	6
prove_var1	-	no disp.	50	5580	2
prove_uv	-	no disp.	250	14916	6
prove_var1_uv	-	no disp.	50	14728	2
prove_oc	-	no disp.	130	6272	6
prove_var1_oc	-	no disp.	30	5452	2
prove_uv_oc	-	no disp.	150	14772	6
prove_var1_uv_oc	-	no disp.	30	14600	2
prove_abd	-	no disp.	260	10524	6
prove_var1_abd	-	no disp.	50	9592	2

prove_abd_uv	-	no disp.	290	19024	6
prove_var1_abd_uv	-	no disp.	60	18740	2
prove_abd_oc	-	no disp.	160	10380	6
prove_var1_abd_oc	-	no disp.	30	9464	2
prove_abd_uv_oc	-	no disp.	190	18880	6
prove_var1_abd_uv_oc	-	no disp.	30	18612	2

Experimento: pel25

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2056	3
prove_var1	-	no disp.	0	1904	1
prove_uv	-	no disp.	0	2880	3
prove_var1_uv	-	no disp.	0	2856	1
prove_oc	-	no disp.	0	2024	3
prove_var1_oc	-	no disp.	0	1872	1
prove_uv_oc	-	no disp.	0	2848	3
prove_var1_uv_oc	-	no disp.	0	2824	1
prove_abd	-	no disp.	0	2876	3
prove_var1_abd	-	no disp.	0	2724	1
prove_abd_uv	-	no disp.	0	3700	3
prove_var1_abd_uv	-	no disp.	0	3676	1
prove_abd_oc	-	no disp.	0	2844	3
prove_var1_abd_oc	-	no disp.	0	2692	1
prove_abd_uv_oc	-	no disp.	0	3668	3
prove_var1_abd_uv_oc	-	no disp.	0	3644	1

Experimento: pel34

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove	-	no disp.	3840	38140	5
prove_var1	-	no disp.	200	37088	2
prove_uv	-	no disp.	14660	63396	5
prove_var1_uv	-	no disp.	640	65548	2
prove_oc	-	no disp.	1890	37088	5
prove_var1_oc	-	no disp.	100	35848	2
prove_uv_oc	-	no disp.	7870	62464	5
prove_var1_uv_oc	-	no disp.	330	64624	2
prove_abd	-	no disp.	4260	57224	5
prove_var1_abd	-	no disp.	220	57876	2
prove_abd_uv	-	no disp.	16230	80272	5
prove_var1_abd_uv	-	no disp.	700	82208	2
prove_abd_oc	-	no disp.	2300	56172	5
prove_var1_abd_oc	-	no disp.	120	56636	2
prove_abd_uv_oc	-	no disp.	9580	79340	5
prove_var1_abd_uv_oc	-	no disp.	410	81284	2

Experimento: pel36

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	2884	6
prove_var1	-	no disp.	0	2504	1
prove_uv	-	no disp.	0	3512	6
prove_var1_uv	-	no disp.	0	3340	1
prove_oc	-	no disp.	0	2828	6
prove_var1_oc	-	no disp.	0	2448	1
prove_uv_oc	-	no disp.	0	3456	6
prove_var1_uv_oc	-	no disp.	0	3284	1

prove_abd	-	no disp.	0	3464	6
prove_var1_abd	-	no disp.	0	3060	1
prove_abd_uv	-	no disp.	0	4068	6
prove_var1_abd_uv	-	no disp.	0	3896	1
prove_abd_oc	-	no disp.	0	3408	6
prove_var1_abd_oc	-	no disp.	0	3004	1
prove_abd_uv_oc	-	no disp.	0	4012	6
prove_var1_abd_uv_oc	-	no disp.	0	3840	1

Experimento: pel37

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	0	14560	7
prove_var1	-	no disp.	0	12980	1
prove_uv	-	no disp.	0	22208	7
prove_var1_uv	-	no disp.	0	21444	1
prove_oc	-	no disp.	0	14368	7
prove_var1_oc	-	no disp.	0	12788	1
prove_uv_oc	-	no disp.	10	22016	7
prove_var1_uv_oc	-	no disp.	0	21252	1
prove_abd	-	no disp.	0	15932	7
prove_var1_abd	-	no disp.	0	14352	1
prove_abd_uv	-	no disp.	0	23580	7
prove_var1_abd_uv	-	no disp.	0	22816	1
prove_abd_oc	-	no disp.	0	15740	7
prove_var1_abd_oc	-	no disp.	10	14160	1
prove_abd_uv_oc	-	no disp.	0	23388	7
prove_var1_abd_uv_oc	-	no disp.	0	22624	1

Experimento: pel38

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	30	38036	4
prove_var1	-	no disp.	20	29232	1
prove_uv	-	no disp.	30	106664	4
prove_var1_uv	-	no disp.	30	86372	1
prove_oc	-	no disp.	0	35840	4
prove_var1_oc	-	no disp.	10	27036	1
prove_uv_oc	-	no disp.	10	104468	4
prove_var1_uv_oc	-	no disp.	0	84144	1
prove_abd	-	no disp.	40	44544	4
prove_var1_abd	-	no disp.	20	35740	1
prove_abd_uv	-	no disp.	30	113172	4
prove_var1_abd_uv	-	no disp.	20	92880	1
prove_abd_oc	-	no disp.	10	42348	4
prove_var1_abd_oc	-	no disp.	10	33544	1
prove_abd_uv_oc	-	no disp.	10	110976	4
prove_var1_abd_uv_oc	-	no disp.	10	90652	1

Experimento: pel43

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	10	31272	5
prove_var1	-	no disp.	10	20580	1
prove_uv	-	no disp.	10	113940	5
prove_var1_uv	-	no disp.	20	97076	1
prove_oc	-	no disp.	0	26768	5
prove_var1_oc	-	no disp.	0	16076	1

prove_uv_oc	-	no disp.	0	109436	5
prove_var1_uv_oc	-	no disp.	0	92572	1
prove_abd	-	no disp.	10	33676	5
prove_var1_abd	-	no disp.	20	22984	1
prove_abd_uv	-	no disp.	10	116344	5
prove_var1_abd_uv	-	no disp.	10	99480	1
prove_abd_oc	-	no disp.	0	29172	5
prove_var1_abd_oc	-	no disp.	0	18480	1
prove_abd_uv_oc	-	no disp.	0	111840	5
prove_var1_abd_uv_oc	-	no disp.	10	94976	1

Experimento: pel45

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove	-	no disp.	10	12728	5
prove_var1	-	no disp.	10	8392	1
prove_uv	-	no disp.	20	38160	5
prove_var1_uv	-	no disp.	0	33872	1
prove_oc	-	no disp.	10	11836	5
prove_var1_oc	-	no disp.	0	7432	1
prove_uv_oc	-	no disp.	0	37268	5
prove_var1_uv_oc	-	no disp.	0	32912	1
prove_abd	-	no disp.	10	16476	5
prove_var1_abd	-	no disp.	10	12140	1
prove_abd_uv	-	no disp.	20	41908	5
prove_var1_abd_uv	-	no disp.	0	37620	1
prove_abd_oc	-	no disp.	0	15584	5
prove_var1_abd_oc	-	no disp.	10	11180	1

prove_abd_uv_oc	-	no disp.	0	41016	5
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
prove_var1_abd_uv_oc	-	no disp.	0	36660	1
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+

Apéndice J

Comparación con fórmulas proposicionales

J.1. Fórmulas empleadas

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fichero: benchmark.pl
%% Código Prolog para generar la colección de fórmulas del
%% trabajo 'Some benchmark formulae for intuitionistic
%% propositional logic'.
%% http://www.dcs.st-and.ac.uk/~rd/logic/IPL.BM.Appx.html
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Se ha incluido sólo el código Prolog, del fichero
% que puede obtenerse en la dirección de arriba. Más
% introducimos seis tipos más de fórmulas. Al final,
% para cada tipo de fórmula de pone un ejemplo de
% muestra, indicando el nombre que tal familia de
% fórmulas tendrá en las pruebas posteriores.
% Los operadores lógicos son diferentes a los usa-
% dos por leanTaP, lo que requerirá una posterior
% transformación de la sintaxis.

% Exportamos en el módulo los predicados que servi-
% ran para generar fórmulas-experimento. Son sólo
% los que nos sirven, pues otros tipos de fórmula
% son interesantes sólo para demostradores intui-
% cionistas.
:-module(benchmark,[dB/2,ph_p/2,con_p/2,schwicht_p/2,kk_p/2,
equiv_p/2,tipo1/2,tipo2/2,tipo3/2,
tipo4/2,tipo5/2,tipo6/2]).

% Prolog operator definitions
:- op(425, fy, ~ ).
:- op(450, yfx, &). % left associative
:- op(475, yfx, v ). % left associative
```

```

:- op(500, xfx, <-> ).           % non associative
:- op(500, xfy, ->).           % right associative
%%% WARNING, this overwrites Prolog's ->

#####
%# de Bruijn formulae
#####
de_bruijn_p(N, F ) :-
    N1 is 2*N + 1,
    dB(N1, F).
de_bruijn_n(N, F ) :-
    N1 is 2*N,
    dB(N1, A -> C),
    F = A -> ( p0 v C v ~p0 ).
dB(N, A -> C) :-
    atoms(N, N, [], Ps),
    conj(Ps, C),
    make(Ps, C, As),
    conj(As, A).

atoms(_, M, Atoms, Atoms) :- M =< 0.
atoms(N, M, Atoms, Result) :-
    M > 0,
    M1 is M-1,
    atoms(N, M1, [p(M) | Atoms], Result).

conj( [H], H).
conj( [H1, H2 | Tail], C) :-
    conj( [H1 & H2 | Tail], C).

make([ P0 | Ps], C, As ) :-
    make4([ P0 | Ps], P0, C, As).
make4( [Pn], P0, C, [(Pn<->P0)->C] ).
make4( [P1, P2 | Tail], P0, C, [(P1<->P2)->C | Rest] ) :-
    make4( [P2 | Tail], P0, C, Rest).
#####
%#9.2 Pigeonhole formulae
#####
ph_p(N, L -> R ) :-
    left(N, pos, L),
    right(N,R).

ph_n(N, L -> R ) :-
    left(N, neg, L),
    right(N,R).

right(N, R) :-
    disj(N, 1, false, R).

disj( N, H, D, D ) :-
    H > N.
disj( N, H, D, D1 ) :-

```

```

H =< N,
N1 is N+1,
disj1(N1, H, 1, D, D0),
H1 is H+1,
disj(N, H1, D0, D1).

disj1( N1, _, P1, D, D ) :-
  P1 > N1.
disj1( N1, H, P1, D, D1 ) :-
  P1 =< N1,
  P11 is P1 + 1,
  disj2(N1, H, P1, P11, D, D0),
  disj1(N1, H, P11, D0, D1).

disj2( N1, _, _, P2, D, D ) :-
  P2 > N1.
disj2( N1, H, P1, P2, D, D1 ) :-
  P2 =< N1,
  P21 is P2+1,
  disjoin(D, o(P1,H) & o(P2,H), DC),
  disj2(N1, H, P1, P21, DC, D1).

left( N, Sign, L ) :-
  N1 is N+1,
  conj(N1, N, 1, Sign, true, L).

conj(N1, _, P, _, C, C ) :-
  P > N1.
conj(N1, N, P, Sign, C, C1 ) :-
  P =< N1,
  cd(N, 1, P, false, Sign, D),
  P1 is P+1,
  conjoin(C, D, CD),
  conj(N1, N, P1, Sign, CD, C1).

cd( N, H, _, D, _, D ) :-
  H > N.
cd( N, H, P, D, Sign, D1 ) :-
  H =< N,
  H1 is H+1,
  signify(Sign, o(P,H), H-N, SOPH),
  disjoin(D, SOPH, DOPH),
  cd(N, H1, P, DOPH, Sign, D1).

signify( neg, X, H - H, ~ ~X ) :- !.
signify( _, X, _ - _, X ).
disjoin(false, D, D ) :- !.
disjoin(D1, D, D1 v D).
conjoin(true, D, D ) :- !.
conjoin(D1, D, D1 & D).

%#####

```

```

%# Examples of Franzen
%# con_p(n) requires n contractions in LJ
%# con_n(n) similar but unprovable
#####
con_p(N, ( (C v D) -> f) -> f ) :-
    disjs(N, pos, D),
    conjs(N, C).
con_n(N, ( (C v D) -> f) -> f ) :-
    disjs(N, neg, D),
    conjs(N, C).

conjs(1, p(1)).
conjs(N, C & p(N) ) :-
    N > 1,
    N1 is N-1,
    conjs(N1, C).

disjs(1, pos, p(1) -> f).
disjs(1, neg, ( ~ ~p(1) -> f).
disjs(N, Sign, D v ( p(N) -> f) ) :-
    N > 1,
    N1 is N-1,
    disjs(N1, Sign, D).

#####
%# Schwichtenberg's formulae
%# normal natural deduction of schwicht_p(n) has size
% exponential in n easy for sequent calculus-based systems
#####
schwicht_p( N, A -> p(0) ) :-
    ant(N, 1, p(N), A).
schwicht_n( N, A -> p(0) ) :-
    ant(N, 1, ~ ~p(N), A).
ant( N, I, A, A ) :- I > N.
ant( N, I, A, A1 ) :-
    I =< N,
    I1 is I+1, IM1 is I-1,
    ant(N, I1, A & ( p(I) -> p(I) -> p(IM1)) , A1).

#####
%# Korn & Kreitz's formulae
%# modified to avoid use of Glivenko theorem
#####
kk_p(N, (A->f) & (AR->f) ) :- kk_p(N, N, A), kkr(N,N,AR).
kk_p(N, 0, (a(0) -> f) & ((b(N)->b(0))-> ( a(N)) ) ).
kk_p(N, I, LHS & ((b(I1)->a(I))->( a(I1))) ) :-
    0 < I,
    I1 is I-1,
    kk_p(N, I1, LHS).

kkr(N, 0, ((b(N)->b(0))-> a(N) ) & (a(0) -> f) ).
kkr(N, I, ((b(I1)->a(I))-> a(I1) ) & LHS ) :-

```

```

0 < I,
I1 is I-1,
kk_r(N, I1, LHS).

kk_n(N, A -> f ) :- kk_n(N, N, A).
kk_n(N, 0, (a(0) -> f) & ((~ ~b(N)->b(0))-> ( a(N)) ) ).
kk_n(N, I, LHS & ((~ ~b(I1)->a(I))->( a(I1))) ) :-
  0 < I,
  I1 is I-1,
  kk_n(N, I1, LHS).

#####
%# 9.6 Equivalences
#####
equiv_p( N, A ) :- equiv( N, p, A).
equiv_n( N, A ) :- equiv( N, n, A).
equiv( 1, n, (~ ~a(1)) <-> a(1) ).
equiv( 1, p, a(1) <-> a(1) ).
equiv(N, S, AB <-> BA ) :-
  N > 1,
  N1 is N-1, AN = a(N),
  equiv( N1, S, A1 <-> A2),
  AB = A1 <-> AN,
  BA = AN <-> A2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Nuevos tipos de fórmulas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primer tipo: Son fórmulas no satisfacibles, de tipo:
% (p1 & ... & pn) & ~(p1 & ... & pn)

tipol(N, A & ~A):-
  conj1(N,A).

conj1(1,p(1)):-!.
conj1(N,p(N) & T):-
  N1 is N-1,
  conj1(N1,T).

% Segundo tipo: Fórmulas no satisfacibles, del tipo:
% ~((p1 <-> p2) & ... & (pn-1 <-> pn)) -> (p1 <-> pn))

tipo2(N, ~(A -> (p(1) <-> p(N)))):-
  conj2(N,A).

conj2(2,p(1)<->p(2)):-!.
conj2(N,A & (p(N1) <-> p(N))):-
  N1 is N-1,
  conj2(N1,A).

% Tercer tipo: Fórmulas no satisfacibles, muy parecidas a

```

```

% las anteriores:
% ~(((p1 -> p2) & ... & (pn-1 -> pn)) -> (p1 -> pn))
tipo3(N, ~(A -> (p(1) -> p(N)))):-
    conj3(N,A).

conj3(2,p(1)->p(2)):-!.
conj3(N,A & (p(N1) -> p(N))):-
    N1 is N-1,
    conj3(N1,A).

% Cuarto tipo: Familia muy interesante. Para valores de 'n'
% impares es no satisfacible. Para 'n' pares, es contingente.
tipo4(N, ~(A -> (p(1) <-> p(N)))):-
    conj4(N,A).

conj4(2,~(p(1)<->p(2))):-!.
conj4(N,A & ~(p(N1) <-> p(N))):-
    N1 is N-1,
    conj4(N1,A).

% Quinto tipo. Fórmulas no satisfacibles, de la forma:
% (p1 v ... v pn) & (p1 v ... v -pn) & ... & (-p1 v ... v -pn)
% En que hay una cláusula por cada interpretación.

tipo5(N,F):-
    crea_cláusulas(N,Clau),
    conj5(Clau,F).

crea_cláusulas(N,Clausulas):-
    findall(Claus,(valoración(N,T),cláusula(1,T,Claus)),
    Clausulas).

valoración(N,T):-
    length(T,N),
    cero_uno(T).

cero_uno([0]).
cero_uno([1]).
cero_uno([0|R]):-
    cero_uno(R).
cero_uno([1|R]):-
    cero_uno(R).

cláusula(N,[1],p(N)):-!.
cláusula(N,[0],~p(N)):-!.
cláusula(N,[1|Vs],p(N) v A):-
    N1 is N+1,
    cláusula(N1,Vs,A).
cláusula(N,[0|Vs],~p(N) v A):-
    N1 is N+1,
    cláusula(N1,Vs,A).

conj5([A],A):-!.

```

```

conj5([A|B],A & CB):-
    conj5(B,CB).

% Tipo 6: Fórmulas satisfacibles. Resultan de quitar una
% cláusula a las fórmulas tipo 5.
tipo6(N,F):-
    tipo5(N,_ & F).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Ejemplos de fórmulas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% NOTA: Excepto para las instancias de tipo1--tipo6, las fórmulas
% que se emplearán en los tests son las negaciones de las que se
% generan en los siguientes ejemplos.

% dB: Son las fórmulas generadas por dB/2. Así, dB_3 será la
% fórmula Fml:
% ?- dB(3,Fml).
% Fml = ((p(1)<->p(2))->p(1)&p(2)&p(3))& ((p(2)<->p(3))->
% p(1)&p(2)&p(3))& ((p(3)<->p(1))->p(1)&p(2)&p(3))->p(1)&p(2)&p(3)

% palomar: Fórmulas generadas con ph_p/2. Así, palomar_2, es:
% ?- ph_p(2,Fml).
% Fml = (o(1, 1)v o(1, 2))& (o(2, 1)v o(2, 2))&
% (o(3, 1)v o(3, 2))->o(1, 1)&o(2, 1)v o(1, 1)&o(3, 1)v o(2, 1)&
% o(3, 1)v o(1, 2)&o(2, 2)v o(1, 2)&o(3, 2)v o(2, 2)&o(3, 2)

% franzen: Fórmulas generadas con con_p/2. franzen_2:
% ?- con_p(2,Fml).
% Fml = (p(1)&p(2)v ((p(1)->f)v (p(2)->f))->f)->f

% schwicht: Fórmulas generadas con schwicht_p/2. schwicht_2:
% ?- schwicht_p(2,Fml).
% Fml = p(2)& (p(1)->p(1)->p(0))& (p(2)->p(2)->p(1))->p(0)

% kk: fórmulas generadas con kk_p/2. Así, kk_2:
% ?- kk_p(2,Fml).
% Fml = ((a(0)->f)& ((b(2)->b(0))->a(2))& ((b(0)->a(1))->a(0))&
% ((b(1)->a(2))->a(1))->f)& (((b(1)->a(2))->a(1))&
% (((b(0)->a(1))->a(0))& ((b(2)->b(0))->a(2))& (a(0)->f)))->f

% equiv: fórmulas generadas con equiv_p/2. Mostramos equiv_2:
% ?- equiv_p(2,Fml).
% Fml = (a(1)<->a(2))<-> (a(2)<->a(1))

% tipo1_2:
% ?- tipo1(2,Fml).
% Fml = p(2)&p(1)& ~ (p(2)&p(1))

% tipo2_2:
% ?- tipo2(2,Fml).

```

```

% Fm1 = ~ ((p(1)<->p(2))->p(1)<->p(2))

% tipo3_2:
% ?- tipo3(2,Fm1).
% Fm1 = ~ ((p(1)->p(2))->p(1)->p(2))

% tipo4_2:
% ?- tipo4(2,Fm1).
% Fm1 = ~ (~ (p(1)<->p(2))->p(1)<->p(2))

% tipo5_2:
% ?- tipo5(2,Fm1).
% Fm1 = (~p(1)v~p(2))& ((~p(1)v p(2))& ((p(1)v~p(2))&
% (p(1)v p(2))))

% tipo6_2:
% ?- tipo6(2,Fm1).
% Fm1 = (~p(1)v p(2))& ((p(1)v~p(2))& (p(1)v p(2)))

```

J.2. Sencillo demostrador proposicional

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Archivo: proposicional.pl                                     %%
%% Demostrador proposicional por tablas de verdad, tomado de: %%
%% J.A. Alonso & Joaquín Borrego, "Deducción Automática",    %%
%% Kronos, Sevilla, 2002                                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Declaración de operadores:
:- op(400,fy,-).      % Negación
:- op(500,xfy,&).    % Conjunción
:- op(600,xfy,v).    % Disyunción
:- op(650,xfy,=>).   % Implicación
:- op(700,xfy,<=>).  % Equivalencia

% valor_de_verdad(?V)
%     tiene éxito si V es un valor de verdad.
valor_de_verdad(0).
valor_de_verdad(1).

% función_de_verdad(-, +V, -VN)
%     tiene éxito si VN es el valor de verdad de la
%     negación del valor V.
función_de_verdad(-, 1, 0).
función_de_verdad(-, 0, 1).

% función_de_verdad(+Op, +V1, +V2, -V)
%     devuelve V, que es el valor de verdad correspondiente
%     a la fila de la tabla de verdad del operador Op siendo
%     V1 y V2 los valores de sus argumentos.

```

```

función_de_verdad(v, 0, 0, 0):-!.
función_de_verdad(v, _, _, 1).
función_de_verdad(&, 1, 1, 1):-!.
función_de_verdad(&, _, _, 0).
función_de_verdad(=>, 1, 0, 0):-!.
función_de_verdad(=>, _, _, 1).
función_de_verdad(<=>, X, X, 1):-!.
función_de_verdad(<=>, _, _, 0).

% valor(+F,+I,-V)
% el valor de la fórmula F en la interpretación I es V
valor(F, I, V):-
    memberchk((F,V),I).
valor(-A, I, V):-
    valor(A, I, VA),
    función_de_verdad(-, VA, V).
valor(F, I, V):-
    F =.. [Op, A, B],
    valor(A, I, VA),
    valor(B, I, VB),
    función_de_verdad(Op, VA, VB, V).

% símbolos_fórmula(+F, ?U)
% tiene éxito si U es el conjunto ordenado de los símbolos
% proposicionales de la fórmula F.
símbolos_fórmula(F, U):-
    símbolos_fórmula_aux(F, U1),
    sort(U1, U).
símbolos_fórmula_aux(-F, U):- !,
    símbolos_fórmula_aux(F, U).
símbolos_fórmula_aux(F, U):-
    F =.. [Op, A, B],
    operador_bi(Op),!,
    símbolos_fórmula_aux(A, UA),
    símbolos_fórmula_aux(B, UB),
    union(UA, UB, U).
símbolos_fórmula_aux(F, [F]).
operador_bi(v).
operador_bi(&).
operador_bi(=>).
operador_bi(<=>).

% interpretación_símbolos(+L, -I)
% I es una interpretación para la lista de átomos L.
interpretación_símbolos([], []).
interpretación_símbolos([A|L], [(A,V)|IL]):-
    valor_de_verdad(V),
    interpretación_símbolos(L, IL).

% es_modelo_fórmula(+I, +F)
% tiene éxito si la interpretación I es un modelo de
% la fórmula F

```



```

:-module(dimacs,[pasa_a_dimacs/2]).

:-      op(400,fy,-),
        op(500,xfy,&),
        op(600,xfy,v),
        op(650,xfy,=>),
        op(700,xfy,<=>).

% pasa_a_dimacs(+F,-FD)
% FD es el resultado de pasar la fórmula proposicional F a formato
% DIMACS. Se trata de una fórmula en forma normal conjuntiva en que
% los átomos se han sustituido por números naturales a partir de 1.
% Posteriormente, FD puede emplearse para escribir ficheros de en-
% trada para anldp y zChaff.
pasa_a_dimacs(Form,FDimacs):-
    nnf(Form,NForm,S1),
    asigna_valores(1,S1,S2),
    forma_normal_conj(NForm,S2,FDimacs).

% nnf(+Fml,?NFml,?Simb)
% tiene éxito si para la fórmula proposicional Fml, su NNF es NFml,
% y además Simb es la lista de símbolos proposicionales de Fml. Los
% símbolos se añaden al encontrar literales. Se emplea union/3 para
% evitar repeticiones.
nnf(Fml,NNF,S) :-
    (Fml = -(-A)      -> Fml1 = A;
     Fml = -all(X,F)  -> Fml1 = ex(X,-F);
     Fml = -ex(X,F)   -> Fml1 = all(X,-F);
     Fml = -(A v B)   -> Fml1 = -A & -B;
     Fml = -(A & B)   -> Fml1 = -A v -B;
     Fml = (A => B)   -> Fml1 = -A v B;
     Fml = -(A => B)  -> Fml1 = A & -B;
     Fml = (A <=> B)  -> Fml1 = (A & B) v (-A & -B);
     Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B)),!,
    nnf(Fml1,NNF,S).
nnf(A & B,(NNF1,NNF2),S) :- !,
    nnf(A,NNF1,S1),
    nnf(B,NNF2,S2),
    union(S1,S2,S).
nnf(A v B,(NNF1;NNF2),S) :- !,
    nnf(A,NNF1,S1),
    nnf(B,NNF2,S2),
    union(S1,S2,S).
nnf(-Lit,-Lit,[Lit]):-!.
nnf(L,L,[L]).

% asigna_valores(+N,+Simbolos,?SimbVals)
% A partir de N, asigna números naturales a cada símbolo de
% Simbolos, y devuelve en SimbVals una lista de términos de
% tipo Nk:Sk, siendo Nk el número que corresponde a Sk.
asigna_valores(_,[],[]):-!.

```

```

asigna_valores(N,[S|R],[N:S|NR):-
    N1 is N+1,
    asigna_valores(N1,R,NR).

% forma_normal_conj(+F1,+Simb,?F2)
% tiene éxito si F2 es el resultado de poner en forma normal con-
% juntiva la fórmula proposicional F1. Además, los símbolos pro-
% posicionales de F1 se sustituyen por los número que le corres-
% ponden según la lista Simb.
forma_normal_conj(A,Simb,AS):-
    disyunc_de_literales(A),!,
    sustituye_símbolos(A,Simb,AS).
forma_normal_conj((A,B);C, Simb, (D,E)):-!,
    forma_normal_conj((A;C),Simb,D),
    forma_normal_conj((B;C),Simb,E).
forma_normal_conj((A;(B,C)),S,(D,E)):-!,
    forma_normal_conj((A;B),S,D),
    forma_normal_conj((A;C),S,E).
forma_normal_conj((A,B),S,(C,D)):-
    forma_normal_conj(A,S,C),
    forma_normal_conj(B,S,D).
forma_normal_conj((A;B),S,E):-
    forma_normal_conj(A,S,C),
    forma_normal_conj(B,S,D),
    forma_normal_conj((C;D),S,E).

% disyunc_de_literales(+A)
% tiene éxito si A es una disyunción de literales.
disyunc_de_literales((C;D)):-!,
    disyunc_de_literales(C),
    disyunc_de_literales(D).
disyunc_de_literales(A):-
    literal(A).

% literal(+A)
% tiene éxito si A es un literal.
literal(-A):-!,
    atomo_logico(A).
literal(A):-
    atomo_logico(A).
atomo_logico((_,_)):-!,fail.
atomo_logico((_,_)):-!,fail.
atomo_logico(-_):-!,fail.
atomo_logico(_).

% sustituye_símbolos(+F,+Simbolos,-FS)
% siendo Simbolos un conjunto de símbolos proposicionales enume-
% rados, y F una disyunción de literales, FS es el resultado de
% sustituir en F cada símbolo proposicional por el número co-
% rrespondiente. Cuando ya se ha sustituido, se deja sin cambiar.
sustituye_símbolos((A;B),Simbolos,(AS;BS)):-!,
    sustituye_símbolos(A,Simbolos,AS),

```

```

sustituye_símbolos(B,Simbolos,BS).
sustituye_símbolos(-F,Simbolos,-FS):-!,
    integer(F) -> FS=F ; member(FS:F,Simbolos).
sustituye_símbolos(F,Simbolos,FS):-!,
    integer(F) -> FS=F ; member(FS:F,Simbolos).

```

J.4. Comparación entre todos los métodos y Otter

Experimento dB_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Sat	3130	10	60	-
prove_abd_oc	Sat	3130	0	60	-
proposic	Sat	282	0	60	-
prueba_tgraph	Sat	2727	0	60	-
prueba_comp	Sat	3493	0	60	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Sat	571	0	84	-
prop_prove_abd	Sat	423	0	84	-
prop_tgraph	Sat	554	0	84	-
prop_comp	Sat	543	0	84	-
prop_bdd	Sat	479	0	84	-

```

Otter: dB_2
Res: Satisf
Tpo: 0 segundos

```

Experimento palomar_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	2349	0	4188	-
prove_abd_oc	-	2349	0	13644	-
proposic	-	12163	20	96	-
prueba_tgraph	-	2110	0	3276	-

prueba_comp	-	2654	0	11808	-
prueba_bdd	-	3116	0	3304	-
prop_prove	-	1197	10	2716	-
prop_prove_abd	-	1251	0	5200	-
prop_tgraph	-	1132	0	2260	-
prop_comp	-	1289	0	9712	-
prop_bdd	-	1057	0	2484	-

Otter: palomar_2
Res: Prueba
Tpo: 1 segundos

Experimento franzen_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	315	0	724	-
prove_abd_oc	-	315	0	1180	-
proposic	-	594	0	96	-
prueba_tgraph	-	323	0	972	-
prueba_comp	-	411	0	2716	-
prueba_bdd	-	421	0	984	-
prop_prove	-	272	0	704	-
prop_prove_abd	-	275	0	884	-
prop_tgraph	-	277	0	884	-
prop_comp	-	334	0	2872	-
prop_bdd	-	297	0	876	-

Otter: franzen_2
Res: Prueba
Tpo: 0 segundos

Experimento schwicht_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove_oc	-	301	0	600	-
prove_abd_oc	-	301	0	1200	-
proposic	-	588	10	96	-
prueba_tgraph	-	318	0	880	-
prueba_comp	-	420	0	3272	-
prueba_bdd	-	380	0	840	-
prop_prove	-	273	0	552	-
prop_prove_abd	-	280	0	792	-
prop_tgraph	-	281	0	792	-
prop_comp	-	352	0	3244	-
prop_bdd	-	287	0	748	-

Otter: schwicht_2
 Res: Prueba
 Tpo: 0 segundos

Experimento kk_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1347	0	2316	-
prove_abd_oc	-	1347	0	5292	-
proposic	-	22616	31	96	-
prueba_tgraph	-	1276	0	2884	-
prueba_comp	-	1666	0	9624	-
prueba_bdd	-	9318	10	5768	-
prop_prove	-	987	0	2172	-
prop_prove_abd	-	1005	0	3192	-
prop_tgraph	-	984	0	2532	-
prop_comp	-	1169	0	9504	-
prop_bdd	-	1908	0	2780	-

Otter: kk_2
 Res: Prueba
 Tpo: 0 segundos

Experimento equiv_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	578	0	1228	-
prove_abd_oc	-	578	0	2404	-
proposic	-	183	0	96	-
prueba_tgraph	-	604	0	1716	-
prueba_comp	-	775	0	5664	-
prueba_bdd	-	1057	0	1840	-
prop_prove	-	519	0	1196	-
prop_prove_abd	-	527	0	1688	-
prop_tgraph	-	524	0	1508	-
prop_comp	-	641	0	5840	-
prop_bdd	-	600	0	1332	-

Otter: equiv_2
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo1_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	113	0	276	-
prove_abd_oc	-	113	0	468	-
proposic	-	183	0	96	-
prueba_tgraph	-	124	0	404	-
prueba_comp	-	164	0	1296	-
prueba_bdd	-	153	0	404	-
prop_prove	-	109	0	276	-

prop_prove_abd	-	111	0	360	-
prop_tgraph	-	112	0	372	-
prop_comp	-	143	0	1384	-
prop_bdd	-	124	0	380	-

Otter: tipo1_2
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo2_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	287	0	648	-
prove_abd_oc	-	287	0	1224	-
proposic	-	183	0	96	-
prueba_tgraph	-	300	0	880	-
prueba_comp	-	384	0	2788	-
prueba_bdd	-	400	0	908	-
prop_prove	-	259	0	632	-
prop_prove_abd	-	263	0	872	-
prop_tgraph	-	261	0	776	-
prop_comp	-	319	0	2888	-
prop_bdd	-	287	0	748	-

Otter: tipo2_2
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo3_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	131	0	340	-
prove_abd_oc	-	131	0	532	-
proposic	-	183	0	96	-

prueba_tgraph	-	142	0	468	-
prueba_comp	-	182	0	1360	-
prueba_bdd	-	171	0	484	-
prop_prove	-	127	0	340	-
prop_prove_abd	-	129	0	424	-
prop_tgraph	-	130	0	436	-
prop_comp	-	161	0	1448	-
prop_bdd	-	141	0	444	-

Otter: tipo3_2
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo4_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Sat	249	0	84	-
prove_abd_oc	Sat	249	0	84	-
proposic	Sat	122	0	84	-
prueba_tgraph	Sat	265	0	84	-
prueba_comp	Sat	345	0	84	-
prueba_bdd	Sat	11983	10	84	-
prop_prove	Sat	234	0	84	-
prop_prove_abd	Sat	235	0	84	-
prop_tgraph	Sat	243	0	84	-
prop_comp	Sat	301	0	84	-
prop_bdd	Sat	265	0	84	-

Otter: tipo4_2
 Res: Satisf
 Tpo: 0 segundos

Experimento tipo5_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	282	0	504	-
prove_abd_oc	-	282	0	1296	-
proposic	-	384	0	96	-
prueba_tgraph	-	302	0	788	-
prueba_comp	-	404	0	3304	-
prueba_bdd	-	338	0	724	-
prop_prove	-	252	0	468	-
prop_prove_abd	-	261	0	828	-
prop_tgraph	-	254	0	636	-
prop_comp	-	326	0	3204	-
prop_bdd	-	245	0	572	-

Otter: tipo5_2
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo6_2

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Sat	214	0	84	-
prove_abd_oc	Sat	214	10	84	-
proposic	Sat	269	0	84	-
prueba_tgraph	Sat	231	0	84	-
prueba_comp	Sat	311	0	84	-
prueba_bdd	Sat	186	0	84	-
prop_prove	Sat	182	0	84	-
prop_prove_abd	Sat	174	0	84	-
prop_tgraph	Sat	185	0	84	-
prop_comp	Sat	229	0	84	-

prop_bdd	Sat	173	0	84	-
----------	-----	-----	---	----	---

Otter: tipo6_2
 Res: Satisf
 Tpo: 0 segundos

Experimento dB_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1313	0	2428	-
prove_abd_oc	-	1313	0	6052	-
proposic	-	1255	0	96	-
prueba_tgraph	-	1239	0	2816	-
prueba_comp	-	1609	0	9416	-
prueba_bdd	-	2043	11	2756	-
prop_prove	-	930	0	2212	-
prop_prove_abd	-	953	0	3688	-
prop_tgraph	-	899	0	2224	-
prop_comp	-	1078	0	9212	-
prop_bdd	-	933	0	1924	-

Otter: dB_3
 Res: Prueba
 Tpo: 0 segundos

Experimento palomar_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	-	1991548	3479	96	-
prueba_tgraph	-	2902023	1391	624276	-
prueba_comp	-	3750356	2760	1944196	-
prueba_bdd	-	100322	30	35920	-

prop_prove	-	683041	1029	940036	-
prop_prove_abd	-	715585	1151	71488	-
prop_tgraph	-	618413	989	165388	-
prop_comp	-	618839	1020	183784	-
prop_bdd	-	12133	10	13216	-

Otter: palomar_3
 Res: Prueba
 Tpo: 0 segundos

Experimento franzen_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	485	0	1028	-
prove_abd_oc	-	485	10	1748	-
proposic	-	1472	0	96	-
prueba_tgraph	-	482	0	1348	-
prueba_comp	-	617	0	3808	-
prueba_bdd	-	661	0	1352	-
prop_prove	-	386	0	972	-
prop_prove_abd	-	390	0	1224	-
prop_tgraph	-	393	0	1224	-
prop_comp	-	471	0	3988	-
prop_bdd	-	420	0	1204	-

Otter: franzen_3
 Res: Prueba
 Tpo: 0 segundos

Experimento schwicht_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	453	0	888	-
prove_abd_oc	-	453	0	1896	-

proposic	-	1460	0	96	-
prueba_tgraph	-	466	0	1224	-
prueba_comp	-	617	0	4784	-
prueba_bdd	-	615	0	1148	-
prop_prove	-	393	0	820	-
prop_prove_abd	-	402	0	1204	-
prop_tgraph	-	399	0	1096	-
prop_comp	-	499	0	4804	-
prop_bdd	-	405	0	1004	-

Otter: schwicht_3
Res: Prueba
Tpo: 0 segundos

Experimento kk_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	2150	0	3312	-
prove_abd_oc	-	2150	0	8256	-
proposic	-	111043	170	96	-
prueba_tgraph	-	1925	0	3804	-
prueba_comp	-	2551	0	12504	-
prueba_bdd	-	30263	11	12152	-
prop_prove	-	1380	0	3024	-
prop_prove_abd	-	1407	0	4632	-
prop_tgraph	-	1358	10	3252	-
prop_comp	-	1587	0	12120	-
prop_bdd	-	4359	0	4284	-

Otter: kk_3
Res: Prueba
Tpo: 0 segundos

Experimento equiv_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1848	0	3468	-
prove_abd_oc	-	1848	0	7716	-
proposic	-	457	0	96	-
prueba_tgraph	-	1804	0	4404	-
prueba_comp	-	2337	10	14736	-
prueba_bdd	-	4232	0	4904	-
prop_prove	-	1411	0	3132	-
prop_prove_abd	-	1435	0	4680	-
prop_tgraph	-	1408	0	3732	-
prop_comp	-	1697	10	14800	-
prop_bdd	-	1652	0	3252	-

Otter: equiv_3

Res: Prueba

Tpo: 0 segundos

Experimento tipo1_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	192	0	420	-
prove_abd_oc	-	192	10	756	-
proposic	-	454	0	96	-
prueba_tgraph	-	204	0	612	-
prueba_comp	-	270	0	2028	-
prueba_bdd	-	269	0	600	-
prop_prove	-	176	0	408	-
prop_prove_abd	-	179	0	540	-
prop_tgraph	-	181	0	564	-

prop_comp	-	228	0	2136	-
prop_bdd	-	198	0	564	-

Otter: tipo1_3
Res: Prueba
Tpo: 0 segundos

Experimento tipo2_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	492	0	976	-
prove_abd_oc	-	492	0	1984	-
proposic	-	457	9	96	-
prueba_tgraph	-	492	0	1292	-
prueba_comp	-	634	0	4224	-
prueba_bdd	-	695	0	1324	-
prop_prove	-	398	0	936	-
prop_prove_abd	-	404	0	1344	-
prop_tgraph	-	398	0	1116	-
prop_comp	-	483	0	4328	-
prop_bdd	-	435	0	1084	-

Otter: tipo2_3
Res: Prueba
Tpo: 0 segundos

Experimento tipo3_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	208	0	460	-
prove_abd_oc	-	208	0	772	-
proposic	-	457	0	96	-
prueba_tgraph	-	220	0	660	-
prueba_comp	-	287	0	2192	-

prueba_bdd	-	285	0	664	-
prop_prove	-	194	0	456	-
prop_prove_abd	-	197	0	588	-
prop_tgraph	-	199	0	612	-
prop_comp	-	247	0	2260	-
prop_bdd	-	215	0	612	-

Otter: tipo3_3
Res: Prueba
Tpo: 0 segundos

Experimento tipo4_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	494	0	992	-
prove_abd_oc	-	494	10	2000	-
proposic	-	523	0	96	-
prueba_tgraph	-	494	0	1308	-
prueba_comp	-	636	0	4240	-
prueba_bdd	-	697	0	1340	-
prop_prove	-	400	0	952	-
prop_prove_abd	-	406	0	1360	-
prop_tgraph	-	400	0	1132	-
prop_comp	-	485	0	4344	-
prop_bdd	-	437	0	1100	-

Otter: tipo4_3
Res: Prueba
Tpo: 1 segundos

Experimento tipo5_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	7130	0	10136	-

prove_abd_oc	-	7130	11	43016	-
proposic	-	2005	0	96	-
prueba_tgraph	-	6260	0	7788	-
prueba_comp	-	7857	10	23332	-
prueba_bdd	-	1513	0	2108	-
prop_prove	-	3193	0	5100	-
prop_prove_abd	-	3534	10	17052	-
prop_tgraph	-	2925	0	2364	-
prop_comp	-	3160	0	15968	-
prop_bdd	-	786	0	1532	-

Otter: tipo5_3
Res: Prueba
Tpo: 0 segundos

Experimento tipo6_3

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Tiempo	-	-	-	-
prove_abd_oc	Tiempo	-	-	-	-
proposic	Sat	1680	0	84	-
prueba_tgraph	Tiempo	-	-	-	-
prueba_comp	Tiempo	-	-	-	-
prueba_bdd	Sat	767	0	84	-
prop_prove	Tiempo	-	-	-	-
prop_prove_abd	Sat	1685	0	84	-
prop_tgraph	Tiempo	-	-	-	-
prop_comp	Sat	1663	0	84	-
prop_bdd	Sat	633	0	84	-

Otter: tipo6_3
Res: Satisf

Tpo: 0 segundos

Experimento dB_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Tiempo	-	-	-	-
prove_abd_oc	Tiempo	-	-	-	-
proposic	Sat	1526	0	84	-
prueba_tgraph	Tiempo	-	-	-	-
prueba_comp	Tiempo	-	-	-	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Tiempo	-	-	-	-
prop_prove_abd	Sat	1110	0	84	-
prop_tgraph	Tiempo	-	-	-	-
prop_comp	Sat	1394	0	84	-
prop_bdd	Sat	1291	0	84	-

Otter: dB_4

Res: Satisf

Tpo: 0 segundos

Experimento palomar_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	-	3740516	1359	687388	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-

prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	-	282736	320	125912	-

Otter: palomar_4
 Res: Prueba
 Tpo: 321 segundos

Experimento franzen_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	681	0	1348	-
prove_abd_oc	-	681	0	2380	-
proposic	-	3524	9	96	-
prueba_tgraph	-	659	0	1724	-
prueba_comp	-	849	0	4916	-
prueba_bdd	-	943	0	1720	-
prop_prove	-	503	0	1240	-
prop_prove_abd	-	508	0	1564	-
prop_tgraph	-	512	0	1564	-
prop_comp	-	611	0	5104	-
prop_bdd	-	546	0	1532	-

Otter: franzen_4
 Res: Prueba
 Tpo: 1 segundos

Experimento schwicht_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	696	0	1344	-
prove_abd_oc	-	696	0	3240	-
proposic	-	3503	10	96	-
prueba_tgraph	-	689	0	1680	-

prueba_comp	-	907	0	6456	-
prueba_bdd	-	970	0	1532	-
prop_prove	-	560	0	1256	-
prop_prove_abd	-	575	0	2000	-
prop_tgraph	-	554	0	1448	-
prop_comp	-	683	0	6324	-
prop_bdd	-	542	0	1324	-

Otter: schwicht_4
 Res: Prueba
 Tpo: 0 segundos

Experimento kk_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	3265	0	4508	-
prove_abd_oc	-	3265	0	11972	-
proposic	-	528948	830	96	-
prueba_tgraph	-	2783	0	4820	-
prueba_comp	-	3750	0	15600	-
prueba_bdd	-	92669	41	26596	-
prop_prove	-	1858	0	4020	-
prop_prove_abd	-	1896	0	6360	-
prop_tgraph	-	1809	0	4020	-
prop_comp	-	2082	0	14848	-
prop_bdd	-	10755	19	6908	-

Otter: kk_4
 Res: Prueba
 Tpo: 0 segundos

Experimento equiv_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove_oc	-	5240	10	8812	-
prove_abd_oc	-	5240	0	21700	-
proposic	-	1094	0	96	-
prueba_tgraph	-	4804	10	10004	-
prueba_comp	-	6313	0	33744	-
prueba_bdd	-	14184	10	11688	-
prop_prove	-	3363	0	7292	-
prop_prove_abd	-	3427	0	11432	-
prop_tgraph	-	3320	11	8180	-
prop_comp	-	3953	0	32912	-
prop_bdd	-	4003	10	7092	-

Otter: equiv_4

Res: Prueba

Tpo: 0 segundos

Experimento tipo1_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	280	0	572	-
prove_abd_oc	-	280	0	1076	-
proposic	-	1085	0	96	-
prueba_tgraph	-	290	0	820	-
prueba_comp	-	385	0	2768	-
prueba_bdd	-	406	0	796	-
prop_prove	-	245	0	540	-
prop_prove_abd	-	249	0	720	-
prop_tgraph	-	252	0	756	-
prop_comp	-	315	0	2888	-
prop_bdd	-	274	0	748	-

Otter: tipo1_4
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo2_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	771	0	1360	-
prove_abd_oc	-	771	0	2944	-
proposic	-	1094	0	96	-
prueba_tgraph	-	739	0	1744	-
prueba_comp	-	960	0	5740	-
prueba_bdd	-	1110	0	1808	-
prop_prove	-	555	0	1288	-
prop_prove_abd	-	563	0	1888	-
prop_tgraph	-	551	0	1480	-
prop_comp	-	663	0	5816	-
prop_bdd	-	600	0	1420	-

Otter: tipo2_4
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo3_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	287	0	580	-
prove_abd_oc	-	287	0	1012	-
proposic	-	1094	0	96	-
prueba_tgraph	-	300	0	852	-
prueba_comp	-	394	0	3024	-
prueba_bdd	-	398	0	860	-
prop_prove	-	261	10	572	-

prop_prove_abd	-	265	0	752	-
prop_tgraph	-	268	0	788	-
prop_comp	-	333	0	3072	-
prop_bdd	-	289	0	780	-

Otter: tipo3_4

Res: Prueba

Tpo: 0 segundos

Experimento tipo4_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Sat	691	0	84	-
prove_abd_oc	Sat	691	0	84	-
proposic	Sat	543	0	84	-
prueba_tgraph	Sat	667	10	84	-
prueba_comp	Sat	881	0	84	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Sat	494	0	84	-
prop_prove_abd	Sat	480	0	84	-
prop_tgraph	Sat	507	0	84	-
prop_comp	Sat	606	0	84	-
prop_bdd	Sat	541	0	84	-

Otter: tipo4_4

Res: Satisf

Tpo: 0 segundos

Experimento tipo5_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	-	9946	21	96	-

prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	-	6735	0	5788	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	-	5122517	5930	154556	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	-	4152618	4891	10696	-
prop_bdd	-	2204	0	3932	-

Otter: tipo5_4
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo6_4

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Sat	9111	10	84	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Sat	2699	0	84	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Sat	1590444	1851	20	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Sat	1288669	1520	20	-
prop_bdd	Sat	1828	0	84	-

Otter: tipo6_4
 Res: Satisf
 Tpo: 0 segundos

Experimento dB_5

-----+

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	5023	10	7808	-
prove_abd_oc	-	5023	0	22880	-
proposic	-	9679	10	96	-
prueba_tgraph	-	4165	11	6744	-
prueba_comp	-	5568	0	21184	-
prueba_bdd	-	7760	10	6420	-
prop_prove	-	2517	0	6716	-
prop_prove_abd	-	2584	0	12212	-
prop_tgraph	-	2291	0	4772	-
prop_comp	-	2642	10	19656	-
prop_bdd	-	2310	0	3940	-

Otter: dB_5

Res: Prueba

Tpo: 0 segundos

Experimento palomar_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Pila local	-	-	-	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-

```

|      prop_bdd | Pila local |      - |      - |      - |      - |
+-----+-----+-----+-----+-----+-----+
Otter: palomar_5
Res: Límite de tiempo
Tpo: 593 segundos

```

Experimento franzen_5

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc |      - |      903 |      0 | 1684 |      - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc |      - |      903 |      0 | 3076 |      - |
+-----+-----+-----+-----+-----+-----+
|      propositic |      - |     8254 |      9 |   96 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph |      - |      854 |      0 | 2100 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp |      - |     1107 |      0 | 6040 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_bdd |      - |     1267 |      0 | 2088 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove |      - |      623 |      0 | 1508 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove_abd |      - |      629 |      0 | 1904 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_tgraph |      - |      634 |      0 | 1904 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_comp |      - |      754 |     10 | 6220 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_bdd |      - |      675 |      0 | 1860 |      - |
+-----+-----+-----+-----+-----+-----+
Otter: franzen_5
Res: Prueba
Tpo: 0 segundos

```

Experimento schwicht_5

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc |      - |     1214 |      0 | 2232 |      - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc |      - |     1214 |      0 | 6264 |      - |
+-----+-----+-----+-----+-----+-----+
|      propositic |      - |     8221 |     10 |   96 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph |      - |     1125 |      0 | 2360 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp |      - |     1476 |      0 | 8608 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_bdd |      - |     1487 |      0 | 1972 |      - |
+-----+-----+-----+-----+-----+-----+

```

prop_prove	-	842	0	2028	-
prop_prove_abd	-	871	0	3564	-
prop_tgraph	-	804	0	1896	-
prop_comp	-	962	10	8132	-
prop_bdd	-	699	0	1676	-

Otter: schwicht_5
 Res: Prueba
 Tpo: 0 segundos

Experimento kk_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	4752	0	5904	-
prove_abd_oc	-	4752	0	16440	-
proposic	-	2458283	4050	96	-
prueba_tgraph	-	3886	0	5932	-
prueba_comp	-	5323	0	18912	-
prueba_bdd	-	267248	90	59064	-
prop_prove	-	2433	10	5160	-
prop_prove_abd	-	2484	0	8376	-
prop_tgraph	-	2349	0	4836	-
prop_comp	-	2666	0	17688	-
prop_bdd	-	27207	31	11772	-

Otter: kk_5
 Res: Prueba
 Tpo: 0 segundos

Experimento equiv_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	14088	10	21484	-
prove_abd_oc	-	14088	9	57412	-

proposic	-	2566	10	96	-
prueba_tgraph	-	12244	0	21652	-
prueba_comp	-	16329	20	73744	-
prueba_bdd	-	43020	21	26536	-
prop_prove	-	7635	0	16188	-
prop_prove_abd	-	7795	10	26472	-
prop_tgraph	-	7464	10	17076	-
prop_comp	-	8785	9	69520	-
prop_bdd	-	9265	10	14772	-

Otter: equiv_5
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo1_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	377	0	732	-
prove_abd_oc	-	377	0	1428	-
proposic	-	2548	0	96	-
prueba_tgraph	-	382	0	1028	-
prueba_comp	-	509	0	3516	-
prueba_bdd	-	564	0	992	-
prop_prove	-	316	0	672	-
prop_prove_abd	-	321	0	900	-
prop_tgraph	-	325	0	948	-
prop_comp	-	404	0	3640	-
prop_bdd	-	352	0	932	-

Otter: tipo1_5
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo2_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1104	0	1744	-
prove_abd_oc	-	1104	0	3904	-
proposic	-	2566	10	96	-
prueba_tgraph	-	1022	0	2196	-
prueba_comp	-	1340	0	7256	-
prueba_bdd	-	1615	0	2292	-
prop_prove	-	716	0	1640	-
prop_prove_abd	-	726	0	2432	-
prop_tgraph	-	708	0	1844	-
prop_comp	-	847	0	7304	-
prop_bdd	-	767	0	1756	-

Otter: tipo2_5

Res: Prueba

Tpo: 0 segundos

Experimento tipo3_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	370	0	700	-
prove_abd_oc	-	370	0	1252	-
proposic	-	2566	0	96	-
prueba_tgraph	-	383	0	1044	-
prueba_comp	-	505	10	3856	-
prueba_bdd	-	520	0	1056	-
prop_prove	-	329	0	688	-
prop_prove_abd	-	334	0	916	-
prop_tgraph	-	338	0	964	-

```

|      prop_comp |      - |      420 |      0 | 3884 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_bdd |      - |      364 |      0 |  948 |      - |
+-----+-----+-----+-----+-----+
Otter: tipo3_5
Res: Prueba
Tpo: 0 segundos

```

Experimento tipo4_5

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc |      - |      1108 |      10 | 1776 |      - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc |      - |      1108 |      0 | 3936 |      - |
+-----+-----+-----+-----+-----+-----+
|      propositic |      - |      3082 |      0 |   96 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph |      - |      1026 |      0 | 2228 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp |      - |      1344 |      0 | 7288 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_bdd |      - |      1619 |      0 | 2324 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove |      - |      720 |      0 | 1672 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove_abd |      - |      730 |      0 | 2464 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_tgraph |      - |      712 |      9 | 1876 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_comp |      - |      851 |      0 | 7336 |      - |
+-----+-----+-----+-----+-----+-----+
|      prop_bdd |      - |      771 |      0 | 1788 |      - |
+-----+-----+-----+-----+-----+-----+
Otter: tipo4_5
Res: Prueba
Tpo: 0 segundos

```

Experimento tipo5_5

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc | Pila local |      - |      - |      - |      - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc | Pila local |      - |      - |      - |      - |
+-----+-----+-----+-----+-----+-----+
|      propositic |      - |      47899 |      61 |   96 |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph | Pila local |      - |      - |      - |      - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp | Pila local |      - |      - |      - |      - |
+-----+-----+-----+-----+-----+-----+

```

prueba_bdd	-	31295	10	15152	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	-	5780	0	9712	-

Otter: tipo5_5
Res: Prueba
Tpo: 0 segundos

Experimento tipo6_5

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Sat	45854	60	84	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Sat	9234	10	84	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Sat	4734	0	84	-

Otter: tipo6_5
Res: Satisf
Tpo: 0 segundos

Experimento dB_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Tiempo	-	-	-	-

prove_abd_oc	Tiempo	-	-	-	-
proposic	Sat	9179	10	84	-
prueba_tgraph	Tiempo	-	-	-	-
prueba_comp	Tiempo	-	-	-	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Tiempo	-	-	-	-
prop_prove_abd	Sat	2049	0	96	-
prop_tgraph	Tiempo	-	-	-	-
prop_comp	Sat	2542	0	96	-
prop_bdd	Sat	2564	0	96	-

Otter: dB_6
 Res: Satisf
 Tpo: 0 segundos

Experimento palomar_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Pila local	-	-	-	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Pila local	-	-	-	-

Otter: palomar_6
 Res: Límite de tiempo

Tpo: 591 segundos

Experimento franzen_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1151	0	2048	-
prove_abd_oc	-	1151	0	3848	-
proposic	-	19006	20	108	-
prueba_tgraph	-	1067	0	2488	-
prueba_comp	-	1391	9	7192	-
prueba_bdd	-	1633	0	2468	-
prop_prove	-	746	0	1788	-
prop_prove_abd	-	753	0	2256	-
prop_tgraph	-	759	0	2256	-
prop_comp	-	900	0	7348	-
prop_bdd	-	807	0	2200	-

Otter: franzen_6

Res: Prueba

Tpo: 0 segundos

Experimento schwicht_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	2399	0	4060	-
prove_abd_oc	-	2399	0	12916	-
proposic	-	18958	31	108	-
prueba_tgraph	-	2062	0	3500	-
prueba_comp	-	2720	0	11796	-
prueba_bdd	-	2214	0	2480	-
prop_prove	-	1379	0	3484	-
prop_prove_abd	-	1438	0	6676	-

prop_tgraph	-	1269	9	2548	-
prop_comp	-	1456	0	10528	-
prop_bdd	-	878	0	2072	-

Otter: schwicht_6
 Res: Prueba
 Tpo: 0 segundos

Experimento kk_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	6671	9	7512	-
prove_abd_oc	-	6671	0	21672	-
proposic	-	11207464	19430	108	-
prueba_tgraph	-	5270	0	7152	-
prueba_comp	-	7330	10	22452	-
prueba_bdd	-	732759	250	131320	-
prop_prove	-	3117	0	6456	-
prop_prove_abd	-	3183	0	10692	-
prop_tgraph	-	2990	11	5712	-
prop_comp	-	3351	0	20652	-
prop_bdd	-	68553	70	21128	-

Otter: kk_6
 Res: Prueba
 Tpo: 0 segundos

Experimento equiv_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	36632	20	51320	-
prove_abd_oc	-	36632	31	146384	-
proposic	-	5929	9	108	-
prueba_tgraph	-	30484	11	45856	-

prueba_comp	-	41209	30	158236	-
prueba_bdd	-	122012	40	58804	-
prop_prove	-	16979	21	35144	-
prop_prove_abd	-	17363	9	59636	-
prop_tgraph	-	16456	20	34880	-
prop_comp	-	19153	20	143516	-
prop_bdd	-	21037	21	30144	-

Otter: equiv_6
Res: Prueba
Tpo: 5 segundos

Experimento tipo1_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	483	0	912	-
prove_abd_oc	-	483	0	1824	-
proposic	-	5899	11	108	-
prueba_tgraph	-	480	0	1248	-
prueba_comp	-	642	0	4284	-
prueba_bdd	-	743	0	1200	-
prop_prove	-	389	0	816	-
prop_prove_abd	-	395	0	1092	-
prop_tgraph	-	400	0	1152	-
prop_comp	-	495	0	4404	-
prop_bdd	-	432	0	1128	-

Otter: tipo1_6
Res: Prueba
Tpo: 0 segundos

Experimento tipo2_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove_oc	-	1491	0	2140	-
prove_abd_oc	-	1491	0	4876	-
proposic	-	5929	11	108	-
prueba_tgraph	-	1341	0	2660	-
prueba_comp	-	1774	0	8784	-
prueba_bdd	-	2204	9	2788	-
prop_prove	-	881	0	2004	-
prop_prove_abd	-	893	0	2988	-
prop_tgraph	-	869	0	2220	-
prop_comp	-	1035	0	8804	-
prop_bdd	-	936	0	2104	-

Otter: tipo2_6
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo3_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	457	0	832	-
prove_abd_oc	-	457	0	1504	-
proposic	-	5929	9	108	-
prueba_tgraph	-	469	0	1248	-
prueba_comp	-	620	0	4700	-
prueba_bdd	-	651	0	1264	-
prop_prove	-	398	0	816	-
prop_prove_abd	-	404	0	1092	-
prop_tgraph	-	409	0	1152	-
prop_comp	-	508	0	4708	-
prop_bdd	-	440	0	1128	-

Otter: tipo3_6
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo4_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Sat	2627	0	96	-
prove_abd_oc	Sat	2627	0	96	-
proposic	Sat	2584	10	96	-
prueba_tgraph	Sat	2150	0	96	-
prueba_comp	Sat	2929	0	96	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Sat	974	0	96	-
prop_prove_abd	Sat	737	0	96	-
prop_tgraph	Sat	945	0	96	-
prop_comp	Sat	922	0	96	-
prop_bdd	Sat	826	0	96	-

Otter: tipo4_6
 Res: Satisf
 Tpo: 0 segundos

Experimento tipo5_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	-	225548	311	108	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	-	150927	49	37904	-
prop_prove	Pila local	-	-	-	-

prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	-	14564	11	22928	-

Otter: tipo5_6
 Res: Prueba
 Tpo: 2 segundos

Experimento tipo6_6

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Sat	220693	310	96	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Sat	32236	9	96	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Sat	11543	0	96	-

Otter: tipo6_6
 Res: Satisf
 Tpo: 2 segundos

Experimento dB_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	13529	11	18104	-
prove_abd_oc	-	13529	9	56216	-
proposic	-	64587	90	108	-

prueba_tgraph	-	10367	11	12716	-
prueba_comp	-	14295	9	38124	-
prueba_bdd	-	20223	10	11556	-
prop_prove	-	5268	0	15132	-
prop_prove_abd	-	5403	11	28344	-
prop_tgraph	-	4603	0	8400	-
prop_comp	-	5166	9	33772	-
prop_bdd	-	4476	11	6640	-

Otter: dB_7

Res: Prueba

Tpo: 0 segundos

Experimento palomar_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Pila local	-	-	-	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Pila local	-	-	-	-

Otter: palomar_7

Res: Límite de tiempo

Tpo: 576 segundos

Experimento franzen_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1425	0	2416	-
prove_abd_oc	-	1425	0	4672	-
proposic	-	43140	59	108	-
prueba_tgraph	-	1298	0	2864	-
prueba_comp	-	1701	0	8348	-
prueba_bdd	-	2041	0	2836	-
prop_prove	-	872	0	2056	-
prop_prove_abd	-	880	11	2596	-
prop_tgraph	-	887	0	2596	-
prop_comp	-	1049	0	8464	-
prop_bdd	-	942	0	2528	-

Otter: franzen_7
 Res: Prueba
 Tpo: 0 segundos

Experimento schwicht_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	5147	0	7860	-
prove_abd_oc	-	5147	0	27300	-
proposic	-	43074	70	108	-
prueba_tgraph	-	4148	0	5524	-
prueba_comp	-	5543	0	17148	-
prueba_bdd	-	3199	0	3032	-
prop_prove	-	2475	11	6272	-
prop_prove_abd	-	2596	0	12848	-
prop_tgraph	-	2213	0	3572	-
prop_comp	-	2429	9	14064	-

```

|      prop_bdd | - |      1081 |      0 | 2488 | - |
+-----+-----+-----+-----+-----+-----+
Otter: schwicht_7
Res: Prueba
Tpo: 0 segundos

```

Experimento kk_7

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc | - |      9082 |      11 | 9308 | - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc | - |      9082 |      9 | 27644 | - |
+-----+-----+-----+-----+-----+-----+
|      propositic | - | 50332587 | 91541 | 108 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph | - |      6971 |      9 | 8456 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp | - |      9831 |      0 | 26196 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_bdd | - | 1929550 | 741 | 269168 | - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove | - |      3922 |      9 | 7884 | - |
+-----+-----+-----+-----+-----+-----+
|      prop_prove_abd | - |      4005 |      0 | 13284 | - |
+-----+-----+-----+-----+-----+-----+
|      prop_tgraph | - |      3744 |      11 | 6624 | - |
+-----+-----+-----+-----+-----+-----+
|      prop_comp | - |      4149 |      0 | 23716 | - |
+-----+-----+-----+-----+-----+-----+
|      prop_bdd | - | 170173 | 189 | 39432 | - |
+-----+-----+-----+-----+-----+-----+
Otter: kk_7
Res: Prueba
Tpo: 0 segundos

```

Experimento equiv_7

```

+-----+-----+-----+-----+-----+-----+
|      MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
|      prove_oc | - |      92856 |      59 | 120952 | - |
+-----+-----+-----+-----+-----+-----+
|      prove_abd_oc | - |      92856 |      70 | 363472 | - |
+-----+-----+-----+-----+-----+-----+
|      propositic | - |      13519 |      20 | 108 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_tgraph | - |      74644 |      41 | 96032 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_comp | - | 102105 | 69 | 337180 | - |
+-----+-----+-----+-----+-----+-----+
|      prueba_bdd | - | 329548 | 121 | 128436 | - |
+-----+-----+-----+-----+-----+-----+

```

prop_prove	-	37395	29	75336	-
prop_prove_abd	-	38291	41	132084	-
prop_tgraph	-	35976	29	70464	-
prop_comp	-	41425	61	293020	-
prop_bdd	-	47333	40	60864	-

Otter: equiv_7
 Res: Prueba
 Tpo: 57 segundos

Experimento tipo1_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	598	0	1088	-
prove_abd_oc	-	598	0	2240	-
proposic	-	13474	20	108	-
prueba_tgraph	-	584	0	1456	-
prueba_comp	-	784	0	5048	-
prueba_bdd	-	943	0	1396	-
prop_prove	-	464	0	948	-
prop_prove_abd	-	471	0	1272	-
prop_tgraph	-	477	9	1344	-
prop_comp	-	588	0	5156	-
prop_bdd	-	514	0	1312	-

Otter: tipo1_7
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo2_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1932	0	2524	-
prove_abd_oc	-	1932	0	5836	-

proposic	-	13519	20	108	-
prueba_tgraph	-	1696	0	3112	-
prueba_comp	-	2262	0	10300	-
prueba_bdd	-	2877	0	3272	-
prop_prove	-	1050	9	2356	-
prop_prove_abd	-	1064	0	3532	-
prop_tgraph	-	1034	0	2584	-
prop_comp	-	1227	0	10292	-
prop_bdd	-	1107	0	2440	-

Otter: tipo2_7
Res: Prueba
Tpo: 0 segundos

Experimento tipo3_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	548	0	952	-
prove_abd_oc	-	548	0	1744	-
proposic	-	13519	19	108	-
prueba_tgraph	-	558	0	1440	-
prueba_comp	-	739	0	5532	-
prueba_bdd	-	791	0	1460	-
prop_prove	-	468	0	932	-
prop_prove_abd	-	475	0	1256	-
prop_tgraph	-	481	0	1328	-
prop_comp	-	597	0	5520	-
prop_bdd	-	517	0	1296	-

Otter: tipo3_7
Res: Prueba
Tpo: 0 segundos

Experimento tipo4_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1938	0	2572	-
prove_abd_oc	-	1938	0	5884	-
proposic	-	16597	29	108	-
prueba_tgraph	-	1702	0	3160	-
prueba_comp	-	2268	0	10348	-
prueba_bdd	-	2877	0	3320	-
prop_prove	-	1056	0	2404	-
prop_prove_abd	-	1070	0	3580	-
prop_tgraph	-	1040	11	2632	-
prop_comp	-	1233	0	10340	-
prop_bdd	-	1113	0	2488	-

Otter: tipo4_7

Res: Prueba

Tpo: 0 segundos

Experimento tipo5_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	-	1042381	1529	108	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	-	741343	241	92060	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-

prop_comp	Tiempo	-	-	-	-
prop_bdd	-	35732	31	52636	-

Otter: tipo5_7
Res: Prueba
Tpo: 57 segundos

Experimento tipo6_7

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Sat	1031116	1509	96	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Sat	116521	49	96	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Sat	27119	20	96	-

Otter: tipo6_7
Res: Satisf
Tpo: 51 segundos

Experimento dB_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Tiempo	-	-	-	-
prove_abd_oc	Tiempo	-	-	-	-
proposic	Sat	54912	81	96	-
prueba_tgraph	Tiempo	-	-	-	-
prueba_comp	Tiempo	-	-	-	-

prueba_bdd	Tiempo	-	-	-	-
prop_prove	Tiempo	-	-	-	-
prop_prove_abd	Sat	3260	0	96	-
prop_tgraph	Tiempo	-	-	-	-
prop_comp	Sat	4006	9	96	-
prop_bdd	Sat	4301	0	96	-

Otter: dB_8
 Res: Satisf
 Tpo: 5 segundos

Experimento palomar_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Pila local	-	-	-	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Pila local	-	-	-	-

Otter: palomar_8
 Res: Límite de tiempo
 Tpo: 265 segundos

Experimento franzen_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	1725	0	2800	-

prove_abd_oc	-	1725	0	5560	-
proposic	-	96720	140	108	-
prueba_tgraph	-	1547	0	3240	-
prueba_comp	-	2037	0	9520	-
prueba_bdd	-	2491	0	3204	-
prop_prove	-	1001	10	2324	-
prop_prove_abd	-	1010	0	2936	-
prop_tgraph	-	1018	0	2936	-
prop_comp	-	1201	0	9580	-
prop_bdd	-	1080	0	2856	-

Otter: franzen_8
 Res: Prueba
 Tpo: 0 segundos

Experimento schwicht_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	11474	10	15884	-
prove_abd_oc	-	11474	11	58196	-
proposic	-	96633	140	108	-
prueba_tgraph	-	8823	10	9340	-
prueba_comp	-	11977	9	27108	-
prueba_bdd	-	4490	0	3640	-
prop_prove	-	4786	0	11748	-
prop_prove_abd	-	5033	10	25164	-
prop_tgraph	-	4212	0	5364	-
prop_comp	-	4457	11	19904	-
prop_bdd	-	1310	0	2936	-

Otter: schwicht_8
 Res: Prueba

Tpo: 0 segundos

Experimento kk_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	12045	10	11304	-
prove_abd_oc	-	12045	11	34368	-
proposic	Tiempo	-	-	-	-
prueba_tgraph	-	9025	0	9856	-
prueba_comp	-	12886	11	30156	-
prueba_bdd	-	4920417	1699	621164	-
prop_prove	-	4860	0	9456	-
prop_prove_abd	-	4962	10	16164	-
prop_tgraph	-	4623	0	7584	-
prop_comp	-	5072	11	26892	-
prop_bdd	-	415187	470	75656	-

Otter: kk_8

Res: Prueba

Tpo: 0 segundos

Experimento equiv_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	230456	131	282232	-
prove_abd_oc	-	230456	300	724000	-
proposic	-	30456	41	108	-
prueba_tgraph	-	180244	89	199968	-
prueba_comp	-	249049	180	717084	-
prueba_bdd	-	857548	311	277940	-
prop_prove	-	81939	79	160328	-
prop_prove_abd	-	83987	80	289268	-

prop_tgraph	-	78344	81	141632	-
prop_comp	-	89297	120	595100	-
prop_bdd	-	105941	100	122304	-

Otter: equiv_8
 Res: Límite de tiempo
 Tpo: 600 segundos

Experimento tipo1_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	722	0	1272	-
prove_abd_oc	-	722	0	2688	-
proposic	-	30393	39	108	-
prueba_tgraph	-	694	0	1664	-
prueba_comp	-	935	10	5820	-
prueba_bdd	-	1164	0	1592	-
prop_prove	-	541	0	1080	-
prop_prove_abd	-	549	0	1452	-
prop_tgraph	-	556	0	1536	-
prop_comp	-	683	11	5908	-
prop_bdd	-	598	0	1496	-

Otter: tipo1_8
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo2_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	2427	0	2908	-
prove_abd_oc	-	2427	0	6796	-
proposic	-	30456	49	108	-
prueba_tgraph	-	2087	0	3564	-

prueba_comp	-	2804	0	11816	-
prueba_bdd	-	3634	0	3756	-
prop_prove	-	1223	11	2708	-
prop_prove_abd	-	1239	0	4076	-
prop_tgraph	-	1203	0	2948	-
prop_comp	-	1423	0	11780	-
prop_bdd	-	1280	0	2776	-

Otter: tipo2_8
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo3_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	-	643	0	1072	-
prove_abd_oc	-	643	0	1984	-
proposic	-	30456	39	108	-
prueba_tgraph	-	650	0	1632	-
prueba_comp	-	862	0	6364	-
prueba_bdd	-	940	0	1656	-
prop_prove	-	539	0	1048	-
prop_prove_abd	-	547	0	1420	-
prop_tgraph	-	554	0	1504	-
prop_comp	-	687	0	6332	-
prop_bdd	-	595	0	1464	-

Otter: tipo3_8
 Res: Prueba
 Tpo: 0 segundos

Experimento tipo4_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

prove_oc	Sat	11840	10	96	-
prove_abd_oc	Sat	11840	11	96	-
proposic	Sat	12797	19	96	-
prueba_tgraph	Sat	8764	0	96	-
prueba_comp	Sat	12312	10	96	-
prueba_bdd	Tiempo	-	-	-	-
prop_prove	Sat	2275	0	96	-
prop_prove_abd	Sat	1000	0	96	-
prop_tgraph	Sat	2050	0	96	-
prop_comp	Sat	1244	0	96	-
prop_bdd	Sat	1115	0	96	-

Otter: tipo4_8

Res: Satisf

Tpo: 0 segundos

Experimento tipo5_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	-	4741902	7511	108	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	-	3650687	1180	219052	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	-	86004	70	118700	-

Otter: tipo5_8
 Res: Límite de tiempo
 Tpo: 600 segundos

Experimento tipo6_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prove_oc	Pila local	-	-	-	-
prove_abd_oc	Pila local	-	-	-	-
proposic	Sat	4716227	7460	96	-
prueba_tgraph	Pila local	-	-	-	-
prueba_comp	Pila local	-	-	-	-
prueba_bdd	Sat	435369	159	96	-
prop_prove	Pila local	-	-	-	-
prop_prove_abd	Tiempo	-	-	-	-
prop_tgraph	Pila local	-	-	-	-
prop_comp	Tiempo	-	-	-	-
prop_bdd	Sat	62150	41	96	-

Otter: tipo6_8
 Res: Límite de tiempo
 Tpo: 600 segundos

J.5. Comparación con Mace, zChaff y Anldp

Experimento dB_91

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	7900378	9980	372556	-

Mace4: 1.86 seg.
 zChaff: 0.014 seg.
 Anldp: 0.21 seg.

Experimento dB_101

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

```

| prop_bdd | - | 11600598 | 14780 | 457768 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 2.38
zChaff: 0.018
Anldp: 0.26

```

Experimento dB_111

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_bdd | - | 16473968 | 21030 | 550744 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 3.18
zChaff: 0.023
Anldp: 0.29

```

Experimento palomar_4

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_bdd | - | 282736 | 320 | 125912 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 0.00
zChaff: 0.001
Anldp: 0.00

```

Experimento palomar_5

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_bdd | Pila local | - | - | - | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 0.02
zChaff: 0.004
Anldp: 0.01

```

Experimento palomar_6

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_bdd | Pila local | - | - | - | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 0.07
zChaff: 0.020
Anldp: 0.01

```

Experimento schwicht_200

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_bdd | - | 2923294 | 3819 | 306304 | - |
+-----+-----+-----+-----+-----+-----+

```

Mace4: 0.07
 zChaff: 0.001
 Anldp: 0.00

Experimento schwicht_300

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	9564894	16030	1507196	-

Mace4: 0.14
 zChaff: 0.003
 Anldp: 0.00

Experimento schwicht_400

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	22326494	29509	1298304	-

Mace4: 0.22
 zChaff: 0.004
 Anldp: 0.00

Experimento tipo2_800

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	766352	1209	281280	-

Mace4: 0.78
 zChaff: 0.007
 Anldp: 0.02

Experimento tipo2_1000

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	1157952	1901	351680	-

Mace4: 1.30
 zChaff: 0.010
 Anldp: 0.02

Experimento tipo2_1200

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	error	-	-	-	-

Mace4: 2.33
 zChaff: 0.013

Anldp: 0.02

Experimento tipo5_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	-	86004	80	118688	-

Mace4: 0.25

zChaff: 0.005

Anldp: 0.01

Experimento tipo5_9

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	error	-	-	-	-

Mace4: 1.40

zChaff: 0.014

Anldp: 0.01

Experimento tipo5_10

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	error	-	-	-	-

Mace4: 8.82

zChaff: 0.029

Anldp: 0.03

Experimento tipo6_8

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	Sat	62150	70	84	-

Mace4: 0.24

zChaff: 0.00

Anldp: 0.01

Experimento tipo6_9

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	error	-	-	-	-

Mace4: 1.41

zChaff: 0.00

Anldp: 0.00

Experimento tipo6_10

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_bdd	error	-	-	-	-

Mace4: 8.81
zChaff: 0.00
Anldp: 0.01

Experimento franzen_800

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	1045253	1919	214568	-

Mace4: 12.58
zChaff: 0.007
Anldp: bug de Mace2

Experimento franzen_1200

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	2287853	4290	321768	-

Mace4: 41.64
zChaff: 0.010
Anldp: bug de Mace2

Experimento franzen_1600

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	4010453	7650	428968	-

Mace4: 102.68
zChaff: 0.015
Anldp: bug de Mace2

Experimento tipo1_1200

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	1514381	2910	158412	-

Mace4: 39.66
zChaff: 0.010
Anldp: bug de Mace2

Experimento tipo1_1600

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	2659181	5011	211212	-

Mace4: 93.71

zChaff: 0.013

Anldp: bug de Mace2

Experimento tipo1_2000

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	4123981	7750	264012	-

Mace4: 192.24

zChaff: 0.016

Anldp: bug de Mace2

Experimento tipo3_800

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	-	370799	771	105312	-

Mace4: 0.69

zChaff: 0.006

Anldp: 0.00

Experimento tipo3_1200

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	error	-	-	-	-

Mace4: 1.59

zChaff: 0.008

Anldp: 0.00

Experimento tipo3_1600

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove	error	-	-	-	-

Mace4: 3.76

zChaff: 0.012

Anldp: 0.01

Experimento tipo4_601

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
--------	-------	-------------	---------	-------	--------

```

| prop_prove | - | 808896 | 1490 | 225452 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 0.47
zChaff: 0.006
Anldp: 0.01

```

Experimento tipo4_1001

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_prove | - | 2148096 | 4069 | 375852 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 1.42
zChaff: 0.010
Anldp: 0.02

```

Experimento tipo4_1401

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_prove | error | - | - | - | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 3.59
zChaff: 0.015
Anldp: 0.03

```

Experimento dB_200

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_prove_abd | Sat | 3819484 | 5920 | 84 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 15.37
zChaff: 0.042
Anldp: 0.99

```

Experimento dB_300

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_prove_abd | Sat | 11569234 | 17831 | 84 | - |
+-----+-----+-----+-----+-----+-----+
Mace4: 54.82
zChaff: 0.103
Anldp: bug de Mace2

```

Experimento dB_400

```

+-----+-----+-----+-----+-----+-----+
| MÉTODO | ERROR | INFERENCIAS | MILISEG | BYTES | LÍMITE |
+-----+-----+-----+-----+-----+-----+
| prop_prove_abd | Sat | 25878984 | 40910 | 84 | - |
+-----+-----+-----+-----+-----+-----+

```

Mace4: max_megs.
 zChaff: 0.167
 Anldp: bug de Mace2

Experimento tipo4_600

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove_abd	Sat	342584	570	84	-

Mace4: 0.47
 zChaff: 0.00
 Anldp: 0.01

Experimento tipo4_1000

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove_abd	Sat	870984	1490	84	-

Mace4: 1.43
 zChaff: 0.00
 Anldp: 0.02

Experimento tipo4_1100

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_prove_abd	error	-	-	-	-

Mace4: 1.82
 zChaff: 0.001
 Anldp: 0.02

Experimento equiv_15

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
proposic	-	7078540	12659	72	-

Mace4: Falta memoria.
 zChaff: No pudo pasarse a DIMACS.
 Anldp: No pudo pasarse a DIMACS.

Experimento equiv_16

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
proposic	-	15073964	27750	84	-

Mace4: Falta memoria.
 zChaff: No pudo pasarse a DIMACS.

Anldp: No pudo pasarse a DIMACS.

Experimento equiv_17

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
proposic	-	31982320	60230	96	-

Mace4: Falta memoria.

zChaff: No pudo pasarse a DIMACS.

Anldp: No pudo pasarse a DIMACS.

Experimento kk_80

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_tgraph	-	1174299	1550	169340	-

Mace4: 86.68

zChaff: 0.011

Anldp: 0.46

Experimento kk_120

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_tgraph	-	3779639	4879	369352	-

Mace4: 445.36

zChaff: 0.023

Anldp: 1.31

Experimento kk_160

MÉTODO	ERROR	INFERENCIAS	MILISEG	BYTES	LÍMITE
prop_tgraph	-	8754579	11420	646152	-

Mace4: 1386.44

zChaff: 0.039

Anldp: 2.79