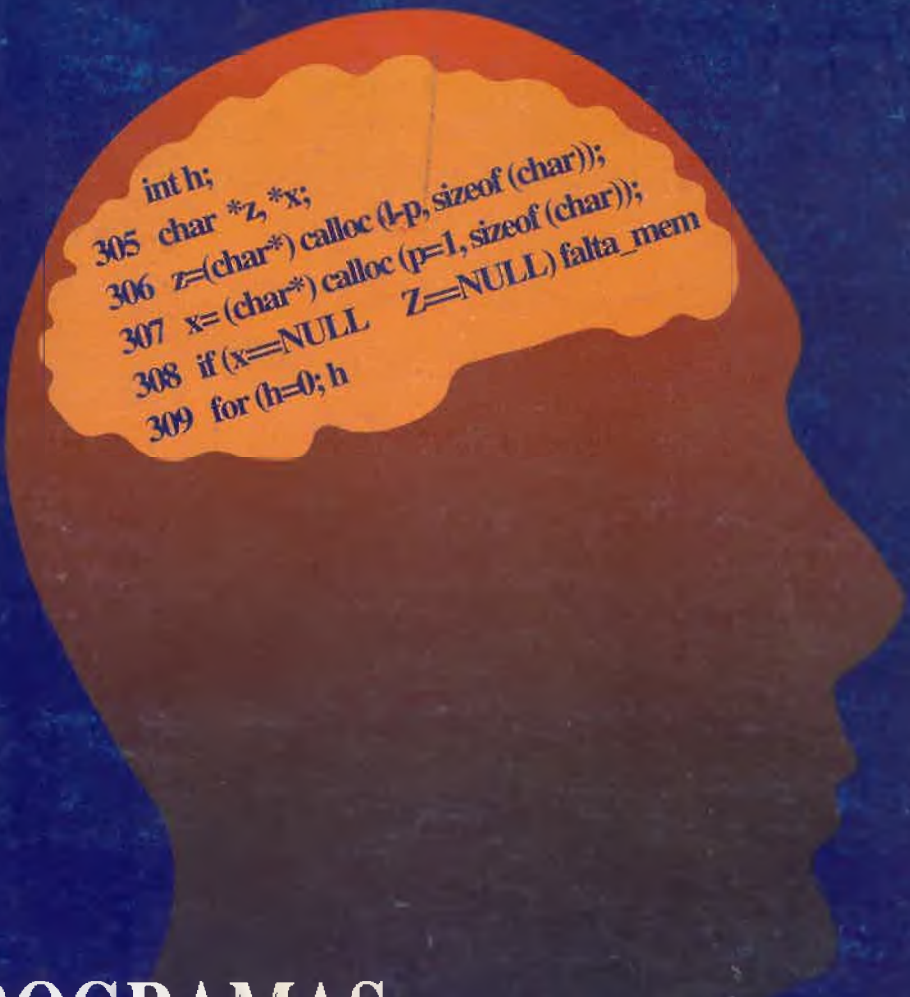
ANTONIO FRIAS DELGADO



PROGRAMAS PARA DEDUCCION AUTOMATIZADA EN LOGICA PROPOSICIONAL

ANTONIO FRIAS DELGADO

Programas para deducción automatizada en lógica proposicional

Universidad de Cádiz
Servicio de Publicaciones
1992

Diseño portada: Juan Candón
Edita: Servicio de Publicaciones. Universidad de Cádiz
ISBN: 84/7786/082/3
Depósito legal: CA-447-92
Imprime: INGRASA. Artes Gráficas. Puerto Real (Cádiz)

INTRODUCCION

El proyecto de controlar el razonamiento y reducirlo a una segura operación de cálculo cuenta ya con siglos; pero no cabe duda de que el desarrollo de los ordenadores le ha dado nuevos impulsos. En la actualidad el objetivo del *razonamiento automatizado* es escribir programas de ordenador que nos ayuden en la solución de problemas en los que se requiera razonar [32].

Aunque el antiguo proyecto leibniziano, por citar el de uno de sus más ilustres defensores [14], en toda su amplitud parece estar lejos todavía de su realización completa —el pensamiento humano es de tal complejidad que no puede esperarse su codificación mediante un conjunto, simple y operativo a la vez, de reglas—, es mucho lo que ya se ha logrado en la automatización del razonamiento. Por lo que respecta a las leyes más generales y elementales del razonamiento deductivo, admitiendo que sean éstas las de la lógica proposicional bivalente clásica, podemos recogerlas completamente en métodos manejables y seguros. A partir de la prueba de completud de la lógica proposicional de E. L. Post [25], y debido a su carácter constructivo, está claro que este tipo de razonamiento puede efectuarse de un modo mecánico, sin necesidad de emplear estrategias de búsqueda y tanteo.

La automatización del razonamiento en sentido estricto empieza en la década de los 50: Davis en 1954; Newell, Shaw y Simon en 1955; Prawitz y Gerlenter en 1957. El lector interesado puede consultar [5], [6], [11], [20], [22], [26], [32], [33]; sobre todo, la importante recopilación de los trabajos clásicos en este campo [29], [30].

En la prueba automatizada de teoremas hay, entre otras muchas, dos sensibilidades e intereses. De una parte, cabe el objetivo de probar simplemente que algo sea un teorema, no importa del todo por qué métodos. De otra parte, cabe centrar la atención básicamente en el modo de la prueba, en las estrategias que se siguen, etc., de forma que dichas estrategias se asemejen lo más posible a las que emplearía un lógico experto. Históricamente pueden encontrarse trabajos más cercanos a una sensibilidad o a otra [11].

Nuestro objetivo al escribir los programas que aquí se ofrecen para deducción automatizada de teoremas en lógica proposicional ha sido múltiple:

1. *Ofrecer al lector interesado pero no especialista unas técnicas no siempre de fácil acceso.* Incluso la bibliografía especializada, las más de las veces, sólo da cuenta de las líneas generales de construcción de los programas de prueba automatizada. No cabe duda de que hay partes tediosas y elementales en un programa de estas características, que no añaden nada al algoritmo de prueba en sí mismo y que el estudioso interesado quizás pueda completar por sí mismo. Hemos creído, no obstante, que para un público más amplio puede ser de interés tener el programa desarrollado en su totalidad.

2. Estando los problemas teóricos resueltos desde hace tiempo, nos hemos propuesto desarrollar unos programas en los que se atienda especialmente a la estrategia seguida

en la demostración de los teoremas. Centrándonos en los cálculos que creemos de mayor uso en nuestro país —[15], [23]—, nos ha interesado construir no un algoritmo de demostración cualquiera, sino precisamente *la estrategia deductiva que más se aproximara en la mayoría de los casos a la deducción ofrecida por un lógico experto.* En este sentido, nuestros programas son lo más parecido a sistemas expertos en estos cálculos.

3. Por su construcción centrada en las estrategias deductivas, creeemos que los programas pueden cumplir también una *función tutorial,* de ayuda en el aprendizaje de la lógica elemental.

Como acabamos de indicar, nos hemos centrado en unos cálculos por la razón de ser los más difundidos entre nosotros. Son, además, cálculos en los que las deducciones no son completamente mecánicas en virtud de las reglas de inferencia, por lo que hay cierto margen para la inventiva, la exploración y la heurística. Como contraposición, hemos escrito programas para métodos analíticos de demostración ([4], [18], [31]). Estos métodos, completamente mecánicos en virtud de sus reglas de inferencia, gozan también de amplia aceptación, precisamente por esta característica.

Los programas están escritos en Lenguaje C y han sido compilados y ejecutados sin error en Turbo C de Borland versión 2.0. Nos ha parecido que la potencia, flexibilidad, portabilidad y difusión del Turbo C lo hacía especialmente indicado para nuestra tarea.

En I se dan las características y reglas de los cálculos que vamos a usar. En II se exponen las estrategias deductivas para cada uno. En III se comentan los programas. En IV se valora su rendimiento. En V se da una mínima bibliografía. Las referencias entre corchetes son a las entradas de la bibliografía.

I. CALCULOS USADOS

I.1. El lenguaje de la lógica proposicional

La lógica se ocupa de la validez de nuestros razonamientos. Su parte más elemental es la lógica proposicional, llamada así porque es la proposición en su totalidad la que se toma, junto con los conectores lógicos, como constituyente.

I.1.1. *Símbolos*

De un modo estándar la lógica proposicional consta, pues, de:

a. *Variables proposicionales* —variables que se refieren a proposiciones cualesquiera—, que pueden ser evaluadas como verdaderas o falsas.

Usaremos como símbolos para variables proposicionales:

p1, p2, p3, p4,...

Lo importante aquí es que podamos contar con un número infinito numerable de tales símbolos.

b. *Conectores lógicos* cuyo significado, por así decirlo, es invariante. El significado de un conector lógico —por ejemplo, "y"— puede entenderse como el peculiar modo de componer veritativamente una proposición compleja a partir de proposiciones simples. En ese sentido, puede admitirse que el significado de los conectores lógicos viene determinado por las reglas que especifican su introducción y su eliminación en el proceso deductivo o por sus tablas de verdad.

Usaremos como símbolos para los conectores:

$\neg$ para la negación —"no"—;

$\wedge$ para la conjunción —"y"—;

$\vee$ para la disyunción —"o" inclusiva—;

$\rightarrow$ para el condicional o implicación material —"si... entonces..."—;

$\leftrightarrow$ para el bicondicional —"... si y sólo si ..."—.

Los programas que veremos después admiten la posibilidad de introducir los conectores mediante letras sólo para facilitar la escritura de las fórmulas. Respectivamente:

N, n;

Y, y, K;

O, o, A;

I, i, C;

B, b, E.

N, K, A, C, E son símbolos estándar en la notación polaca.

Usaremos paréntesis para indicar el alcance de los conectores siguiendo las reglas de dominancia usuales: $\rightarrow$,$\leftrightarrow$; $\wedge$, $\vee$; $\neg$. Es sabido que los paréntesis pueden eliminarse,

usando por ejemplo la notación polaca, pero la lectura de las fórmulas parece resultar menos intuitiva.

I.1.2. *Reglas de formación de fórmulas*

Qué sea una fórmula viene definido por las reglas:
a. Una variable proposicional es una fórmula.
b: Si α y β son fórmulas, también lo son: $\neg \alpha$, $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \rightarrow \beta$, $\alpha \leftrightarrow \beta$.

Demostrar la validez de un razonamiento consiste en probar que la conclusión es obtenible de acuerdo a un cierto conjunto de axiomas y/o reglas que hemos especificado. Existen diversos métodos de deducción, aunque todos ellos son equivalentes en el sentido de que lo que se deduce en uno puede deducirse en los demás. Entenderemos por cálculo, en este contexto, un procedimiento deductivo que carece de axiomas. Un cálculo consta, pues, sólo de reglas. El procedimiento de los cálculos parece más cercano a nuestro razonar que el de los sistemas axiomáticos; pero esta diferencia no afecta en nada al valor lógico de unos y otros. Los cálculos para los que hemos construído programas de deducción automática serán denominados en adelante G [15], M [23], J [18]. Remitimos a esta bibliografía para un estudio más detallado y completo de los mismos. Aquí sólo daremos la información indispensable para desarrollar después las estrategias deductivas y los programas.

I.2. El cálculo G

En el cálculo G, una deducción de la fórmula β a partir de las premisas $\alpha_1,...,\alpha_n$ ($n \geq 0$) es una serie finita de fórmulas tales que:
a. Cada fórmula es una premisa o es un supuesto o se obtiene por las reglas de inferencia a partir de otras fórmulas anteriores que estén utilizables.
b. β es la última fórmula y todos los supuestos están clausurados.

A la hora de efectuar una deducción, escribimos las fórmulas en líneas numeradas por la izquierda consecutivamente. Indicamos después si es un supuesto ($\lceil$), clausura de un supuesto ($\lfloor$) o un bloque de fórmulas que constituyen una deducción subsidiaria y queda, por tanto, inutilizable para inferencias posteriores ($\mid$). Escribimos a continuación la fórmula. Por último indicamos la regla de inferencia que justifica la obtención de la fórmula. Las premisas se justifican como tales premisas; los supuestos no han de justificarse. En una deducción pueden suponerse cualesquiera fórmulas cuando se desee.

Las reglas de inferencia son dos para cada conector: una que lo introduce (I) y otra que lo elimina (E).

$$I\wedge\ \frac{\alpha\quad \beta}{\alpha\wedge\beta} \qquad E\wedge\ \frac{\alpha\wedge\beta}{\alpha\quad \beta} \qquad I\vee\ \frac{\alpha}{\alpha\vee\beta} \qquad E\vee\ \frac{\alpha\vee\beta\quad \neg\alpha}{\beta}$$

$$I\rightarrow\ \frac{[\alpha\ \ \beta]}{\alpha\rightarrow\beta} \qquad E\rightarrow\ \frac{\alpha\rightarrow\beta\quad \alpha}{\beta} \qquad I\leftrightarrow\ \frac{\alpha\rightarrow\beta\quad \beta\rightarrow\alpha}{\alpha\leftrightarrow\beta} \qquad E\leftrightarrow\ \frac{\alpha\leftrightarrow\beta}{\alpha\rightarrow\beta\quad \beta\rightarrow\alpha}$$

$$I\neg\ \frac{[\alpha\ \ \beta\wedge\neg\beta]}{\neg\alpha} \qquad E\neg\ \frac{\neg\neg\alpha}{\alpha}$$

Se observará una diferencia entre las reglas I→, I ¬ y las restantes. La introducción del condicional y la introducción de la negación son llamadas, a veces, reglas estructurales en el sentido de que especifican las condiciones en que pueden introducirse un condicional y una negación en el proceso deductivo.

I.3. El cálculo M

El cálculo M mantiene analogías con el anterior aunque las deducciones ofrezcan diferente aspecto.

Definamos en primer lugar las líneas de una deducción como:

a. Líneas utilizables: las constituídas por una fórmula sola o una fórmula precedida de un interrogante tachado (?̶).

b. Líneas marcadas: las contituídas por una línea utilizable precedida de n (n≥1) marcas |.

c. Líneas interrogadas: las constituídas por una fórmula precedida de un interrogante no tachado (?).

Las reglas de construcción de las deducciones son:

a. Para cualquier fórmula α puede escribirse como línea

? α

b. Si α es una premisa puede escribirse como línea

α

c. Si ? α es una línea ya escrita, como línea inmediatamente siguiente puede escribirse

¬α

d. Si ? α → β es una línea ya escrita, como línea inmediatamente siguiente puede escribirse

α

e. Si α es inferible de líneas utilizables anteriores por una regla de inferencia, puede escribirse como línea

α

f. Si ? α es la última línea interrogada, pueden marcarse todas las líneas siguientes y tachar el interrogante si ocurre uno de estos casos:

f.1. Una de las líneas utilizables siguientes es α;

f.2. una de las líneas utilizables siguientes es la negación de otra de ellas;

f.3. α es un condicional y una de las líneas utilizables siguientes es su consecuente.

Una deducción de β a partir de las premisas $\alpha 1,...,\alpha n$ ($n \geq 0$) es una serie finita de líneas obtenida conforme a las reglas anteriores de construcción tal que la primera línea es ? β y las restantes líneas están todas marcadas.

Las reglas de inferencia del cálculo M son las I$\wedge$, E$\wedge$, I$\vee$, E$\vee$, I $\leftrightarrow$, E $\leftrightarrow$ anteriores del cálculo G, más:

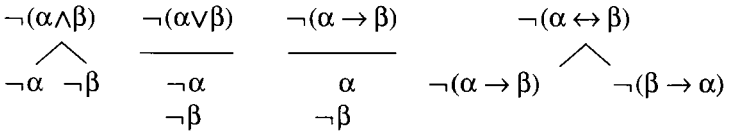
Repetición (R) $\dfrac{\alpha}{\alpha}$ Doble Negación (DN) $\dfrac{\neg\neg\alpha}{\alpha}$ $\dfrac{\alpha}{\neg\neg\alpha}$

Modus ponens (MP) $\dfrac{\alpha \rightarrow \beta \quad \alpha}{\beta}$ Modus tollens (MT) $\dfrac{\alpha \rightarrow \beta \quad \neg\beta}{\neg\alpha}$

I.4. El cáculo J

Un método diferente de demostración es el método que denominaremos analítico y que puede verse en [4], [18], [31]. Las demostraciones suelen tener el tipo de árboles, debido a que algunas reglas abren ramas nuevas en el proceso deductivo. Es un método de prueba indirecta que empieza con la negación de la fórmula a demostrar y la escritura de las premisas, si las hubiera. El procedimiento deductivo consiste en aplicar la regla pertinente a cada fórmula hasta llegar a las variables proposicionales. Si cada rama resultante contiene una contradicción, la fórmula es deducible. El método es analítico porque todas las reglas descomponen una fórmula en otras más elementales; ninguna regla genera fórmulas de más complejidad que las de partida. Las reglas son:

$\dfrac{\neg\neg\alpha}{\alpha}$	$\dfrac{\alpha\wedge\beta}{\alpha \quad \beta}$	$\alpha\vee\beta$ $\wedge$ $\alpha \quad \beta$	$\alpha \rightarrow \beta$ $\wedge$ $\neg\alpha \quad \beta$	$\dfrac{\alpha \leftrightarrow \beta}{\alpha \rightarrow \beta \quad \beta \rightarrow \alpha}$


¬(α∧β)
¬α ¬β
¬(α∨β)
¬α
¬β
¬(α → β)
α
¬β
¬(α ↔ β)
¬(α → β)
¬(β → α)

II. ESTRATEGIAS DE DEDUCCION

La lógica proposicional es completa. Ya hemos indicado que la prueba de Post [25] ofrece a la vez una estrategia que nos permite deducir toda fórmula válida. La estrategia deductiva de Post para sistemas axiomáticos es bastante complicada; para fórmulas relativamente simples es en exceso larga y no puede decirse de ella que constituya una deducción "inteligente". Exige unos pasos intermedios de reducción a fórmulas normales que complican el proceso deductivo.

II.1. Estrategias de deducción en el cálculo J

El método analítico de árboles no necesita ninguna estrategia especial, excepto la que conlleva el propio método deductivo. De hecho, en [18] aparecen algoritmos elementales para deducir fórmulas cuyos pasos son simples y claros. Por proceder sistemáticamente, indicamos la estrategia deductiva en unas operaciones básicas:

a. Escribir las premisas, si la hubiere, y la negación de la conclusión.
b. Aplicar mientras sea posible, es decir, hasta llegar a las variables proposicionales, las reglas de inferencia. Ha de tenerse especialmente en cuenta que tanto la escritura lineal de fórmulas como la apertura de ramas ha de efectuarse en todas las ramas distintas que tengan como nudo la fórmula a la que se aplica la regla de inferencia.
c. Examinar si hay contradicciones en cada rama.

Una rama abierta, en la que no hay contradicciones, basta para que la fórmula no sea deducible de las premisas utilizadas. El significado de una rama abierta es el de una asignación veritativa consistente, un modelo, a las premisas y la negación de la conclusión. (Ver gráfico 1).

Una variante de la estrategia deductiva anterior, que acortará el proceso en bastantes ocasiones, es comprobar la existencia de contradicciones en cada rama abierta inmediatamente después de aplicar una regla. Esto nos evitará ampliar innecesariamente las ramas. Si quedaran ramas abiertas, se repite el paso de la aplicación de reglas (b). Así hasta que todas las ramas estén clausuradas, y la fórmula es válida, o ya no puedan aplicarse más reglas, y la fórmula no es válida. (Ver gráfico 2).

Cuando se trata de una deducción manual, esta variante es preferible. No es tan claro que ocurra lo mismo cuando se trata de pruebas automatizadas. El tiempo utilizado en comprobar si existen contradicciones en cada rama después de la aplicación de cada regla puede superar al empleado en la escritura de todas las ramas en un primer paso y una única comprobación posterior.

Gráfico 1

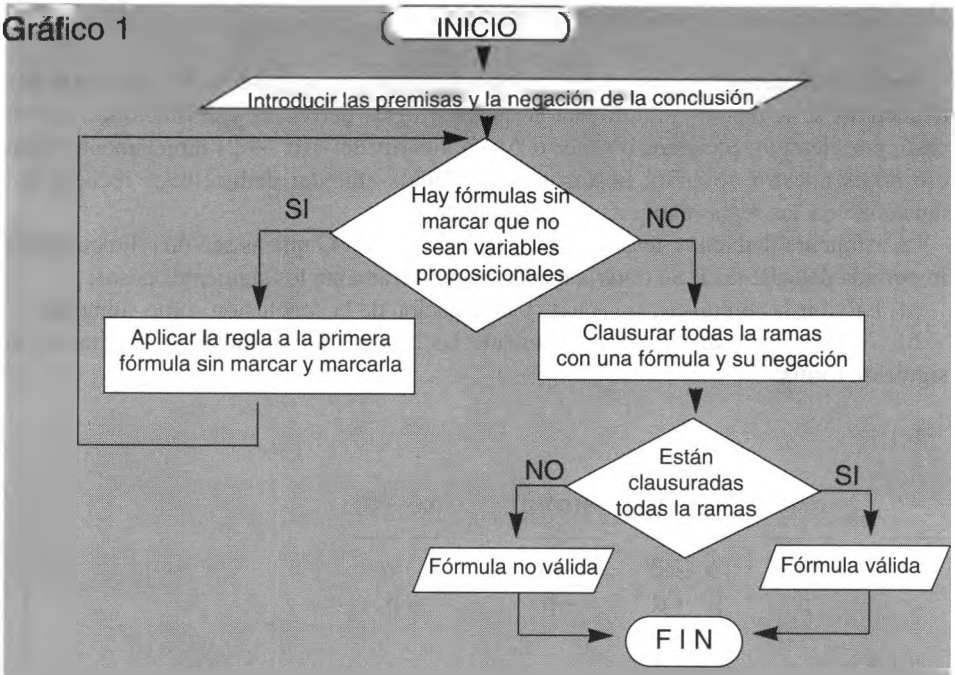
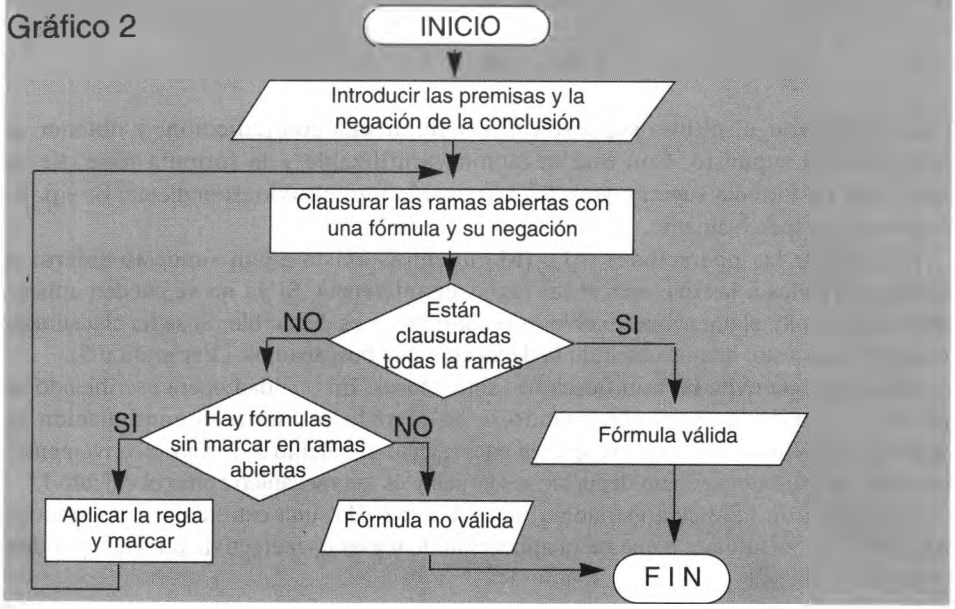


Gráfico 2



II.2. Estrategias de deducción en el cálculo G

Como regla general, tanto para el cálculo G como para el cálculo M, las estrategias deductivas se reducen y simplifican si usamos reglas derivadas y definiciones. En tal caso, por ejemplo, podemos obtener $\alpha \wedge \neg\beta$ a partir de $\neg(\alpha \rightarrow \beta)$ directamente. Pero ese no es nuestro objetivo; nosotros pretendemos efectuar deducciones recurriendo únicamente a las reglas expuestas anteriormente.

La estrategia deductiva más mecánica para el cálculo G quizás sea una directamente importada del cálculo J. Se trataría de dar sistemáticamente los siguientes pasos:

a1. Escribir las premisas, si las hay, y la negación de la conclusión como supuesto.

b1. Aplicar por orden y exhaustivamente las reglas a las fórmulas ya escritas de la siguiente forma:

b.1.1.

$\neg\neg\alpha$	$\alpha\wedge\beta$	$\alpha \leftrightarrow \beta$	$\neg(\alpha\vee\beta)$	$\neg(\alpha \rightarrow \beta)$
α	α β	$\alpha \rightarrow \beta$ $\beta \rightarrow \alpha$	$\neg\alpha$ $\neg\beta$	α $\neg\beta$

b.1.2.

$\alpha\vee\beta$	$\neg(\alpha\wedge\beta)$	$\alpha \rightarrow \beta$	$\neg(\alpha \leftrightarrow \beta)$
$\lceil\ \alpha$	$\lceil\ \neg\alpha$	$\lceil\ \neg\alpha$	$\lceil\ \neg(\alpha \rightarrow \beta)$

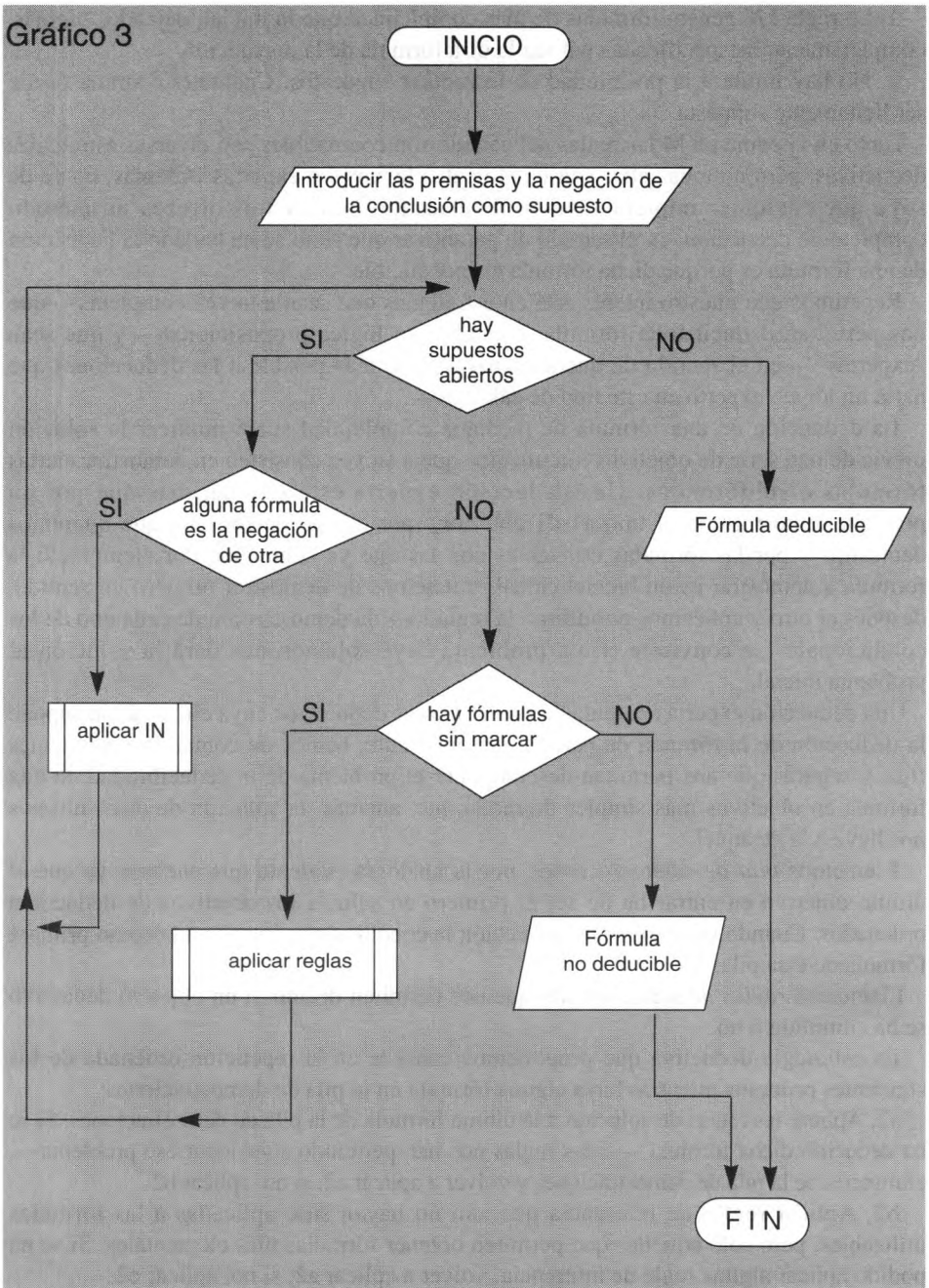
c1. Clausurar el último supuesto, al encontrar una contradicción, y obtener la negación del supuesto. Con esta negación ya utilizable y la fórmula base de la inferencia (la fórmula superior en b.1.2.) obtener la fórmula correspondiente: β, $\neg\beta$, β, $\neg(\beta \rightarrow \alpha)$, respectivamente.

d1. Repetir las operaciones (c1), (b1) mientras exista algún supuesto abierto y queden fórmulas a las que aplicar las reglas de inferencia. Si ya no se pueden aplicar más reglas y hay algún supuesto abierto la fórmula no es deducible. Si se ha clausurado el primer supuesto, hemos obtenido la deducción que buscábamos. (Ver gráfico 3).

Esta estrategia evita las ramificaciones simultáneas. En realidad opera escribiendo la primera rama de la izquierda; cuando se ha cerrado se escribe a continuación la segunda, etc. Puede demostrarse que es equivalente al cálculo J. Como inconveniente, este tipo de estrategia genera deducciones largas y es tan mecánico como el cálculo J.

Dejando a un lado esta estrategia mecánica (a1-d1), una estrategia deductiva no orientada que se limitara a una mera aplicación de reglas es inefectiva al menos por dos razones:

Gráfico 3


INICIO
Introducir las premisas y la negación de la conclusión como supuesto
hay supuestos abiertos
SI
NO
alguna fórmula es la negación de otra
SI
NO
Fórmula deducible
aplicar IN
hay fórmulas sin marcar
SI
NO
aplicar reglas
Fórmula no deducible
F I N

1. La regla I∧ genera fórmulas de más complejidad que la inicial, dejando, además, completamente inespecificada cuál sea la otra fórmula de la disyunción.

2. No hay límite a la posibilidad de introducir supuestos. Cualquier fórmula puede ser lícitamente supuesta.

Tanto en G como en M las reglas del cálculo son compatibles con diversas estrategias deductivas, pero no todas ellas son igualmente elegantes o expertas. Además, no va de suyo que cualquier implementación de este tipo de cálculos ofrezca un método completo de deducción, en el sentido de garantizar que si no se ha hallado la deducción de una fórmula es porque dicha fórmula es indeducible.

Repetimos que nuestro interés está en estrategias que sean a la vez completas —que nos permitan deducir toda fórmula deducible en lógica proposicional— y que sean "expertas" —en el sentido de que se aproximen lo más posible a las deducciones que haría un lógico experto en este tipo de cálculos—.

La deducción de una fórmula de mediana complejidad suele implicar la solución previa de una serie de objetivos intermedios que a su vez consisten en demostrar ciertas fórmulas o subfórmulas. Una deducción experta está guiada, más que por un procedimiento rígido del tipo a1-d1 anterior, por el tipo de fórmula que queremos demostrar y por las fórmulas utilizables con las que ya contamos. Por ejemplo, si la fórmula a demostrar es un bicondicional, trataremos de demostrar primero un sentido, después el otro y aplicamos por último la regla I ↔; la demostración de cada uno de los condicionales se convierte en un problema cuya solución nos dará la solución al problema inicial.

Una deducción experta acumula ciertos objetivos deductivos cuya eliminación supone la deducción de la fórmula de partida. Naturalmente, hemos de contar con estrategias fijas y seguras que nos permitan descomponer el problema de la deducibilidad de una fórmula en objetivos más simples de modo que, además, la solución de éstos últimos nos lleve a la de aquél.

Llamemos *pila de demostraciones,* por la analogía evidente que encierra ya que el último objetivo en entrar ha de ser el primero en salir, a los objetivos de deducción ordenados. Cuando iniciamos una deducción la conclusión es introducida como primera fórmula de esta pila.

Llamemos *reglas de solución* a las que nos permiten decidir si un objetivo deductivo se ha cumplido o no.

La estrategia deductiva que proponemos consiste en la repetición ordenada de los siguientes procesos mientras haya alguna fórmula en la pila de demostraciones:

a2. Aplicar las reglas de solución a la última fórmula de la pila de demostraciones. Si se ha deducido dicha fórmula —si las reglas nos han permitido solucionar ese problema—, eliminarla de la pila de demostraciones y volver a aplicar a2; si no, aplicar b2.

b2. Aplicar reglas de inferencia que aún no hayan sido aplicadas a las fórmulas utilizables, pero sólo aquéllas que permiten obtener fórmulas más elementales. Si se ha podido aplicar alguna regla de inferencia, volver a aplicar a2; si no, aplicar c2.

c2. Aplicar una estrategia de subojetivos. Si no se puede escribir ninguna nueva fórmula utilizable ni se ha incorporado ninguna nueva fórmula en la pila de demostraciones, la fórmula de partida es indemostrable. (Ver gráfico 4).

Como se observará inmediatamente, la estrategia de subojetivos es la clave para que podamos garantizar que la fórmula es indemostrable, no sólo que hemos sido incapaces de encontrar una deducción para ella.

Las reglas de solución están jerarquizadas en tres niveles:

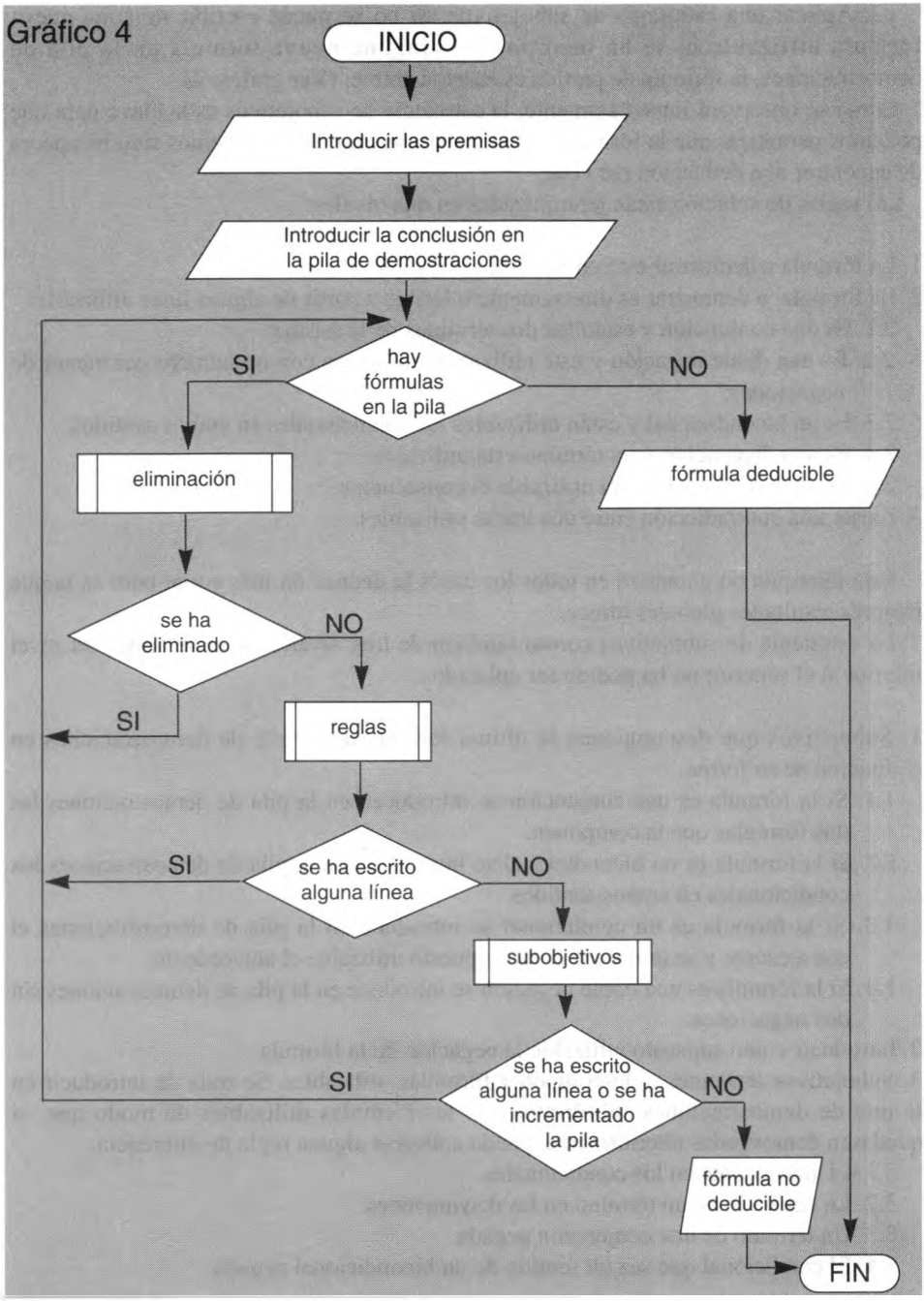
1. La fórmula a demostrar está en una línea utilizable.
2. La fórmula a demostrar es directamente inferible a partir de alguna línea utilizable:
   2.1. Es una conjunción y están los dos términos de la misma.
   2.2. Es una doble negación y está utilizable la fórmula con un número par menor de negaciones.
   2.3. Es un bicondicional y están utilizables los condicionales en ambos sentidos.
   2.4. Es una disyunción y un término está utilizable.
   2.5. Es un condicional y está utilizable el consecuente.
3. Existe una contradicción entre dos líneas utilizables.

Esta jerarquía no garantiza en todos los casos la deducción más corta; pero es la que mejores resultados globales ofrece.

La estrategia de subojetivos consta también de tres niveles. Sólo se aplica un nivel inferior si el superior no ha podido ser aplicado.

1. Subojetivos que descomponen la última fórmula de la pila de demostraciones en función de su forma.
   1.1. Si la fórmula es una conjunción se introducen en la pila de demostraciones las dos fórmulas que la componen.
   1.2. Si la fórmula es un bicondicional se introducen en la pila de demostraciones los condicionales en ambos sentidos.
   1.3. Si la fórmula es un condicional se introduce en la pila de demostraciones el consecuente y se introduce como supuesto utilizable el antecedente.
   1.4. Si la fórmula es una doble negación se introduce en la pila de demostraciones sin dos negaciones.
2. Introducir como supuesto utilizable la negación de la fórmula.
3. Subojetivos tendentes a descomponer fórmulas utilizables. Se trata de introducir en la pila de demostraciones subfórmulas de las fórmulas utilizables de modo que, si quedasen demostradas ulteriormente, pueda aplicarse alguna regla de inferencia.
   3.1. El antecedente en los condicionales.
   3.2. La negación de un término en las disyunciones.
   3.3. Un término de una conjunción negada.
   3.4. El condicional que sea un sentido de un bicondicional negado.

Gráfico 4


INICIO
Introducir las premisas
Introducir la conclusión en la pila de demostraciones
hay fórmulas en la pila
SI
NO
eliminación
fórmula deducible
se ha eliminado
NO
SI
reglas
se ha escrito alguna línea
SI
NO
subobjetivos
se ha escrito alguna línea o se ha incrementado la pila
SI
NO
fórmula no deducible
FIN

Se puede ver que esta estrategia es completa; es decir que garantiza la no deduciblidad de una fórmula en cuyo proceso deductivo ya no se puede seguir aplicando ninguna regla de las anteriores y queden fórmulas en la pila de demostraciones. En realidad, la regla 3 anterior escribe el equivalente a una rama en el cálculo J que permanece abierta.

II.3. Estrategias de deducción en el cálculo M

Las estrategias deductivas para el cálculo M guardan parecido con las del cálculo G. La construcción de la deducción permite evitar el distinguir entre fórmulas utilizables escritas y subojetivos de demostración —la pila de demostraciones anterior—. Estos pueden reflejarse en la deducción como fórmulas interrogadas sin más. El cumplimiento de un objetivo viene dado por el tachado de la interrogación.

La estrategia general es semejante a la anterior. Consiste en aplicar sistemáticamente, mientras haya algún interrogante sin tachar, los siguientes pasos:

a. Buscar la última fórmula interrogada. Si está utilizable, tachar y volver a aplicar (a).

b. Si la última fórmula interrogada es un condicional y está utilizable el consecuente, tachar y aplicar (a).

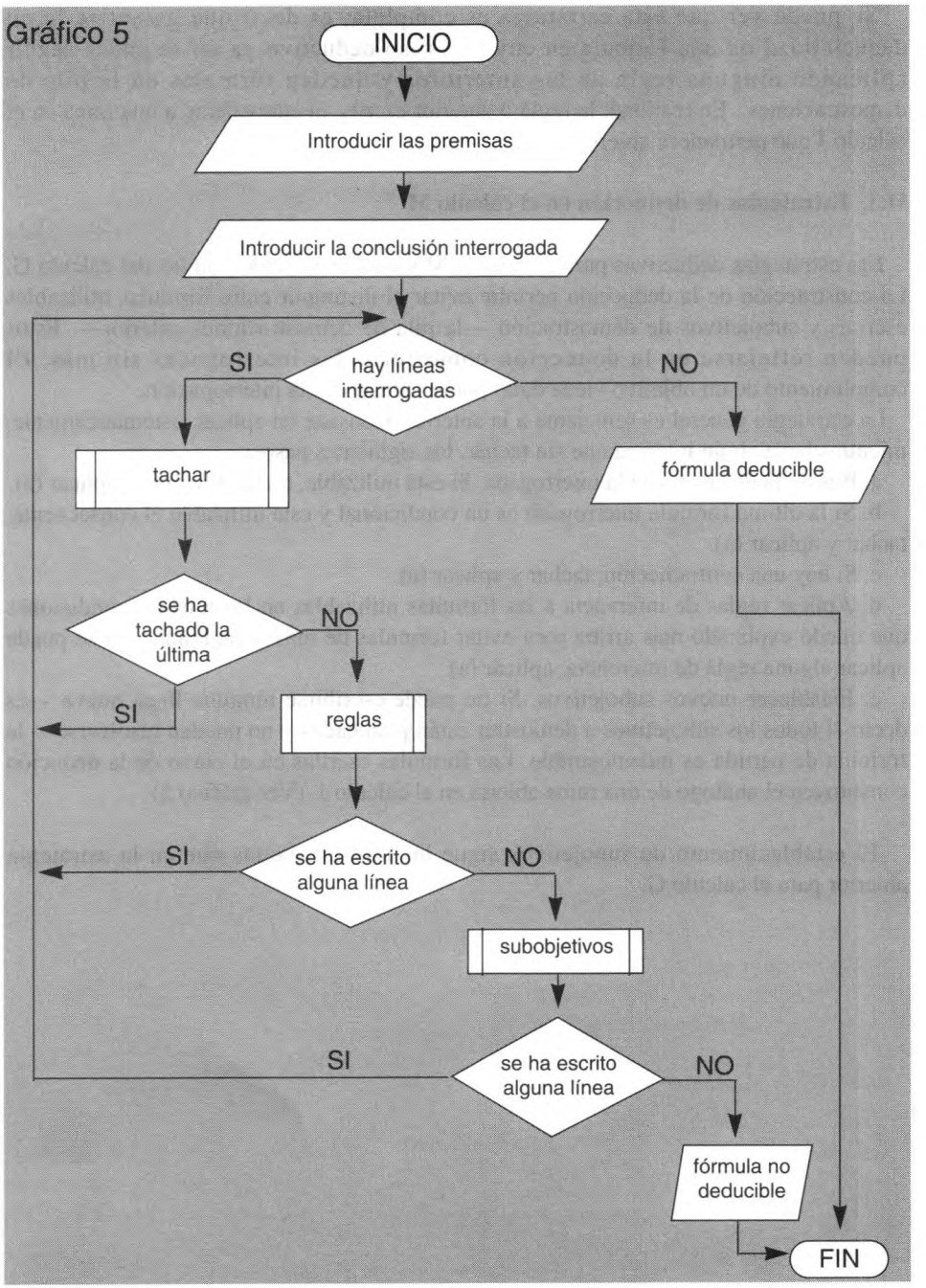
c. Si hay una contradicción, tachar y aplicar (a).

d. Aplicar reglas de inferencia a las fórmulas utilizables en las mismas condiciones que quedó explicado más arriba para evitar fórmulas de más complejidad. Si se puede aplicar alguna regla de inferencia, aplicar (a).

e. Establecer nuevos subojetivos. Si no puede escribirse ninguna línea nueva —es decir: si todos los subojetivos a demostrar están planteados y no pueden resolverse— la fórmula de partida es indemostrable. Las fórmulas escritas en el curso de la dedución constituyen el análogo de una rama abierta en el cálculo J. (Ver gráfico 5).

El establecimiento de subojetivos sigue las mismas pautas que en la estrategia anterior para el cálculo G.

Gráfico 5


INICIO
Introducir las premisas
Introducir la conclusión interrogada
hay líneas interrogadas
SI
NO
tachar
fórmula deducible
se ha tachado la última
NO
SI
reglas
se ha escrito alguna línea
SI
NO
subobjetivos
se ha escrito alguna línea
SI
NO
fórmula no deducible
FIN

III. PROGRAMAS PARA DEDUCCION AUTOMATIZADA

La estructura general de los programas es la siguiente:

1. Instrucciones sobre la notación empleada (opcional).
2. Introducción de la información: la conclusión y las premisas, si las hubiere.
3. Comprobación de la información recibida: verificación de que los símbolos y las fórmulas son correctos.
4. Rutinas de prueba propiamente dicha.
5. Salida de la deducción efectuada.

Dado que los programas son bastante largos, los hemos dividido en ficheros para facilitar su compilación y posible modificación. Para ejecutarlos pueden seguirse los siguientes pasos:

a. Escribir los ficheros del programa correspondiente. Para ello deben suprimirse los números de las líneas, que nosotros usamos sólo para comodidad en los comentarios. Los textos de anotaciones, los que van entre los símbolos /*...*/, pueden suprimirse, así como las sangrías y líneas en blanco.

b. Escribir en un fichero con la extensión .prj los ficheros .c que componen el programa, con su dependencia del fichero cabecera .h. Por ejemplo, el fichero pdalp00.prj contendría:

pdalp01.c (pdalp00.h)
pdalp02.c (pdalp00.h)
pdalp03.c (pdalp00.h)
pdalp04.c (pdalp00.h)
pdalp05.c (pdalp00.h)
pdalp06.c (pdalp00.h)

c. Una vez cargado el Turbo C elegir consecutivamente la opción Project y Project Name e introducir el nombre correspondiente (pdalp00).

d. Pulsar la tecla de función F9.(O elegir sucesivamente la opción Compile y Make EXE file).

e. Elegir la opción Run para ejecutar el programa.

III.1. Programa PDALP00

PDALP00 efectúa deducciones en el cálculo G. El programa PDALP00 consta de los ficheros PDALP00.h, PDALP01.c–PDALP06.c. Comentaremos sus características a continuación.

El fichero PDALP00.h es el fichero cabecera que incluye las estructuras usadas para almacenar la información sobre las líneas de la deducción, definiciones y el prototipo de las funciones empleadas por el programa.

```
 1 /* Programa PDALP00.
 2    Fichero PDALP00.h */

 3 /* Estructuras */

 4 struct FORMULAS_VALIDAS{
 5    char *f;          /*fórmula */
 6    char u;           /*utilizable o no */
 7    char r;           /*reglas */
 8    int l1;           /*líneas de inferencia*/
 9    int l2;           /*líneas de inferencia*/
10    };

11 struct MARCAS{
12    char ma;           /*marca */
13    int l;             /*número de línea*/
14    };

15/* Definiciones */

16 #define TFV sizeof(struct FORMULAS_VALIDAS)
17 #define TM sizeof(struct MARCAS)

18 /* Funciones prototipo */

19 /* Fichero PDALP01.c */
20 void instrucciones(void);
21 void calculos_dn(void);
22 int simbolos(char*);
23 int es_formula(char*);
24 int simbolo_do(char*);
25 void pausa(void);
```

```
26 void falta_memoria(int);
27 void salida_formulas(void);
28 void salida_formulas(void);

29 /* Fichero PDALP02.c */
30 void demostrada(void);
31 int sacar_fd(void);
32 void introducir_disyuncion(char*,int);
33 void introducir_conjuncion(int,int);
34 void introducir_bicondicional(int,int);
35 void introducir_condicional(char*,int);
36 void tachar(int);
37 void ultimo_supuesto(void);

38 /* Fichero PDALP03.c */
39 int inferencias(void);
40 void inf_conj(int);
41 void inf_dis(int);
42 void inf_cond(int);
43 void inf_bi(int);
44 void inf_do_ne(int);
45 void inf_ne_conj(int);

46 /* Fichero PDALP04.c */
47 int supuestos(void);
48 int supuestos_conj(void);
49 int supuestos_bi(void);
50 int supuestos_cond(void);
51 int supuestos_ne(void);
52 int suponer_negacion(void);
53 int supuestos_nuevos(void);
54 int subformulas(char*);

55 /* Fichero PDALP05.c */
56 int posicion_d(char*);
57 int buscar(char*,int,int);
58 int negacion(char*,int);
59 int contradiccion(void);
60 void repeticion(int);
61 void eliminar(void);
62 void introducir_negacion(int);
63 int negar(char*,int,int);
```

```
64 /* Fichero PDALP06.c */
65 void inf_ne_bi(int);
66 void inf_ne_dis(int);
67 void inf_ne_cond(int);
```

Las líneas de la deducción están tratadas como estructuras que constan de una cadena de caracteres (la fórmula), la indicación de si está o no utilizable, así como la regla y las líneas por cuya aplicación se haya obtenido (líneas 4–10 de PDALP00.h). En la estructura podría haberse incluído también la información sobre las marcas, pero hemos optado por almacenar las marcas en una estructura independiente en la que se especifica la marca y la línea a la que corresponde (11–14). La razón es doble: por economía de memoria y por rapidez en la ejecución.

Si se desea aumentar la rapidez de ejecución, puede incluirse en la estructura más información, por ejemplo, sobre la longitud de la fórmula, las subfórmulas que la componen, etc.

El fichero PDALP01.c contiene las rutinas de introducción y salida de fórmulas.

```
68    /*Programa PDALP00.
69      Fichero PDALP01.c. */

70  /* Librerías */
71  #include <stdio.h>
72  #include <stdlib.h>
73  #include <string.h>
74  #include <ctype.h>
75  #include <conio.h>
76  #include <alloc.h>
77  #include <dos.h>
78  #include <time.h>
79  #include "pdalp00.h"

80  /* Variables globales */
81  /* Cabeza de pila de fórmulas a demostrar: cp;
82   * contadores de fórmulas escritas y marcas: ul,m;
83   * último supuesto: us;
84   * marcas,fórmulas a demostrar, fórmulas válidas
85   * disponibles por asignación dinámica en curso:
       md,fdd,fvd.
86   */
87  int cp,ul,m,us,md,fdd,fvd;
88  struct FORMULAS_VALIDAS *fv; /*puntero a fórmulas
    escritas*/
```

```
 89  struct MARCAS *mar;    /*puntero a marcas       */
 90  char **fd;         /*puntero a fórmulas a demostrar */
 91  int *s;               /*puntero a supuestos       */
```

Las líneas 80-91 contienen las variables globales utilizadas a lo largo del programa PDALP00, así como la información necesaria acerca de su uso.

```
 92  /* Inicio */
 93  void main()
 94  {
 95  clrscr();
 96  putch(7);
 97  instrucciones();
 98  calculos_dn();
 99  clrscr();
100  exit(0);
101  }
102  /* Instrucciones generales */
103  void instrucciones()
104  {
105  cputs("Lenguaje proposicional usado:\n\n\r");
106  cputs("Variables proposicionales: p; P; más un
     número.");
107  cputs("\n\n\rNegación: n; N; ¬.\n\n\r");
108  cputs("Conjunción: y; Y; K; /\\.\n\n\r");
109  cputs("Disyunción: o; O; A; \\/.\n\n\r");
110  cputs("Condicional: i; I; C; ->.\n\n\r");
111  cputs("Bicondicional: b; B; E; <->.\n\n\r");
112  pausa();
113  }
```

La función inicial (92–101) gestiona el comienzo de ejecución del programa, la deducción propiamente dicha y la salida al sistema operativo. Antes de la entrada de información, se ofrecen unas instrucciones generales que recuerdan los símbolos que el programa puede aceptar (102–113). Estas dos funciones se repiten en cada uno los programas posteriores sin alteración.

```
114  /* Introducción de conclusión y premisas */
115  void calculos_dn()
116  {
117  int j;
```

```
118  do{
119   j=cp=ul=us=m=0;
120   fv=(struct FORMULAS_VALIDAS*)malloc(300*TFV);
121   if(fv==NULL) falta_memoria(1);
122   fd=(char**)malloc(50*sizeof(char*));
123   if(fd==NULL) falta_memoria(1);
124   mar=(struct MARCAS*)malloc(300*TM);
125   if(mar==NULL) falta_memoria(1);
126   s=(int*)calloc(50,sizeof(int));
127   if(s==NULL) falta_memoria(1);
128   fvd=300;fdd=50;md=300;
129   clrscr();
130   cputs("Introduzca la conclusión y las premisas, \
131            si las hubiere.\n\r");
132   cputs("Pulse [ENTER] para abandonar o finalizar \
133            la introducción.\n\r");
134   do{          /*Introducción de fórmulas */
135    int x,y;
136    char *a,*b;
137    if(j==0) cputs("Conclusión:  ");
138    else cprintf("Premisa  %d:  ",j);
139    y=wherey();x=wherex();
140    gotoxy(x+60,y);
141    putch(42);
142    gotoxy(x,y);
143    a=(char*)calloc(62,sizeof(char));
144    if(!a) falta_memoria(1);
145    fflush(stdin);
146    gets(a);
147    if(!*a){
148      for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
149      free(a);
150      break;
151      }
152    b=(char*)calloc(strlen(a)+1,sizeof(char));
153    if(b==NULL) falta_memoria(1);
154    strcpy(b,a);
155    free(a);
156    if(!simbolos(b)){
157      putch(7);
158      cputs("\n\rError en la escritura de la fórmula.");
```

```
159       pausa();
160       free(b);
161       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
162       continue;
163       }
164     if(!es_formula(b)){
165       putch(7);
166       cputs("\n\rError en la escritura de la fórmula.");
167       pausa();
168       free(b);
169       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
170       continue;
171       }
172     if(j==0){
173       *(fd+cp)=b;
174       *(s+cp)=-1;
175       cp++;
176       }
177     else{
178       (fv+ul)->f=b;
179       (fv+ul)->u=99;
180       (fv+ul)->r=98;
181       (fv+ul)->l1=(fv+ul)->l2=-1;
182       ul++;
183       }
184     j++;
185     } while(1);
186    if(j>0) {
187     struct time *z;
188     z=(struct time*)calloc(1,sizeof(struct time));
189     gettime(z);
190     cprintf("\n\r%u:%u:%u",z->ti_min,z->ti_sec,
       z->ti_hund);
191     demostrada();
192     gettime(z);
193     cprintf("\n\r%u:%u:%u",z->ti_min,z->ti_sec,
       z->ti_hund);
194     salida_formulas();
195     }
196   free(fd);free(fv);free(mar);free(s);/*liberar memoria*/
197   } while(j!=0);
198 }
```

En 114–198 se gestiona básicamente la introducción de información. En 119–128 se inicializan las variables globales y se asigna la memoria dinámica. Para deducciones muy largas pueden modificarse los valores, establecidos inicialmente en 50 para unas variables y 300 para otras. Del mismo modo, puede modificarse la longitud de las fórmulas que se vayan a introducir, establecida en 61 caracteres (143). En 156 se envía la fórmula escrita por teclado a una rutina que analiza los símbolos y los paréntesis. Si son correctos, en 164 se envía la fórmula a una rutina que verifica que los símbolos constituyan efectivamente una fórmula. Una vez examinada la corrección de las fórmulas introducidas, se almacenan bien como la conclusión, 172–176, bien como líneas utilizables de premisas, 177–183. Se ha introducido una doble lectura de reloj (187–190, 192–193) para saber el tiempo empleado en la deducción. En 191 el programa pasa a las rutinas de deducción y en 194 a la salida de las líneas obtenidas, tanto si la fórmula es deducible como si no lo es.

```
199  /*  Verificación de símbolos */
200  int simbolos(char *a)
201  {
202   int b,i,k,z;
203   b=k=0;
204   for(i=0;*(a+i);i++){      /*cerrar espacios*/
205    if(*(a+i)==32) {k++;continue;}
206    *(a+i-k)=*(a+i);
207    }
208  *(a+i-k)='\0';
209  k=0;z=1;
210  for(i=0;*(a+i);i++){
211     switch(*(a+i)){
212        case -86:
213        case 78:
214        case 110: *(a+i-k)=78;break;
215        case 40: b++;*(a+i-k)=40;break;
216        case 41: b--;*(a+i-k)=41;break;
217        case 45: if(*(a+i+1)!=62) z=0;
218          else {*(a+i-k)=67;k++;i++;}
219          break;
220        case 47: if(*(a+i+1)!=92) z=0;
221          else {*(a+i-k)=75;k++;i++;}
222          break;
223        case 48:
224        case 49:
225        case 50:
```

```
226         case 51:
227         case 52:
228         case 53:
229         case 54:
230         case 55:
231         case 56:
232         case 57: *(a+i-k)=*(a+i);break;
233         case 60: if(*(a+i+1)!=45 || *(a+i+2)!=62) z=0;
234           else {*(a+i-k)=69;k+=2;i+=2;}
235           break;
236         case 65:
237         case 79:
238         case 111: *(a+i-k)=65;break;
239         case 66:
240         case 69:
241         case 98: *(a+i-k)=69;break;
242         case 67:
243         case 73:
244         case 105: *(a+i-k)=67;break;
245         case 75:
246         case 89:
247         case 121: *(a+i-k)=75;break;
248         case 80:
249         case 112: *(a+i-k)='p';break;
250         case 92: if(*(a+i+1)!=47) z=0;
251           else {*(a+i-k)=65;k++;i++;}
252           break;
253         default: z=0;
254         }
255       if(!z) break;
256       }
257   if(z){
258       if(b!=0) z=0;
259       else *(a+i-k)='\0';
260       }
261   return z;
262   }
```

En 169–262 se efectúa la primera comprobación de las fórmulas introducidas. En 204–207 se eliminan los espacios en blanco. Esta verificación de símbolos rechaza cualquier información que no conste únicamente de los caracteres permitidos o que presente paréntesis desbalanceados.

```
263  /*  Verificación de fórmulas */
264  int es_formula(char *a)
265  {
266  int l,p,r,h,i;
267  l=strlen(a);r=1;h=0;
268  if(*a=='(' && *(a+l-1)==')'){
269     for(i=0;i<l-1;i++){
270        if(*(a+i)=='(') {h++;continue;}
271        if(*(a+i)==')') h-;
272        if(h==0) break;
273        }
274     }
275  if(h==1){
276     for(i=0;i<l-2;i++) *(a+i)=*(a+i+1);
277     *(a+i)='\0';
278     r=es_formula(a);
279     }
280  else{
281     p=simbolo_do(a);
282     if(p<0) r=0;
283     if(p==0){
284        char *z;
285        if(*a=='p'){
286    for(i=1;*(a+i);i++){
287       if(isdigit(*(a+i))) continue;
288       else {r=0;break;}
289       }
290    }
291        if(*a=='N'){
292    z=(char*)calloc(l,sizeof(char));
293    if(z==NULL) falta_memoria(1);
294    for(i=0;(*(z+i)=*(a+i+1));i++);
295    if((r=es_formula(z))) for(i=0;(*(a+i+1)=*(z+i));i++);
296    free(z);
297    }
298   if(*a!='p' && *a!='N') r=0;
299   }
300  if(p>0){
301   char j;
302   j=*(a+p);
303   if(j=='K'||j=='A'||j=='C'||j=='E'){
```

```
304     int h;
305     char *z,*x;
306     z=(char*)calloc(l-p,sizeof(char));
307     x=(char*)calloc(p+1,sizeof(char));
308     if(x==NULL || z==NULL) falta_memoria(1);
309     for(h=0;h<p;h++) *(x+h)=*(a+h);*(x+h)='\0';
310     for(;(*(z+(h-p))=*(a+h+1));h++) ;
311     if((r=es_formula(x)) && (r=es_formula(z))){
312        l=strlen(x);
313        *a=j;
314        for(h=1;h<=l;h++) *(a+h)=*(x+h-1);
315        for(p=0;(*(a+h)=*(z+p));h++,p++) ;
316        }
317     free(z);free(x);
318     }
319    else r=0;
320    }
321  }
322  return r;
323 }
```

En 263–323 se verifica que los caracteres introducidos constituyen efectivamente una fórmula, a la vez que traduce la fórmula a la notación polaca. El programa opera con fórmulas en tal notación no porque ello sea imprescindible, sino por motivos técnicos de rapidez de los procesos de identificación. En la deducción automatizada es normal hacerlo así. El algoritmo de verificación y traducción es recursivo, empleando una función auxiliar para establecer el símbolo dominante de una fórmula en virtud de los convenios usuales de dominancia (324–355).

```
324  /* Símbolo dominante de una fórmula */
325  int simbolo_do(char *a)
326  {
327  int h,i,w,nc,nd,ni,nb,z;
328  w=strlen(a);z=0;
329  h=nc=nd=ni=nb=0;
330  for(i=0;i<w;i++){
331     switch(*(a+i)){
332        case 40: {h++;break;}
333        case 41: {h--;break;}
334        case 75: if(h>0) break;
335          if(nc!=0 || nd!=0) z=1;
```

```
336           else nc=i;
337           break;
338         case 65: if(h>0) break;
339           if(nc!=0 || nd!=0) z=1;
340           else nd=i;
341           break;
342         case 67: if(h>0) break;
343           if(ni!=0 || nb!=0) z=1;
344           else {ni=i;nc=nd=0;}
345           break;
346         case 69: if(h>0) break;
347           if(ni!=0 || nb!=0) z=1;
348           else{nb=i;nc=nd=0;}
349         }
350       if(z) break;
351       }
352   if(z) z=-1;
353   else z=nb!=ni?(nb<ni?ni:nb):(nc<nd?nd:nc);
354   return z;
355   }
```

La siguiente (356-362) es una función simple para emitir mensajes en las pausas.

```
356   /* Pausa */
357   void pausa()
358   {
359   fflush(stdin);
360   cputs("\n\rPulse [ENTER] para continuar");
361   getchar();
362   }
```

Las cuatro funciones anteriores (verificación de símbolos, verificación de fórmulas, establecimiento del símbolo dominante de una fórmula y pausa) se repiten sin alteraciones en todos los programas posteriores.

La siguiente función (363–386) permite reasignar memoria dinámica, mientras ésta exista, a aquellas variables que lo necesiten. En deducciones largas puede acortarse el tiempo de reasignación ampliando su valor.

```
363   /* Interrupción por falta de memoria */
364   void falta_memoria(int i)
365   {
```

```
366  switch(i){
367     case 1: {cputs("Falta de memoria. ");break;}
368     case 2: fv=(struct FORMULAS_VALIDAS*)\
369             realloc(fv,(fvd+50)*TFV);
370      if(fv==NULL) {cputs("Falta de memoria. ");break;}
371      fvd+=50;return;
372  case 3:fd=(char**)realloc(fd,(fdd+10)*sizeof(char*));
373      s=(int*)realloc(s,(fdd+10)*sizeof(int));
374      if(fd==NULL || s==NULL) {
375       cputs("Falta de memoria. ");break;}
376      fdd+=10;return;
377  case 4: mar=(struct MARCAS*)realloc(mar,(md+50)*TM);
378     if(mar==NULL) {cputs("Falta de memoria. ");break;}
379     md+=50;return;
380     }
381  putch(7);
382  cprintf("\n\rMemoria libre %u bytes",coreleft());
383  pausa();
384  clrscr();
385  exit(1);
386  }
```

La siguiente función (387–544) gestiona la salida de fórmulas por pantalla y por impresora, si se desea. En caso de que la fórmula haya sido deducida se eliminan las líneas superfluas, previamente señaladas por una función que ya comentaremos posteriormente. En general, una función de salida de las líneas de la deducción ha de escribir (en pantalla, en impresora, en un archivo) las marcas (406–416), las fórmulas reescritas de la notación polaca (417–501), las reglas (502–526) y las líneas de aplicación (527–535). La longitud relativamente larga de la presente función se debe a que efectúa la traducción de la notación polaca directamente.

```
387  /* Salida de demostraciones */
388  void salida_formulas()
389  {
390  int i,l,*x,p,j,k,w;
391  char *d,e;
392  cprintf("\n\r¿Desea imprimir el resultado? (S/N): ");
393  fflush(stdin);
394  e=toupper(getchar());
395  l=0;           /*Contador de líneas superfluas*/
396  x=(int*)calloc(ul,sizeof(int));
```

```
397  if(x==NULL) falta_memoria(1);
398  for(i=0;i<ul;i++){
399     if((fv+i)->u==98){
400        l++;
401        continue;
402        }
403     *(x+i)=i+1-l;
404     cprintf("\n\r%3d. ",i+1-l);
405     if(e==83) fprintf(stdprn,"\n\r%3d. ",i+1-l);
406     for(p=m-1;p>=0;p--){
407        if((mar+p)->l!=i) continue;
408        switch((mar+p)->ma){
409    case 40: cputs("┌");
410           if(e==83) fputs("┌",stdprn);break;
411    case 50: cputs("│");
412           if(e==83) fputs("│",stdprn);break;
413    case 60: cputs("└");
414           if(e==83) fputs("└",stdprn);
415    }
416        }
417     d=(fv+i)->f;
418     for(j=0;*(d+j);j++){
419        switch(*(d+j)){
420    case 78: putchar('¬');if(e==83) fputs("¬",stdprn);
421          if(*(d+j+1)=='N' || *(d+j+1)=='p') *(d+j)=45;
422           else {
423             putchar(40);
424             if(e==83) fputs("(",stdprn);
425             *(d+j)=41;
426             }
427            break;
428    case 65:
429    case 75: if(*(d+j+1)!='N' && *(d+j+1)!='p'){
430              putchar(40);if(e==83) fputs("(",stdprn);}
431              break;
432    case 67:
433    case 69: if(*(d+j+1)=='C' || *(d+j+1)=='E'){
434             putchar(40); if(e==83) fputs("(",stdprn);}
435             break;
436    case 112:putchar(*(d+j));
437           if(e==83) fprintf(stdprn,"%c",*(d+j));
```

```
438             *(d+j)=45;
439               while(*(d+(++j))>47 && *(d+j)<58){
440                putchar(*(d+j));
441                if(e==83) fprintf(stdprn,"%c",*(d+j));
442                *(d+j)=45;
443                }
444             for(k=j-1;k>=0;k--){
445                if(*(d+k)==45) continue;
446                if(*(d+k)==41){
447                 putchar(41);
448                 if(e==83) fputs(")",stdprn);
449                 *(d+k)=45;
450                 continue;
451                 }
452                break;
453                }
454              switch(*(d+k)){
455                case 65:
456                case 75: if(*(d+k)==75) {
457                        cputs("/\\");
458                        if(e==83) fputs("/\\",stdprn);
459                        }
460                       else {
461                        cputs("\\/");
462                        if(e==83) fputs("\\/",stdprn);
463                        }
464                       for(w=k-1;w>=0;w--){
465                        if(*(d+w)==45) continue;
466                        if(*(d+w)=='A'
467                        ||*(d+w)=='K') *(d+k)=41;
468                        break;
469                        }
470                       if(*(d+j)!='N'&&*(d+j)!='p'){
471                        putchar(40);
472                        if(e==83) fputs("(",stdprn);
473                        if(*(d+k)==41) *(d+k+1)=41;
474                        else *(d+k)=41;
475                        }
476                       if(*(d+k)>42) *(d+k)=45;break;
477                case 69:
478                case 67: if(*(d+k)==67) {
```

```
479       cputs("->");
480       if(e==83) fputs("->",stdprn);
481       }
482      else {
483       cputs("<->");
484       if(e==83) fputs("<->",stdprn);
485       }
486      for(w=k-1;w>=0;w--){
487       if(*(d+w)==45) continue;
488       if(*(d+w)!=')') *(d+k)=41;
489       break;
490       }
491      if(*(d+j)=='C'||*(d+j)=='E'){
492       putchar(40);
493       if(e==83) fputs("(",stdprn);
494       if(*(d+k)==41) *(d+k+1)=41;
495       else *(d+k)=41;
496       }
497      if(*(d+k)>42) *(d+k)=45;break;
498      }
499     j--;
500     }
501   }
502  if((fv+i)->r!=99){
503    switch((fv+i)->r){
504    case 98: cprintf(" Premisa");
505     if(e==83) fputs(" Premisa",stdprn);break;
506    case 40: cputs(" I¬  ");
507     if(e==83) fputs(" I¬ ",stdprn);break;
508    case 41: cputs(" I/\\  ");
509     if(e==83) fputs(" I/\\ ",stdprn);break;
510    case 42: cputs(" I\\/  ");
511     if(e==83) fputs(" I\\/ ",stdprn);break;
512    case 43: cputs(" I->   ");
513     if(e==83) fputs(" I-> ",stdprn);break;
514    case 44: cputs(" I<->  ");
515     if(e==83) fputs(" I<-> ",stdprn);break;
516    case 50: cputs(" E¬   ");
517     if(e==83) fputs(" E¬ ",stdprn);break;
518    case 51: cputs(" E/\\  ");
519     if(e==83) fputs(" E/\\ ",stdprn);break;
```

```
520     case 52: cputs(" E\\/  ");
521      if(e==83) fputs(" E\\/ ",stdprn);break;
522     case 53: cputs(" E->  ");
523      if(e==83) fputs(" E-> ",stdprn);break;
524     case 54: cputs(" E<->  ");
525      if(e==83) fputs(" E<-> ",stdprn);
526     }
527    if((fv+i)->l1>=0){
528      cprintf("%d",*(x+(fv+i)->l1));
529      if(e==83) fprintf(stdprn,"%d",*(x+(fv+i)->l1));
530      }
531    if((fv+i)->l2>=0){
532      cprintf("  %d",*(x+(fv+i)->l2));
533      if(e==83) fprintf(stdprn," %d",*(x+(fv+i)->l2));
534      }
535     }
536    if(wherey()<22) continue;
537    pausa();
538    clrscr();
539    }
540  if(e==83) fputs("\n\r",stdprn);
541  free(x);
542  for(j=ul-1;j>=0;j--) free((fv+j)->f);
543  pausa();
544  }
```

Una alternativa más elegante es encargar el cometido de traducción de la notación polaca a una función específica, como la siguiente:

```
 1 void traducir(char *a,char *b)
 2 {
 3 int l,p,h;
 4 char *z,*x;
 5 l=strlen(a);
 6 if(*a=='p') strcat(b,a);
 7 if(*a=='N'){
 8    z=(char*)calloc(l,sizeof(char));
 9    if(z==NULL) falta_memoria(1);
10    for(h=0;(*(z+h)=*(a+h+1));h++);
11    strncat(b,"¬",1);
12    if(*z!='p' && *z!='N'){
```

```
13          strncat(b,"(",1);
14          traducir(z,b);
15          strncat(b,")",1);
16          }
17       else traducir(z,b);
18       free(z);
19       }
20    if(*a=='K' || *a=='A'){
21       p=posicion_d(a);
22       z=(char*)calloc(l-p+1,sizeof(char));
23       x=(char*)calloc(p,sizeof(char));
24       if(x==NULL || z==NULL) falta_memoria(1);
25       for(h=0;h<p-1;h++) *(x+h)=*(a+h+1);*(x+h)='\0';
26       for(h=p;(*(z+h-p)=*(a+h));h++);
27       if(*x!='p' && *x!='N'){
28          strncat(b,"(",1);
29          traducir(x,b);
30          strncat(b,")",1);
31          }
32       else traducir(x,b);
33       if(*a=='K') strncat(b,"/\\",2);
34       else strncat(b,"\\/",2);
35       if(*z!='p' && *z!='N'){
36          strncat(b,"(",1);
37          traducir(z,b);
38          strncat(b,")",1);
39          }
40       else traducir(z,b);
41       free(z);free(x);
42       }
43    if(*a=='C' || *a=='E'){
44       p=posicion_d(a);
45       z=(char*)calloc(l-p+1,sizeof(char));
46       x=(char*)calloc(p,sizeof(char));
47       if(x==NULL || z==NULL) falta_memoria(1);
48       for(h=0;h<p-1;h++) *(x+h)=*(a+h+1);*(x+h)='\0';
49       for(h=p;(*(z+h-p)=*(a+h));h++) ;
50       if(*x=='C' || *x=='E'){
51          strncat(b,"(",1);
52          traducir(x,b);
53          strncat(b,")",1);
```

```
54        }
55     else traducir(x,b);
56     if(*a=='C') strncat(b,"->",2);
57     else strncat(b,"<->",3);
58     if(*z=='C' || *z=='E'){
59        strncat(b,"(",1);
60        traducir(z,b);
61        strncat(b,")",1);
62        }
63     else traducir(z,b);
64     free(z);free(x);
65     }
66 }
```

Para usar esta función, las líneas 417-502 han de ser sustituídas por las siguientes:

```
417 d=(char*)calloc(3*strlen((fv+i)->f),sizeof(char));
418 if(d==NULL) falta_memoria(1);
419 traducir((fv+i)->f,d);
420 cprintf(" %s",d);
421 if(e==83)fprintf(stdprn," %s",d);
422 free(d);
```

Debemos, además, incluir en el fichero cabecera, PDALP00.h, el prototipo de la función:

```
void traducir(char*,char*);
```

En adelante nos referiremos, en general, a una función de traducción de los símbolos, como parte integrante de una función de salida, que puede ser o bien la función traducir con las modificaciones acabadas de señalar o bien una traducción directa como la escrita en las líneas 417–501.

El fichero PDALP02.c contiene la rutina principal que controla el proceso de deducción y varias funciones de aplicación de reglas.

```
1 /* Programa PDALP00.
2    Fichero PDALP02.c */

3 /* Librerías */
```

```
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <conio.h>
 7 #include <ctype.h>
 8 #include <string.h>
 9 #include "pdalp00.h"

10 /* Variables globales  externas */

11 extern int cp,ul,m,us,md,fdd,fvd,*s;
12 extern struct FORMULAS_VALIDAS *fv;
13 extern struct MARCAS *mar;
14 extern char **fd;
```

Las líneas 15–91 contienen el algoritmo básico de demostrabilidad. Mientras exista alguna fórmula que demostrar (es decir: en tanto que la pila de demostraciones no esté vacía, condición en 89), se ejecuta el proceso siguiente:

a. Buscar si está utilizable la fórmula y puede extraerse de la pila (22). La condición adicional que se impone para extraer una fórmula de la pila de demostraciones (92–103) no es redundante. Puede ocurrir, en efecto, que una fórmula esté utilizable y que esa misma fórmula sea la última fórmula de la pila, sin que ello garantice la corrección de la deducción al eliminarla. Un ejemplo simple es éste: supóngase que queremos demostrar p1→p1; para ello escribimos el antecedente como supuesto:
1.⌈ p1
e introducimos en la pila de demostraciones el consecuente: p1.
Nuestro objetivo ahora es demostrar p1 (el consecuente) contando como línea utilizable 1. Pero si eliminamos de la pila p1 porque ya está utilizable no podremos aplicar la regla I→ y no habremos obtenido la deducción de p1→p1.

b. Si se trata de una conjunción, buscar si están utilizables los dos términos de la misma; en tal caso aplicar la regla I∧ (26–32).

c. Si se trata de un bicondicional, buscar los dos condicionales necesarios para aplicar I↔ (34–49).

d. Si se trata de una disyunción, buscar si está utilizable algún término de la misma para aplicar I∨ (51–59).

e. Si se trata de un condicional, buscar si está utilizable el consecuente para aplicar I→ (61–71).

f. Si se trata de una doble negación, buscar si está utilizable la fórmula con dos negaciones menos para aplicar I ¬(73–79).

Si se ha tenido éxito en la búsqueda, se aplican las reglas correspondientes. Si se puede extraer de la pila la última fórmula, se intenta la extracción de la fórmula siguiente, etc.

Si no se tiene éxito en la búsqueda, se examina la posible existencia de contradicciones (81). Si éstas no existen, se trata de aplicar reglas de inferencias para obtener nuevas líneas utilizables (82). Si se ha podido aplicar alguna regla de inferencia, el programa vuelve a examinar la última fórmula de la pila de demostraciones por si estuviera ya demostrada; si no ha sido éste el caso, se pasa a establecer nuevos supuestos (83).

Esta estrategia de volver a examinar los objetivos de demostración cada vez que se aplica una regla hace más lento el proceso deductivo, pero evita que se obtengan demasiadas líneas innecesarias que no conducen a nada. Hemos preferido perder en rapidez para ganar en elegancia.

Si, como resultado de la ejecución de la rutina de supuestos, no se ha obtenido ninguna línea utilizable nueva ni se ha incrementado la pila de demostraciones, la fórmula es indemostrable (84–88) y devuelve el control a la rutina de salida de fórmulas. Si la fórmula ha sido demostrada se pasa a eliminar las líneas superfluas que hayan podido obtenerse en el proceso de la deducción y se devuelve el control a la rutina de salida de fórmulas (90).

```
15 /* Búsqueda de demostraciones para la última fórmula */

16 void demostrada()
17 {
18 int i,l;
19 i=0;
20 do{
21    l=strlen(*(fd+cp-1));
22   if(buscar(*(fd+cp-1),0,l)>=0 && sacar_fd()) continue;
23    switch(**(fd+cp-1)){
24       int i,k,p;
25       char *a,*b;
26       case 75:
27        p=posicion_d(*(fd+cp-1));
28        if((i=buscar(*(fd+cp-1),1,p))>=0 &&
29         (k=buscar(*(fd+cp-1),p,l))>=0){
30         introducir_conjuncion(i,k);
31         if(sacar_fd()) continue;
32         }
33        break;
34       case 69:
35        p=posicion_d(*(fd+cp-1));
36        if((b=(char*)calloc(l+1,sizeof(char)))==NULL)
37        falta_memoria(1);
```

```
38          if((a=(char*)calloc(l+1,sizeof(char)))==NULL)
39          falta_memoria(1);
40          *a=*b='C';
41          for(i=1;(*(a+i)=*(*(fd+cp-1)+i));i++);
42          for(i=0;i+p<l;i++) *(b+i+1)=*(*(fd+cp-1)+i+p);
43          k=l-p;
44          for(i=1;i<p;i++) *(b+k+i)=*(*(fd+cp-1)+i);
            *(b+k+i)='\0';
45        if((i=buscar(a,0,l))>=0 && (k=buscar(b,0,l))>=0){
46         introducir_bicondicional(i,k);
47         free(a);free(b);
48         if(sacar_fd()) continue;
49         }
50        break;
51       case 65:
52        p=posicion_d(*(fd+cp-1));
53        i=k=-1;
54        if((i=buscar(*(fd+cp-1),1,p))>=0 ||
55        (k=buscar(*(fd+cp-1),p,l))>=0){
56         i=i<k?k:i;
57         introducir_disyuncion(*(fd+cp-1),i);
58         if(sacar_fd()) continue;
59         }
60        break;
61       case 67:
62        p=posicion_d(*(fd+cp-1));
63        if((k=buscar(*(fd+cp-1),p,l))>=0){
64         a=(char*)calloc(p,sizeof(char));
65         if(a==NULL) falta_memoria(1);
66         for(i=0;i<p-1;i++) *(a+i)=*(*(fd+cp-1)+i+1);
67         *(a+i)='\0';
68         introducir_condicional(a,k);
69         free(a);
70         if(sacar_fd()) continue;
71         }
72        break;
73       case 78:
74        if(*(*(fd+cp-1)+1)==78){
75         int p;
76         p=buscar(*(fd+cp-1),2,strlen(*(fd+cp-1)));
77         if(p>=0){introducir_negacion(p);
```

```
78            if(sacar_fd()) continue;}
79            }
80         }
81      if(contradiccion() && sacar_fd()) continue;
82      if(inferencias()) continue;
83      if(supuestos()) continue;
84      putch(7);
85      cputs("\n\rFórmula no demostrable. ");
86      for(i=0;i<cp;i++) free(*(fd+i));
87      i=1;
88      break;
89      } while(cp);
90   if(!i){eliminar();putch(7);
      cputs("\n\rFórmula demostrada");}
91 }

92 /* Eliminar de la pila de demostraciones */

93 int sacar_fd()
94 {
95 int i;
96 i=0;
97 if(us==0 || *(s+cp-1)<0){
98    cp--;
99    free(*(fd+cp));
100   i=1;
101   }
102 return i;
103 }
```

La siguiente función (104–117) obtiene una nueva línea utilizable por aplicación de la regla IV.

```
104 /* Introducir disyunción */

105 void introducir_disyuncion(char*a,int k)
106 {
107 unsigned int l;
108 l=strlen(a);
109 (fv+ul)->f=(char*)calloc(l+1,sizeof(char));
110 if((fv+ul)->f==NULL) falta_memoria(1);
```

```
111 strcpy((fv+ul)->f,a);
112 (fv+ul)->u=99;
113 (fv+ul)->r=42;
114 (fv+ul)->l1=k;
115 (fv+ul)->l2=-1;
116 ul++;if(ul==fvd) falta_memoria(2);
117 }
```

Cada vez que escribimos una nueva línea de deducción escribimos varias líneas de código. La longitud del Programa puede disminuirse sensiblemente si utilizamos una función especial para escribir cada línea nueva de la deducción; a esta función se envía como parámetros toda la información necesaria en cada caso (fórmula, tipo de utilización, regla y líneas de inferencia). La ejecución del programa se ralentizará, pero el código será más manejable y corto.

Una posibilidad es ésta:

```
void linea_nueva(char *a,char ut,char re,int l11,int l12)
{
(fv+ul)->f=(char*)calloc(strlen(a)+1,sizeof(char));
if((fv+ul)->f==NULL) falta_memoria(1);
strcpy((fv+ul)->f,a);
(fv+ul)->u=ut;
(fv+ul)->r=re;
(fv+ul)->l1=l11;
(fv+ul)->l2=l12;
if(fvd==++ul) falta_memoria(2);
}
```

La siguiente función introduce un condicional cuando tenemos el consecuente (119–159). Podría haberse ampliado, o haberse escrito otra función diferente, para recoger el caso en que estuviese utilizable la negación del antecedente. Esto en nada afecta a la completud del algoritmo deductivo general.

```
118 /* Introducción de condicional */

119 void introducir_condicional(char *a,int k)
120 {
121 int i;
122 if(us-1!=*(s+cp-1)){
123     if(((fv+ul)->f=(char*)
124     calloc(strlen(a)+1,sizeof(char)))==NULL)
```

```
      falta_memoria(1);
125   strcpy((fv+ul)->f,a);
126   (mar+m)->l=ul;(mar+m)->ma=40;
127   m++;if(m==md) falta_memoria(4);
128   (fv+ul)->u=35;
129   us=ul+1;
130   (fv+ul)->r=99;
131   (fv+ul)->l1=(fv+ul)->l2=-1;
132   ul++;if(ul==fvd) falta_memoria(2);
133   repeticion(k);
134   k=ul-1;
135   }
136 if(k<us){/*El consecuente está antes del antecedente*/
137   repeticion(k);
138   k=ul-1;
139   }
140 tachar(k);
141 (fv+us-1)->u=35;       /*Introducir el condicional*/
142 (fv+k)->u=35;
143 (mar+m)->l=k;(mar+m)->ma=60;
144 m++;if(m==md) falta_memoria(4);
145 if(k<ul-1) for(i=k+1;i<ul;i++) free((fv+i)->f);
146 ul=k+1;
147 if(((fv+ul)->f=(char*)
148 calloc(strlen(a)+strlen((fv+k)->f)+2,
    sizeof(char)))==NULL)
149 falta_memoria(1);
150 *((fv+ul)->f)='C';strcat((fv+ul)->f,a);
151 strcat((fv+ul)->f,(fv+k)->f);
152 (fv+ul)->u=99;
153 (fv+ul)->r=43;
154 (fv+ul)->l1=us-1;
155 (fv+ul)->l2=k;
156 ul++;if(ul==fvd) falta_memoria(2);
157 ultimo_supuesto();
158 *(s+cp-1)=-1;
159 }
```

La siguiente función (161-173) obtiene una nueva línea utilizable por aplicación de la regla I∧ .

```
160 /* Introducir /\ */

161 void introducir_conjuncion(int h,int k)
162 {
163 (fv+ul)->f=(char*)calloc(strlen((fv+h)->f)+\
164 strlen((fv+k)->f)+2,sizeof(char)));
165 if(((fv+ul)->f)==NULL) falta_memoria(1);
166 *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+h)->f);
167 strcat((fv+ul)->f,(fv+k)->f);
168 (fv+ul)->u=99;
169 (fv+ul)->r=41;
170 (fv+ul)->l1=k;
171 (fv+ul)->l2=h;
172 ul++;if(ul==fvd) falta_memoria(2);
173 }
```

La siguiente función (174–185) obtiene una nueva línea utilizable por aplicación de la regla I ↔.

```
174 /* Introducir <-> */

175 void introducir_bicondicional(int h,int k)
176 {
177 if(((fv+ul)->f=(char*)calloc(strlen((fv+h)->f)+1,\
178 sizeof(char)))==NULL) falta_memoria(1);
179 *((fv+ul)->f)='E';strcat((fv+ul)->f,(fv+h)->f+1);
180 (fv+ul)->u=99;
181 (fv+ul)->r=44;
182 (fv+ul)->l1=k;
183 (fv+ul)->l2=h;
184 ul++;if(ul==fvd) falta_memoria(2);
185 }
```

La siguiente función (187–195) inutiliza un bloque de fórmulas, tachándolo, al ser clausurado un supuesto.

```
186 /* Tachar fórmulas */

187 void tachar(int k)
188 {
189 int i;
```

```
190 for(i=us;i<k;i++){
191     (mar+m)->l=i;(mar+m)->ma=50;
192     m++;if(m==md) falta_memoria(4);
193     (fv+i)->u=35;
194     }
195 }
```

La siguiente función (196–205) determina el último supuesto que permanece aún sin calusurar.

```
196 /* Ultimo_supuesto */

197 void ultimo_supuesto()
198 {
199 int i;
200 for(i=m-1;i>=0;i--){
201     if((mar+i)->ma!=40) continue;
202     if((fv+(mar+i)->l)->u!=35){us=(mar+i)->l+1; break;}
203     }
204 if(i<0) us=0;
205 }
```

El fichero PDALP03.c contiene la rutina principal que controla la aplicación de las reglas de inferencia, así como diversas rutinas de aplicación de reglas de inferencia.

```
 1 /* Programa PDALP00.
 2    Fichero PDALP03.c*/

 3 /* Librerías */
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <conio.h>
 7 #include <string.h>
 8 #include "pdalp00.h"

 9 /* Variables globales externas */

10 extern int cp,ul,us,m,md,fdd,fvd,*s;
11 extern char **fd;
12 extern struct FORMULAS_VALIDAS *fv;
13 extern struct MARCAS *mar;
```

La siguiente función (14–37) dirige la aplicación de reglas de inferencia. Caben diversas estrategias a la hora de escribir una función de este tipo. Por ejemplo, se pueden jerarquizar las reglas y tratar de aplicarlas buscando si se cumplen las condiciones para ello. Nosotros hemos optado aquí por examinar las líneas y en función del símbolo dominante de la fórmula elegir la regla de inferencia que puede ser aplicada. Hemos optado también por una búsqueda sistemática cada vez que el flujo del programa llega a esta función. Ya hemos comentado que con ello se pierde rapidez en la demostración; pero nuestro programa funciona bastante bien por lo que a exigencias de tiempo se refiere. Se puede ganar algo de rapidez marcando las fórmulas a las que ya haya sido aplicada la correspondiente regla de inferencia, de modo que no vuelvan a ser examinadas. En este caso hay que preveer la reutilización de la fórmula cuando la(s) fórmula(s) obtenida(s) a partir de ella quede(n) inutilizable(s) por haber sido tachada(s). Tal información puede almacenarse, por ejemplo, en la variable que utilizamos para indicar la utilización de la fórmula.

```
14 /* Decidir inferencias */

15 int inferencias()
16 {
17 int x,r;
18 r=ul;
19 for(x=0;x<ul;x++){
20    if((fv+x)->u==35) continue;
21    switch(*((fv+x)->f)){
22       case 75: inf_conj(x); break;
23       case 65: inf_dis(x); break;
24       case 67: inf_cond(x);break;
25       case 69: inf_bi(x); break;
26       case 78: switch(*(((fv+x)->f)+1)){
27           case 78: inf_do_ne(x);break;
28           case 75: inf_ne_conj(x);break;
29           case 69: inf_ne_bi(x);break;
30           case 65: inf_ne_dis(x);break;
31           case 67: inf_ne_cond(x);
32           }
33       }
34    if(ul-r) break;
35    }
36 return ul-r;
37 }
```

La siguiente función (38–65) aplica la regla E∧ y obtiene los términos de la conjunción si no estaban utilizables. En general, en la aplicación de reglas se busca primero la fórmula que se obtendría mediante dicha regla para evitar repetirla si ya está utilizable.

```
38 /* Eliminación de conjunción */

39 void inf_conj(int x)
40 {
41 int j,l,p;
42 l=strlen((fv+x)->f);
43 p=posicion_d((fv+x)->f);
44 if(buscar((fv+x)->f,1,p)<0){
45    (fv+ul)->u=99;
46    (fv+ul)->r=51;
47    (fv+ul)->l1=x;
48    (fv+ul)->l2=-1;
49    (fv+ul)->f=(char*)calloc(p,sizeof(char));
50    if((fv+ul)->f==NULL) falta_memoria(1);
51   for(j=0;j<p-1;j++) *((fv+ul)->f+j)=*((fv+x)->f+j+1);
52    *((fv+ul)->f+j)='\0';
53    ul++;if(ul==fvd) falta_memoria(2);
54    }
55 if(buscar((fv+x)->f,p,l)<0){
56    (fv+ul)->u=99;
57    (fv+ul)->r=51;
58    (fv+ul)->l1=x;
59    (fv+ul)->l2=-1;
60    (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
61    if((fv+ul)->f==NULL) falta_memoria(1);
62    for(j=p;(*((fv+ul)->f+j-p)=*((fv+x)->f+j));j++);
63    ul++;if(ul==fvd) falta_memoria(2);
64    }
65 }
```

La siguiente función (66–100) aplica la regla E∨ . Puesto que no usamos reglas derivadas, han de darse pasos previos (74, 89) en algunos casos para la aplicación directa de la regla.

```
66 /* Eliminación de disyunción */
```

```
 67 void inf_dis(int x)
 68 {
 69 int i,p,k,l;
 70 l=strlen((fv+x)->f);
 71 p=posicion_d((fv+x)->f);
 72 if(buscar((fv+x)->f,1,p)<0){
 73    if((k=negar((fv+x)->f,p,l))>=0){
 74 if(l-p>strlen((fv+k)->f)){introducir_negacion(k);k=ul-1;}
 75       (fv+ul)->u=99;
 76       (fv+ul)->r=52;
 77       (fv+ul)->l1=x;
 78       (fv+ul)->l2=k;
 79       (fv+ul)->f=(char*)calloc(p,sizeof(char));
 80       if((fv+ul)->f==NULL) falta_memoria(1);
 81   for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*((fv+x)->f+i+1);
 82       *((fv+ul)->f+i)='\0';
 83       ul++;if(ul==fvd) falta_memoria(2);
 84       return;
 85       }
 86    }
 87 if(buscar((fv+x)->f,p,l)<0){
 88    if((k=negar((fv+x)->f,1,p))>=0){
 89 if(p-1>strlen((fv+k)->f)){introducir_negacion(k);k=ul-1;}
 90       (fv+ul)->u=99;
 91       (fv+ul)->r=52;
 92       (fv+ul)->l1=x;
 93       (fv+ul)->l2=k;
 94       (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
 95       if((fv+ul)->f==NULL) falta_memoria(1);
 96       for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++) ;
 97       ul++;if(ul==fvd) falta_memoria(2);
 98       }
 99    }
100 }
```

La siguiente función (101–137) aplica la regla E $\rightarrow$ en sentido estricto (107–115) y replantea la deducción para obtener la negación del antecedente si tenemos utilizable la negación del consecuente (116–136). Adviértase que no se trata en propiedad de un *modus tollens*. Supongamos que tenemos utilizables en j y k:

j. $\alpha \rightarrow \beta$

...

k. $\neg\beta$
Lo que hacemos es escribir:
m. $\lceil \alpha$
para que el programa posteriormente obtenga:
n. β E $\rightarrow$, j,m.
...
p. $\lfloor \beta \wedge \neg\beta$ I$\wedge$, k,n.
p+1 $\neg\alpha$ I$\neg$ m-p.

Desde luego que podría escribirse directamente una rutina para obtener la negación del antecedente.

```
101 /* Eliminación de condicional */

102 void inf_cond(int x)
103 {
104 int i,p,k,l;
105 l=strlen((fv+x)->f);
106 p=posicion_d((fv+x)->f);
107 if(buscar((fv+x)->f,p,l)<0
    && (k=buscar((fv+x)->f,1,p))>=0){
108    (fv+ul)->u=99;
109    (fv+ul)->r=53;
110    (fv+ul)->l1=x;
111    (fv+ul)->l2=k;
112    (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
113    for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++) ;
114    ul++;if(ul==fvd) falta_memoria(2);
115    }
116 else{
117    if((negar((fv+x)->f,1,p))<0
       && (negar((fv+x)->f,p,l))>=0){
118       (mar+m)->l=ul;(mar+m)->ma=40;
119       m++;if(m==md) falta_memoria(4);
120       (fv+ul)->u=99;
121       (fv+ul)->r=99;
122       (fv+ul)->l1=(fv+ul)->l2=-1;
123  if(((fv+ul)->f=(char*)calloc(p,sizeof(char)))==NULL)\
124       falta_memoria(1);
125   for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*((fv+x)->f+i+1);
126       *((fv+ul)->f+i)='\0';
127       ul++;if(ul==fvd) falta_memoria(2);
```

```
128         us=ul;
129         *(fd+cp)=(char*)calloc(p+1,sizeof(char));
130         if(*(fd+cp)==NULL) falta_memoria(1);
131         **(fd+cp)='N';
132         strcat(*(fd+cp),(fv+ul-1)->f);
133         *(s+cp)=ul-1;
134         cp++;if(cp==fdd) falta_memoria(3);
135         }
136      }
137 }
```

La siguiente función (138–173) aplica la regla E↔.

```
138 /* Eliminación de bicondicional */

139 void inf_bi(int x)
140 {
141 int i,k,l,p;
142 char *d,*e;
143 l=strlen((fv+x)->f);
144 p=posicion_d((fv+x)->f);
145 d=(char*)calloc(l+1,sizeof(char));e=(char*)calloc(l+1,\
146 sizeof(char));if(d==NULL||e==NULL) falta_memoria(1);
147 *d=*e='C';
148 for(i=1;(*(d+i)=*((fv+x)->f+i));i++);
149 k=l-p;
150 for(i=0;i+p<l;i++) *(e+i+1)=*((fv+x)->f+i+p);
151 for(i=1;i<p;i++) *(e+k+i)=*((fv+x)->f+i);*(e+k+i)='\0';
152 if(buscar(d,0,l)<0){
153    (fv+ul)->u=99;
154    (fv+ul)->r=54;
155    (fv+ul)->l1=x;
156    (fv+ul)->l2=-1;
157 if(((fv+ul)->f=(char*)calloc(l+1,sizeof(char)))==NULL)\
158    falta_memoria(1);
159    strcpy((fv+ul)->f,d);
160    ul++;if(ul==fvd) falta_memoria(2);
161    }
162 if(buscar(e,0,l)<0){
163    (fv+ul)->u=99;
164    (fv+ul)->r=54;
```

```
165     (fv+ul)->l1=x;
166     (fv+ul)->l2=-1;
167 if(((fv+ul)->f=(char*)calloc(l+1,sizeof(char)))==NULL)\
168     falta_memoria(1);
169     strcpy((fv+ul)->f,e);
170     ul++;if(ul==fvd) falta_memoria(2);
171     }
172 free(d); free(e);
173 }
```

La siguiente función (174–189) aplica la regla E¬.

```
174 /* Eliminación de doble negación */

175 void inf_do_ne(int x)
176 {
177 int i,l;
178 l=strlen((fv+x)->f);
179 if(buscar((fv+x)->f,2,l)<0){
180     (fv+ul)->u=99;
181     (fv+ul)->r=50;
182     (fv+ul)->l1=x;
183     (fv+ul)->l2=-1;
184     (fv+ul)->f=(char*)calloc(l-1,sizeof(char));
185     if((fv+ul)->f==NULL) falta_memoria(1);
186     for(i=0;(*((fv+ul)->f+i)=*((fv+x)->f+i+2));i++);
187     ul++;if(ul==fvd) falta_memoria(2);
188     }
189 }
```

Aunque no existe ninguna regla de inferencia que pueda aplicarse directamente a la negación de una conjunción, la siguiente función (190–234) permite obtener líneas utilizables en algunos casos. Si existen utilizables los dos términos de la conjunción negada, les aplica la regla I∧ para, posteriormente, obtener una contradicción (199–202). Si está utilizable un término, supone el otro para obtener posteriormente su negación (206–234).

```
190 /* Negación de la conjunción */

191 void inf_ne_conj(int x)
192 {
```

```
193 int i,p,k,h,l;
194 char *a;
195 l=strlen((fv+x)->f);
196 a=(fv+x)->f+1;
197 p=posicion_d(a);
198 k=buscar(a,1,p);h=buscar(a,p,l);
199 if(k>=0 && h>=0){
200    introducir_conjuncion(k,h);
201    return;
202    }
203 if(h<0 && k<0) return;
204 if((k>=0 && negar(a,p,l-1)>=0)
205 || (h>=0 && negar(a,1,p)>=0)) return;
206 (mar+m)->l=ul;(mar+m)->ma=40;
207 m++;if(m==md) falta_memoria(4);
208 (fv+ul)->u=99;
209 (fv+ul)->r=99;
210 (fv+ul)->l1=(fv+ul)->l2=-1;
211 if(k>=0){
212 if(((fv+ul)->f=(char*)calloc(l-p,sizeof(char)))==NULL)\
213    falta_memoria(1);
214    *(fd+cp)=(char*)calloc(l-p+1,sizeof(char));
215    if(*(fd+cp)==NULL) falta_memoria(1);
216    **(fd+cp)='N';
217    for(i=p;(*((fv+ul)->f+i-p)=*(a+i));i++) ;
218    strcat(*(fd+cp),(fv+ul)->f);
219    }
220 else{
221  if(((fv+ul)->f=(char*)calloc(p,sizeof(char)))==NULL)\
222    falta_memoria(1);
223    for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*(a+i+1);
224    *((fv+ul)->f+i)='\0';
225    *(fd+cp)=(char*)calloc(p+1,sizeof(char));
226    if(*(fd+cp)==NULL) falta_memoria(1);
227    **(fd+cp)='N';
228    strcat(*(fd+cp),(fv+ul)->f);
229    }
230 *(s+cp)=ul;
231 ul++;if(ul==fvd) falta_memoria(2);
232 us=ul;
233 cp++;if(cp==fdd) falta_memoria(3);
234 }
```

El fichero PDALP04.c contiene las rutinas básicas que controlan la apertura de supuestos.

```
1 /* Programa PDALP00.
2    Fichero PDALP04.c */

3 /* Librerías */

4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <conio.h>
8 #include "pdalp00.h"

9 /* Variables globales externas */

10 extern int cp,ul,us,m,md,fdd,fvd,*s;
11 extern char **fd;
12 extern struct FORMULAS_VALIDAS *fv;
13 extern struct MARCAS *mar;
```

La siguiente función establece la apertura de supuestos en función del tipo de la fórmula que en ese momento sea la última de la pila de demostraciones y decide la apertura de nuevos supuestos para subfórmulas de fórmulas utilizables.

```
14 /* Rutina de abrir supuestos */

15 int supuestos()
16 {
17 int r;
18 r=0;
19 switch(*(*(fd+cp-1))){
20    case 75: r=supuestos_conj();break;
21    case 69: r=supuestos_bi(); break;
22    case 67: r=supuestos_cond(); break;
23    case 78: r=supuestos_ne(); break;
24    case 65:
25    case 112: r=suponer_negacion();
26    }
27 if(!r) r=supuestos_nuevos();
28 return r;
29 }
```

La siguiente función (30–48) escribe como supuesto la negación de una fórmula a demostrar.

```
30 /* Suponer la negación */

31 int suponer_negacion()
32 {
33 int l;
34 l=strlen(*(fd+cp-1));
35 if(negar(*(fd+cp-1),0,l)>=0) return 0;
36 (fv+ul)->f=(char*)calloc(l+2,sizeof(char));
37 if((fv+ul)->f==NULL) falta_memoria(1);
38 *((fv+ul)->f)=78;
39 strcat((fv+ul)->f,*(fd+cp-1));
40 (mar+m)->l=ul;(mar+m)->ma=40;
41 m++;if(m==md) falta_memoria(4);
42 (fv+ul)->u=99;
43 (fv+ul)->r=99;
44 (fv+ul)->l1=(fv+ul)->l2=-1;
45 us=++ul;if(ul==fvd) falta_memoria(2);
46 *(s+cp-1)=ul-1;
47 return 1;
48 }
```

La función siguiente (49–94) gestiona las subfórmulas que han de suponerse con vistas a obtener nuevas fórmulas utilizables. Queda como último recurso para garantizar que, si la fórmula es deducible, obtengamos efectivamente una deducción de la misma. Para que la estrategia deductiva sea completa, esta función ha de aplicarse sistemáticamente. Si se admiten reglas derivadas para la negación del bicondicional y la conjunción —por ejemplo, en términos de una disyunción—, y se admite también la equivalencia por definición entre condicional y disyunción, una rutina eficiente para apertura de nuevos supuestos se limitará a suponer un término de todas las disyunciones (66–74).

```
49 /* Apertura de nuevos supuestos */

50 int supuestos_nuevos()
51 {
52 int h,r,p,i,l;
53 char *a,*b;
54 r=0;
```

```
55 for(h=0;h<ul;h++){
56    if((fv+h)->u==35) continue;
57    l=strlen((fv+h)->f);
58    if(*((fv+h)->f)=='C'){
59       p=posicion_d((fv+h)->f);
60       a=(char*)calloc(p,sizeof(char));
61       if(!a) falta_memoria(1);
62       for(i=0;i<p-1;i++) *(a+i)=*((fv+h)->f+i+1);
         *(a+i)='\0';
63       if((r=subformulas(a))) break;
64       else {free(a); continue;}
65       }
66    if(*((fv+h)->f)=='A'){
67       p=posicion_d((fv+h)->f);
68       a=(char*)calloc(p+1,sizeof(char));
69       if(!a) falta_memoria(1);
70       *a='N';
71    for(i=1;i<p;i++) *(a+i)=*((fv+h)->f+i);*(a+i)='\0';
72       if((r=subformulas(a))) break;
73       else {free(a); continue;}
74       }
75    if(!strncmp((fv+h)->f,"NK",2)){
76       b=(fv+h)->f+1;
77       p=posicion_d(b);
78       a=(char*)calloc(p,sizeof(char));
79       if(!a) falta_memoria(1);
80       for(i=0;i<p-1;i++) *(a+i)=*(b+i+1);*(a+i)='\0';
81       if((r=subformulas(a))) break;
82       else {free(a); continue;}
83       }
84    if(!strncmp((fv+h)->f,"NE",2)){
85       a=(char*)calloc(l,sizeof(char));
86       if(!a) falta_memoria(1);
87       *a='C';
88       for(i=1;(*(a+i)=*((fv+h)->f+i+1));i++);
89       if((r=subformulas(a))) break;
90       else {free(a); continue;}
91       }
92    }
93 return r;
94 }
```

La siguiente función (95–120) introduce en la pila de demostraciones los términos no demostrados de una conjunción, que pasan así a convertirse en los objetivos deductivos inmediatos.

```
 95 /* Supuestos para la conjunción */
 96 int supuestos_conj()
 97 {
 98 int p,i,l;
 99 l=strlen(*(fd+cp-1));
100 for(i=0;(*(*(fd+cp-1)+i));i++);
101 p=posicion_d(*(fd+cp-1));
102 if(buscar(*(fd+cp-1),1,p)<0){
103    *(fd+cp)=(char*)calloc(p,sizeof(char));
104    if(*(fd+cp)==NULL) falta_memoria(1);
105    for(i=0;i<p-1;i++) *(*(fd+cp)+i)=*(*(fd+cp-1)+i+1);
106    *(*(fd+cp)+i)='\0';
107    *(s+cp)=-1;
108    cp++;if(cp==fdd) falta_memoria(3);
109    return 1;
110    }
111 if(buscar(*(fd+cp-1),p,l)<0){
112    *(fd+cp)=(char*)calloc(l-p+1,sizeof(char));
113    if(*(fd+cp)==NULL) falta_memoria(1);
114    for(i=p;(*(*(fd+cp)+i-p)=*(*(fd+cp-1)+i));i++) ;
115    *(s+cp)=-1;
116    cp++;if(cp==fdd) falta_memoria(3);
117    return 1;
118    }
119 return 0;
120 }
```

La siguiente función (121–156) introduce en la pila de demostraciones los condicionales no demostrados en que se descompone lógicamente un bicondicional.

```
121 /* Supuestos para el bicondicional */

122 int supuestos_bi()
123 {
124 int i,p,l;
125 char *b,*c,*d,*e;
126 l=strlen(*(fd+cp-1));
```

```
127 p=posicion_d(*(fd+cp-1));
128 if((b=(char*)calloc(p,sizeof(char)))==NULL)
129 falta_memoria(1);
130 if((c=(char*)calloc(l-p+1,sizeof(char)))==NULL)
131 falta_memoria(1);
132 for(i=0;i<p-1;i++)*(b+i)=*(*(fd+cp-1)+i+1);*(b+i)='\0';
133 for(i=p;(*(c+i-p)=*(*(fd+cp-1)+i));i++) ;
134 if((d=(char*)calloc(l+1,sizeof(char)))==NULL)
135 falta_memoria(1);
136 if((e=(char*)calloc(l+1,sizeof(char)))==NULL)
137 falta_memoria(1);
138 *d=*e='C';strcat(d,b);strcat(d,c);
    strcat(e,c);strcat(e,b);
139 free(b);free(c);
140 if(buscar(d,0,1)<0){
141    *(fd+cp)=d;
142    *(s+cp)=-1;
143    cp++;if(cp==fdd) falta_memoria(3);
144    free(e);
145    return 1;
146    }
147 if(buscar(e,0,1)<0){
148    *(fd+cp)=e;
149    *(s+cp)=-1;
150    cp++;if(cp==fdd) falta_memoria(3);
151    free(d);
152    return 1;
153    }
154 free(d);free(e);
155 return 0;
156 }
```

La siguiente función (157–180) escribe el antecedente de un condicional a demostrar como línea utilizable e introduce en la pila de demostraciones el consecuente.

```
157 /* Supuestos para el condicional */

158 int supuestos_cond()
159 {
160 int i,p,l;
161 l=strlen(*(fd+cp-1));
```

```
162 p=posicion_d(*(fd+cp-1));
163 *(fd+cp)=(char*)calloc(l-p+1,sizeof(char));
164 if(*(fd+cp)==NULL) falta_memoria(1);
165 for(i=p;(*(*(fd+cp)+i-p)=*(*(fd+cp-1)+i));i++) ;
166 cp++;if(cp==fdd) falta_memoria(3);
167 (mar+m)->l=ul;(mar+m)->ma=40;
168 m++;if(m==md) falta_memoria(4);
169 (fv+ul)->u=99;
170 (fv+ul)->r=99;
171 (fv+ul)->l1=(fv+ul)->l2=-1;
172 (fv+ul)->f=(char*)calloc(p,sizeof(char));
173 if((fv+ul)->f==NULL) falta_memoria(1);
174 for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*(*(fd+cp-2)+i+1);
175 *((fv+ul)->f+i)='\0';
176 us=++ul;if(ul==fvd) falta_memoria(2);
177 *(s+cp-2)=ul-1;
178 *(s+cp-1)=-1;
179 return 1;
180 }
```

La siguiente función (181–206) determina el procedimiento a seguir cuando la última fórmula a demostrar sea una negación. Si se trata de una doble negación, se introduce en la pila de demostraciones sin dos negaciones (186–193). En otro caso se supone la fórmula sin la negación tratando de obtener una contradicción.

```
181 /* Supuestos para la negación */

182 int supuestos_ne()
183 {
184 int i,l;
185 l=strlen(*(fd+cp-1));
186 if(*(*(fd+cp-1)+1)=='N'){    /*Doble negación*/
187  if((*(fd+cp)=(char*)calloc(l-1,sizeof(char))) ==NULL)
188    falta_memoria(1);
189    for(i=0;(*(*(fd+cp)+i)=*(*(fd+cp-1)+i+2));i++) ;
190    *(s+cp)=-1;
191    cp++;if(cp==fdd) falta_memoria(3);
192    return 1;
193    }
194 if(negar(*(fd+cp-1),0,1)>=0) return 0;
195 if(((fv+ul)->f=(char*)calloc(l,sizeof(char)))==NULL)
```

```
196 falta_memoria(1);
197 for(i=0;(*((fv+ul)->f+i)=*(*(fd+cp-1)+1+i));i++) ;
198 (mar+m)->l=ul;(mar+m)->ma=40;
199 m++;if(m==md) falta_memoria(4);
200 (fv+ul)->u=99;
201 (fv+ul)->r=99;
202 (fv+ul)->l1=(fv+ul)->l2=-1;
203 us=++ul;if(ul==fvd) falta_memoria(2);
204 *(s+cp-1)=ul-1;
205 return 1;
206 }
```

La siguiente función (207–221) complementa la anterior que gestionaba la apertura de nuevos supuestos. Básicamente, se encarga de que no se repitan los objetivos de deducción innecesariamente.

```
207 /* Suponer las fórmulas elementales */

208 int subformulas(char *a)
209 {
210 int l,x;
211 l=strlen(a);
212 if(buscar(a,0,l)>=0 || negar(a,0,l)>=0) return 0;
213 for(x=0;x<cp;x++) if(!strcmp(a,*(fd+x))) return 0;
214 *(fd+cp)=(char*)calloc(l+1,sizeof(char));
215 if(*(fd+cp)==NULL) falta_memoria(1);
216 strcpy(*(fd+cp),a);
217 free(a);
218 *(s+cp)=-1;
219 cp++; if(cp==fdd) falta_memoria(3);
220 return 1;
221 }
```

El fichero PDALP05.c contiene diversas funciones usadas en el proceso de demostración.

```
1 /* Programa PDALP00.
2    Fichero PDALP05.c */

3 /* Librerías */

4 #include <stdio.h>
```

```
 5 #include <stdlib.h>
 6 #include <string.h>
 7 #include <ctype.h>
 8 #include <conio.h>
 9 #include "pdalp00.h"

10 /* Variables globales  externas */

11 extern int cp,ul,m,us,md,fdd,fvd,*s;
12 extern struct FORMULAS_VALIDAS *fv;
13 extern struct MARCAS *mar;
14 extern char **fd;
```

La siguiente función (15–29) es una de las más usadas en la ejecución del programa. Establece las subfórmulas de una fórmula. De hecho, es tanto su uso que una de las formas de ganar tiempo en las demostraciones puede pasar por su eliminación. Pero ello ha de hacerse en detrimento de la memoria disponible, ya que ha de almacenarse la información necesaria para la determinación de las subfórmulas de una fórmula.

```
15 /* Establecimiento de subfórmulas */

16 int posicion_d(char *a)
17 {
18 int i,h;
19 if(*a=='p'||*a=='N') return 0;
20 h=1;
21 for(i=1;*(a+i);i++){
22    if(*(a+i)=='N' || isdigit(*(a+i))) continue;
23    if(*(a+i)=='p') h--;
24    else h++;
25    if(h==0) break;
26    }
27 for(i++;*(a+i);i++) if(!isdigit(*(a+i))) return i;
28 exit(1);
29 }
```

La siguiente función (30–96) es el equivalente a una regla de repetición, que en nuestro cálculo no es una regla de inferencia como tal.

```
30 /* Repetición de fórmulas */

31 void repeticion(int h)
32 {
```

```
33 int p,l;
34 char a;
35 a=(fv+h)->r;
36 if(a!=99 && a!=40 && a!=43){
37    int j,k;
38    j=(fv+h)->l1;k=(fv+h)->l2;
39    if(a==98 || (j>=0 && (fv+j)->u!=35 && (k<0 ||
40    (fv+k)->u!=35))){
41       (fv+ul)->f=(fv+h)->f;
42       (fv+ul)->r=(fv+h)->r;
43       (fv+ul)->u=99;
44       (fv+ul)->l1=(fv+h)->l1;
45       (fv+ul)->l2=(fv+h)->l2;
46       ul++;if(ul==fvd) falta_memoria(2);
47       return;
48       }
49    }
50 l=strlen((fv+h)->f);
51 (fv+ul)->u=35;
52 (mar+m)->l=ul;(mar+m)->ma=40;
53 m++;if(m==md) falta_memoria(4);
54 if(*((fv+h)->f)=='N'){
55   if(((fv+ul)->f=(char*)calloc(l,sizeof(char)))==NULL)
56    falta_memoria(1);
57    for(p=0;(*((fv+ul)->f+p)=*((fv+h)->f+p+1));p++) ;
58    }
59 else{
60 if(((fv+ul)->f=(char*)calloc(l+2,sizeof(char)))==NULL)
61    falta_memoria(1);
62    *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+h)->f);
63    }
64 (fv+ul)->r=99;
65 (fv+ul)->l1=(fv+ul)->l2=-1;
66 ul++;if(ul==fvd) falta_memoria(2);
67 (mar+m)->l=ul;(mar+m)->ma=60;
68 m++;if(m==md) falta_memoria(4);
69 (fv+ul)->u=35;
70 (fv+ul)->r=41;
71 (fv+ul)->l1=h;
72 (fv+ul)->l2=ul-1;
73 if(((fv+ul)->f=(char*)calloc(2*l+3,sizeof(char)))==NULL)
```

```
74 falta_memoria(1);
75 *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+h)->f);
76 strcat((fv+ul)->f,(fv+ul-1)->f);
77 ul++;if(ul==fvd) falta_memoria(2);
78 (fv+ul)->u=99;
79 (fv+ul)->r=40;
80 (fv+ul)->l1=ul-2;
81 (fv+ul)->l2=ul-1;
82 if(((fv+ul)->f=(char*)calloc(l+3,sizeof(char)))==NULL)
83 falta_memoria(1);
84 *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-2)->f);
85 ul++;if(ul==fvd) falta_memoria(2);
86 if(*((fv+h)->f)!='N'){
87    (fv+ul)->f=(char*)calloc(l+1,sizeof(char));
88    if((fv+ul)->f==NULL) falta_memoria(1);
89    strcpy((fv+ul)->f,(fv+h)->f);
90    (fv+ul)->u=99;
91    (fv+ul)->r=50;
92    (fv+ul)->l1=ul-1;
93    (fv+ul)->l2=-1;
94    ul++;if(ul==fvd) falta_memoria(2);
95    }
96 }
```

La siguiente función (97–167) busca si hay dos fórmulas utilizables una de las cuales sea la negación de la otra. Si es éste el caso, escribe las líneas necesarias para que la última fórmula de la pila de demostraciones quede demostrada.

```
97 /* Buscando contradicciones */

98 int contradiccion()
99 {
100 int i,j,r,n,l;
101 r=0;
102 for(i=0;i<ul;i++){
103    if((fv+i)->u==35) continue;
104    if(*((fv+i)->f)!='N') continue;
105    if((n=negar((fv+i)->f,0,strlen((fv+i)->f)))>=0){
106       l=strlen(*(fd+cp-1));
107       if(us-1!=*(s+cp-1)){
108        (fv+ul)->u=35;
```

```
109     (mar+m)->l=ul;(mar+m)->ma=40;
110     m++;if(m==md) falta_memoria(4);
111     if(*(*(fd+cp-1))=='N'){
112        if(((fv+ul)->f=(char*)calloc(l,\
113        sizeof(char)))==NULL) falta_memoria(1);
114       for(j=0;(*((fv+ul)->f+j)=*(*(fd+cp-1)+j+1));j++);
115        }
116     else{
117        if(((fv+ul)->f=(char*)calloc(l+2,\
118        sizeof(char)))==NULL) falta_memoria(1);
119        *((fv+ul)->f)='N';strcat((fv+ul)->f,*(fd+cp-1));
120        }
121     us=ul+1;
122     (fv+ul)->r=99;
123     (fv+ul)->l1=(fv+ul)->l2=-1;
124     ul++;if(ul==fvd) falta_memoria(2);
125     }
126        else{
127     (fv+us-1)->u=35;
128     tachar(ul);
129           }
130        (mar+m)->l=ul;(mar+m)->ma=60;
131        m++;if(m==md) falta_memoria(4);
132        (fv+ul)->u=35;
133        (fv+ul)->r=41;
134        (fv+ul)->l1=i;
135        (fv+ul)->l2=n;
136      if(((fv+ul)->f=(char*)calloc(2+strlen((fv+i)->f)+\
137        strlen((fv+n)->f),sizeof(char)))==NULL)
138        falta_memoria(1);
139        *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+i)->f);
140        strcat((fv+ul)->f,(fv+n)->f);
141        ul++;if(ul==fvd) falta_memoria(2);
142        (fv+ul)->u=99;
143        (fv+ul)->r=40;
144        (fv+ul)->l1=us-1;
145        (fv+ul)->l2=ul-1;
146  if(((fv+ul)->f=(char*)calloc(strlen((fv+us-1)->f)+2,\
147        sizeof(char)))==NULL) falta_memoria(1);
148      *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+us-1)->f);
149        ul++;if(ul==fvd) falta_memoria(2);
```

```
150       if(**(fd+cp-1)!='N'){
151     if(((fv+ul)->f=(char*)calloc(l+1,\
152     sizeof(char)))==NULL) falta_memoria(1);
153     strcpy((fv+ul)->f,*(fd+cp-1));
154     (fv+ul)->u=99;
155     (fv+ul)->r=50;
156     (fv+ul)->l1=ul-1;
157     (fv+ul)->l2=-1;
158     ul++;if(ul==fvd) falta_memoria(2);
159     }
160        ultimo_supuesto();
161        r=1;
162        *(s+cp-1)=-1;
163        }
164     if(r) break;
165     }
166 return r;
167 }
```

La siguiente función (168–180) elimina las líneas no necesarias en una deducción cuando ésta ha obtenido éxito. El procedimiento de eliminación es simple y consiste en revisar la deducción desde el final, la fórmula que queríamos demostrar, anotando las líneas que se han utilizado y las que no.

```
168 /* Eliminar líneas superfluas */

169 void eliminar()
170 {
171 register int i;
172 for(i=0;i<ul-1;i++) (fv+i)->u=98;
173 for(i=ul-1;i>=0;i--){
174     if((fv+i)->u==98) continue;
175     if((fv+i)->l1<0) continue;
176     (fv+(fv+i)->l1)->u=99;
177     if((fv+i)->l2<0) continue;
178     (fv+(fv+i)->l2)->u=99;
179     }
180 }
```

La siguiente función (181–192) realiza una búsqueda sistemática en las líneas utilizables escritas para encontrar una fórmula dada.

```
181 /* Buscar una fórmula utilizable */

182 int buscar(char *a,int h,int k)
183 {
184 register int i,j,z;
185 z=-1;
186 for(i=0;i<ul;i++){
187    if((fv+i)->u==35) continue;
188   for(j=h;j<k;j++) if(*(a+j)!=*((fv+i)->f+j-h)) break;
189    if(j==k) {z=i;break;}
190    }
191 return z;
192 }
```

La siguiente función (193–216) determina si una fórmula dada es la negación de otra fórmula dada.

```
193 /* Una fórmula es la negación de otra */

194 int negacion(char *a,int h)
195 {
196 int l,w,j,z;
197 char *b;
198 z=0;
199 if(h<ul) b=(fv+h)->f;
200 else b=*(fd+h-ul);
201 if(*a=='N' || *b=='N'){
202    l=strlen(a);w=strlen(b);
203    j=l<w?w-l:l-w;
204    if(j==1){
205       if(l>w && *a=='N'){
206    for(j=0;j<w;j++) if(*(b+j)!=*(a+j+1)) break;
207    if(j==w) z=1;
208    }
209       if(w>l && *b=='N'){
210    for(j=0;j<l;j++) if(*(a+j)!=*(b+j+1)) break;
211    if(j==l) z=1;
212    }
213       }
214    }
215 return z;
216 }
```

La siguiente función (217–239) determina si una fórmula dada es la negación de alguna de las fórmulas utilizables. Obsérvese que no es idéntica a la anterior.

```
217 /* Buscar una negación */

218 int negar(char *a, int h, int k)
219 {
220 int j,i,l,w,z;
221 z=-1;
222 for(i=0;i<ul;i++){
223    if((fv+i)->u==35) continue;
224    if(*(a+h)!='N' && *((fv+i)->f)!='N') continue;
225    l=k-h;w=strlen((fv+i)->f);
226    j=l<w?w-l:l-w;
227    if(j==0 || j>1) continue;
228    if(l>w && *(a+h)=='N'){
229 for(j=0;j<w;j++) if(*((fv+i)->f+j)!=*(a+j+h+1)) break;
230       if(j==w) z=i;
231       }
232    if(w>l && *((fv+i)->f)=='N'){
233 for(j=0;j<l;j++) if(*(a+j+h)!=*((fv+i)->f+j+1)) break;
234       if(j==l) z=i;
235       }
236    if(z>=0) break;
237    }
238 return z;
239 }
```

La siguiente función (240–273) introduce una doble negación.

```
240 /* Introducir negación */

241 void introducir_negacion(int i)
242 {
243 int l;
244 l=strlen((fv+i)->f);
245 (mar+m)->l=ul;(mar+m)->ma=40;
246 m++;if(m==md) falta_memoria(4);
247 (fv+ul)->u=35;
248 (fv+ul)->r=99;
249 (fv+ul)->l1=(fv+ul)->l2=-1;
```

```
250 if(((fv+ul)->f=(char*)calloc(l+2,sizeof(char)))==NULL)
251 falta_memoria(1);
252 *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+i)->f);
253 ul++;if(ul==fvd) falta_memoria(2);
254 (mar+m)->l=ul;(mar+m)->ma=60;
255 m++;if(m==md) falta_memoria(4);
256 (fv+ul)->u=35;
257 (fv+ul)->r=41;
258 (fv+ul)->l1=i;
259 (fv+ul)->l2=ul-1;
260 if(((fv+ul)->f=(char*)calloc(l*2+3,sizeof(char)))==NULL)
261 falta_memoria(1);
262 *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+i)->f);
263 strcat((fv+ul)->f,(fv+ul-1)->f);
264 ul++;if(ul==fvd) falta_memoria(2);
265 (fv+ul)->u=99;
266 (fv+ul)->r=40;
267 (fv+ul)->l1=ul-2;
268 (fv+ul)->l2=ul-1;
269 if(((fv+ul)->f=(char*)calloc(l+3,sizeof(char)))==NULL)
270 falta_memoria(1);
271 *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-2)->f);
272 ul++;if(ul==fvd) falta_memoria(2);
273 }
```

El fichero PDALP06.c contiene contiene diversas rutinas de inferencia que se comportan en realidad como reglas derivadas de inferencia escribiendo varias líneas a partir de ciertas negaciones para las cuales no hay ninguna regla que sea aplicable directamente.

```
1 /*  Programa PDALP00.
2     Fichero PDALP06.c */
3 /* Librerías */

4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <conio.h>
7 #include <string.h>
8 #include "pdalp00.h"

9 /* Variables globales externas */
```

```
10 extern int cp,ul,us,m,md,fdd,fvd;
11 extern char **fd;
12 extern struct FORMULAS_VALIDAS *fv;
13 extern struct MARCAS *mar;
```

La siguiente función (14–85) gestiona la posibilidad de obtener nuevas líneas utilizables a partir de la negación de un bicondicional. Si están utilizables los dos sentidos, se introduce un bicondicional para obtener una contradicción (33–37). Si está utilizable algún sentido, se obtiene la negación del otro (44–85).

```
14 /* Negación de bicondicional */

15 void inf_ne_bi(int x)
16 {
17 int i,p,l,k,h,r,s;
18 char *a,*b,*c,*d,*e;
19 l=strlen((fv+x)->f);
20 if((a=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
21 for(i=0;(*(a+i)=*((fv+x)->f+1+i))!=0;i++) ;
22 p=posicion_d(a);
23 if((b=(char*)calloc(p,sizeof(char)))==NULL)
   falta_memoria(1);
24 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
25 falta_memoria(1);
26 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
27 for(i=p;(*(c+i-p)=*(a+i));i++) ;
28 if((d=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
29 if((e=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
30 *d=*e='C';strcat(d,b);strcat(d,c);strcat(e,c);
   strcat(e,b);
31 free(b);free(c);
32 k=buscar(d,0,l-1);h=buscar(e,0,l-1);
33 if(k>=0 && h>=0){
34    introducir_bicondicional(k,h);
35    free(a);free(d);free(e);
36    return;
37    }
38 if(h<0 && k<0){
```

```
39     free(a);free(d);free(e);
40     return ;
41     }
42 r=negar(d,0,l-1);
43 s=negar(e,0,l-1);
44 if((k>=0 && s>=0) || (h>=0 && r>=0)){
45     free(a);free(d);free(e);return;}
46 (mar+m)->l=ul;(mar+m)->ma=40;
47 m++;if(m==md) falta_memoria(4);
48 (fv+ul)->u=35;
49 (fv+ul)->r=99;
50 (fv+ul)->l1=(fv+ul)->l2=-1;
51 if(((fv+ul)->f=(char*)calloc(l,sizeof(char)))==NULL)
52 falta_memoria(1);
53 if(k>=0) strcpy((fv+ul)->f,e);
54 else strcpy((fv+ul)->f,d);
55 ul++;if(ul==fvd) falta_memoria(2);
56 (mar+m)->l=ul;(mar+m)->ma=50;
57 m++;if(m==md) falta_memoria(4);
58 (fv+ul)->u=35;
59 (fv+ul)->r=44;
60 if(k>=0) (fv+ul)->l1=k;
61 else (fv+ul)->l1=h;
62 (fv+ul)->l2=ul-1;
63 (fv+ul)->f=a;
64 ul++;if(ul==fvd) falta_memoria(2);
65 (mar+m)->l=ul;(mar+m)->ma=60;
66 m++;if(m==md) falta_memoria(4);
67 (fv+ul)->u=35;
68 (fv+ul)->r=41;
69 (fv+ul)->l1=x;
70 (fv+ul)->l2=ul-1;
71 if(((fv+ul)->f=(char*)calloc(l*2+1,sizeof(char)))==NULL)
72 falta_memoria(1);
73 *((fv+ul)->f)='K';strcat((fv+ul)->f,a);
74 strcat((fv+ul)->f,(fv+x)->f);
75 ul++;if(ul==fvd) falta_memoria(2);
76 (fv+ul)->u=99;
77 (fv+ul)->r=40;
78 (fv+ul)->l1=ul-3;
79 (fv+ul)->l2=ul-1;
```

```
 80 if(((fv+ul)->f=(char*)calloc(l+1,sizeof(char)))==NULL)
 81 falta_memoria(1);
 82 *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-3)->f);
 83 ul++;if(ul==fvd) falta_memoria(2);
 84 free(d);free(e);
 85 }
```

La siguiente función (86–190) permite obtener nuevas líneas utilizables a partir de la negación de una disyunción. Si existe un término de la disyunción, se introduce a partir de ella la disyunción para obtener una contradicción (101–110). Si no está utilizable la negación de cualquier término, se obtiene (111–150 y 152–190).

```
 86 /* Negación de la disyunción */

 87 void inf_ne_dis(int x)
 88 {
 89 int i,h,k,l,p;
 90 char *a,*b,*c;
 91 l=strlen((fv+x)->f);
 92 if((a=(char*)calloc(l,sizeof(char)))==NULL)
    falta_memoria(1);
 93 for(i=0;(*(a+i)=*((fv+x)->f+1+i))!='\0';i++) ;
 94 p=posicion_d(a);
 95 if((b=(char*)calloc(p,sizeof(char)))==NULL)
    falta_memoria(1);
 96 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
 97 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
 98 falta_memoria(1);
 99 for(i=p;(*(c+i-p)=*(a+i));i++) ;
100 k=h=-1;
101 if((h=buscar(a,1,p))>=0 || (k=buscar(a,p,l-1))>=0){
102    (fv+ul)->f=a;
103    (fv+ul)->u=99;
104    (fv+ul)->r=42;
105    (fv+ul)->l1=h<k?k:h;
106    (fv+ul)->l2=-1;
107    ul++;if(ul==fvd) falta_memoria(2);
108    free(b);free(c);
109    return;
110    }
111 if((h=negar(a,1,p))<0){
```

```
112     (mar+m)->l=ul;(mar+m)->ma=40;
113     m++;if(m==md) falta_memoria(4);
114     (fv+ul)->u=35;
115     (fv+ul)->r=99;
116     (fv+ul)->l1=(fv+ul)->l2=-1;
117    if(((fv+ul)->f=(char*)calloc(p,sizeof(char)))==NULL)
118     falta_memoria(1);
119     strcpy((fv+ul)->f,b);
120     ul++;if(ul==fvd) falta_memoria(2);
121     (mar+m)->l=ul;(mar+m)->ma=50;
122     m++;if(m==md) falta_memoria(4);
123     (fv+ul)->u=35;
124     (fv+ul)->r=42;
125     (fv+ul)->l1=ul-1;
126     (fv+ul)->l2=-1;
127     (fv+ul)->f=a;
128     ul++;if(ul==fvd) falta_memoria(2);
129     (mar+m)->l=ul;(mar+m)->ma=60;
130     m++;if(m==md) falta_memoria(4);
131     (fv+ul)->u=35;
132     (fv+ul)->r=41;
133     (fv+ul)->l1=ul-1;
134     (fv+ul)->l2=x;
135 if(((fv+ul)->f=(char*)calloc(l*2+1,sizeof(char)))==NULL)
136     falta_memoria(1);
137     *((fv+ul)->f)='K';strcat((fv+ul)->f,a);
138     strcat((fv+ul)->f,(fv+x)->f);
139     ul++;if(ul==fvd) falta_memoria(2);
140     (fv+ul)->u=99;
141     (fv+ul)->r=40;
142     (fv+ul)->l1=ul-3;
143     (fv+ul)->l2=ul-1;
144 if(((fv+ul)->f=(char*)calloc(p+1,sizeof(char)))==NULL)
145     falta_memoria(1);
146     *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
147     ul++;if(ul==fvd) falta_memoria(2);
148     free(b);free(c);
149     return;
150     }
151 if((k=negar(a,p,l-1))>=0)
```

```
    {free(a);free(b);free(c);return;}
152 (mar+m)->l=ul;(mar+m)->ma=40;
153 m++;if(m==md) falta_memoria(4);
154 (fv+ul)->u=35;
155 (fv+ul)->r=99;
156 (fv+ul)->l1=(fv+ul)->l2=-1;
157 if(((fv+ul)->f=(char*)calloc(l-p,sizeof(char)))==NULL)
158 falta_memoria(1);
159 strcpy((fv+ul)->f,c);
160 ul++;if(ul==fvd) falta_memoria(2);
161 (mar+m)->l=ul;(mar+m)->ma=50;
162 m++;if(m==md) falta_memoria(4);
163 (fv+ul)->u=35;
164 (fv+ul)->r=42;
165 (fv+ul)->l1=ul-1;
166 (fv+ul)->l2=-1;
167 (fv+ul)->f=a;
168 ul++;if(ul==fvd) falta_memoria(2);
169 (mar+m)->l=ul;(mar+m)->ma=60;
170 m++;if(m==md) falta_memoria(4);
171 (fv+ul)->u=35;
172 (fv+ul)->r=41;
173 (fv+ul)->l1=ul-1;
174 (fv+ul)->l2=x;
175 if(((fv+ul)->f=(char*)calloc(l*2+1,sizeof(char)))==NULL)
176 falta_memoria(1);
177 *((fv+ul)->f)='K';strcat((fv+ul)->f,a);
178 strcat((fv+ul)->f,(fv+x)->f);
179 ul++;if(ul==fvd) falta_memoria(2);
180 (fv+ul)->u=99;
181 (fv+ul)->r=40;
182 (fv+ul)->l1=ul-3;
183 (fv+ul)->l2=ul-1;
184 if(((fv+ul)->f=(char*)calloc(l-p+1,sizeof(char)))==NULL)
185 falta_memoria(1);
186 *((fv+ul)->f)='N';strcat((fv+ul)->f,c);
187 ul++;if(ul==fvd) falta_memoria(2);
188 free(c);free(b);
189 return;
190 }
```

La siguiente función (191–351) permite obtener líneas utilizables a partir de la negación de un condicional: el antecedente y la negación del consecuente. La longitud relativamente larga de esta función se debe a que se contemplan todos los casos que pueden ocurrir para que la deducción resultante sea la más elegante posible.

```
191 /* Negación de condicional */
192 void inf_ne_cond(int x)
193 {
194 int i,l,p,k,h;
195 char *a,*b,*c;
196 l=strlen((fv+x)->f);
197 if((a=(char*)calloc(l,sizeof(char)))==NULL)
    falta_memoria(1);
198 for(i=0;(*(a+i)=*((fv+x)->f+1+i))!=0;i++) ;
199 p=posicion_d(a);
200 if((b=(char*)calloc(p,sizeof(char)))==NULL)
    falta_memoria(1);
201 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
202 falta_memoria(1);
203 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
204 for(i=p;(*(c+i-p)=*(a+i));i++) ;
205 if((k=buscar(a,1,p))>=0 && (h=negar(a,p,l-1))>=0) {
206    free(a);free(b);free(c);return;}
207 (mar+m)->l=ul;(mar+m)->ma=40;
208 m++;if(m==md) falta_memoria(4);
209 (fv+ul)->u=35;
210 (fv+ul)->r=99;
211 (fv+ul)->l1=(fv+ul)->l2=-1;
212 if(k<0){
213    if(*b=='N'){
214 if(((fv+ul)->f=(char*)calloc(p-1,sizeof(char)))==NULL)
215       falta_memoria(1);
216       for(i=0;(*((fv+ul)->f+i)=*(b+i+1))!=0;i++) ;
217       }
218    else{
219 if(((fv+ul)->f=(char*)calloc(p+1,sizeof(char)))==NULL)
220       falta_memoria(1);
221       *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
222       }
223    }
224 else{
```

```
225 if(((fv+ul)->f=(char*)calloc(l-p+1,
    sizeof(char)))==NULL)
226    falta_memoria(1);
227    strcpy((fv+ul)->f,c);
228    }
229 ul++;if(ul==fvd) falta_memoria(2);
230 (mar+m)->l=ul;(mar+m)->ma=40;
231 m++;if(m+15==md) falta_memoria(4);
232 (mar+m)->l=ul;(mar+m)->ma=50;m++;
233 (fv+ul)->u=35;
234 (fv+ul)->r=99;
235 (fv+ul)->l1=(fv+ul)->l2=-1;
236 (fv+ul)->f=b;
237 ul++;if(ul==fvd) falta_memoria(2);
238 (mar+m)->l=ul;(mar+m)->ma=40;m++;
239 (mar+m)->l=ul;(mar+m)->ma=50;m++;
240 (mar+m)->l=ul;(mar+m)->ma=50;m++;
241 (fv+ul)->u=35;
242 (fv+ul)->r=99;
243 (fv+ul)->l1=(fv+ul)->l2=-1;
244 if(*c=='N'){
245 if(((fv+ul)->f=(char*)calloc(l-p,sizeof(char)))==NULL)
246    falta_memoria(1);
247    for(i=0;(*((fv+ul)->f+i)=*(c+i+1))!=0;i++) ;
248    }
249 else{
250    if(((fv+ul)->f=(char*)calloc(l-p+2,
       sizeof(char)))==NULL)
251    falta_memoria(1);
252    *((fv+ul)->f)='N';strcat((fv+ul)->f,c);
253    }
254 ul++;if(ul==fvd) falta_memoria(2);
255 (mar+m)->l=ul;(mar+m)->ma=60;m++;
256 (mar+m)->l=ul;(mar+m)->ma=50;m++;
257 (mar+m)->l=ul;(mar+m)->ma=50;m++;
258 (fv+ul)->u=35;
259 (fv+ul)->r=41;
260 (fv+ul)->l1=ul-3;
261 if(k<0){
262 if(((fv+ul)->f=(char*)calloc(p*2+1,sizeof(char)))==NULL)
263    falta_memoria(1);
```

```
264     *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+ul-3)->f);
265     strcat((fv+ul)->f,(fv+ul-2)->f);
266     (fv+ul)->l2=ul-2;
267     }
268 else{
269     if(((fv+ul)->f=(char*)calloc((l-p)*2+3,\
270     sizeof(char)))==NULL)
271     falta_memoria(1);
272     *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+ul-3)->f);
273     strcat((fv+ul)->f,(fv+ul-1)->f);
274     (fv+ul)->l2=ul-1;
275     }
276 ul++;if(ul==fvd) falta_memoria(2);
277 if(*c=='N'){
278     (mar+m)->l=ul;(mar+m)->ma=60;m++;
279     (mar+m)->l=ul;(mar+m)->ma=50;m++;
280     }
281 else{
282     (mar+m)->l=ul;(mar+m)->ma=50;m++;
283     (mar+m)->l=ul;(mar+m)->ma=50;m++;
284     }
285 (fv+ul)->u=35;
286 (fv+ul)->r=40;
287 (fv+ul)->l1=ul-2;
288 (fv+ul)->l2=ul-1;
289 if(((fv+ul)->f=(char*)calloc(strlen((fv+ul-2)->f)+2,\
290 sizeof(char)))==NULL) falta_memoria(1);
291 *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-2)->f);
292 ul++;if(ul==fvd) falta_memoria(2);
293 if(*c!='N'){
294     (mar+m)->l=ul;(mar+m)->ma=60;m++;
295     (mar+m)->l=ul;(mar+m)->ma=50;m++;
296     (fv+ul)->u=35;
297     (fv+ul)->r=50;
298     (fv+ul)->l1=ul-1;
299     (fv+ul)->l2=-1;
300 if(((fv+ul)->f=(char*)calloc(l-p+1,sizeof(char)))==NULL)
301     falta_memoria(1);
302     strcpy((fv+ul)->f,c);
303     ul++;if(ul==fvd) falta_memoria(2);
304     }
```

```
305 (mar+m)->l=ul;(mar+m)->ma=50;m++;
306 (fv+ul)->u=35;
307 (fv+ul)->r=43;
308 if(*c=='N') (fv+ul)->l1=ul-4;
309 else (fv+ul)->l1=ul-5;
310 (fv+ul)->l2=ul-1;
311 (fv+ul)->f=a;
312 ul++;if(ul==fvd) falta_memoria(2);
313 (mar+m)->l=ul;(mar+m)->ma=60;m++;
314 (fv+ul)->u=35;
315 (fv+ul)->r=41;
316 (fv+ul)->l1=x;
317 (fv+ul)->l2=ul-1;
318 if(((fv+ul)->f=(char*)calloc(l*2+1,sizeof(char)))==NULL)
319 falta_memoria(1);
320 *((fv+ul)->f)='K';strcat((fv+ul)->f,a);
321 strcat((fv+ul)->f,(fv+x)->f);
322 ul++;if(ul==fvd) falta_memoria(2);
323 (fv+ul)->u=99;
324 (fv+ul)->r=40;
325 (fv+ul)->l2=ul-1;
326 if(*c=='N'){
327  if(((fv+ul)->f=(char*)calloc(strlen((fv+ul-7)->f)+2,\
328    sizeof(char)))==NULL) falta_memoria(1);
329    (fv+ul)->l1=ul-7;
330    *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-7)->f);
331    }
332 else{
333  if(((fv+ul)->f=(char*)calloc(strlen((fv+ul-8)->f)+2,\
334    sizeof(char)))==NULL) falta_memoria(1);
335    (fv+ul)->l1=ul-8;
336    *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-8)->f);
337    }
338 ul++;if(ul==fvd) falta_memoria(2);
339 if((k<0 && *b!='N') || (h<0 && *c=='N')){
340    (fv+ul)->u=99;
341    (fv+ul)->r=50;
342    (fv+ul)->l1=ul-1;
343    (fv+ul)->l2=-1;
344  if(((fv+ul)->f=(char*)calloc(strlen((fv+ul-1)->f)-1,\
345    sizeof(char)))==NULL) falta_memoria(1);
```

```
346     for(i=0;(*((fv+ul)->f+i)=*((fv+ul-1)->f+i+2))!=0;
        i++) ;
347     ul++;if(ul==fvd) falta_memoria(2);
348     }
349 free(c);
350 return;
351 }
```

III. 2. Programa PDALP01

Los métodos de deducción en el cálculo G pueden simplificarse si se adopta una estrategia simple y única que parta de la negación de la fórmula a demostrar y trate de obtener contradicciones. Tal método no tiene la elegancia del anterior y, en muchos casos, genera deducciones notablemente más largas. El siguiente programa, PDALP01, está escrito para obtener deducciones conforme a esta estrategia. PDALP01 tiene funciones que son iguales a las de PDALP00, otras contienen modificaciones que afectan al modo como se almacena la información, ya que en PDALP01 se usan menos variables globales. Presentamos los ficheros que componen el programa completo, aunque sólo comentaremos aquellas diferencias significativas respecto a PDALP00. PDALP01 está formado por los ficheros PDALP10.h, PDALP11.c–PDALP14.c.

```
/* Programa PDALP01.
   Fichero PDALP10.h */

/* Estructuras */

struct FORMULAS_VALIDAS{
   char *f;     /*fórmula              */
   char u;      /*utilizable o no      */
   char r;      /*reglas               */
   int l1;      /*líneas de inferencia*/
   int l2;      /*líneas de inferencia*/
   };

struct MARCAS{
   char ma;           /*marca          */
   int l;             /*número de línea*/
   };

/* Definiciones */

#define TFV sizeof(struct FORMULAS_VALIDAS)
#define TM sizeof(struct MARCAS)

/* Funciones prototipo */

/* Fichero pdalp11.c */
void instrucciones(void);
```

```
21  void calculos_dn(void);
22  int simbolos(char*);
23  int es_formula(char*);
24  int simbolo_do(char*);
25  void pausa(void);
26  void falta_memoria(int);
27  void salida_formulas(void);

28  /* Fichero pdalp12.c */
29  void demostrada(void);
30  void introducir_disyuncion(char*,int);
31  void introducir_conjuncion(int,int);
32  void introducir_bicondicional(int,int);
33  void tachar(int,int);
34  int inferencias(void);
35  void inf_conj(int);
36  void inf_dis(int);
37  void inf_cond(int);
38  void inf_bi(int);
39  void inf_do_ne(int);
40  void inf_ne_conj(int);

41  /* Fichero pdalp13.c */
42  int supuestos(void);
43  int subformulas(char*);
44  int posicion_d(char*);
45  int buscar(char*,int,int);
46  int negacion(char*,int);
47  int contradiccion(void);
48  void introducir_negacion(int);
49  int negar(char*,int,int);

50  /* Fichero pdalp14.c */
51  void inf_ne_bi(int);
52  void inf_ne_dis(int);
53  void inf_ne_cond(int);
```

Se observará que han desaprecido varias funciones de PDALP00.
El fichero PDALP11.c contiene las funciones básicas de entrada y salidas con ligeras modificaciones.

```
54  /* Programa PDALP01.
55     Fichero PDALP11.c */

56  /* Librerías */
57  #include <stdio.h>
58  #include <stdlib.h>
59  #include <string.h>
60  #include <ctype.h>
61  #include <conio.h>
62  #include <alloc.h>
63  #include <time.h>
64  #include "pdalp10.h"

65  /* Variables globales */
66  /* Contadores de fórmulas escritas y marcas: ul,m;
67   * último supuesto: us;
68   * marcas y fórmulas válidas
69  * disponibles por asignación dinámica en curso: md, fvd.
70   */
71  int ul,m,us,md,fvd;
72  struct FORMULAS_VALIDAS *fv;
    /*puntero a fórmulas escritas*/
73  struct MARCAS *mar;   /*puntero a marcas   */
```

En este método de deducción no necesitamos una pila de fórmulas a demostrar por lo que todas las variables necesarias para su control han desaparecido.

Las funciones de inicio de programa y de instrucciones generales, que ocuparían las líneas 74–83 y 84–95 del programa, son las mismas que las de PDALP00.

```
 96  /* Introducción de conclusión y premisas */
 97  void calculos_dn()
 98  {
 99  int j;
100  do{
101     j=ul=us=m=0;
102     fv=(struct FORMULAS_VALIDAS*)calloc(300,TFV);
103     if(fv==NULL) falta_memoria(1);
104     mar=(struct MARCAS*)calloc(600,TM);
105     if(mar==NULL) falta_memoria(1);
106     fvd=300;md=600;
107     clrscr();
```

```
108     cputs("Introduzca la conclusión y las premisas, \
109     si las hubiere.\n\r");
110     cputs("Pulse [ENTER] para abandonar o finalizar \
111     la introducción.\n\r");
112     do{
113        int x,y;
114        char *a,*b;
115        if(j==0) cputs("Conclusión:  ");
116        else cprintf("Premisa  %d:  ",j);
117        y=wherey();x=wherex();
118        gotoxy(x+60,y);
119        putch(42);
120        gotoxy(x,y);
121        a=(char*)calloc(62,sizeof(char));
122        if(!a) falta_memoria(1);
123        fflush(stdin);
124        gets(a);
125        if(!*a){
126    for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
127    free(a);
128    break;
129    }
130        b=(char*)calloc(strlen(a)+1,sizeof(char));
131        if(b==NULL) falta_memoria(1);
132        strcpy(b,a);
133        free(a);
134        if(!simbolos(b)){
135    putch(7);
136    cputs("\n\rError en la escritura de la fórmula.");
137           pausa();
138           free(b);
139       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
140           continue;
141           }
142        if(!es_formula(b)){
143    putch(7);
144    cputs("\n\rError en la escritura de la fórmula.");
145           pausa();
146           free(b);
147       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
148           continue;
```

```
149     }
150         if(j==0){
151     (mar+m)->l=ul;(mar+m)->ma=40;
152     m++;
153     (fv+ul)->f=(char*)calloc(strlen(b)+2,sizeof(char));
154     if((fv+ul)->f==NULL) falta_memoria(1);
155     *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
156     free(b);
157     (fv+ul)->u=80;
158     (fv+ul)->r=99;
159     (fv+ul)->l1=(fv+ul)->l2=-1;
160     us=ul;
161     ul++;
162     }
163         else{
164     (fv+ul)->f=b;
165     (fv+ul)->u=80;
166     (fv+ul)->r=98;
167     (fv+ul)->l1=(fv+ul)->l2=-1;
168     ul++;
169     }
170         j++;
171         } while(1);
172      if(j>0) {
173         time_t x;
174         struct tm *z;
175         int t1,t2,t3,t4;
176         x=time(NULL);
177         z=localtime(&x);
178         t1=z->tm_min;t2=z->tm_sec;
179         demostrada();
180         x=time(NULL);
181         z=localtime(&x);
182         t3=z->tm_min;t4=z->tm_sec;
183         cprintf("   (%d:%d)",t3-t1,t4-t2);
184         salida_formulas();
185         }
186      free(fv);free(mar);      /*liberar memoria*/
187      } while(j!=0);
188  }
```

Al haber suprimido la pila de demostraciones, la conclusión se introduce directamente negada como la primera fórmula utilizable (150–162). Las modificaciones en las líneas 173–178, 180–183 son meramente accidentales; se refieren sólo al modo de presentar el tiempo empleado en la deducción por el programa.

Las líneas 189–353 del programa corresponden a las funciones de verificación de símbolos, verificación de fórmulas, establecimiento del símbolo dominante de una fórmula y pausa; todas son semejantes a las del programa PDALP00.

```
354  /* Interrupción por falta de memoria */
355  void falta_memoria(int i)
356  {
357  switch(i){
358     case 1: {cputs("Falta de memoria. ");break;}
359     case 2: fv=(struct FORMULAS_VALIDAS*)
360      realloc(fv,(fvd+50)*TFV);
361      if(fv==NULL) {cputs("Falta de memoria. ");break;}
362      fvd+=50;return;
363  case 4: mar=(struct MARCAS*)realloc(mar,(md+200)*TM);
364     if(mar==NULL) {cputs("Falta de memoria. ");break;}
365      md+=200;return;
366     }
367  putch(7);
368  cprintf("\n\rMemoria libre %u bytes",coreleft());
369  pausa();
370  clrscr();
371  exit(1);
372  }

373  /* Salida de demostraciones */
374  void salida_formulas()
375  {
376  int i,p,j,k,w;
377  char *d,e;
378  cprintf("\n\r(Desea imprimir el resultado? (S/N): ");
379  fflush(stdin);
380  e=toupper(getchar());
381  for(i=0;i<ul;i++){
382     cprintf("\n\r%3d. ",i+1);
383     if(e==83) fprintf(stdprn,"\n\r%3d. ",i+1);
384     for(p=m-1;p>=0;p--){
385        if((mar+p)->l!=i) continue;
```

```
386         switch((mar+p)->ma){
387     case 40: cputs("┌");
388              if(e==83) fputs("┌",stdprn);break;
389     case 50: cputs("│");
390              if(e==83) fputs("│",stdprn);break;
391     case 60: cputs("└");
392              if(e==83) fputs("└",stdprn);
393     }
394         }
```

Las líneas 395–468 corresponden a la traducción de las fórmulas de la notación polaca; puede usarse cualquiera de los mecanismos ya mencionados en el programa PDALP00. La salida de las reglas (469–491) es idéntica a la de PDALP00. Las modificaciones de la presente función se deben a que PDALP01 carece de una rutina para eliminar las líneas no necesarias de la deducción. Si se incorpora dicha función al programa (eliminar líneas superfluas), la salida de las deducciones ha de ser idéntica a la de PDALP00. La razón por la que hemos suprimido la función de eliminación de PDALP01 es la siguiente: PDALP01 sigue una estrategia de prueba enteramente mecánica, sin atender a subojetivos; nos ha interesado comparar las deducciones que ejecuta una estrategia de este tipo frente a las más expertas de PDALP00, en tiempo de ejecución y en líneas.

```
492          if((fv+i)->l1>=0){ cprintf("%d",(fv+i)->l1+1);
493           if(e==83) fprintf(stdprn,"%d",(fv+i)->l1+1);}
494         if((fv+i)->l2>=0){ cprintf("  %d",(fv+i)->l2+1);
495          if(e==83) fprintf(stdprn," %d",(fv+i)->l2+1);}
496          }
497      if(wherey()<22) continue;
498      pausa();
499      clrscr();
500      }
501  if(e==83) fputs("\n\n\n\n\r",stdprn);
502  for(j=ul-1;j>=0;j--) free((fv+j)->f);
503  pausa();
504  }
```

El fichero PDALP12.c contiene las rutinas básicas de la deducción y algunas reglas de inferencia.

```
1  /* Programa PDALP01.
2     Fichero PDALP12.c */
```

```
 3  /* Librerías */
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <string.h>
 7  #include "padlp10.h"

 8  /* Variables globales externas */
 9  extern int ul,m,us,md,fvd;
10  extern struct FORMULAS_VALIDAS *fv;
11  extern struct MARCAS *mar;
```

La rutina de demostraciones es extremadamente simple. Mientras exista algún supuesto abierto se busca la existencia de contradicciones, se aplican reglas de inferencia y se abren nuevos supuestos.

```
12  /* Búsqueda de demostraciones para la última fórmula */
13  void demostrada()
14  {
15  int i;
16  i=0;
17  do{
18     if(contradiccion()) continue;
19     if(inferencias()) continue;
20     if(supuestos()) continue;
21     putch(7);
22     cputs("\n\rFórmula no demostrable. ");
23     i=1;
24     break;
25    } while(us>=0 ||
       buscar((fv+0)->f,1,strlen((fv+0)->f))<0);
26  if(!i) {putch(7);cputs("\n\rFórmula demostrada. ");}
27  }
```

Las funciones de introducción de la disyunción, la conjunción y el bicondicional (28–66) son iguales a las correspondientes de PDALP00 con la siguiente salvedad: para unificar el tratamiento dado —debido a razones técnicas— a la variable que indica la utilización o no de una fórmula, sustitúyase siempre el valor 99 por el valor 80. Es decir, donde aparece en PDALP00

```
(fv+ul)->u=99;
```

escribir en PDALP01

```
(fv+ul)->u=80;
```

La siguiente función (67–107) es más compleja que la de PDALP00. A partir de las líneas en las que aparece una misma fórmula afirmada y negada obtiene la negación del supuesto último, tacha las líneas correspondientes y busca el supuesto que sigue.

```
 67  /* Tachar fórmulas */
 68  void tachar(int h,int k)
 69  {
 70  int i,x;
 71  (fv+us)->u=99;
 72  x=h<k?k:h;
 73  if(x<us) ul=us+1;
 74  else ul=x+1;
 75  for(i=us+1;i<ul;i++){
 76     (mar+m)->l=i;(mar+m)->ma=50;
 77     m++;if(m==md) falta_memoria(4);
 78     (fv+i)->u=99;
 79     }
 80  (mar+m)->l=ul;(mar+m)->ma=60;
 81  m++;if(m==md) falta_memoria(4);
 82  (fv+ul)->u=99;
 83  (fv+ul)->r=41;
 84  (fv+ul)->l1=h;
 85  (fv+ul)->l2=k;
 86  (fv+ul)->f=(char*)calloc(strlen((fv+h)->f)+\
 87  strlen((fv+k)->f)+2,sizeof(char));
 88  if((fv+ul)->f==NULL) falta_memoria(1);
 89  *((fv+ul)->f)='K';strcat((fv+ul)->f,(fv+h)->f);
 90  strcat((fv+ul)->f,(fv+k)->f);
 91  ul++;if(ul==fvd) falta_memoria(2);
 92  (fv+ul)->u=80;
 93  (fv+ul)->r=40;
 94  (fv+ul)->l1=us;
 95  (fv+ul)->l2=ul-1;
 96  (fv+ul)->f=(char*)calloc(strlen((fv+us)->f)+2,
     sizeof(char));
 97  if((fv+ul)->f==NULL) falta_memoria(1);
 98  *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+us)->f);
 99  ul++;if(ul==fvd) falta_memoria(2);
100  for(i=m-1;i>=0;i--){
101     if((mar+i)->ma==40 && (fv+(mar+i)->l)->u==80){
102        us=(mar+i)->l;
```

```
103         break;
104         }
105       }
106   if(i<0) us=-1;
107   }
```

La rutina que decide la aplicación de las reglas de inferencia es esencialmente la misma que en PDALP00.

```
108   /* Decidir inferencias */
109   int inferencias()
110   {
111   int x,r;
112   r=ul;
113   for(x=0;x<ul;x++){
114      if((fv+x)->u!=80) continue;
115      if(!strncmp((fv+x)->f,"NN",2)) inf_do_ne(x);
116      }
117   for(x=0;x<ul;x++){
118      if((fv+x)->u!=80) continue;
119      switch(*((fv+x)->f)){
120         case 75: inf_conj(x); break;
121         case 65: inf_dis(x); break;
122         case 67: inf_cond(x);break;
123         case 69: inf_bi(x); break;
124         case 78: switch(*((fv+x)->f+1)){
125                  case 78: inf_do_ne(x);break;
126                  case 75: inf_ne_conj(x);break;
127                  case 69: inf_ne_bi(x);break;
128                  case 65: inf_ne_dis(x);break;
129                  case 67: inf_ne_cond(x);
130                  }
131         }
132      if(ul-r) break;
133      }
134   return ul-r;
135   }
```

Las funciones de eliminación de la conjunción y la disyunción (136–193) son iguales que las correspondientes de PDALP00, excepto en el valor de la variable de utilización, que ha de modificarse en el sentido comentado anteriormente.

La siguiente función (194–209) de eliminación del condicional es más simple que la correspondiente función de PDALP00; corresponde al clásico *modus ponens.*

```
194   /* Eliminación de condicional */
195   void inf_cond(int x)
196   {
197   int i,p,k,l;
198   l=strlen((fv+x)->f);
199   p=posicion_d((fv+x)->f);
200   if(buscar((fv+x)->f,p,l)<0 &&
      (k=buscar((fv+x)->f,1,p))>=0){
201      (fv+ul)->u=80;
202      (fv+ul)->r=53;
203      (fv+ul)->l1=x;
204      (fv+ul)->l2=k;
205      (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
206      if((fv+ul)->f==NULL) falta_memoria(1);
207      for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++) ;
208     ul++;if(ul==fvd) falta_memoria(2); }
209   }
```

Las funciones de eliminación del bicondicional y doble negación (210–262) son iguales a las correspondientes funciones de PDALP00, excepto en el valor de la variable de utilización, que ha de modificarse en el sentido comentado más arriba.

La siguiente función (263–290), aunque más simple a primera vista, actúa técnicamente igual que la función correspondiente de PDALP00.

```
263   /* Negación de la conjunción */
264   void inf_ne_conj(int x)
265   {
266   int i,p,k,h,l,j;
267   char *a;
268   l=strlen((fv+x)->f);
269   a=(fv+x)->f+1;
270   p=posicion_d(a);
271   k=buscar(a,1,p);h=buscar(a,p,l-1); j=-1;
272  if(k>=0 && h>=0) {introducir_conjuncion(k,h);return;}
273   if(k>=0 && h<0 && negar(a,p,l-1)<0) j=k;
274   if(h>=0 && k<0 && negar(a,1,p)<0) j=h;
275   if(j>=0){
276      (mar+m)->l=ul;(mar+m)->ma=40;
```

```
277       m++;if(m==md) falta_memoria(4);
278       (fv+ul)->u=80;
279       (fv+ul)->r=99;
280       (fv+ul)->l1=(fv+ul)->l2=-1;
281       i=(j==k)?l-p:p;
282       (fv+ul)->f=(char*)calloc(i,sizeof(char));
283       if((fv+ul)->f==NULL) falta_memoria(1);
284       if(j==h){ for(j=0;j<p-1;j++)
          *((fv+ul)->f+j)=*(a+j+1);
285       *((fv+ul)->f+j)='\0';}
286       else for(j=p;(*((fv+ul)->f+j-p)=*(a+j));j++);
287       us=ul;
288       ul++;if(ul==fvd) falta_memoria(2);
289       }
290    }
```

El fichero PDALP13.c contiene las rutinas de supuestos y funciones diversas de búsqueda.

```
1    /* Programa PDALP01.
2    Fichero PDALP13.c */

3    /* Librerías */
4    #include <stdio.h>
5    #include <stdlib.h>
6    #include <string.h>
7    #include "pdalp10.h"

8    /* Variables globales  externas */
9    extern int ul,us,m,md,fvd;
10   extern struct FORMULAS_VALIDAS *fv;
11   extern struct MARCAS *mar;
```

Al carecer de pila de demostraciones, PDALP01 no abre supuestos para subojetivos en función del tipo de la fórmula a demostrar. La siguiente función (12–57) corresponde a la función de apertura de nuevos supuestos de PDALP00.

```
12   /* Apertura de nuevos supuestos */
13   int supuestos()
14   {
15   int h,r,p,i,l;
```

```
16   char *a,*b;
17   r=0;
18   for(h=0;h<ul;h++){
19      if((fv+h)->u!=80) continue;
20      l=strlen((fv+h)->f);
21      if(*((fv+h)->f)=='C'){
22         p=posicion_d((fv+h)->f);
23         a=(char*)calloc(p+1,sizeof(char));
24         if(!a) falta_memoria(1);
25         *a='N';
26         for(i=1;i<p;i++) *(a+i)=*((fv+h)->f+i);
           *(a+i)='\0';
27         if((r=subformulas(a))) break;
28         else {free(a); continue;}
29         }
30      if(*((fv+h)->f)=='A'){
31         p=posicion_d((fv+h)->f);
32         a=(char*)calloc(p,sizeof(char));
33         if(!a) falta_memoria(1);
34         for(i=0;i<p-1;i++) *(a+i)=*((fv+h)->f+i+1);
           *(a+i)='\0';
35         if((r=subformulas(a))) break;
36         else {free(a); continue;}
37         }
38      if(!strncmp((fv+h)->f,"NK",2)){
39         b=(fv+h)->f+1;
40         p=posicion_d(b);
41         a=(char*)calloc(p+1,sizeof(char));
42         if(!a) falta_memoria(1);
43         *a='N';
44         for(i=1;i<p;i++) *(a+i)=*(b+i);*(a+i)='\0';
45         if((r=subformulas(a))) break;
46         else {free(a); continue;}
47         }
48      if(!strncmp((fv+h)->f,"NE",2)){
49         a=(char*)calloc(l+1,sizeof(char));
50         if(!a) falta_memoria(1);
51         strcpy(a,(fv+h)->f);*(a+1)='C';
52         if((r=subformulas(a))) break;
53         else {free(a); continue;}
54         }
```

```
55      }
56   return r;
57   }
```

La siguiente función (58–76) es básicamente igual a la de PDALP00.

```
58   /* Suponer las fórmulas elementales */
59   int subformulas(char *a)
60   {
61   int l,r,x;
62   l=strlen(a);
63   r=0;
64   if(buscar(a,0,l)<0 && negar(a,0,l)<0){
65      (mar+m)->l=ul;(mar+m)->ma=40;
66      m++;if(m==md) falta_memoria(4);
67      (fv+ul)->u=80;
68      (fv+ul)->r=99;
69      (fv+ul)->l1=(fv+ul)->l2=-1;
70      (fv+ul)->f=a;
71      us=ul;
72      ul++;if(ul==fvd) falta_memoria(2);
73      r=1;
74      }
75   return r;
76   }
```

La función que establece las subfórmulas de una fórmula (77–91) es idéntica a la de PDALP00. La función de búsqueda de contradicciones (92–107) es más simple ya que se limita a enviar las líneas en las que está la contradicción a una función que se encarga de clausurar el último supuesto.

```
92   /* Buscando contradicciones */
93   int contradiccion()
94   {
95   int i,r,n;
96   r=0;
97   for(i=0;i<ul;i++){
98      if((fv+i)->u!=80) continue;
99      if(*((fv+i)->f)!='N') continue;
100     if((n=negar((fv+i)->f,0,strlen((fv+i)->f)))>=0){
101        tachar(n,i);
```

```
102          r=1;
103          }
104       if(r) break;
105       }
106    return r;
107    }
```

Las funciones de búsqueda de una fórmula utilizable, búsqueda de una negación y el establecimiento de que una fórmula sea la negación de otra (108–165), son iguales a las correspondientes funciones de PDALP00 excepto en el valor de la variable de utilización que en PDALP00 se iguala a 35 y en PDALP01 se diferencia de 80. Es decir, donde aparece en PDALP00

`(fv+i)->u==35,`

en PDALP01 debe convertirse en

`(fv+i)->u!=80.`

La siguiente función introduce (166–177) una doble negación.

```
166    /* Introducir negación */
167    void introducir_negacion(int i)
168    {
169    (fv+ul)->u=80;
170    (fv+ul)->r=61;
171    (fv+ul)->l1=i;
172    (fv+ul)->l2=-1;
173    if(((fv+ul)->f=(char*)calloc(strlen((fv+i)->f)+3,\
174    sizeof(char)))==NULL) falta_memoria(1);
175    strncat((fv+ul)->f,"NN",2);strcat((fv+ul)->f,(fv+i)->f);
176    ul++;if(ul==fvd) falta_memoria(2);
177    }
```

El fichero PDALP14.c contiene rutinas que se comportan como reglas derivadas de inferencia en algunos caso en que no puede aplicarse una regla directa a la fórmula.

```
1 /* Programa PDALP01.
2    Fichero PDALP14.c */

3 /* Librerías */
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
```

```
 7 #include "pdalp10.h"

 8 /* Variables globales externas */
 9 extern int ul,us,m,md,fvd;
10 extern struct FORMULAS_VALIDAS *fv;
11 extern struct MARCAS *mar;
```

Las funciones siguientes de negación del bicondicional, la disyunción y el condicional (12–206) se diferencian básicamente de las funciones correspondientes de PDALP00 en que efectúan llamadas recursivas a sí mismas.

```
12 /* Negación de bicondicional */
13 void inf_ne_bi(int x)
14 {
15 int i,p,l,k,h,r,s;
16 char *a,*b,*c,*d,*e;
17 l=strlen((fv+x)->f);
18 if((a=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
19 for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
20 p=posicion_d(a);
21 if((b=(char*)calloc(p,sizeof(char)))==NULL)
   falta_memoria(1);
22 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
23 falta_memoria(1);
24 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
25 for(i=p;(*(c+i-p)=*(a+i));i++) ;
26 if((d=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
27 if((e=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
28 *d=*e='C';strcat(d,b);strcat(d,c);
   strcat(e,c);strcat(e,b);
29 free(b);free(c);
30 k=buscar(d,0,l-1);h=buscar(e,0,l-1);
31 if(k>=0 && h>=0){
32    introducir_bicondicional(k,h);
33    free(a);free(d);free(e);
34    return;
35    }
36 if(h<0 && k<0){
```

```
37     free(a);free(d);free(e);
38     return ;
39     }
40 r=negar(d,0,l-1);
41 s=negar(e,0,l-1);
42 if( (k>=0 && s>=0) || (h>=0 && r>=0))
   {free(a);free(d);
43     free(e);return;}
44 (mar+m)->l=ul;(mar+m)->ma=40;
45 m++;if(m==md) falta_memoria(4);
46 (fv+ul)->u=80;
47 (fv+ul)->r=99;
48 (fv+ul)->l1=(fv+ul)->l2=-1;
49 (fv+ul)->f=(char*)calloc(l,sizeof(char));
50 if((fv+ul)->f==NULL) falta_memoria(1);
51 if(k>=0) strcpy((fv+ul)->f,e);
52 else strcpy((fv+ul)->f,d);
53 free(a);free(d);free(e);
54 us=ul;
55 ul++;if(ul==fvd) falta_memoria(2);
56 inf_ne_bi(x);
57 }

58 /* Negación de la disyunción */
59 void inf_ne_dis(int x)
60 {
61 int i,h,k,l,p;
62 char *a,*b,*c;
63 l=strlen((fv+x)->f);
64 if((a=(char*)calloc(l,sizeof(char)))==NULL)
   falta_memoria(1);
65 for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
66 p=posicion_d(a);
67 if((b=(char*)calloc(p,sizeof(char)))==NULL)
   falta_memoria(1);
68 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
69 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
70 falta_memoria(1);
71 for(i=p;(*(c+i-p)=*(a+i));i++) ;
72 k=h=-1;
73 if((h=buscar(a,1,p))>=0 || (k=buscar(a,p,l-1))>=0){
```

```
 74     h=h<k?k:h;
 75     introducir_disyuncion(a,h);
 76     free(a);free(b);free(c);
 77     return;
 78     }
 79 if((k=negar(a,1,p))>=0 && (h=negar(a,p,l-1))>=0){
 80     free(a);free(b);free(c);
 81     return;
 82     }
 83 (mar+m)->l=ul;(mar+m)->ma=40;
 84 m++;if(m==md) falta_memoria(4);
 85 (fv+ul)->u=80;  86 (fv+ul)->r=99;
 87 (fv+ul)->l1=(fv+ul)->l2=-1;
 88 (fv+ul)->f=(char*)calloc(l,sizeof(char));
 89 if((fv+ul)->f==NULL) falta_memoria(1);
 90 if(k<0) strcpy((fv+ul)->f,b);
 91 else strcpy((fv+ul)->f,c);
 92 free(a);free(b);free(c);
 93 us=ul;
 94 ul++;if(ul==fvd) falta_memoria(2);
 95 inf_ne_dis(x);
 96 }

 97 /* Negación de condicional */
 98 void inf_ne_cond(int x)
 99 {
100 int i,l,p,k,h;
101 char *a,*b,*c;
102 l=strlen((fv+x)->f);
103 if((a=(char*)calloc(l,sizeof(char)))==NULL)
    falta_memoria(1);
104 for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
105 p=posicion_d(a);
106 if((b=(char*)calloc(p,sizeof(char)))==NULL)
    falta_memoria(1);
107 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
108 falta_memoria(1);
109 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
110 for(i=p;(*(c+i-p)=*(a+i));i++) ;
111 k=h=-1;
112 if((k=buscar(a,p,l-1))>=0 || (h=negar(a,1,p))>=0){
```

```
113     (mar+m)->l=ul;(mar+m)->ma=40;
114     m++;if(m==md) falta_memoria(4);
115     (fv+ul)->u=99;
116     (fv+ul)->r=99;
117     (fv+ul)->l1=(fv+ul)->l2=-1;
118     (fv+ul)->f=b;
119     ul++;if(ul==fvd) falta_memoria(2);
120     (mar+m)->l=ul;(mar+m)->ma=40;
121     m++;if(m==md) falta_memoria(4);
122     (mar+m)->l=ul;(mar+m)->ma=50;
123     m++;if(m==md) falta_memoria(4);
124     (fv+ul)->u=99;
125     (fv+ul)->r=99;
126     (fv+ul)->l1=(fv+ul)->l2=-1;
127     (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
128     if((fv+ul)->f==NULL) falta_memoria(1);
129     *((fv+ul)->f)='N';strcat((fv+ul)->f,c);
130     ul++;if(ul==fvd) falta_memoria(2);
131     (mar+m)->l=ul;(mar+m)->ma=60;
132     m++;if(m==md) falta_memoria(4);
133     (mar+m)->l=ul;(mar+m)->ma=50;
134     m++;if(m==md) falta_memoria(4);
135     (fv+ul)->u=99;
136     (fv+ul)->r=41;
137     if(k>=0){
138        (fv+ul)->l1=k;
139        (fv+ul)->l2=ul-1;
140        (fv+ul)->f=(char*)calloc(2*(l-p+2),sizeof(char));
141        if((fv+ul)->f==NULL) falta_memoria(1);
142        *((fv+ul)->f)='K';strcat((fv+ul)->f,c);
143        strcat((fv+ul)->f,(fv+ul-1)->f);
144        }
145     else{
146        (fv+ul)->l1=h;
147        (fv+ul)->l2=ul-2;
148        (fv+ul)->f=(char*)calloc(2*(p+2),sizeof(char));
149        if((fv+ul)->f==NULL) falta_memoria(1);
150        *((fv+ul)->f)='K';strcat((fv+ul)->f,b);
151        strcat((fv+ul)->f,(fv+h)->f);
152        }
153     ul++;if(ul==fvd) falta_memoria(2);
```

```
154     (mar+m)->l=ul;(mar+m)->ma=50;
155     m++;if(m==md) falta_memoria(4);
156     (fv+ul)->u=99;
157     (fv+ul)->r=40;
158     (fv+ul)->l1=ul-2;
159     (fv+ul)->l2=ul-1;
160     (fv+ul)->f=(char*)calloc(2*(l-p+1),sizeof(char));
161     if((fv+ul)->f==NULL) falta_memoria(1);
162     *((fv+ul)->f)='N';strcat((fv+ul)->f,(fv+ul-2)->f);
163     ul++;if(ul==fvd) falta_memoria(2);
164     (mar+m)->l=ul;(mar+m)->ma=60;
165     m++;if(m==md) falta_memoria(4);
166     (fv+ul)->u=99;
167     (fv+ul)->r=50;
168     (fv+ul)->l1=ul-1;
169     (fv+ul)->l2=-1;
170     (fv+ul)->f=(char*)calloc(l-p,sizeof(char));
171     if((fv+ul)->f==NULL) falta_memoria(1);
172     strcpy((fv+ul)->f,c);
173     ul++;if(ul==fvd) falta_memoria(2);
174     (fv+ul)->u=80;
175     (fv+ul)->r=43;
176     (fv+ul)->l1=ul-5;
177     (fv+ul)->l2=ul-1;
178     (fv+ul)->f=(char*)calloc(l,sizeof(char));
179     if((fv+ul)->f==NULL) falta_memoria(1);
180     strcpy((fv+ul)->f,a);
181     ul++;if(ul==fvd) falta_memoria(2);
182     return;
183     }
184 k=h=0;
185 if((k=negar(a,p,l-1))<0 || (h=buscar(a,1,p))<0){
186     (mar+m)->l=ul;(mar+m)->ma=40;
187     m++;if(m==md) falta_memoria(4);
188     (fv+ul)->u=80;
189     (fv+ul)->r=99;
190     (fv+ul)->l1=(fv+ul)->l2=-1;
191     if(k<0){
192        (fv+ul)->f=(char*)calloc(l-p,sizeof(char));
193        if((fv+ul)->f==NULL) falta_memoria(1);
194        strcpy((fv+ul)->f,c);
```

```
195         }
196      else{
197         (fv+ul)->f=(char*)calloc(p+1,sizeof(char));
198         if((fv+ul)->f==NULL) falta_memoria(1);
199         *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
200         }
201      free(a);free(b);free(c);
202      us=ul;
203      ul++;if(ul==fvd) falta_memoria(2);
204      inf_ne_cond(x);
205      }
206   }
```

III. 3. Programa PDALP02

El programa PDALP02 efectúa deducciones en el cálculo M. Las reglas de construcción de las líneas deductivas en este cálculo permiten que se simplifiquen las estrategias deductivas. La estrategia que aquí presentamos carece de subojetivos para las deducciones que atiendan a todas las formas de las fórmulas a demostrar. El programa PDALP02 consta de los ficheros PDALP20.h, PDALP21.c y PDALP22.c.

```
1  /* Programa PDALP02.
2     Fichero PDALP20.h */

3  /* Estructuras */
4  struct FORMULAS_VALIDAS{
5     char *f;      /*fórmula              */
6     char u;       /*utilizable o no      */
7     char r;       /*reglas               */
8     int l1;       /*líneas de inferencia*/
9     int l2;       /*líneas de inferencia*/
10    };

11 struct MARCAS{
12    char ma;           /*marca            */
13    int l;             /*número de línea*/
14    };

15 /* Definiciones */
16 #define TFV sizeof(struct FORMULAS_VALIDAS)
17 #define TM sizeof(struct MARCAS)

18 /* Funciones prototipo */
19 /* Fichero pdalp21.c */
20 void calculos_dn(void);
21 int simbolos(char*);
22 int es_formula(char*);
23 int simbolo_do(char*);
24 void pausa(void);
25 void falta_memoria(int);
26 void salida_formulas(void);

27 /* Fichero pdalp22.c */
28 void demostrada(void);
```

```
29  void introducir_disyuncion(char*,int);
30  void introducir_conjuncion(int,int);
31  void introducir_bicondicional(int,int);
32  void tachar(void);
33  void escribir_supuestos(char*);
34  int inferencias(void);
35  void inf_conj(int);
36  void inf_dis(int);
37  void inf_cond(int);
38  void inf_bi(int);
39  void inf_do_ne(int);
40  void inf_ne_conj(int);
41  int supuestos(void);
42  int subformulas(char*);
43  int interrogada(char*,int,int);
44  int posicion_d(char*);
45  int buscar(char*,int,int);
46  int negacion(char*,int);
47  int contradiccion(void);
48  void repeticion(int);
49  void introducir_negacion(int);
50  int negar(char*,int,int);
51  void inf_ne_bi(int);
52  void inf_ne_dis(int);
53  void inf_ne_cond(int);
54 int neg_int(char*);
```

El cálculo M permite aproximar las líneas interrogadas a los subojetivos de la demostración; esto posibilita eliminar toda la información necesaria para mantener una pila de fórmulas por demostrar. De todos modos, la deducción es siempre más elegante si se utiliza una pila independiente de fórmulas a demostrar.

```
55  /* Programa PDALP02.
56     Fichero PDALP21.c */

57  /* Librerías */
58  #include <stdio.h>
59  #include <stdlib.h>
60  #include <string.h>
61  #include <ctype.h>
62  #include <conio.h>
```

```
63  #include <alloc.h>
64  #include "pdalp20.h"
65  #include <time.h>

66  /* Variables globales */
67  /* Contadores de fórmulas escritas y marcas: ul,m;
68   * último supuesto: us;
69   * marcas y fórmulas válidas
70   * disponibles por asignación dinámica en curso: md, fvd.
71   */
72  int ul,m,us,md,fvd;
73  struct FORMULAS_VALIDAS *fv;
    /*puntero a fórmulas escritas*/
74  struct MARCAS *mar;   /*puntero a marcas     */
```

La función inicial y, si se quiere, una función de instrucciones generales son las ya conocidas.

La introducción de conclusión y premisas (84–164) es igual a la de PDALP01, excepto en el almacenamiento de las nuevas fórmulas que queda así:

```
138         if(j==0) escribir_supuestos(b);
139         else{
140     (fv+ul)->f=b;
141     (fv+ul)->u=80;
142     (fv+ul)->r=98;
143     (fv+ul)->l1=(fv+ul)->l2=-1;
144     ul++;
145     }
```

La escritura de las premisas, si las hay, sigue pautas ya conocidas de los programas anteriores (139–145). El tratamiento de la conclusión es gestionado por una función especial (138).

Las funciones de verificación de símbolos, fórmulas, establecimiento del símbolo dominante de una fórmula, pausa e interrupción por falta de memoria son las mismas que las de PDALP01 (165–349).

```
350  /* Salida de demostraciones */
351  void salida_formulas()
352  {
353  int i,p,j,k,w;
354  char *d,e;
```

```
355  cprintf("\n\r¿Desea imprimir el resultado? (S/N): ");
356  fflush(stdin);
357  e=toupper(getchar());
358  for(i=0;i<ul;i++){
359     cprintf("\n\r%3d. ",i+1);
360     if(e==83) fprintf(stdprn,"\n\r%3d. ",i+1);
361     for(p=m-1;p>=0;p--){
362        if((mar+p)->l!=i) continue;
363        switch((mar+p)->ma){
364    case 40: cputs("?");
365           if(e==83) fputs("?",stdprn);break;
366    case 50: cputs("|");
367           if(e==83) fputs("|",stdprn);break;
368    case 60: cputs("#");
369           if(e==83) fputs("#",stdprn);
370    }
371        }
***
***
***
445     if((fv+i)->r!=99){
446        switch((fv+i)->r){
447    case 98: cprintf(" Premisa");
448           if(e==83) fputs(" Premisa",stdprn);break;
449    case 41: cputs(" I/\\  ");
450           if(e==83) fputs(" I/\\ ",stdprn);break;
451    case 42: cputs(" I\\/  ");
452           if(e==83) fputs(" I\\/ ",stdprn);break;
453    case 44: cputs(" I<->  ");
454           if(e==83) fputs(" I<-> ",stdprn);break;
455    case 50: cputs(" DN   ");
456           if(e==83) fputs(" DN ",stdprn);break;
457    case 51: cputs(" E/\\  ");
458           if(e==83) fputs(" E/\\ ",stdprn);break;
459    case 52: cputs(" E\\/  ");
460           if(e==83) fputs(" E\\/ ",stdprn);break;
461    case 53: cputs(" MP  ");
462           if(e==83) fputs(" MP ",stdprn);break;
463    case 54: cputs(" E<->  ");
464           if(e==83) fputs(" E<-> ",stdprn);break;
465    case 60: cputs(" MT  ");
```

```
466             if(e==83) fputs(" MT ",stdprn);break;
467       case 61: cputs(" R  ");
468             if(e==83) fputs(" R ",stdprn);
469             }
470           if((fv+i)->l1>=0){ cprintf("%d",(fv+i)->l1+1);
471       if(e==83) fprintf(stdprn,"%d",(fv+i)->l1+1);}
472       if((fv+i)->l2>=0){ cprintf("  %d",(fv+i)->l2+1);
473       if(e==83) fprintf(stdprn," %d",(fv+i)->l2+1);}
474           }
475        if(wherey()<22) continue;
476        pausa();
477        clrscr();
478        }
479     if(e==83) fputs("\n\r",stdprn);
480     for(j=ul-1;j>=0;j--) free((fv+j)->f);
481     pausa();
482   }
```

En la salida de las fórmulas, empleamos por conveniencia el símbolo # en vez de una interrogación tachada. Las líneas 372–444 corresponden a la traduccción de las fórmulas de la notación polaca. Téngase en cuenta lo que ya hemos dicho sobre la posibilidad de sustituirlas por otra función.

El fichero PDALP22.c contiene el resto de las funciones del programa.

```
1   /* Programa PDALP02.
2      Fichero PDALP22.c */

3   /* Librerías */
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <conio.h>
7   #include <ctype.h>
8   #include <string.h>
9   #include "pdalp20.h"

10  /* Variables globales  externas */
11  extern int ul,m,us,md,fvd;
12  extern struct FORMULAS_VALIDAS *fv;
13  extern struct MARCAS *mar;
```

La rutina principal de las demostraciones es bastante simple. Tacha el último interrogante (22–35); busca alguna contradicción; aplica reglas de inferencia; gestiona la posibilidad de interrogar nuevas fórmulas.

```
14  /* Búsqueda de demostraciones para la última fórmula */
15  void demostrada()
16  {
17  int i,l;
18  i=0;
19  do{
20     int k;
21     l=strlen((fv+us)->f);
22     if((k=buscar((fv+us)->f,0,l))>0) {
23        if(k<us) repeticion(k);
24        tachar();
25        continue;
26        }
27     if(*((fv+us)->f)=='C'){
28        int p;
29        p=posicion_d((fv+us)->f);
30        if((k=buscar((fv+us)->f,p,l))>0){
31    if(k<us) repeticion(k);
32    tachar();
33    continue;
34    }
35        }
36     if(contradiccion()) continue;
37     if(inferencias()) continue;
38     if(supuestos()) continue;
39     putch(7);
40     cputs("\n\rFórmula no demostrable. ");
41     i=1;
42     break;
43     } while(us>=0);
44  if(!i) {putch(7);cputs("\n\rFórmula demostrada. ");}
45  }
```

Las funciones de introducción de disyunción, conjunción y bicondicional (46–84) son las mismas que las del programa PDALP01.

La siguiente función tacha la última fórmula interrogada y establece, de nuevo, el último supuesto.

```
/* Tachar fórmulas */
void tachar(void)
{
int i,r;
r=us;
(fv+us)->u=80;
for(i=m-1;i>=0;i--){
   if((mar+i)->l==us){ (mar+i)->ma=60;continue;}
   if((mar+i)->ma==40) {
      us=(mar+i)->l;
      break;
      }
   }
if(i<0) us=-1;
for(i=r+1;i<ul;i++){
   (mar+m)->l=i;(mar+m)->ma=50;
   m++;if(m==md) falta_memoria(4);
   (fv+i)->u=99;
   }
}
```

La siguiente función escribe las líneas pertinentes para cada objetivo de deducción.

```
/* Escribir supuestos */
void escribir_supuestos(char *a)
{
int l;
l=strlen(a);
(mar+m)->l=ul;(mar+m)->ma=40;
m++;if(m==md) falta_memoria(4);
us=ul;
(fv+ul)->u=95;
(fv+ul)->r=99;
(fv+ul)->l1=(fv+ul)->l2=-1;
(fv+ul)->f=a;
ul++;if(ul==fvd)falta_memoria(2);
if(*a=='C'){
   int i,p;
   char *b;
   p=posicion_d(a);
   (fv+ul)->u=80;
```

```
123      (fv+ul)->r=99;
124      (fv+ul)->l1=(fv+ul)->l2=-1;
125    if(((fv+ul)->f=(char*)calloc(p,sizeof(char)))==NULL)
126      falta_memoria(1);
127      for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*(a+i+1);
128      *((fv+ul)->f+i)='\0';
129      ul++;if(ul==fvd) falta_memoria(2);
130      if((b=(char*)calloc(l-p+1,sizeof(char)))==NULL)
131      falta_memoria(1);
132      for(i=p;(*(b+i-p)=*(a+i));i++);
133      escribir_supuestos(b);
134      }
135   else{
136      (fv+ul)->u=80;
137      (fv+ul)->r=99;
138      (fv+ul)->l1=(fv+ul)->l2=-1;
139  if(((fv+ul)->f=(char*)calloc(l+2,sizeof(char)))==NULL)
140      falta_memoria(1);
141      *((fv+ul)->f)='N';
142      strcat((fv+ul)->f,a);
143      ul++;if(ul==fvd) falta_memoria(2);
144      if(*a=='N'){
145         int j;
146         (fv+ul)->u=80;
147         (fv+ul)->r=50;
148         (fv+ul)->l1=ul-1;
149         (fv+ul)->l2=-1;
150    if(((fv+ul)->f=(char*)calloc(l,sizeof(char)))==NULL)
151         falta_memoria(1);
152         for(j=0;(*((fv+ul)->f+j)=*(a+j+1));j++);
153         ul++;if(ul==fvd) falta_memoria(2);
154         }
155      }
156   }
```

La siguiente función decide la aplicación de inferencias siguiendo el criterio de dar prioridad a la forma de las fórmulas de cada línea en vez de concedérsela a las reglas de inferencia mismas.

```
157   /* Decidir inferencias */
158   int inferencias()
```

```
159  {
160  int x,r;
161  r=ul;
162  for(x=0;x<ul;x++){
163     if((fv+x)->u!=80) continue;
164     switch(*((fv+x)->f)){
165        case 75: inf_conj(x); break;
166        case 65: inf_dis(x); break;
167        case 67: inf_cond(x);break;
168        case 69: inf_bi(x); break;
169        case 78: switch(*((fv+x)->f+1)){
170           case 78: inf_do_ne(x);break;
171           case 75: inf_ne_conj(x);break;
172           case 69: inf_ne_bi(x);break;
173           case 65: inf_ne_dis(x);break;
174           case 67: inf_ne_cond(x);
175           }
176        }
177     if(ul-r) break;
178     }
179  return ul-r;
180  }
```

Las funciones de eliminación de la conjunción y de la disyunción (181–239) son las mismas que las correspondientes de PDALP01.

La siguiente función (240–270) aplica las reglas de *modus ponens* y de *modus tollens.*

```
240  /* Eliminación de condicional */
241  void inf_cond(int x)
242  {
243  int i,p,k,l;
244  l=strlen((fv+x)->f);
245  p=posicion_d((fv+x)->f);
246  if(buscar((fv+x)->f,p,l)<0 &&
     (k=buscar((fv+x)->f,1,p))>=0){
247     (fv+ul)->u=80;
248     (fv+ul)->r=53;
249     (fv+ul)->l1=x;
250     (fv+ul)->l2=k;
251     (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
252     if((fv+ul)->f==NULL) falta_memoria(1);
```

```
253     for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++) ;
254     ul++;if(ul==fvd) falta_memoria(2);
255     }
256  else{
257     if((negar((fv+x)->f,1,p))<0 &&
        (negar((fv+x)->f,p,l))>=0){
258        (fv+ul)->u=80;
259        (fv+ul)->r=60;
260        (fv+ul)->l1=x;
261        (fv+ul)->l2=k;
262        (fv+ul)->f=(char*)calloc(p+1,sizeof(char));
263        if((fv+ul)->f==NULL) falta_memoria(1);
264        *((fv+ul)->f)='N';
265       for(i=1;i<p;i++) *((fv+ul)->f+i)=*((fv+x)->f+i);
266        *((fv+ul)->f+i)='\0';
267        ul++;if(ul==fvd) falta_memoria(2);
268        }
269     }
270  }
```

Las funciones de eliminación del bicondicional y de doble negación (271–324) son las mismas que las de PDALP01.

La siguiente función (325–362) gestiona el tratamiento de las conjunciones negadas. Aquí utilizamos la función de escritura de supuestos. Si se quiere disminuir el tiempo de ejecución, pueden escribirse directamente todas las líneas de deducción que el programa habrá de escribir ejecutando varias funciones varias veces. Para deducciones normales no parece, sin embargo, que la ganancia de tiempo sea significativa.

```
325  /* Negación de la conjunción */
326  void inf_ne_conj(int x)
327  {
328  int i,p,k,h,l;
329  char *a,*b,*c;
330  l=strlen((fv+x)->f);
331  a=(fv+x)->f+1;
332  p=posicion_d(a);
333  k=buscar(a,1,p);h=buscar(a,p,l);
334  if(k>=0 && h>=0){
335     introducir_conjuncion(k,h);
336     return;
337     }
```

```
338  if(h<0 && k<0) return;
339  b=(char*)calloc(p,sizeof(char));
340  c=(char*)calloc(l-p,sizeof(char));
341  if(b==NULL || c==NULL) falta_memoria(1);
342  for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
343  for(i=p;(*(c+i-p)=*(a+i));i++);
344  if( (k>=0 && ( negar(a,p,l-1)>=0 || neg_int(c) ) )\
345  ||  (h>=0 && ( negar(a,1,p)>=0 || neg_int(b) ))){
346     free(b);free(c);return;}
347  if(k>=0){
348     free(b);
349     b=(char*)calloc(l-p+1,sizeof(char));
350     if(!b) falta_memoria(1);
351     *b='N';strcat(b,c);
352     escribir_supuestos(b);
353     free(c);
354     return;
355     }
356  free(c);
357  c=(char*)calloc(p+1,sizeof(char));
358  if(!c) falta_memoria(1);
359  *c='N';strcat(c,b);
360  escribir_supuestos(c);
361  free(b);
362  }
```

La siguiente función (363–408) abre nuevos supuestos de un modo que es lógicamente igual al de los programas anteriores. Sin embargo, dada la diferencia en la construcción de las deduccciones, se observará que las fórmulas que aparecían negadas en los programas anteriores aquí no lo están y viceversa.

```
363  /* Apertura de nuevos supuestos */
364  int supuestos()
365  {
366  int h,r,p,i,l;
367  char *a,*b;
368  r=0;
369  for(h=0;h<ul;h++){
370     if((fv+h)->u!=80) continue;
371     l=strlen((fv+h)->f);
372     if(*((fv+h)->f)=='C'){
```

```
373         p=posicion_d((fv+h)->f);
374         a=(char*)calloc(p,sizeof(char));
375         if(!a) falta_memoria(1);
376         for(i=0;i<p-1;i++) *(a+i)=*((fv+h)->f+i+1);
            *(a+i)='\0';
377         if((r=subformulas(a))) break;
378         else {free(a); continue;}
379         }
380      if(*((fv+h)->f)=='A'){
381         p=posicion_d((fv+h)->f);
382         a=(char*)calloc(p+1,sizeof(char));
383         if(!a) falta_memoria(1);
384         *a='N';
385         for(i=1;i<p;i++) *(a+i)=*((fv+h)->f+i);
            *(a+i)='\0';
386         if((r=subformulas(a))) break;
387         else {free(a); continue;}
388         }
389      if(!strncmp((fv+h)->f,"NK",2)){
390         b=(fv+h)->f+1;
391         p=posicion_d(b);
392         a=(char*)calloc(p,sizeof(char));
393         if(!a) falta_memoria(1);
394         for(i=0;i<p-1;i++) *(a+i)=*(b+i+1);*(a+i)='\0';
395         if((r=subformulas(a))) break;
396         else {free(a); continue;}
397         }
398      if(!strncmp((fv+h)->f,"NE",2)){
399         a=(char*)calloc(1,sizeof(char));
400         if(!a) falta_memoria(1);
401         *a='C';
402         for(i=1;(*(a+i)=*((fv+h)->f+i+1));i++);
403         if((r=subformulas(a))) break;
404         else {free(a); continue;}
405         }
406      }
407   return r;
408   }
```

La siguiente función (409–418) escribe como supuestos subfórmulas de fórmulas utilizables.

```
409  /* Suponer las fórmulas elementales */
410  int subformulas(char *a)
411  {
412  int l,x;
413  l=strlen(a);
414  if(buscar(a,0,l)>=0 || negar(a,0,l)>=0 ||
     interrogada(a,0,l))
415     return 0;
416  escribir_supuestos(a);
417  return 1;
418  }
```

La siguiente función (419–432) busca si una fórmula está interrogada.

```
419  /* Buscar una fórmula interrogada */
420  int interrogada(char *a,int h, int k)
421  {
422  int x,j,l;
423  l=0;
424  for(x=0;x<ul;x++){
425     if((fv+x)->u!=95) continue;
426     for(j=h;j<k;j++)
       if(*(a+j)!=*((fv+x)->f+j-h))break;
427     if(j<k) continue;
428     l=1;
429     break;
430     }
431  return l;
432  }
```

La función que establece las subfórmulas de una fórmula (433–447) es la misma que la de los programas anteriores.

La siguiente función (448–459) corresponde a la regla de repetición.

```
448  /* Repetición de fórmulas */
449  void repeticion(int h)
450  {
451  (fv+ul)->u=80;
452  (fv+ul)->r=61;
453  (fv+ul)->l1=h;
454  (fv+ul)->l2=-1;
```

```
455  (fv+ul)->f=(char*)calloc(strlen((fv+h)->f)+1,
     sizeof(char));
456  if((fv+ul)->f==NULL)falta_memoria(1);
457  strcpy((fv+ul)->f,(fv+h)->f);
458  ul++;if(ul==fvd) falta_memoria(2);
459  }
```

La siguiente función (460–477) efectúa una búsqueda de contradicciones.

```
460  /* Buscando contradicciones */
461  int contradiccion()
462  {
463  int i,r,n;
464  r=0;
465  for(i=0;i<ul;i++){
466     if((fv+i)->u!=80) continue;
467     if(*((fv+i)->f)!='N') continue;
468     if((n=negar((fv+i)->f,0,strlen((fv+i)->f)))>=0){
469        if(i<us) repeticion(i);
470        if(n<us) repeticion(n);
471        tachar();
472        r=1;
473        }
474     if(r) break;
475     }
476  return r;
477  }
```

Las funciones de búsqueda de una fórmula utilizable, de una negación, el establecimiento de si una fórmula es la negación de otra y la introducción de la negación (478–547) son iguales a las funciones correspondientes de PDALP01.

Las siguientes funciones (548–655) efectúan el tratamiento de un bicondicional, una disyunción y un condicional negados. Lógicamente no hay diferencias esenciales respecto a las funciones correspondientes de los programas anteriores; las diferencias obedecen sólo a las peculiaridades de la construcción de las deducciones en el cálculo M.

```
548  /* Negación de bicondicional */
549  void inf_ne_bi(int x)
550  {
551  int i,p,l,k,h,r,s;
552  char *a,*b,*c,*d,*e;
```

```
553  l=strlen((fv+x)->f);
554  if((a=(char*)calloc(l,sizeof(char)))==NULL)
     falta_memoria(1);
555  for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
556  p=posicion_d(a);
557  if((b=(char*)calloc(p,sizeof(char)))==NULL)
     falta_memoria(1);
558  if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
559  falta_memoria(1);
560  for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
561  for(i=p;(*(c+i-p)=*(a+i));i++) ;
562  if((d=(char*)calloc(l,sizeof(char)))==NULL)
     falta_memoria(1);
563  if((e=(char*)calloc(l,sizeof(char)))==NULL)
     falta_memoria(1);
564  *d=*e='C';strcat(d,b);strcat(d,c);strcat(e,c);
     strcat(e,b);
565  free(b);free(c);
566  k=buscar(d,0,l-1);h=buscar(e,0,l-1);
567  if(k>=0 && h>=0){
568     introducir_bicondicional(k,h);
569     free(a);free(d);free(e);
570     return;
571     }
572  if(h<0 && k<0){
573     free(a);free(d);free(e);
574     return ;
575     }
576  r=negar(d,0,l-1);
577  s=negar(e,0,l-1);
578  if( (k>=0 && ( s>=0 || neg_int(e) ) ) \
579  ||  (h>=0 && ( r>=0 || neg_int(d) ) ) ) {
580     free(a);free(d);free(e);return;}
581  free(a);
582  a=(char*)calloc(l+1,sizeof(char));
583  if(!a) falta_memoria(1);
584  *a='N';
585  if(k>=0) strcat(a,e);
586  else strcat(a,d);
587  free(d);free(e);
588  escribir_supuestos(a);
589  }
```

```
590  /* Negación de la disyunción */
591  void inf_ne_dis(int x)
592  {
593  int i,h,k,l,p;
594  char *a,*b,*c;
595  l=strlen((fv+x)->f);
596  if((a=(char*)calloc(l,sizeof(char)))==NULL)
     falta_memoria(1);
597  for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
598  p=posicion_d(a);
599  if((b=(char*)calloc(p,sizeof(char)))==NULL)
     falta_memoria(1);
600  for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
601  if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
602     falta_memoria(1);
603  for(i=p;(*(c+i-p)=*(a+i));i++) ;
604  k=h=-1;
605  if((h=buscar(a,1,p))>=0 || (k=buscar(a,p,l-1))>=0){
606     h=h<k?k:h;
607     introducir_disyuncion(a,h);
608     free(a);free(b);free(c);
609     return;
610     }
611  if(negar(a,1,p)<0 && !neg_int(b) && !neg_int(c)){
612     free(b);free(c);
613     *a='N';
614     *(a+p)='\0';
615     escribir_supuestos(a);
616     return;
617     }
618  if(negar(a,p,l-1)>=0 || neg_int(c) || neg_int(b)){
619     free(a);free(b);free(c);return;}
620  free(b);free(c);
621  *a='N';
622  for(i=1;(*(a+i)=*(a+p));i++,p++);
623  escribir_supuestos(a);
624  }

625  /* Negación de condicional */
626  void inf_ne_cond(int x)
627  {
```

```
628  int i,l,p,k,h;
629  char *a,*b,*c;
630  l=strlen((fv+x)->f);
631  if((a=(char*)calloc(l,sizeof(char)))==NULL)
     falta_memoria(1);
632  for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
633  p=posicion_d(a);
634  if((b=(char*)calloc(p,sizeof(char)))==NULL)
     falta_memoria(1);
635  if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
636  falta_memoria(1);
637  for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
638  for(i=p;(*(c+i-p)=*(a+i));i++) ;
639  if((buscar(a,p,l-1)>=0 || negar(a,1,p)>=0)\
640     && !neg_int(a) && !interrogada(a,0,1)){
641     escribir_supuestos(a);
642     free(b);free(c);
643     return;
644     }
645  if(negar(a,p,l-1)<0 && !interrogada(a,1,p) &&
     !neg_int(c)){
646     *a='N';*(a+1)='\0';strcat(a,c);
647     escribir_supuestos(a);
648     free(b);free(c);
649     return;
650     }
651  if(buscar(a,1,p)<0 && !interrogada(a,1,p) &&
     !neg_int(c)){
652     escribir_supuestos(b);
653     free(a);free(c);
654     }
655  }
```

La siguiente función (656–678) busca la negación de una fórmula interrogada. Su necesidad obedece a la exigencia de no repetir supuestos de demostración a nivel de subfórmulas.

```
656  /* Buscar la negación de una fórmula interrogada */
657  int neg_int(char *a)
658  {
659  int j,i,l,w,z;
```

```
660  z=0;
661  for(i=0;i<ul;i++){
662     if((fv+i)->u!=95) continue;
663     if(*a!='N' && *((fv+i)->f)!='N') continue;
664     l=strlen(a);w=strlen((fv+i)->f);
665     j=l<w?w-l:l-w;
666     if(j==0 || j>1) continue;
667     if(l>w && *a=='N'){
668   for(j=0;j<w;j++) if(*((fv+i)->f+j)!=*(a+j+1)) break;
669        if(j==w) z=1;
670        }
671     if(w>l && *((fv+i)->f)=='N'){
672   for(j=0;j<l;j++) if(*(a+j)!=*((fv+i)->f+j+1)) break;
673        if(j==l) z=1;
674        }
675     if(z) break;
676     }
677  return z;
678 }
```

III. 4. Programa PDALP03

El programa siguiente, PDALP03, efectúa deducciones en el cálculo M, con modificaciones respecto a PDALP02 por lo que se refiere al modo de aplicación de las reglas de inferencia, como comentaremos después. PDALP03 consta de los ficheros PDALP30.h, PDALP31.c y PDALP32.c.

```
1  /* Programa PDALP03.
2     Fichero PDALP30.h */

3  /* Estructuras */
4  struct FORMULAS_VALIDAS{
5     char *f;
6     char u;
7     char r;
8     int l1;
9     int l2;
10    };
11 struct MARCAS{
12    char ma;
13    int l;
14    };
15 /* Definiciones */
16 #define TFV sizeof(struct FORMULAS_VALIDAS)
17 #define TM sizeof(struct MARCAS)

18 /* Funciones prototipo */
19 /* Fichero pdalp31.c */
20 void calculos_dn(void);
21 int simbolos(char*);
22 int es_formula(char*);
23 int simbolo_do(char*);
24 void pausa(void);
25 void falta_memoria(int);
26 void salida_formulas(void);
27 void salida_formulas(void);

28 /* Fichero pdalp32.c */
29 void demostrada(void);
30 void introducir_disyuncion(char*,int);
31 void introducir_conjuncion(int,int);
```

```
32  void introducir_bicondicional(int,int);
33  void tachar(void);
34  void escribir_supuestos(char*);
35  int inferencias(void);
36  void inf_conj(void);
37  void inf_dis(void);
38  void inf_cond(void);
39  void inf_bi(void);
40  void inf_do_ne(void);
41  void inf_ne_conj(void);
42  void inf_ne_conj_2(void);
43  int supuestos(void);
44  int subformulas(char*);
45  int interrogada(char*,int,int);
46  int posicion_d(char*);
47  int buscar(char*,int,int);
48  int negacion(char*,int);
49  int contradiccion(void);
50  void repeticion(int);
51  void introducir_negacion(int);
52  int negar(char*,int,int);
53  void inf_ne_bi(void);
54  void inf_ne_dis(void);
55  void inf_ne_cond(void);
56  int neg_int(char*);
57  void inf_ne_bi_2(void);
58  void inf_ne_dis_2(void);
59  void inf_ne_cond_2(void);
```

El fichero PDALP31.c es idéntico al fichero PDALP21.c. Las líneas 1–156 del fichero PDALP32.c son idénticas a las del fichero PDALP22.c.

La siguiente función (157–186) aplica las reglas de inferencia de modo diferente a todos los programas anteriores. Aquí hemos optado por escribir todas las reglas de inferencia que pueden ser aplicadas a las fórmulas y proceder a su ejecución consecutiva y sistemática.

```
157  /* Decidir inferencias */
158  int inferencias()
159  {
160  int r;
161  r=ul;
```

```
162  inf_ne_conj();
163  if(ul-r) return 1;
164  inf_ne_dis();
165  if(ul-r) return 1;
166  inf_ne_bi();
167  if(ul-r) return 1;
168  inf_do_ne();
169  if(ul-r) return 1;
170  inf_cond();
171  if(ul-r) return 1;
172  inf_conj();
173  if(ul-r) return 1;
174  inf_dis();
175  if(ul-r) return 1;
176  inf_bi();
177  if(ul-r) return 1;
178  inf_ne_dis_2();
179  if(ul-r) return 1;
180  inf_ne_cond_2();
181  if(ul-r) return 1;
182  inf_ne_conj_2();
183  if(ul-r) return 1;
184  inf_ne_bi_2();
185  return ul-r;
186  }
```

La estrategia que hemos seguido obliga a una reescritura de las funciones de aplicación de las reglas de inferencia. Las modificaciones, no obstante, son más bien técnicas y no afectan en lo sustancial al modo de aplicación. Por ello suprimiremos los comentarios, remitiéndonos a las observaciones realizadas a los programas anteriores.

```
187  /* Eliminación de conjunción */
188  void inf_conj(void)
189  {
190  int j,l,p,x;
191  for(x=0;x<ul;x++){
192     if((fv+x)->u!=80) continue;
193     if(*((fv+x)->f)!='K') continue;
194     l=strlen((fv+x)->f);
195     p=posicion_d((fv+x)->f);
196     if(buscar((fv+x)->f,1,p)<0){
```

```
197         (fv+ul)->u=80;
198         (fv+ul)->r=51;
199         (fv+ul)->l1=x;
200         (fv+ul)->l2=-1;
201         (fv+ul)->f=(char*)calloc(p,sizeof(char));
202         if((fv+ul)->f==NULL) falta_memoria(1);
203    for(j=0;j<p-1;j++) *((fv+ul)->f+j)=*((fv+x)->f+j+1);
204         *((fv+ul)->f+j)='\0';
205         ul++;if(ul==fvd) falta_memoria(2);
206         break;
207         }
208       if(buscar((fv+x)->f,p,l)<0){
209         (fv+ul)->u=80;
210         (fv+ul)->r=51;
211         (fv+ul)->l1=x;
212         (fv+ul)->l2=-1;
213         (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
214         if((fv+ul)->f==NULL) falta_memoria(1);
215        for(j=p;(*((fv+ul)->f+j-p)=*((fv+x)->f+j));j++);
216         ul++;if(ul==fvd) falta_memoria(2);
217         break;
218         }
219       }
220   }

221   /* Eliminación de disyunción */
222   void inf_dis(void)
223   {
224   int i,p,k,l,x;
225   for(x=0;x<ul;x++){
226      if((fv+x)->u!=80) continue;
227      if(*((fv+x)->f)!='A') continue;
228      l=strlen((fv+x)->f);
229      p=posicion_d((fv+x)->f);
230      if(buscar((fv+x)->f,1,p)<0 &&
         (k=negar((fv+x)->f,p,l))>=0){
231         if(l-p>strlen((fv+k)->f)) {
232         introducir_negacion(k);k=ul-1;}
233         (fv+ul)->u=80;
234         (fv+ul)->r=52;
235         (fv+ul)->l1=x;
```

```
236         (fv+ul)->l2=k;
237         (fv+ul)->f=(char*)calloc(p,sizeof(char));
238         if((fv+ul)->f==NULL) falta_memoria(1);
239    for(i=0;i<p-1;i++) *((fv+ul)->f+i)=*((fv+x)->f+i+1);
240         *((fv+ul)->f+i)='\0';
241         ul++;if(ul==fvd) falta_memoria(2);
242         break;
243         }
244       if(buscar((fv+x)->f,p,l)<0 &&
          (k=negar((fv+x)->f,1,p))>=0){
245         if(p-1>strlen((fv+k)->f))
246         {introducir_negacion(k);k=ul-1;}
247         (fv+ul)->u=80;
248         (fv+ul)->r=52;
249         (fv+ul)->l1=x;
250         (fv+ul)->l2=k;
251         (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
252         if((fv+ul)->f==NULL) falta_memoria(1);
253        for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++);
254         ul++;if(ul==fvd) falta_memoria(2);
255         break;
256         }
257       }
258    }

259    /* Eliminación de condicional */
260    void inf_cond(void)
261    {
262    int i,p,k,l,x;
263    for(x=0;x<ul;x++){
264       if((fv+x)->u!=80) continue;
265       if(*((fv+x)->f)!='C') continue;
266       l=strlen((fv+x)->f);
267       p=posicion_d((fv+x)->f);
268       if(buscar((fv+x)->f,p,l)<0 && \
269       (k=buscar((fv+x)->f,1,p))>=0){
270         (fv+ul)->u=80;
271         (fv+ul)->r=53;
272         (fv+ul)->l1=x;
273         (fv+ul)->l2=k;
274         (fv+ul)->f=(char*)calloc(l-p+1,sizeof(char));
```

```
275         if((fv+ul)->f==NULL) falta_memoria(1);
276       for(i=p;(*((fv+ul)->f+i-p)=*((fv+x)->f+i));i++) ;
277         ul++;if(ul==fvd) falta_memoria(2);
278         break;
279         }
280       if((negar((fv+x)->f,1,p))<0 &&
          (negar((fv+x)->f,p,1))>=0){
281         (fv+ul)->u=80;
282         (fv+ul)->r=60;
283         (fv+ul)->l1=x;
284         (fv+ul)->l2=k;
285         (fv+ul)->f=(char*)calloc(p+1,sizeof(char));
286         if((fv+ul)->f==NULL) falta_memoria(1);
287         *((fv+ul)->f)='N';
288        for(i=1;i<p;i++) *((fv+ul)->f+i)=*((fv+x)->f+i);
289         *((fv+ul)->f+i)='\0';
290         ul++;if(ul==fvd) falta_memoria(2);
291         break;
292         }
293       }
294   }

295   /* Eliminación de bicondicional */
296   void inf_bi(void)
297   {
298   int i,k,l,p,x;
299   char *d,*e;
300   for(x=0;x<ul;x++){
301      if((fv+x)->u!=80) continue;
302      if(*((fv+x)->f)!='E') continue;
303      l=strlen((fv+x)->f);
304      p=posicion_d((fv+x)->f);
305      if((d=(char*)calloc(l+1,sizeof(char)))==NULL)
306      falta_memoria(1);
307      if((e=(char*)calloc(l+1,sizeof(char)))==NULL)
308      falta_memoria(1);
309      *d=*e='C';
310      for(i=1;(*(d+i)=*((fv+x)->f+i));i++);
311      k=l-p;
312      for(i=0;i+p<l;i++) *(e+i+1)=*((fv+x)->f+i+p);
313      for(i=1;i<p;i++) *(e+k+i)=*((fv+x)->f+i);
         *(e+k+i)='\0';
```

```
314     if(buscar(d,0,1)<0){
315        (fv+ul)->u=80;
316        (fv+ul)->r=54;
317        (fv+ul)->l1=x;
318        (fv+ul)->l2=-1;
319        (fv+ul)->f=d;
320        free(e);
321        ul++;if(ul==fvd) falta_memoria(2);
322        break;
323        }
324     if(buscar(e,0,1)<0){
325        (fv+ul)->u=80;
326        (fv+ul)->r=54;
327        (fv+ul)->l1=x;
328        (fv+ul)->l2=-1;
329        (fv+ul)->f=e;
330        free(d);
331        ul++;if(ul==fvd) falta_memoria(2);
332        break;
333        }
334     free(d); free(e);
335     }
336  }

337  /* Doble negación */
338  void inf_do_ne(void)
339  {
340  int i,l,x;
341  for(x=0;x<ul;x++){
342     if((fv+x)->u!=80) continue;
343     if(strncmp((fv+x)->f,"NN",2)) continue;
344     l=strlen((fv+x)->f);
345     if(buscar((fv+x)->f,2,l)<0){
346        (fv+ul)->u=80;
347        (fv+ul)->r=50;
348        (fv+ul)->l1=x;
349        (fv+ul)->l2=-1;
350        (fv+ul)->f=(char*)calloc(l-1,sizeof(char));
351        if((fv+ul)->f==NULL) falta_memoria(1);
352      for(i=0;(*((fv+ul)->f+i)=*((fv+x)->f+i+2)); i++);
353        ul++;if(ul==fvd) falta_memoria(2);
```

```
354         break;
355         }
356      }
357   }

358   /* Negación de la conjunción */
359   void inf_ne_conj(void)
360   {
361   int i,p,k,h,l,x;
362   char *a;
363   for(x=0;x<ul;x++){
364      if((fv+x)->u!=80) continue;
365      if(strncmp((fv+x)->f,"NK",2)) continue;
366      l=strlen((fv+x)->f);
367      a=(fv+x)->f+1;
368      p=posicion_d(a);
369      k=buscar(a,1,p);h=buscar(a,p,l);
370      if(k>=0 && h>=0){
371         introducir_conjuncion(k,h);
372         break;
373         }
374      }
375   }
```

Se observará que para algunas reglas de inferencia se han escrito dos funciones distintas. La razón es la de salvar lo más posible el aspecto experto de la deducción. Cuando hay dos funciones para una regla de inferencia, la primera en aplicarse es la más simple, la que efectuaría la deducción en un número menor de líneas; la segunda es más compleja y puede entenderse como una regla derivada.

```
376   /* Negación de la conjunción 2 */
377   void inf_ne_conj_2(void)
378   {
379   int i,p,k,h,l,x;
380   char *a,*b,*c;
381   for(x=0;x<ul;x++){
382      if((fv+x)->u!=80) continue;
383      if(strncmp((fv+x)->f,"NK",2)) continue;
384      l=strlen((fv+x)->f);
385      a=(fv+x)->f+1;
386      p=posicion_d(a);
```

```
387     k=buscar(a,1,p);h=buscar(a,p,1);
388     if(h<0 && k<0) continue;
389     b=(char*)calloc(p,sizeof(char));
390     c=(char*)calloc(l-p,sizeof(char));
391     if(b==NULL || c==NULL) falta_memoria(1);
392     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
393     for(i=p;(*(c+i-p)=*(a+i));i++);
394    if( (k>=0 && ( negar(a,p,l-1)>=0 || neg_int(c) ) )\
395     ||  (h>=0 && ( negar(a,1,p)>=0 || neg_int(b) )))
396     {free(b);free(c);continue;}
397     if(k>=0){
398        free(b);
399        b=(char*)calloc(l-p+1,sizeof(char));
400        if(!b) falta_memoria(1);
401        *b='N';strcat(b,c);
402        escribir_supuestos(b);
403        free(c);
404        break;
405        }
406     free(c);
407     c=(char*)calloc(p+1,sizeof(char));
408     if(!c) falta_memoria(1);
409     *c='N';strcat(c,b);
410     escribir_supuestos(c);
411     free(b);
412     break;
413     }
414  }
```

Las funciones de apertura de nuevos supuestos, suponer fórmulas elementales, establecimiento de subfórmulas, buscar una fórmula interrogada, repetir fórmulas, buscar una contradicción, buscar fórmulas utilizables, buscar una negación, establecimiento de que una fórmula es la negación de otra, e introducir negación (415–599) son todas ellas idénticas a las del programa PDALP02.

```
600  /* Negación de bicondicional */
601  void inf_ne_bi()
602  {
603  int i,p,l,k,h,x;
604  char *a,*b,*c,*d,*e;
605  for(x=0;x<ul;x++){
```

```
606     if((fv+x)->u!=80) continue;
607     if(strncmp((fv+x)->f,"NE",2)) continue;
608     l=strlen((fv+x)->f);
609     if((a=(char*)calloc(l,sizeof(char)))==NULL)
610     falta_memoria(1);
611     for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
612     p=posicion_d(a);
613     if((b=(char*)calloc(p,sizeof(char)))==NULL)
614     falta_memoria(1);
615     if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
616     falta_memoria(1);
617     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
618     for(i=p;(*(c+i-p)=*(a+i));i++) ;
619     if((d=(char*)calloc(l,sizeof(char)))==NULL)
620     falta_memoria(1);
621     if((e=(char*)calloc(l,sizeof(char)))==NULL)
622     falta_memoria(1);
623     *d=*e='C';strcat(d,b);strcat(d,c);strcat(e,c);
        strcat(e,b);
624     free(b);free(c);
625     k=buscar(d,0,l-1);h=buscar(e,0,l-1);
626     if(k>=0 && h>=0){
627        introducir_bicondicional(k,h);
628        free(a);free(d);free(e);
629        break;
630        }
631     free(a);free(d);free(e);
632     }
633  }

634  /* Negación de bicondicional 2 */
635  void inf_ne_bi_2()
636  {
637  int i,p,l,k,h,r,s,x;
638  char *a,*b,*c,*d,*e;
639  for(x=0;x<ul;x++){
640     if((fv+x)->u!=80) continue;
641     if(strncmp((fv+x)->f,"NE",2)) continue;
642     l=strlen((fv+x)->f);
643     if((a=(char*)calloc(l,sizeof(char)))==NULL)
644     falta_memoria(1);
```

```
645     for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
646     p=posicion_d(a);
647     if((b=(char*)calloc(p,sizeof(char)))==NULL)
648     falta_memoria(1);
649     if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
650     falta_memoria(1);
651     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
652     for(i=p;(*(c+i-p)=*(a+i));i++) ;
653     if((d=(char*)calloc(l,sizeof(char)))==NULL)
654     falta_memoria(1);
655     if((e=(char*)calloc(l,sizeof(char)))==NULL)
656     falta_memoria(1);
657     *d=*e='C';strcat(d,b);strcat(d,c);strcat(e,c);
        strcat(e,b);
658     free(b);free(c);
659     k=buscar(d,0,l-1);h=buscar(e,0,l-1);
660     if(h<0 && k<0){
661        free(a);free(d);free(e);
662        continue;
663        }
664     r=negar(d,0,l-1);
665     s=negar(e,0,l-1);
666     if( (k>=0 && ( s>=0 || neg_int(e) ) ) \
667     ||  (h>=0 && ( r>=0 || neg_int(d) ) ) )
668     {free(a);free(d);free(e);continue;}
669     free(a);
670     a=(char*)calloc(l+1,sizeof(char));
671     if(!a) falta_memoria(1);
672     *a='N';
673     if(k>=0) strcat(a,e);
674     else strcat(a,d);
675     free(d);free(e);
676     escribir_supuestos(a);
677     break;
678     }
679  }

680  /* Negación de la disyunción */
681  void inf_ne_dis()
682  {
683  int i,h,k,l,p,x;
```

```
684  char *a,*b,*c;
685  for(x=0;x<ul;x++){
686     if((fv+x)->u!=80) continue;
687     if(strncmp((fv+x)->f,"NA",2)) continue;
688     l=strlen((fv+x)->f);
689     if((a=(char*)calloc(l,sizeof(char)))==NULL)
690     falta_memoria(1);
691     for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
692     p=posicion_d(a);
693     if((b=(char*)calloc(p,sizeof(char)))==NULL)
694     falta_memoria(1);
695     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
696     if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
697     falta_memoria(1);
698     for(i=p;(*(c+i-p)=*(a+i));i++) ;
699     k=h=-1;
700     if((h=buscar(a,1,p))>=0 ||
       (k=buscar(a,p,l-1))>=0){
701        h=h<k?k:h;
702        introducir_disyuncion(a,h);
703        free(a);free(b);free(c);
704        break;
705        }
706     free(a);free(b);free(c);
707     }
708  }

709  /* Negación de la disyunción  2 */
710  void inf_ne_dis_2()
711  {
712  int i,h,k,l,p,x;
713  char *a,*b,*c;
714  for(x=0;x<ul;x++){
715     if((fv+x)->u!=80) continue;
716     if(strncmp((fv+x)->f,"NA",2)) continue;
717     l=strlen((fv+x)->f);
718     if((a=(char*)calloc(l,sizeof(char)))==NULL)
719     falta_memoria(1);
720     for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
721     p=posicion_d(a);
722     if((b=(char*)calloc(p,sizeof(char)))==NULL)
```

```
723     falta_memoria(1);
724     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
725     if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
726     falta_memoria(1);
727     for(i=p;(*(c+i-p)=*(a+i));i++) ;
728     if((k=negar(a,1,p))<0 || (h=negar(a,p,l-1))<0){
729        (mar+m)->l=ul;(mar+m)->ma=60;
730        m++;if(m==md) falta_memoria(4);
731        (fv+ul)->u=80;
732        (fv+ul)->r=99;
733        (fv+ul)->l1=(fv+ul)->l2=-1;
734        if(k<0){
735    if(((fv+ul)->f=(char*)calloc(p+1,\
736    sizeof(char)))==NULL) falta_memoria(1);
737    *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
738    }
739        else{
740    if(((fv+ul)->f=(char*)calloc(l-p+1,\
741    sizeof(char)))==NULL) falta_memoria(1);
742    *((fv+ul)->f)='N';strcat((fv+ul)->f,c);
743    }
744        ul++;if(ul==fvd) falta_memoria(2);
745        (mar+m)->l=ul;(mar+m)->ma=50;
746        m++;if(m==md) falta_memoria(4);
747        (fv+ul)->u=99;
748        (fv+ul)->r=99;
749        (fv+ul)->l1=(fv+ul)->l2=-1;
750        if(k<0){
751    if(((fv+ul)->f=(char*)calloc(p+2,\
752    sizeof(char)))==NULL) falta_memoria(1);
753    strncat((fv+ul)->f,"NN",2);strcat((fv+ul)->f,b);
754    }
755        else{
756    if(((fv+ul)->f=(char*)calloc(l-p+2,\
757    sizeof(char)))==NULL) falta_memoria(1);
758    strncat((fv+ul)->f,"NN",2);strcat((fv+ul)->f,c);
759    }
760        ul++;if(ul==fvd) falta_memoria(2);
761        (mar+m)->l=ul;(mar+m)->ma=50;
762        m++;if(m==md) falta_memoria(4);
763        (fv+ul)->u=99;
```

```
764         (fv+ul)->r=50;
765         (fv+ul)->l1=l-1;
766         (fv+ul)->l2=-1;
767         if(k<0){
768     if(((fv+ul)->f=(char*)calloc(p,\
769     sizeof(char)))==NULL) falta_memoria(1);
770     strcpy((fv+ul)->f,b);
771     }
772         else{
773     if(((fv+ul)->f=(char*)calloc(l-p,\
774     sizeof(char)))==NULL) falta_memoria(1);
775     strcpy((fv+ul)->f,c);
776     }
777         ul++;if(ul==fvd) falta_memoria(2);
778         free(b);free(c);
779         (mar+m)->l=ul;(mar+m)->ma=50;
780         m++;if(m==md) falta_memoria(4);
781         (fv+ul)->u=99;
782         (fv+ul)->r=42;
783         (fv+ul)->l1=l-1;
784         (fv+ul)->l2=-1;
785         (fv+ul)->f=a;
786         ul++;if(ul==fvd) falta_memoria(2);
787         (mar+m)->l=ul;(mar+m)->ma=50;
788         m++;if(m==md) falta_memoria(4);
789         (fv+ul)->u=99;
790         (fv+ul)->r=61;
791         (fv+ul)->l1=x;
792         (fv+ul)->l2=-1;
793         (fv+ul)->f=(char*)calloc(l+1,sizeof(char));
794         if((fv+ul)->f==NULL) falta_memoria(1);
795         strcpy((fv+ul)->f,(fv+x)->f);
796         ul++;if(ul==fvd) falta_memoria(2);
797         break;
798         }
799      free(a);free(b);free(c);
800      }
801  }

802  /* Negación de condicional */
803  void inf_ne_cond()
```

```
804  {
805  int i,l,p,k,x;
806  char *a,*b,*c;
807  for(x=0;x<ul;x++){
808     if((fv+x)->u!=80) continue;
809     if(strncmp((fv+x)->f,"NC",2)) continue;
810     l=strlen((fv+x)->f);
811     if((a=(char*)calloc(l,sizeof(char)))==NULL)
812     falta_memoria(1);
813     for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
814     p=posicion_d(a);
815     if((b=(char*)calloc(p,sizeof(char)))==NULL)
816     falta_memoria(1);
817     if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
818     falta_memoria(1);
819     for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
820     for(i=p;(*(c+i-p)=*(a+i));i++) ;
821     if((k=buscar(a,p,l-1))>=0){
822        (mar+m)->l=ul;(mar+m)->ma=60;
823        m++;if(m==md) falta_memoria(4);
824        (fv+ul)->u=80;
825        (fv+ul)->r=99;
826        (fv+ul)->l1=(fv+ul)->l2=-1;
827        (fv+ul)->f=a;
828        ul++;if(ul==fvd) falta_memoria(2);
829        repeticion(k);
830        (mar+m)->l=ul-1;(mar+m)->ma=50;
831        m++;if(m==md) falta_memoria(4);
832        free(b);free(c);
833        break;
834        }
835     if((k=negar(a,1,p))>=0){
836        (mar+m)->l=ul;(mar+m)->ma=60;
837        m++;if(m==md) falta_memoria(4);
838        (fv+ul)->u=80;
839        (fv+ul)->r=99;
840        (fv+ul)->l1=(fv+ul)->l2=-1;
841        (fv+ul)->f=a;
842        ul++;if(ul==fvd) falta_memoria(2);
843        (mar+m)->l=ul;(mar+m)->ma=50;
844        m++;if(m==md) falta_memoria(4);
```

```
845         (fv+ul)->u=99;
846         (fv+ul)->r=99;
847         (fv+ul)->l1=(fv+ul)->l2=-1;
848         (fv+ul)->f=b;
849         ul++;if(ul==fvd) falta_memoria(2);
850         (mar+m)->l=ul;(mar+m)->ma=50;
851         m++;if(m==md) falta_memoria(4);
852         (fv+ul)->u=99;
853         (fv+ul)->r=61;
854         (fv+ul)->l1=ul-1;
855         (fv+ul)->l2=-1;
856         (fv+ul)->f=(char*)calloc(p+1,sizeof(char));
857         if((fv+ul)->f==NULL) falta_memoria(1);
858         strcpy((fv+ul)->f,(fv+k)->f);
859         ul++;if(ul==fvd) falta_memoria(2);
860         break;
861         }
862      free(a);free(b);free(c);
863      }
864   }

865   /* Negación de condicional  2 */
866   void inf_ne_cond_2()
867   {
868   int i,l,p,k,h,x;
869   char *a,*b,*c;
870   for(x=0;x<ul;x++){
871      if((fv+x)->u!=80) continue;
872      if(strncmp((fv+x)->f,"NC",2)) continue;
873      l=strlen((fv+x)->f);
874      if((a=(char*)calloc(l,sizeof(char)))==NULL)
875      falta_memoria(1);
876      for(i=0;(*(a+i)=*((fv+x)->f+1+i));i++) ;
877      p=posicion_d(a);
878      if((b=(char*)calloc(p,sizeof(char)))==NULL)
879      falta_memoria(1);
880      if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
881      falta_memoria(1);
882      for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
883      for(i=p;(*(c+i-p)=*(a+i));i++) ;
884      if((h=buscar(a,1,p))<0 || (k=negar(a,p,l-1))<0){
```

```
885         (mar+m)->l=ul;(mar+m)->ma=40;
886         m++;if(m==md) falta_memoria(4);
887         us=ul;
888         (fv+ul)->u=99;
889         (fv+ul)->r=99;
890         (fv+ul)->l1=(fv+ul)->l2=-1;
891         if(h<0){
892    if(((fv+ul)->f=(char*)calloc(p,sizeof(char)))==NULL)
893     falta_memoria(1);
894     strcpy((fv+ul)->f,b);
895     }
896         else{
897     if(((fv+ul)->f=(char*)calloc(l-p+1,\
898     sizeof(char)))==NULL) falta_memoria(1);
899     *((fv+ul)->f)='N';strcat((fv+ul)->f,c);
900     }
901         ul++;if(ul==fvd) falta_memoria(2);
902         (fv+ul)->u=80;
903         (fv+ul)->r=99;
904         (fv+ul)->l1=(fv+ul)->l2=-1;
905         if(h<0){
906     if(((fv+ul)->f=(char*)calloc(p+1,\
907     sizeof(char)))==NULL) falta_memoria(1);
908     *((fv+ul)->f)='N';strcat((fv+ul)->f,b);
909     }
910         else{
911     if(((fv+ul)->f=(char*)calloc(l-p+2,\
912     sizeof(char)))==NULL) falta_memoria(1);
913     strncat((fv+ul)->f,"NN",2);strcat((fv+ul)->f,c);
914     }
915         ul++;if(ul==fvd) falta_memoria(2);
916         if(h>=0 && k<0){
917     (fv+ul)->u=80;
918     (fv+ul)->r=50;
919     (fv+ul)->l1=ul-1;
920     (fv+ul)->l2=-1;
921     if(((fv+ul)->f=(char*)calloc(l-p,\
922     sizeof(char)))==NULL) falta_memoria(1);
923     strcpy((fv+ul)->f,c);
924     ul++;if(ul==fvd) falta_memoria(2);
925     }
```

```
926         inf_ne_cond();
927         free(a);free(b);free(c);
928         break;
929         }
930      free(a);free(b);free(c);
931      }
932   }

933   /* Buscar la negación de una fórmula interrogada */
934   int neg_int(char *a)
935   {
936   int j,i,l,w,z;
937   z=0;
938   for(i=0;i<ul;i++){
939      if((fv+i)->u!=95) continue;
940      if(*a!='N' && *((fv+i)->f)!='N') continue;
941      l=strlen(a);w=strlen((fv+i)->f);
942      j=l<w?w-l:l-w;
943      if(j==0 || j>1) continue;
944      if(l>w && *a=='N'){
945    for(j=0;j<w;j++) if(*((fv+i)->f+j)!=*(a+j+1)) break;
946         if(j==w) z=1;
947         }
948      if(w>l && *((fv+i)->f)=='N'){
949    for(j=0;j<l;j++) if(*(a+j)!=*((fv+i)->f+j+1)) break;
950         if(j==l) z=1;
951         }
952      if(z) break;
953      }
954   return z;
955   }
```

III. 5. Programa PDALP04

El programa PDALP04 efectúa deducciones en el cálculo J. Consta de los ficheros PDALP40.h, PDALP41.c–PDALP44.c.

La única estructura del programa es la que hemos denominado RAMA. Una deducción es un conjunto de estas estructuras. Contiene información sobre las fórmulas que la componen, si la rama está abierta o cerrada y sobre los nodos —el número de nodos disponibles por la asignación de memoria, el nodo de la última fórmula escrita y el nodo sobre el que se aplican las reglas de inferencia—. Las fórmulas se almacenan con independencia de las ramas (4–10).

```
 1 /* Programa PDALP04.
 2    Fichero PDALP40.h */

 3 /* Estructuras */
 4 struct RAMA{
 5    int *r;          /*fórmulas de la rama          */
 6    int n;           /*número de nodos en curso     */
 7    int nd;          /*número de nodos disponibles  */
 8    int ni;          /*nodo de inferencia           */
 9    char u;          /*abierta o cerrada            */
10    };

11 /* Definiciones */
12 #define TR sizeof(struct RAMA)

13 /* Funciones prototipo */

14 /* Fichero pdalp41.c */
15 void calculos_dn(void);
16 int simbolos(char*);
17 int es_formula(char*);
18 int simbolo_do(char*);
19 void pausa(void);
20 void falta_memoria(int);

21 /* Fichero pdalp42.c */
22 void demostrada(void);
23 int contradiccion(int);
24 int negar(int,int);
```

```
25 void salida_formulas(void);
26 void traducir(char*,char*);
27 void espacio(int);
28 int posicion_d(char*);

29 /* Fichero pdalp43.c */
30 int inferencias(int);
31 void inf_conj(int,int);
32 void inf_dis(int,int);
33 void inf_cond(int,int);
34 void inf_bi(int,int);
35 void inf_do_ne(int,int);
36 void inf_ne_conj(int,int);

37 /* Fichero pdalp44.c */
38 void inf_ne_bi(int,int);
39 void inf_ne_dis(int,int);
40 void inf_ne_cond(int,int);
41 void pedir(int);
```

El fichero PDALP41.c contiene las rutinas de entrada y verificación de fórmulas.

```
 1 /* Programa PDALP04.
 2    Fichero PDALP41.c */

 3 /* Librerías */
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <string.h>
 7 #include <ctype.h>
 8 #include <conio.h>
 9 #include <alloc.h>
10 #include "pdalp40.h"
11 #include <time.h>

12 /* Variables globales */
13 /* Contadores de fórmulas escritas y ramas: ul,m;
14  * última fórmula para inferencias: us;
15  * ramas y fórmulas válidas
16  * disponibles por asignación dinámica en curso: md, fvd.
17  */
```

```
18 int ul,m,md,fvd;
19 char **fv;          /*puntero a fórmulas escritas    */
20 struct RAMA *ra;    /*puntero a ramas                */
```

Las funciones de inicio e instrucciones generales, si se desea ésta última, son las ya conocidas en anteriores programas.

La siguiente función (30–114) gestiona la introducción de premisas, si las hay, y la conclusión. El procedimiento consiste simplemente en escribir las premisas y la negación de la conclusión (84–94).

```
30 /* Introducción de conclusión y premisas */
31 void calculos_dn()
32 {
33 int j;
34 do{
35    j=ul=m=0;
36    fv=(char**)calloc(500,sizeof(char*));
37    if(fv==NULL) falta_memoria(1);
38    ra=(struct RAMA*)calloc(100,TR);
39    if(ra==NULL) falta_memoria(1);
40    fvd=500;md=100;
41    clrscr();
42    cputs("Introduzca la conclusión y las premisas, \
43    si las hubiere.\n\r");
44    cputs("Pulse [ENTER] para abandonar o finalizar \
45    la introducción.\n\r");
46    do{                    /*Introducción de fórmulas */
47       int x,y;
48       char *a,*b;
49       if(j==0) cputs("Conclusión:  ");
50       else cprintf("Premisa  %d:  ",j);
51       y=wherey();x=wherex();
52       gotoxy(x+60,y);
53       putch(42);
54       gotoxy(x,y);
55       a=(char*)calloc(62,sizeof(char));
56       if(!a) falta_memoria(1);
57       fflush(stdin);
58       gets(a);
59       if(!*a){
60    for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
```

```
61    free(a);
62    break;
63    }
64       b=(char*)calloc(strlen(a)+1,sizeof(char));
65       if(b==NULL) falta_memoria(1);
66       strcpy(b,a);
67       free(a);
68       if(!simbolos(b)){
69    putch(7);
70    cputs("\n\rError en la escritura de la fórmula.");
71          pausa();
72          free(b);
73       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
74          continue;
75          }
76       if(!es_formula(b)){
77    putch(7);
78     cputs("\n\rError en la escritura de la fórmula.");
79          pausa();
80          free(b);
81       for(x=wherey();x>=y;x--) {gotoxy(1,x);clreol();}
82          continue;
83    }
84       if(j==0){
85    *(fv+ul)=(char*)calloc(strlen(b)+2,sizeof(char));
86    if(*(fv+ul)==NULL) falta_memoria(1);
87    **(fv+ul)='N';strcat(*(fv+ul),b);
88    free(b);
89    ul++;
90    }
91       else{
92    *(fv+ul)=b;
93    ul++;
94    }
95       j++;
96       } while(1);
97    if(j>0) {
98       time_t x;
99       struct tm *z;
100      int t1,t2,t3,t4;
101      /* lectura del reloj*/
```

```
102         x=time(NULL);
103         z=localtime(&x);
104         t1=z->tm_min;t2=z->tm_sec;
105         demostrada();
106         x=time(NULL);
107         z=localtime(&x);
108         t3=z->tm_min;t4=z->tm_sec;
109         cprintf("  (%d:%d)",t3-t1,t4-t2);
110         salida_formulas();
111         }
112      free(fv);free(ra);      /*liberar memoria*/
113      } while(j!=0);
114  }
```

Las funciones de verificación de símbolos, de fórmulas, establecimiento del símbolo dominante de una fórmula y pausa son las ya conocidas de los programas anteriores (115–279).

```
280 /* Interrupción por falta de memoria */
281 void falta_memoria(int i)
282 {
283 switch(i){
284    case 1: {cputs("Falta de memoria. ");break;}
285    case 2: fv=(char**)
                realloc(fv,(fvd+50)*sizeof(char*));
286      if(fv==NULL) {cputs("Falta de memoria. ");break;}
287      fvd+=50;return;
288    case 4: ra=(struct RAMA*)realloc(ra,(md+200)*TR);
289      if(ra==NULL) {cputs("Falta de memoria. ");break;}
290      md+=200;return;
291    }
292 putch(7);
293 cprintf("\n\rMemoria libre %u bytes",coreleft());
294 pausa();
295 clrscr();
296 exit(1);
297 }
```

El fichero PDALP42.c contiene rutinas de búsqueda y de salida de las deducciones.

```
 1 /* Programa PDALP04.
 2    Fichero PDALP42.c */

 3 /* Librerías */
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <string.h>
 7 #include <conio.h>
 8 #include <ctype.h>
 9 #include "pdalp40.h"

10 /* Variables globales externas */
11 extern int ul,m,md,fvd;
12 extern char **fv;
13 extern struct RAMA *ra;
```

La función siguiente (14–38) controla el procedimiento de demostración. Escribe las premisas y la negación de la conclusión en la primera rama (18–25). La rutina básica de las deducciones consiste en la búsqueda de contradicciones y en la aplicación de las reglas de inferencia (28–34).

```
14 /* Búsqueda de demostraciones */
15 void demostrada()
16 {
17 int i,r;
18 if(((ra+0)->r=(int*)calloc(100,sizeof(int)))==NULL)
19 falta_memoria(1);
20 for(i=0;i<ul;i++) *((ra+0)->r+i)=i;
21 (ra+0)->n=ul;
22 (ra+0)->nd=100;
23 (ra+0)->u=40;
24 (ra+0)->ni=0;
25 m++;
26 for(i=0;i<m;i++){
27    int j;
28    do{
29       if((r=contradiccion(i))) break;
30       j=inferencias(i);
31       } while(j);
32    if(r) continue;
33    else break;
```

```
34     }
35 putch(7);
36 if(i<m) cputs("\n\rFórmula no demostrable. ");
37 else cputs("\n\rFórmula demostrada. ");
38 }
```

La siguiente función (39–54) busca contradicciones en una rama.

```
39 /* Buscando contradicciones */
40 int contradiccion(int i)
41 {
42 int r,x,y;
43 r=0;
44 for(x=0;x<(ra+i)->n;x++){
45    y=*((ra+i)->r+x);
46    if(**(fv+y)!='N') continue;
47    if((negar(y,i))>=0){
48       (ra+i)->u=50;
49       r=1;
50       break;
51       }
52    }
53 return r;
54 }
```

La siguiente función (55–80) busca la negación de una fórmula en una rama.

```
55 /* Buscar una negación */
56 int negar(int y,int h)
57 {
58 int j,i,l,w,z,x;
59 char *a;
60 z=-1;
61 a=*(fv+y);
62 for(i=0;i<(ra+h)->n;i++){
63    x=*((ra+h)->r+i);
64    if(x==y) continue;
65    if(*a!='N' && **(fv+x)!='N') continue;
66    l=strlen(a);w=strlen(*(fv+x));
67    j=l<w?w-l:l-w;
68    if(j==0 || j>1) continue;
```

```
 69     if(l>w && *a=='N'){
 70      for(j=0;j<w;j++) if(*(*(fv+x)+j)!=*(a+j+1)) break;
 71        if(j==w) z=x;
 72        }
 73     if(w>l && **(fv+x)=='N'){
 74      for(j=0;j<l;j++) if(*(a+j)!=*(*(fv+x)+j+1)) break;
 75        if(j==l) z=x;
 76        }
 77     if(z>=0) break;
 78     }
 79 return z;
 80 }
```

La salida de las fórmulas de la deducción en el cálculo J es bastante complicada. Para fórmulas de mediana complejidad es imposible la típica salida en árbol, al no caber en pantalla. Hemos optado por una salida lineal, de modo que aparezcan todas las fórmulas de cada rama y todas las ramas. Si se quiere reconstruir manualmente el árbol deductivo a partir de esta información, puede hacerse.

```
 81 /* Salida de demostraciones */
 82 void salida_formulas()
 83 {
 84 int i,j,l,n,z,p;
 85 char **f,e;
 86 f=(char**)calloc(ul,sizeof(char*));
 87 if(!f) falta_memoria(1);
 88 for(i=0;i<ul;i++){
 89    *(f+i)=(char*)calloc(3*strlen(*(fv+i)),sizeof(char));
 90    if(*(f+i)==NULL) falta_memoria(1);
 91    traducir(*(fv+i),*(f+i));
 92    }
 93 for(i=0;i<ul;i++) free(*(fv+i));
 94 for(i=0;i<m;i++) (ra+i)->nd=0;
 95 cprintf("\n\r¿Imprimir el resultado? (S/N): ");
 96 e=toupper(getchar());
 97 l=strlen(*(f+0));
 98 if(l<9) l=10;
 99 else l+=2;
100 z=0;p=0;
101 pausa();
102 do{
```

```
103    int x,y;
104    clrscr();
105    x=1;
106    if(z+(80/l)>=m) n=m;
107    else n=z+(80/l);
108    for(j=z;j<n;j++){
109       if((ra+j)->nd==-1){x+=l;continue;}
110       gotoxy(x,1);
111       cprintf("Rama %2d",j+1);
112       x+=l;
113       if(e==83 && !p){
114    fprintf(stdprn,"Rama %2d",j+1);
115    espacio(l-7);
116    }
117       }
118    cputs("\n\r");
119    if(e==83 && !p) fputs("\n\r",stdprn);
120    do{
121       int q=0;
122       x=1;p=1;
123       for(j=z;j<n;j++){
124    int w=(ra+j)->nd;
125    if(w==-1){
126       x+=l;q++;
127       if(e==83) espacio(l);
128       continue;
129       }
130    gotoxy(x,wherey());
131    if(w==-2){
132       if((ra+j)->u==50) cputs(" X");
133       x+=l;
134       if(e==83){
135          if((ra+j)->u==50)
136          {fputs(" X",stdprn);espacio(l-2);}
137          else espacio(l);
138          }
139       (ra+j)->nd=-1;
140       continue;
141       }
142    cputs(*(f+*((ra+j)->r+w)));
143    x+=l;
```

```
144     if(e==83){
145        fprintf(stdprn,"%s",*(f+*((ra+j)->r+w)));
146        espacio(l-strlen(*(f+*((ra+j)->r+w))));
147        }
148     if(w==(ra+j)->n-1) (ra+j)->nd=-2;
149     else (ra+j)->nd=w+1;
150     }
151        cputs("\n\r");
152        if(e==83) fputs("\n\r",stdprn);
153        if(q==n-z) {p=0;break;}
154        }while(wherey()<24);
155     if(e==83 && !p) fputs("\n\n\r",stdprn);
156     if(!p) z=n;
157     getchar();
158     }while(z<m);
159  for(j=0;j<ul;j++) free(*(f+j));free(f);
160  for(j=0;j<m;j++) free((ra+j)->r);
161  }

162  /* Escribir espacios */
163  void espacio(int j)
164  {
165  int i;
166  for(i=0;i<j;i++) fputs(" ",stdprn);
167  }
```

Las funciones de traducción de la notación polaca y de establecimiento de subfórmulas de una fórmula (168–249) son las ya conocidas de programas anteriores.

El fichero PDALP43.c contiene la rutina de inferencias y varias rutinas de aplicación de reglas. Las reglas de inferencia se aplican del mismo modo que en una deducción manual: las fórmulas obtenidas se escriben en todas las ramas de las que la fórmula a la que se aplica la regla es un nodo, o bien se abren ramas nuevas para las alternativas —caso de la disyunción, por ejemplo—.

```
1  /* Programa PDALP04.
2     Fichero PDALP43.c */

3  /* Librerías */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
```

```
 7 #include "pdalp40.h"

 8 /* Variables globales externas */
 9 extern int ul,m,md,fvd;
10 extern char **fv;
11 extern struct RAMA *ra;

12 /* Decidir inferencias */
13 int inferencias(int j)
14 {
15 int r,us;
16 r=0;
17 us=(ra+j)->ni;
18 if(us<(ra+j)->n){
19    switch(**(fv+*((ra+j)->r+us))){
20       case 75: inf_conj(j,us); break;
21       case 65: inf_dis(j,us); break;
22       case 67: inf_cond(j,us);break;
23       case 69: inf_bi(j,us); break;
24       case 78: switch(*(*(fv+*((ra+j)->r+us))+1)){
25           case 78: inf_do_ne(j,us);break;
26           case 75: inf_ne_conj(j,us);break;
27           case 69: inf_ne_bi(j,us);break;
28           case 65: inf_ne_dis(j,us);break;
29           case 67: inf_ne_cond(j,us);
30           }
31       }
32    (ra+j)->ni=++us;r=1;
33    }
34 return r;
35 }

36 /* Eliminación de conjunción */
37 void inf_conj(int i,int k)
38 {
39 int j,l,p,us,x;
40 us=*((ra+i)->r+k);
41 l=strlen(*(fv+us));
42 p=posicion_d(*(fv+us));
43 *(fv+ul)=(char*)calloc(p,sizeof(char));
44 if(*(fv+ul)==NULL) falta_memoria(1);
```

```
45 for(j=0;j<p-1;j++) *(*(fv+ul)+j)=*(*(fv+us)+j+1);
46 *(*(fv+ul)+j)='\0';
47 for(j=0;j<ul;j++) if(!strcmp(*(fv+ul),*(fv+j)))
   {x=j;break;}
48 if(j<ul) free(*(fv+ul));
49 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
50 *((ra+i)->r+((ra+i)->n))=x;
51 (ra+i)->n++;if((ra+i)->n==(ra+i)->nd) pedir(i);
52 *(fv+ul)=(char*)calloc(l-p+1,sizeof(char));
53 if(*(fv+ul)==NULL) falta_memoria(1);
54 for(j=p;(*(*(fv+ul)+j-p)=*(*(fv+us)+j));j++);
55 for(j=0;j<ul;j++) if(!strcmp(*(fv+ul),*(fv+j)))
   {x=j;break;}
56 if(j<ul) free(*(fv+ul));
57 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
58 *((ra+i)->r+((ra+i)->n))=x;
59 (ra+i)->n++;if((ra+i)->n==(ra+i)->nd) pedir(i);
60 }

61 /* Eliminación de disyunción */
62 void inf_dis(int h,int k)
63 {
64 int i,p,l,x,us;
65 us=*((ra+h)->r+k);
66 l=strlen(*(fv+us));
67 p=posicion_d(*(fv+us));
68 *(fv+ul)=(char*)calloc(p,sizeof(char));
69 if(*(fv+ul)==NULL) falta_memoria(1);
70 for(i=0;i<p-1;i++) *(*(fv+ul)+i)=*(*(fv+us)+i+1);
71 *(*(fv+ul)+i)='\0';
72 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
73 if(i<ul) free(*(fv+ul));
74 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
75 *((ra+h)->r+((ra+h)->n))=x;
76 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
77 *(fv+ul)=(char*)calloc(l-p+1,sizeof(char));
78 if(*(fv+ul)==NULL) falta_memoria(1);
79 for(i=p;(*(*(fv+ul)+i-p)=*(*(fv+us)+i));i++);
80 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
```

```
 81 if(i<ul) free(*(fv+ul));
 82 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
 83 for(i=m-1;i>h;i—) memcpy((ra+i+1),(ra+i),TR);
 84 (ra+h+1)->r=(int*)calloc((ra+h)->nd,sizeof(int));
 85 if((ra+h+1)->r==NULL) falta_memoria(1);
 86 for(i=0;i<(ra+h)->n-1;i++)
    *((ra+h+1)->r+i)=*((ra+h)->r+i);
 87 *((ra+h+1)->r+i)=x;
 88 (ra+h+1)->n=(ra+h)->n;
 89 (ra+h+1)->nd=(ra+h)->nd;
 90 (ra+h+1)->u=40;
 91 (ra+h+1)->ni=(ra+h)->ni+1;
 92 m++;if(m==md)falta_memoria(4);
 93 }

 94 /* Eliminación de condicional */
 95 void inf_cond(int h, int k)
 96 {
 97 int i,p,l,x,us;
 98 us=*((ra+h)->r+k);
 99 l=strlen(*(fv+us));
100 p=posicion_d(*(fv+us));
101 *(fv+ul)=(char*)calloc(p+1,sizeof(char));
102 if(*(fv+ul)==NULL) falta_memoria(1);
103 **(fv+ul)='N';
104 for(i=1;i<p;i++) *(*(fv+ul)+i)=*(*(fv+us)+i);
105 *(*(fv+ul)+i)='\0';
106 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
107 if(i<ul) free(*(fv+ul));
108 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
109 *((ra+h)->r+((ra+h)->n))=x;
110 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
111 *(fv+ul)=(char*)calloc(l-p+1,sizeof(char));
112 if(*(fv+ul)==NULL) falta_memoria(1);
113 for(i=p;(*(*(fv+ul)+i-p)=*(*(fv+us)+i));i++) ;
114 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
115 if(i<ul) free(*(fv+ul));
116 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
117 for(i=m-1;i>h;i- —) memcpy((ra+i+1),(ra+i),TR);
```

```
118 (ra+h+1)->r=(int*)calloc((ra+h)->nd,sizeof(int));
119 if((ra+h+1)->r==NULL) falta_memoria(1);
120 for(i=0;i<(ra+h)->n-1;i++)
    *((ra+h+1)->r+i)=*((ra+h)->r+i);
121 *((ra+h+1)->r+i)=x;
122 (ra+h+1)->n=(ra+h)->n;
123 (ra+h+1)->nd=(ra+h)->nd;
124 (ra+h+1)->u=40;
125 (ra+h+1)->ni=(ra+h)->ni+1;
126 m++;if(m==md)falta_memoria(4);
127 }

128 /* Eliminación de bicondicional */
129 void inf_bi(int h, int q)
130 {
131 int i,k,l,p,x,us;
132 char *d,*e;
133 us=*((ra+h)->r+q);
134 l=strlen(*(fv+us));
135 p=posicion_d(*(fv+us));
136 if((d=(char*)calloc(l+1,sizeof(char)))==NULL)
137 falta_memoria(1);
138 if((e=(char*)calloc(l+1,sizeof(char)))==NULL)
139 falta_memoria(1);
140 *d=*e='C';
141 for(i=1;(*(d+i)=*(*(fv+us)+i));i++);
142 k=l-p;
143 for(i=0;i+p<l;i++) *(e+i+1)=*(*(fv+us)+i+p);
144 for(i=1;i<p;i++) *(e+k+i)=*(*(fv+us)+i);*(e+k+i)='\0';
145 *(fv+ul)= d;
146 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
147 if(i<ul) free(*(fv+ul));
148 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
149 *((ra+h)->r+((ra+h)->n))=x;
150 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
151 *(fv+ul)=e;
152 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
153 if(i<ul) free(*(fv+ul));
154 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
```

```
155 *((ra+h)->r+((ra+h)->n))=x;
156 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
157 }

158 /* Doble negación */
159 void inf_do_ne(int h, int k)
160 {
161 int i,l,x,us;
162 us=*((ra+h)->r+k);
163 l=strlen(*(fv+us));
164 *(fv+ul)=(char*)calloc(l-1,sizeof(char));
165 if(*(fv+ul)==NULL) falta_memoria(1);
166 for(i=0;(*(*(fv+ul)+i)=*(*(fv+us)+i+2));i++);
167 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
168 if(i<ul) free(*(fv+ul));
169 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
170 *((ra+h)->r+((ra+h)->n))=x;
171 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
172 }

173 /* Negación de la conjunción */
174 void inf_ne_conj(int h, int k)
175 {
176 int p,l,i,x,us;
177 char *a;
178 us=*((ra+h)->r+k);
179 l=strlen(*(fv+us));
180 a=*(fv+us)+1;
181 p=posicion_d(a);
182 *(fv+ul)=(char*)calloc(p+1,sizeof(char));
183 if(*(fv+ul)==NULL) falta_memoria(1);
184 **(fv+ul)='N';
185 for(i=1;i<p;i++) *(*(fv+ul)+i)=*(a+i);
    *(*(fv+ul)+i)='\0';
186 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
187 if(i<ul) free(*(fv+ul));
188 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
189 *((ra+h)->r+((ra+h)->n))=x;
190 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
```

```
191 *(fv+ul)=(char*)calloc(l-p+1,sizeof(char));
192 if(*(fv+ul)==NULL) falta_memoria(1);
193 **(fv+ul)='N';
194 for(i=p;(*(*(fv+ul)+i-p+1)=*(a+i));i++);
195 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
196 if(i<ul) free(*(fv+ul));
197 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
198 for(i=m-1;i>h;i--) memcpy((ra+i+1),(ra+i),TR);
199 (ra+h+1)->r=(int*)calloc((ra+h)->nd,sizeof(int));
200 if((ra+h+1)->r==NULL) falta_memoria(1);
201 for(i=0;i<(ra+h)->n-1;i++)
    *((ra+h+1)->r+i)=*((ra+h)->r+i);
202 *((ra+h+1)->r+i)=x;
203 (ra+h+1)->n=(ra+h)->n;
204 (ra+h+1)->nd=(ra+h)->nd;
205 (ra+h+1)->u=40;
206 (ra+h+1)->ni=(ra+h)->ni+1;
207 m++;if(m==md)falta_memoria(4);
208 }
```

El fichero PDALP44.c contiene rutinas de aplicación de reglas de inferencia y una función para ampliar la memoria disponible para ramas (112–117).

```
 1 /* Programa PDALP04.
 2    Fichero PDALP44.c */

 3 /* Librerías */
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <string.h>
 7 #include "pdalp40.h"

 8 /* Variables globales externas */
 9 extern int ul,m,md,fvd;
10 extern char **fv;
11 extern struct RAMA *ra;

12 /* Negación de bicondicional */
13 void inf_ne_bi(int h, int k)
14 {
```

```
15 int i,p,l,x,us;
16 char *a,*b,*c,*d,*e;
17 us=*((ra+h)->r+k);
18 l=strlen(*(fv+us));
19 a=*(fv+us)+1;
20 p=posicion_d(a);
21 if((b=(char*)calloc(p,sizeof(char)))==NULL)
   falta_memoria(1);
22 if((c=(char*)calloc(l-p,sizeof(char)))==NULL)
23 falta_memoria(1);
24 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
25 for(i=p;(*(c+i-p)=*(a+i));i++) ;
26 if((d=(char*)calloc(l+1,sizeof(char)))==NULL)
27 falta_memoria(1);
28 if((e=(char*)calloc(l+1,sizeof(char)))==NULL)
29 falta_memoria(1);
30 strncat(d,"NC",2);strncat(e,"NC",2);
31 strcat(d,b);strcat(d,c);strcat(e,c);strcat(e,b);
32 free(b);free(c);
33 *(fv+ul)=d;
34 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
35 if(i<ul) free(*(fv+ul));
36 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
37 *((ra+h)->r+((ra+h)->n))=x;
38 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
39 *(fv+ul)=e;
40 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
41 if(i<ul) free(*(fv+ul));
42 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
43 for(i=m-1;i>h;i--) memcpy((ra+i+1),(ra+i),TR);
44 (ra+h+1)->r=(int*)calloc((ra+h)->nd,sizeof(int));
45 if((ra+h+1)->r==NULL) falta_memoria(1);
46 for(i=0;i<(ra+h)->n-1;i++)
   *((ra+h+1)->r+i)=*((ra+h)->r+i);
47 *((ra+h+1)->r+i)=x;
48 (ra+h+1)->n=(ra+h)->n;
49 (ra+h+1)->nd=(ra+h)->nd;
50 (ra+h+1)->u=40;
51 (ra+h+1)->ni=(ra+h)->ni+1;
```

```
52 m++;if(m==md)falta_memoria(4);
53 }

54 /* Negación de la disyunción */
55 void inf_ne_dis(int h, int k)
56 {
57 int i,l,p,x,us;
58 char *a,*b,*c;
59 us=*((ra+h)->r+k);
60 l=strlen(*(fv+us));
61 a=*(fv+us)+1;
62 p=posicion_d(a);
63 if((b=(char*)calloc(p+1,sizeof(char)))==NULL)
64 falta_memoria(1);
65 *b='N';
66 for(i=1;i<p;i++) *(b+i)=*(a+i);*(b+i)='\0';
67 if((c=(char*)calloc(l-p+1,sizeof(char)))==NULL)
68 falta_memoria(1);
69 *c='N';
70 for(i=p;(*(c+i-p+1)=*(a+i));i++) ;
71 *(fv+ul)=b;
72 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
73 if(i<ul) free(*(fv+ul));
74 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
75 *((ra+h)->r+((ra+h)->n))=x;
76 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
77 *(fv+ul)=c;
78 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
   {x=i;break;}
79 if(i<ul) free(*(fv+ul));
80 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
81 *((ra+h)->r+((ra+h)->n))=x;
82 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
83 }

84 /* Negación de condicional */
85 void inf_ne_cond(int h, int k)
86 {
87 int i,l,p,x,us;
88 char *a,*b,*c;
```

```
 89 us=*((ra+h)->r+k);
 90 l=strlen(*(fv+us));
 91 a=*(fv+us)+1;
 92 p=posicion_d(a);
 93 if((b=(char*)calloc(p,sizeof(char)))==NULL)
    falta_memoria(1);
 94 if((c=(char*)calloc(l-p+1,sizeof(char)))==NULL)
 95 falta_memoria(1);
 96 for(i=0;i<p-1;i++) *(b+i)=*(a+i+1);*(b+i)='\0';
 97 *c='N';
 98 for(i=p;(*(c+i-p+1)=*(a+i));i++) ;
 99 *(fv+ul)=b;
100 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
101 if(i<ul) free(*(fv+ul));
102 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
103 *((ra+h)->r+((ra+h)->n))=x;
104 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
105 *(fv+ul)=c;
106 for(i=0;i<ul;i++) if(!strcmp(*(fv+ul),*(fv+i)))
    {x=i;break;}
107 if(i<ul) free(*(fv+ul));
108 else{x=ul;ul++;if(ul==fvd) falta_memoria(2);}
109 *((ra+h)->r+((ra+h)->n))=x;
110 (ra+h)->n++;if((ra+h)->n==(ra+h)->nd) pedir(h);
111 }

112 /* Ampliar memoria para ramas */
113 void pedir(int h)
114 {
115 (ra+h)->r=(int*)
    realloc((ra+h)->r,(ra+h)->nd+50*sizeof(int));
116 if((ra+h)->r==NULL) falta_memoria(1);
117 }
```

IV. RENDIMIENTO DE LOS PROGRAMAS

En la escritura de programas para razonamiento automatizado hay diversas variables a las que atender, de modo que optimizar los resultados respecto a una de ellas conlleva, las más de las veces, sacrificar alguna de las otras. Estas variables, básicamente, son: tiempo de ejecución, memoria utilizable y estrategia inteligente —que la deducción sea lo más parecida a la de un lógico experto—. Cuanto mayor sea el carácter experto que quiera conferirse a una estrategia deductiva, tanto mayor será el tiempo de ejecución, la complicación del programa y la memoria a utilizar.

Cualquier estrategia empleada presenta, además, puntos ciegos; funciona mejor con un tipo de fórmulas que con otro. Por ejemplo, no es irrelevante el orden en que se introduce un bicondicional. Supóngase que queremos deducir $\alpha \leftrightarrow \beta$, donde $\alpha \rightarrow \beta$ es deducible, pero $\beta \rightarrow \alpha$ no lo es. Si un programa descompone la prueba de $\alpha \leftrightarrow \beta$ en la prueba de $\alpha \rightarrow \beta$ y en la de $\beta \rightarrow \alpha$, tardará, obviamente más tiempo en la ejecución si le introducimos la fórmula como $\alpha \leftrightarrow \beta$ que si la introducimos como $\beta \leftrightarrow \alpha$. Quizás pueda demostrarse que toda estrategia deductiva presenta puntos ciegos como el que acabamos de mencionar; que para toda estrategia deductiva es posible encontrar fórmulas cuya deducción sea más inexperta que la del lógico. Incluso aunque no fuese éste el caso, hay una enorme diferencia entre el modo de deducción del lógico y el de la máquina. Por decirlo de esta forma, el lógico tiene una visión *gestaltística* donde la máquina sólo puede operar paso a paso mediante el análisis de los componentes de la totalidad.

Para hacernos una idea aproximada del rendimiento de los programas en lo que se refiere a *tiempo de ejecución,* hemos escogido unas fórmulas al azar y hemos intentado probarlas.

1. $p1 \leftrightarrow p1$
2. $p1 \vee \neg p1$
3. $(p1 \rightarrow p2) \rightarrow (\neg p2 \rightarrow \neg p1)$
4. $(p1 \vee p2) \leftrightarrow \neg(\neg p1 \wedge \neg p2)$
5. $p1 \wedge (p2 \vee p3) \leftrightarrow (p1 \wedge p2) \vee (p1 \wedge p3)$
6. $p1 \vee (p2 \wedge p3) \leftrightarrow (p1 \vee p2) \wedge (p1 \vee p3)$
7. $p1 \wedge (p2 \vee p3) \leftrightarrow (p1 \wedge p2) \vee (p1 \wedge p1)$
8. $(p1 \leftrightarrow p2) \leftrightarrow ((p1 \leftrightarrow p3) \leftrightarrow (p2 \leftrightarrow p3))$
9. $(p1 \leftrightarrow p2) \leftrightarrow ((p1 \leftrightarrow p3) \leftrightarrow (p2 \leftrightarrow p4))$
10. $(p1 \leftrightarrow p2) \leftrightarrow ((p1 \leftrightarrow p3) \leftrightarrow (p2 \leftrightarrow p3)) \wedge ((p1 \wedge p2) \leftrightarrow \neg(\neg p1 \vee \neg p2))$

Las fórmulas 7 y 9 no son deducibles.

Ejecutados los programas en un PC 386 (20 Mhz) se obtuvo la deducción en un tiempo igual o inferior a 1 segundo, excepto para las fórmulas 8 y 10 cuyos tiempos en segundos fueron:

	PDALP00	PDALP01	PDALP02	PDALP03	PDALP04
Fórmula 8	1	4	2	2	1
Fórmula 10	1	–	5	6	2

PDALP01 no pudo terminar la deducción de la fórmula 10 por falta de memoria.

Ejecutados los programas en un PC XT (8 Mhz) dan unos tiempos sólo ligeramente superiores para fórmulas cuya deducción esté en torno a las 100 líneas; a partir de aquí los tiempos se elevan sensiblemente. Por ejemplo, la deducción de la fórmula 8 —267 líneas en PDALP02 y 254 en PDALP03— se efectúa en torno a los 12 segundos. Un programa con la misma estrategia deductiva que PDALP00 escrito en BASIC ejecuta, por ejemplo, la fórmula 4 en 30 segundos y la fórmula 8 en 12 minutos y 15 segundos.

La conclusión que extraemos es que paralos cálculos G y M las estrategias inteligentes dan mejor resultado, por lo que se refiere a tiempo de ejecución que las estrategias más mecánicas.

Por lo que se refiere al *carácter experto* de los programas —su aspecto de deducción inteligente, podríamos decir—, dejando a un lado PDALP04 que, obviamente, es una estrategia mecánica por su analiticidad, podemos extraer algunas conclusiones.

Para cualquier programa, incluir una rutina que elimine fórmulas no necesarias para el resultado final aumenta el aspecto de deducción inteligente aunque la estrategia en sí misma no lo sea. El número de fórmulas que así se eliminan es muy variable; por ejemplo, en PDALP00 se eliminan 3 líneas en la deducción de la fórmula 8 y 10 líneas en la de la fórmula 5. Hecha esta salvedad, las diferencias entre PDALP00 y PDALP01 —con una estrategia deductiva no orientada— por lo que se refiere al número de líneas empleadas en la deducción y a la elegancia de la misma son muy notables. He aquí las líneas de deducción de las fórmulas anteriores:

	PDALP00	PDALP01	PDALP02	PDALP03
1	7	24	8	8
2	9	9	8	8
3	8	44	9	9
4	27	63	47	38
5	48	97	83	62
6	42	95	74	62
7	49	89	81	64
8	114	382	267	254
9	10	72	38	35

Por lo que se refiere a la diferencia en la aplicación de las reglas de inferencia —por exploración sistemática de la forma de las fórmulas para decidir qué regla aplicar o de las reglas para decidir a qué fórmula aplicarlas—, reflejada en las estrategias de PDALP02 y PDALP03, podemos observar una longitud menor en las deducciones de PDALP03; en cuanto a tiempos de ejecución ambos programas son similares.

Las estrategias desarrolladas en los programas PDALP02 y PDALP03 penalizan las fórmulas con bicondicionales al no descomponer el bicondicional en dos subobjetivos de prueba, los dos condicionales que lo forman lógicamente. Es por esta razón y no por peculiaridades del cálculo M por lo que el número de líneas de la deducción es sensiblemente superior a partir de la fórmula 4 en estos programas por comparación a PDALP00.

A continuación exponemos algunos ejemplos de las deducciones efectuadas.

Programa PDALP00:

```
1. ┌ p1
2. │┌ ¬p1
3. │└ p1∧¬p1   I∧ 1,2
4. │ ¬¬p1      I¬ 2-3
5. └ p1        E¬ 4
6.  p1 → p1    I→ 1-5
7.  p1 ↔ p1    I↔ 6,6
```

```
1. ┌ ¬(p1∨ ¬p1)
2. │┌ p1
3. ││ p1∨ ¬p1            I∨ 2
4. │└ (p1∨ ¬p1)∧ ¬(p1∨ ¬p1)  I∧3,1
5. │ ¬p1                 I¬ 2-4
6. │ p1∨ ¬p1            I∨ 5
7. └¬(p1∨ ¬p1)∧ (p1∨ ¬p1)  I∧ 1,6
8.  ¬¬(p1∨ ¬p1)          I¬ 1-7
9.  p1∨ ¬p1              E¬ 8
```

```
1. ┌ p1 → p2
2. │┌ ¬p2
3. ││┌ p1
4. │││ p2            E→ 1,3
5. ││└ ¬p2∧ p2        I∧ 2,4
6. │└ ¬p1             I¬3-5
7. └¬p2 → ¬p1         I→ 2-6
8.  (p1 → p2) → ( ¬p2 → ¬p1)  I→ 1-7
```

```
1. ┌ p1∨p2
2. │┌ ¬p1∧¬p2
3. ││ ¬p1            E∧2
```

```
4.  |  ¬p2                E∧ 2
5.  |  p1                 E∨ 1,4
6.  |  ¬p1 ∧ p1           I∧ 3,5
7.  ¬(¬p1 ∧ ¬p2)          I¬ 2-6
8.  p1 ∨ p2 → ¬( ¬p1 ∧ ¬p2)   I→ 1-7
9.  ¬( ¬p1 ∧ ¬p2)
10.    ¬(p1 ∨ p2)
11.       p1
12.       p1 ∨ p2          I∨ 11
13.       (p1 ∨ p2)∧ ¬(p1 ∨ p2) I∧ 12,10
14.    ¬p1                 I¬ 11-13
15.       ¬p2
16.       ¬p1 ∧ ¬p2        I∧ 15,14
17.       ¬( ¬p1 ∧ ¬p2)∧ ( ¬p1 ∧ ¬p2)  I∧ 9,16
18.    ¬¬p2                I¬ 15-17
19.       p2
20.       p1 ∨ p2          I∨ 19
21.       (p1∨ p2)∧ ¬(p1 ∨ p2) I∧ 20,10
22.    ¬p2                 I¬ 19-21
23.    ¬¬p2∧ ¬p2           I∧ 18,22
24. ¬¬(p1 ∨ p2)            I¬ 10-23
25. p1 ∨ p2                E¬ 24
26. ¬( ¬p1 ∧ ¬p2) → p1 ∨ p2   I→ 9-25
27. p1 ∨ p2 ↔ ¬( ¬p1 ∧ ¬p2)   I↔ 26,8
```

Programa PDALP01:

```
1.  ¬((p1 → p2)->( ¬p2 → ¬p1))
2.     ¬p2 → ¬p1
3.        p1 → p2
4.           ¬( ¬p2 → ¬p1)
5.           ( ¬p2 → ¬p1)∧ ¬( ¬p2 → ¬p1)    I∧ 2,4
6.        ¬¬( ¬p2 → ¬p1)          I¬ 4-5
7.        ¬p2 → ¬p1               E¬ 6
8.     (p1 → p2) → ( ¬p2 → ¬p1)      I→ 3-7
9.     ((p1 → p2) → ( ¬p2 → ¬p1)) ∧ ¬((p1 → p2) → ( ¬p2 → ¬p1)) I∧ 8, 1
10.  ¬( ¬p2 → ¬p1)                I¬ 2-9
11.    ¬(p1 → p2)
12.       p1 → p2
13.       ¬( ¬p2 → ¬p1)
14.       (p1 → p2) ∧ ¬(p1 → p2)     I∧ 11,12
```

15. ¬¬ (¬p2 → ¬p1) I ¬13-14
16. ¬p2 → ¬p1 E ¬15
17. (p1 → p2) → (¬p2 → ¬p1) I → 12-16
18. ((p1 → p2) → (¬p2 → ¬p1)) ∧ ¬((p1 → p2) → (¬p2 → ¬p1)) I ∧ 17, 1
19. ¬¬(p1 → p2) I ¬11-18
20. p1 → p2 E ¬19
21. ¬p1
22. ¬p2
23. ¬¬p1
24. ¬p1 ∧ ¬¬p1 I ∧ 21,23
25. ¬¬¬p1 I ¬23-24
26. ¬p1 E ¬25
27. ¬p2 → ¬p1 I → 22-26
28. (¬p2 → ¬p1) ∧ ¬(¬p2 → ¬p1) I ∧ 27,10
29. ¬¬p1 I ¬21-28
30. p1 E ¬29
31. ¬¬p2
32. ¬p2
33. ¬¬p1
34. ¬p2 ∧ ¬¬p2 I ∧ 31,32
35. ¬¬¬p1 I ¬33-34
36. ¬p1 E ¬35
37. ¬p2 → ¬p1 I → 32-36
38. (¬p2 → ¬p1) ∧ ¬(¬p2 → ¬p1) I ∧ 37,10
39. ¬¬¬p2 I ¬31-38
40. ¬p2 E ¬39
41. p2 E → 20,30
42. p2 ∧ ¬p2 I ∧ 41,40
43. ¬¬((p1 → p2) → (¬p2 → ¬p1)) I ¬1-42
44. (p1 → p2) → (¬p2 → ¬p1) E ¬43

Programa PDALP02: (Recuérdese que utilizamos # como símbolo sustituto de la interrogación tachada.)

1. # p1∨¬p1
2. ¬(p1∨¬p1)
3. # ¬p1
4. ¬¬p1
5. p1 DN 4
6. p1∨¬p1 IV 5
7. ¬(p1∨¬p1) R 2
8. p1∨¬p1 IV 3

1. # (p1 → p2) → (¬p2 → ¬p1)
2. | p → p2
3. | # ¬p2 → ¬p1
4. | | ¬p2
5. | | # ¬p1
6. | | | ¬¬p1
7. | | | p1 DN 6
8. | | | p2 MP 2,7
9. | | | ¬p2 R 4

Programa PDALP03:

1. # p1 ∨ p2 ↔ ¬(¬p1 ∧¬p2)
2. | ¬(p1 ∨ p ↔ ¬(¬p1 ∧¬p2))
3. | # p1 ∨p2 → ¬(¬p1 ∧¬p2)
4. | | p1 ∨ p2
5. | | # ¬(¬p1 ∧ ¬p2)
6. | | | ¬¬(¬p1 ∧¬p2)
7. | | | ¬p1 ∧¬p2 DN 6
8. | | | ¬p1 E ∧ 7
9. | | | ¬p2 E ∧ 7
10. | | | p1 E ∨ 4,9
11. | # ¬(¬(¬p1 ∧¬p2) → p1 ∨ p2)
12. | | ¬¬(¬(¬p1 ∧¬p2) → p1 ∨ p2)
13. | | ¬(¬p1 ∧¬p2) → p1 ∨ p2 DN 12
14. | | p1 ∨ p2 ↔ ¬(¬p1 ∧¬p2) I ↔ 13,3
15. | | ¬(p1 ∨ p2 ↔ ¬(¬p1 ∧¬p2)) R 2
16. | # ¬(¬p1 ∧¬p2)
17. | | ¬¬(¬p1 ∧¬p2)
18. | | # ¬(¬p1 ∧¬p2) → p1 ∨ p2
19. | | | ¬(¬p1 ∧¬p2)
20. | | | ¬¬(¬p1 ∧¬p2) R 19
21. | | ¬(¬(¬p1 ∧¬p2) → p1 ∨ p2) R 11
22. | # ¬(p1 ∨ p2)
23. | | ¬¬(p1 ∨ p2)
24. | | p1 ∨ p2 DN 23
25. | | # ¬(¬p1 ∧¬p2) → p1 ∨ p2
26. | | | p1 ∨ p2 R 24
27. | | ¬(¬(¬p1 ∧¬p2) → p1 ∨ p2) R 11
28. | # ¬p1

29. | | ¬¬p1
30. | | p1 DN 29
31. | | p1∨p2 I ∨30
32. | | ¬(p1 ∨ p2) R 22
33. | # ¬p2
34. | | ¬¬p2
35. | | p2 DN 34
36. | | p1 ∨ p2 I ∨ 35
37. | | ¬(p1 ∨ p2) R 22
38. | | ¬p1 ∧¬p2 I ∧33,28

Programa PDALP04. Deducción de la fórmula 4 escribiendo todas las fórmulas de cada rama y clausurando después:

Rama 1

¬(p1∨p2 ↔ ¬(¬p1∧¬p2))
¬(p1∨p2→ ¬(¬p1 ∧¬p2))
p1∨ p2
¬¬(¬p1 ∧¬p2)
p1
¬p1 ∧¬p2
¬p1
¬p2
X

Rama 2

¬(p1∨ p2 ↔ ¬(¬p1∧¬p2))
¬(p1 ∨ p2→ ¬(¬p1 ∧ ¬p2))
p1 ∨p2
¬¬(¬p1∧¬p2)
p2
¬p1∧¬ p2
¬p1
¬p2
X

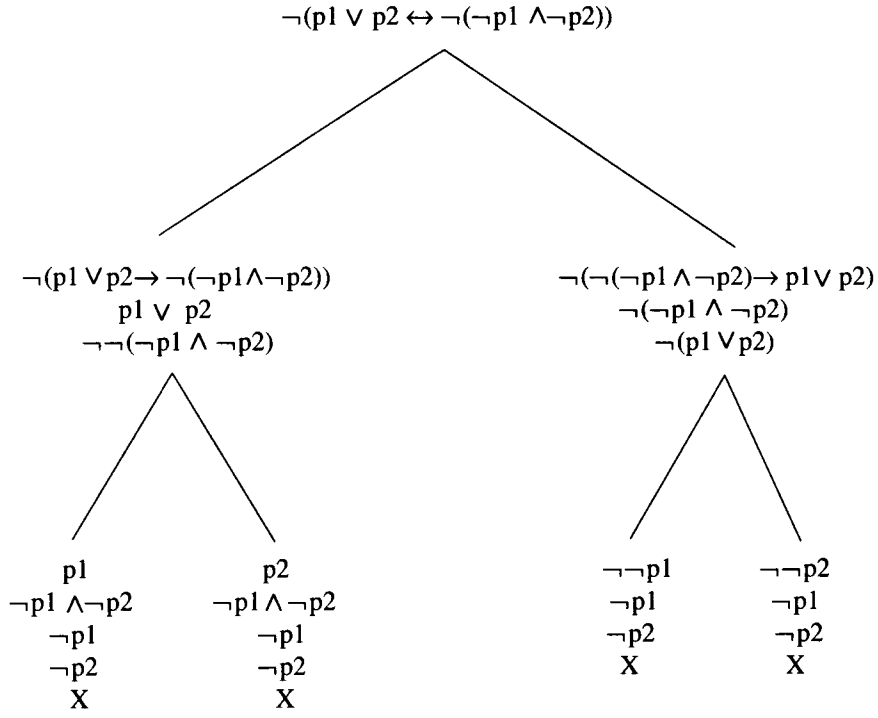
Rama 3

¬(p1∨p2 ↔ ¬(¬p1∧¬p2))
¬(¬ (¬p1 ∧ ¬p2) → p1∨ p2)
¬(¬p1 ∧¬p2)
¬(p1 ∨ p2)
¬¬p1
¬p1
¬p2
X

Rama 4

¬(p1 ∨ p2 ↔ ¬(¬p1 ∧ ¬p2))
¬(¬(¬p1 ∧¬p2)→ p1 ∨ p2)
¬(¬p1 ∧¬p2)
¬(p1∨ p2)
¬¬p2
¬p1
¬p2
X

Esta deducción equivale al árbol:



V. BIBLIOGRAFIA

1. AGAZZI, EVANDRO (Ed.): *Modern-logic. A survey. Historical, philosophical and mathematical aspects of modern logic and its applications.* Reidel. Dordrecht.

2. ANDERSON, J.M. and JOHNSTONE, H.W. (1962): *Natural Deduction. The logical basis of axioms systems.* Wadsworth. Belmont.

BARWISE, JON (Ed.) (1985): *Handbook of mathematical logic.* North Holland. Amsterdam.

3. BETH, E.W. (1962): *Formal methods. An introduction to symbolic logic and to the study of effective operations in arithmetical logic.* D. Reidel. Dordrecht.

4. BETH, E.W. (1978): *Entrañamiento semántico y derivabilidad formal.* Teorema. Valencia.

5. BIBEL, W. (1987): *Automated theorem proving.* Friedr. Vieweg & Sohn. Braunschweig.

6. BIERMANN, A. GUIHO, G. and KODRATOFF, I. (Eds.) (1984): *Automatic program construction techniques.* MacMillan. New York.

7. BLEDSOE, W.W. (1971): "Splitting and reduction heuristics in automatic theorem proving." *Artificial Intelligence.* 2; pags. 55-77.

8. BOHM, CORRADO (1980): "Logic and computer". En AGAZZI (1980) pgs. 297-309.

9. BOOLOS, GEORGE and JEFFREY, RICHARD (1982): *Computability and logic.* Cambridge University Press. Cambridge.

10. BRAFFORT, P. and HIRSCHBERG (Eds.) (1963): *Computer programming and formal systems.* North-Holland. Amsterdam.

11. CABRERA CALVO-SOTELO, JAIME (1984): "Prueba automática de teoremas: un panorama." *Teorema.* 14; pags. 475-496.

12. CHANG, C.L. and LEE, R.C. (1973): *Symbolic logic and mechanical theorem proving.* Academic Press.

13. CHARNIAK, E. and McDERMOTT, D. (1985): *Introduction to artificial intelligence.* Addison-Wesley. Reading, Mass.

14. DAVIS, M. (1983): "The prehistory and early history of automated deduction." Pags. 1-28 de [29].

15. GARRIDO, MANUEL (1978): *Lógica simbólica.* Tecnos. Madrid.

16. HEIJENOORT, JEAN van (1977): *From Frege to Gödel. A source book in mathematical logic, 1879-1931.* Harvard University Press. Cambridge, Mass.

17. HOGGER, C.J. (1984): *Introduction to logic programming.* Academic Press. London.

18. JEFFREY, RICHARD C. (1986): *Lógica formal: su alcance y sus límites.* EUNSA. Pamplona.

19. KALISH, DONALD and MONTAGUE, RICHARD (1980): *Logic. Techniques of formal reasoning.* Harcourt. New York.

20. KORFHAGE, R.R. (1966): *Logic and algorithms with applications to the computer and information sciences*. John Wiley. London.
21. KOWALSKI, R. (1979): *Logic for problem solving*. North-Holland. New York.
22. LOVELAND, DONALD W. (1978): *Automated theorem proving. A logical basis.* North-Holland. Amsterdam.
23. MOSTERIN, JESUS (1970): *Lógica de primer orden.* Ariel. Barcelona.
24. NILSSON, N.J. (1980): *Principles of artificial intelligence*. Tioga. Palo Alto.
25. POST, EMIL LEON (1921): "Introduction to a general theory of elementary propositions." En HEIJENOORT (1977), pags. 264-283.
26. PURTILL, RICHARD L. (1969): "Doing logic by computer." *Notre Dame Journal of Formal Logic.* 10; pags. 150-162.
27. ROBINSON, J.A. (1964): "On automatic deduction." *Rice University Studies*. 50; pags. 69-89.
28. ROBINSON, J.A. (1979): *Logic: form and function. The mechanization of deductive reasoning.* North-Holland. New York.
29. SIEKMANN, J.H. and WRIGHTSON, G. (Eds.) (1983): *The automation of reasoning: Collected Papers from 1957 to 1970, vol. I.* Springer Verlag. New York.
30. SIEKMANN, J.H. and WRIGHTSON, G. (Eds.) (1983): *The automation of reasoning: Collected Papers from 1957 to 1970, vol. II.* Springer Verlag. New York.
31. SMULLYAN, R.M. (1968): *First-order Logic*. Springer Verlag. Berlin-Heidelberg-New York.
32. WOS, L. OVERBEEK, R. LUSK, E. and BOYLE, J. (1984): *Automated reasoning. Introduction and applications.* Prentice-Hall. Englewood Cliffs.
33. WOSS, L. (1988): *Automated reasoning: 33 basic research problems*. Prentice Hall. Englewood Cliffs.

INDICE

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE CADIZ