Lógica Computacional

José Luis Fernández Vindel
Ángeles Manjarrés Riesco
Francisco Javier Díez Vegas

Dpto. Inteligencia Artificial
E.T.S.I. Informática
UNED
2007

Presentación

Estos apuntes se facilitan como material de estudio de la asignatura *Lógica Computacional*, impartida por los autores en cuarto curso de Ingeniería Informática en la UNED.

Los alumnos de este curso encontrarán, en su grupo de tutorización en la Red, otros documentos con ejemplos, actividades, ampliaciones o exámenes previos resueltos.

Los capítulos 1-2 y 5-6 han sido desarrollados por José Luis Fernández Vindel, el capítulo 3 por Ángeles Manjarrés Riesco y el capítulo 4 por Francisco Javier Díez Vegas, todos ellos profesores del Departamento de Inteligencia Artificial de la UNED.

Agradeceríamos sinceramente cualquier comunicación sobre erratas así como sugerencias sobre los contenidos y su exposición. En particular nos preocupa cómo mantener un tono expositivo amigable sin renunciar al rigor: hasta qué punto facilitar demostraciones formales (y dónde situarlas), cómo reubicar ejemplos y ejercicios para no entorpecer la exposición, etc.

Estas aportaciones se están canalizado a través del grupo de tutorización en la Red. La página `www.ia.uned.asignaturas/logica4` resume los detalles de planificación de este curso y facilita la dirección electrónica del grupo.

Los autores
UNED, Madrid, 1 de octubre de 2007

Sobre esta edición 2007

Esta edición, de Octubre de 2007, difiere de la utilizada el pasado curso 06-07 en dos puntos:

1. algunas actualizaciones menores en los apartados sobre el uso de los apuntes (presentación, localización de ejercicios, direcciones de la asignatura, etc.)

2. una variación sustancial en el capítulo 6: se ha sustituido íntegramente su contenido. El nuevo capítulo 6 corresponde al contenido del artículo sobre lógica temporal CTL que se facilitó separadamente el pasado curso.

Si el estudiante dispone de la versión impresa previa de estos apuntes y del citado artículo, no es necesario que imprima esta nueva versión. Cualquier duda puede consultarse en los Foros habilitados en el grupo de tutorización telemática.

Índice general

Índice de figuras

Índice de tablas

Introducción

Objetivos y contenidos

El objetivo fundamental de este libro es exponer **los métodos de la lógica** (concretamente de la lógica de predicados y de la lógica modal) que más se utilizan hoy en día en ciencias de la computación, inteligencia artificial e ingeniería del software.

Por ello, algunos de los temas son de índole puramente teórica, mientras que otros tratan de establecer el puente con las aplicaciones concretas. Así, la primera parte, que es de naturaleza teórica, estudia dos aspectos de la lógica de predicados: la lógica de proposiciones (tema 1) y la de predicados de primer orden (tema 2), mientras que la segunda parte se centra en dos formalismos lógicos aplicables a problemas de computación: el tema 3 estudia el fundamento de la programación lógica y el tema 4 la lógica de Hoare como formalismo para la verificación de programas secuenciales.

Análogamente, dentro de la tercera parte, dedicada a la lógica modal, el tema 5 (fundamentos de lógica modal) es puramente teórico, mientras que el tema **??** estudia la lógica modal temporal, que sirve, entre otras aplicaciones, para la verificación de sistemas concurrentes y de tiempo real (software) y de componentes físicos (hardware).

La figura 1 muestra la dependencia conceptual entre los temas tratados en el libro.

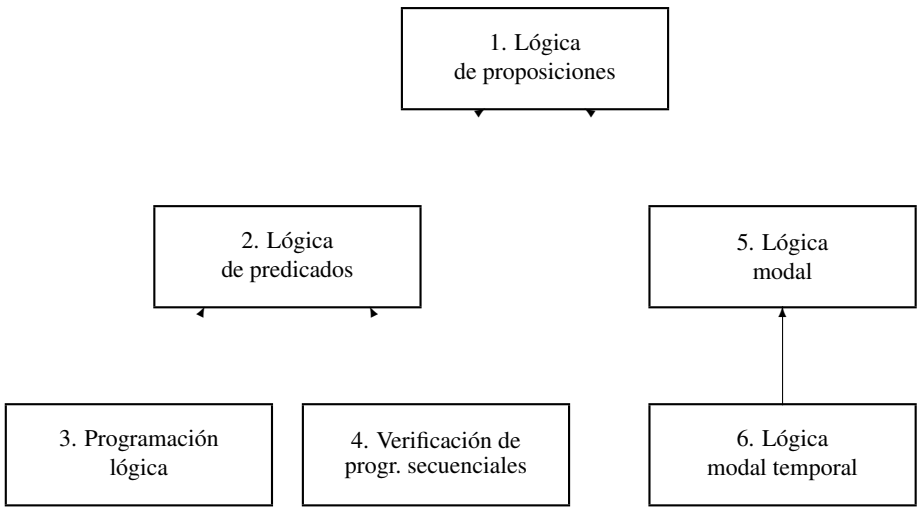


Figura 1: Dependencia conceptual entre los temas.

Conviene señalar, sin embargo, que el objeto de este libro no es ninguno de los campos de aplicación anteriores, sino los formalismos lógicos. Por eso no vamos a estudiar aquí los detalles de cada una de estas técnicas aplicadas, sino que insistiremos ante todo en el fundamento lógico común en

que se apoyan. En la bibliografía complementaria de cada uno de los temas encontrará el lector las referencias necesarias para conocer a fondo las técnicas que más le interesen.

Motivación para los alumnos de Ingeniería Informática

La lógica constituye la herramienta formal de razonamiento de la mayor parte de las asignaturas de la carrera de informática, sobre todo de las que están más relacionadas con **las matemáticas y la programación**, tales como *Álgebra*, *Análisis Matemático*, *Matemática Discreta*, *Electrónica Digital*, *Teoría de Autómatas y Lenguajes Formales*, *Programación* (incluida la *Programación Concurrente* y la *Programación Declarativa*), *Bases de Datos*, etc.

En cuanto a la **inteligencia artificial**, la lógica es el fundamento de todos los métodos de representación del conocimiento y del razonamiento, especialmente en sistemas expertos, razonamiento con incertidumbre (encadenamiento de reglas, lógica difusa, etc.), procesado del lenguaje natural, razonamiento espacial y temporal, visión artificial, robótica, lógica epistémica,[1] etc.

En tercer lugar, los **métodos formales** tienen una aplicación inmediata en la ingeniería del software. El uso de lenguajes de especificación formal es beneficioso en todos los desarrollos, ya que promueve la definición de modelos estructurados, concisos y precisos en diferentes niveles de abstracción, y facilita el razonamiento sobre ellos incluso a un nivel informal. Cuando a las notaciones formales se les asigna una semántica operacional, es posible diseñar herramientas automáticas que detecten ambigüedades en los requisitos iniciales, verifiquen y validen modelos a lo largo del ciclo de desarrollo, ayuden en la evolución y el mantenimiento de los productos y generen automática o semi-automáticamente prototipos o incluso partes del código final. Finalmente, los métodos formales sirven también para propósitos de documentación, ingeniería inversa y reutilización de componentes.

No obstante, aun siendo reconocidas las importantes contribuciones de los métodos formales a la ingeniería del software, su uso no se ha extendido tanto como cabía prever hace un par de décadas. Lo cierto es que los proyectos de desarrollo formal han fracasado con frecuencia, si bien por razones más ergonómicas que técnicas. La experiencia ha mostrado que introducir métodos formales en un proceso de desarrollo puede provocar dificultades no relacionadas con la potencia de los métodos. En primer lugar, pueden resultar excesivamente restrictivos, tanto porque obligan a sobreespecificar ciertos aspectos como porque constriñen el proceso de desarrollo, que tienen tendencia a monopolizar. En segundo lugar, su introducción puede ser problemática desde un punto de vista práctico, debido a su dificultad real y percibida, de forma que el coste inicial de entrenamiento en estas técnicas puede ser significativo, y en ocasiones los métodos son infrautilizadas incluso por parte de ingenieros instruidos en su uso.

El incremento en costes de desarrollo que los métodos formales suponen en las primeras fases del ciclo software se puede ver no obstante compensado por el decremento de los costes de las últimas fases, particularmente de la fase de validación, así como de las subsecuentes fases de mantenimiento y evolución. El problema fundamental de los métodos formales ha sido siempre que la comunidad científica dedicada al desarrollo métodos formales ha invertido un escaso esfuerzo en transferir la tecnología a los entornos de ingeniería. Las dificultades antes referidas podrían solventarse si se difundiesen pautas para seleccionar notaciones adecuadas para cada problema, evitar la sobre-formalización, identificar dominios adecuados, integrar las técnicas formales con técnicas semi-formales de uso extendi-

[1] La lógica epistémica es una rama de la lógica modal que sirve para razonar sobre el conocimiento (propio y ajeno), y se utiliza sobre todo para modelar el conocimiento de varios agentes. Hay que tener en cuenta que el enfoque más actual de la inteligencia artificial, tanto en el estudio teórico como en el desarrollo de aplicaciones, consiste en analizar y construir los sistemas inteligentes como *agentes* que interactúan entre sí (por ejemplo, cooperando o compitiendo, según el dominio de que se trate).

do, comercializar herramientas de desarrollo amigables, desarrollar técnicas específicas de estimación de costes de desarrollo, etc. Una buena estrategia para promover la integración de técnicas formales en un proceso de desarrollo es construir una base formal detrás de un lenguaje semi-formal bien conocido y difundido. Tal fue la estrategia seguida en el caso del exitoso lenguaje SDL en el campo de las telecomunicaciones, y es asimismo la estrategia seguida actualmente por algunos investigadores del lenguaje UML, con el propósito de introducir estas técnicas en los procesos de desarrollo orientados a objeto, actualmente tan en boga.

De los métodos formales se ha afirmado, irónicamente, que "siempre han tenido un gran futuro y siempre lo tendrán". No obstante, recientemente hay motivos razonables para confiar al fin en un lanzamiento definitivo de estas técnicas, particularmente en los cada vez más numerosos complejos dominios que exigen un alto grado de calidad del software, tales como: telecomunicaciones, transacciones bancarias, comercio electrónico, votaciones electrónicas, tarjetas inteligentes, dispositivos inalámbricos, etc.

En resumen, los avances realizados en la última década en este campo han hecho de los métodos formales de verificación una herramienta cada vez más utilizada, y es de prever que su uso se extienda significativamente en los próximos años.

Por eso, el alumno debe tener presente que, además del valor educativo de la lógica en la formación de una "mente ordenada", las técnicas estudiadas en este libro han llegado a ser una herramienta cada vez más utilizada en aplicaciones industriales en la última década; las empresas más importantes del mundo de la informática y las telecomunicaciones, como AT&T, BT, IBM, Intel, Motorola, Siemens, SRI, etc., cuentan con equipos especializados en este tema y los profesionales que conocen estas técnicas están cada vez más solicitados.

En este sentido, puede ser útil la lectura del artículo —disponible en Internet— en que Clarke y Wing [1996] enumeran algunas de las herramientas de verificación formal que se están utilizando en la actualidad. Como anécdota curiosa, este artículo menciona que el error que el primer procesador Pentium presentaba al efectuar ciertas divisiones, y que provocó unas pérdidas económicas considerables para Intel, se podría haber evitado con las técnicas disponibles en la actualidad; las referencias bibliográficas se encuentran en dicho artículo.

Entre las aplicaciones desarrolladas recientemente utilizando un proceso de desarrollo enteramente formal cabe destacar la automatización de la nueva línea de metro de París, METEOR, que ha obtenido una enorme publicidad. Esta línea funciona sin conductores y dejando un pequeño margen temporal entre trenes, de modo que los requisitos de seguridad requieren una alta confianza en el software.

En la misma línea, recomendamos también los sitios de Internet `www.afm.sbu.ac.uk/` y `www.cs.indiana.edu/formal-methods-education/`, donde el alumno podrá encontrar numerosos recursos sobre métodos formales aplicados a la especificación y verificación de sistemas (hardware y software), tales como herramientas gratuitas, abundante bibliografía e incluso ofertas de empleo.

Finalmente, aunque el alumno no llegue a ser un experto en verificación, el estudio de la lógica computacional puede aportar al alumno dos beneficios importantes: por un lado, le ayudará a formar una mente lógica, lo cual le servirá tanto para estudiar las asignaturas de esta carrera como para en su práctica profesional, especialmente en las labores de diseño y programación.

Bibliografía recomendada

Además de la *bibliografía complementaria* que hemos incluido en cada capítulo de este libro, queremos ofrecer una selección general de bibliografía que puede ser útil para el alumno que quiera ampliar los temas tratados en esta asignatura.

• *Logic in Computer Science* [Huth y Ryan, 2000]

Es un libro excelente, que explica la aplicación de las técnicas de la lógica computacional a problemas del mundo real. Lo recomendamos encarecidamente a los alumnos que quieran profundizar en los temas que tratamos en este texto y conocer otras técnicas nuevas. Aunque el nivel de complejidad es mayor que el de nuestro texto, se encuentra al alcance cualquier estudiante de pregrado que decida estudiarlo detenidamente. Recomendamos a quienes estén interesados que visiten la página web del libro, `http://www.cs.bham.ac.uk/research/lics/`, creada por sus autores; en ella podrán encontrar una fe de erratas, ejercicios de autoevaluación en forma de test (con soluciones explicadas), enlaces de interés, etc.

• *Mathematical Logic for Computer Science* [Ben-Ari, 2001]

Desde su aparición, en 1993, este libro se ha convertido en uno de los textos más conocidos de lógica matemática. La segunda edición, publicada en junio de 2001, amplía y actualiza sensiblemente los contenidos de la primera. Se trata de un texto para pregraduados, que intenta ser bastante didáctico, y en general lo consigue, aunque a nosotros nos gustaría que fuera más intuitivo en las explicaciones; por ejemplo, dando la "traducción" a lenguaje natural de las expresiones matemáticas, especialmente de los teoremas (lo hace en algunas ocasiones, pero en otras muchas no). Los ejemplos y ejercicios del libro están bien escogidos y constituyen un apoyo importante para el estudio.

La página web `http://stwww.weizmann.ac.il/g-cs/benari/books.htm#ml2`, creada por el autor, contiene una fe de erratas, enlaces muy interesantes sobre lógica matemática y el código fuente en PROLOG de algunos algoritmos para los formalismos lógicos (demostradores de teoremas, etc.), descritos en el libro, cuyo estudio puede resultar muy interesante para asentar los conceptos y los métodos expuestos en nuestro texto.

• *Handbook of Logic in Artificial Intelligence and Logic Programming* [Gabbay et al., 1993, 1994b, 1994c, 1995, 1998]

Es una colección de cinco volúmenes, en que cada capítulo está escrito por un autor diferente. El primer volumen se ocupa de los fundamentos lógicos (incluida la lógica modal básica y las cláusulas de Horn), el segundo de las metodologías de deducción, el tercero del razonamiento no-monótono, el cuarto de las lógicas epistémicas y temporales y el quinto de la programación lógica. Constituye una excelente obra de consulta, pero puede resultar demasiado compleja para alumnos pregraduados.

(No se debe confundir esta obra con el *Handbook of Logic in Computer Science* [Abramsky et al., 1992-2001]. Ambas constan de cinco volúmenes y están editadas por Oxford University Press, pero esta última es de una complejidad aún mayor, por lo que no la incluimos como bibliografía complementaria para nuestros alumnos.)

• *Introduction to Mathematical Logic* [Mendelson, 1997]

Uno de los mejores libros sobre lógica por la claridad de las explicaciones y la amplitud de los temas que cubre. Muchos de estos temas superan ampliamente el nivel de una asignatura de pregrado.

Para mayor información, en `http://aracne.usal.es/results/biblioes.html` hay una lista de más de 40 libros de lógica en inglés y en español. A través de esa página puede acceder también a recursos muy interesantes desarrollados dentro del Proyecto Aracne.

Parte I

LÓGICA DE PREDICADOS

Capítulo 1

LÓGICA DE PROPOSICIONES

Resumen

Se presenta el primero de los sistemas lógicos de este texto. Como en todos los restantes, se cubren tres etapas:

- *primero se facilita un lenguaje formal,*
- *después se define con precisión cómo evaluar el valor de verdad de una expresión del lenguaje*
- *y por último se facilitan sistemas deductivos para decidir, mediante cálculos, sobre propiedades o relaciones entre unas expresiones y otras (más precisamente, entre sus valores de verdad).*

Este capítulo tiene, por un lado, un interés intrínseco por sus aplicaciones en Matemáticas, Computación y en la formalización incipiente de razonamientos cotidianos, en lenguaje natural.

Adicionalmente ofrece un marco formal sencillo donde plantearse propiedades e interrelaciones sobre los valores de las expresiones (satisfacibilidad, validez, consecuencia, equivalencia), sobre sus métodos de decisión y sobre los sistemas deductivos creados para abordar estas cuestiones. Y lo que es más importante: tales conceptos son todos exportables a otros sistemas lógicos, con las restricciones propias de cada sistema.

Objetivos

1. *Respecto a la comprensión y uso del lenguaje formal:*
    - *es primordial ser capaz de generar, a mano, expresiones correctas (con todos los paréntesis que conllevan) y sus árboles sintácticos. Asimismo se debe poder analizar cualquier expresión correcta facilitada.*
    - *adicionalmente, se requiere la comprensión de la estructura inductiva del lenguaje con dos fines: primero porque facilita la automatización del lenguaje y segundo porque facilita el seguimiento de demostraciones sobre el propio lenguaje.*
2. *Respecto a la semántica, a la evaluación de los valores de verdad de una expresión*
    - *es básico ser capaz de evaluar, a mano, el valor de verdad de una expresión frente a una determinada interpretación; y ser capaz de listar todas las interpretaciones posibles y evaluar la expresión sobre cada una de ellas (es decir, de construir la tabla de verdad de la expresión).*

- *tan básico como lo anterior resulta comprender, trabajando sobre tablas de verdad, los conceptos semánticos que manejaremos permanentemente: satisfacción, satisfacibilidad, validez (tautologías), equivalencia. Es crítico tanto entender estos conceptos por separado como* empaparse de sus interrelaciones.
- *adicionalmente se puede considerar una primera automatización de los procesos de decisión de estas cuestiones, basada en el recorrido extensivo de la tabla de verdad hasta donde se requiera.*

3. *Sobre los sistemas deductivos:*

   - *es importante darse cuenta de que tanto la Resolución como las Tablas Analíticas se utilizan para confirmar que el conjunto de fórmulas de partida es insatisfacible. Y de cómo se utiliza esta confirmación para detectar relaciones de consecuencia.*
   - *sería deseable que el lector trabajara sobre al menos uno de los sistemas deductivos hasta sentirse cómodo desarrollando demostraciones en ese marco. No obstante, para problemas complejos se requiere en seguida la asistencia de un sistema automatizado de apoyo.*
   - *como objetivo más avanzado se propone la reflexión sobre la automatización de estos sistemas, principalmente de los basados en Resolución o en Tablas Analíticas*
   - *otro objetivo avanzado, éste muy formal, conlleva la comprensión de los teoremas de corrección y completud (que aún no facilita el texto) para entender cómo se engarzan y convergen los conceptos semánticos con 'esos cálculos sintácticos de los sistemas deductivos'. En particular, cómo confluyen el concepto de consecuencia lógica y el de derivación, prueba o demostración sintáctica.*

Metodología

De un texto lineal conviene saber qué debe leerse primero, qué se debe posponer y cómo contrastar el nivel de conocimiento en cada instante.

Utilice el apartado Objetivos como referencia: trate de abordar, por ese orden, los objetivos básicos de cada uno de los tres puntos analizados. En el estado actual de estas notas no es difícil: no hay muchos contenidos sobre automatización ni demostraciones formales.

Le recomendamos que aborde mínimamente el capítulo de sintaxis, hasta donde formalmente le resulte cómodo. Y que, con ese conocimiento intuitivo sobre la generación correcta de expresiones dedique tanto tiempo como requiera a la comprensión en profundidad de los conceptos semánticos básicos. Sólo entonces puede comenzar a trabajar sobre los sistemas deductivos.

Lamentamos que, salvo los ejemplos y figuras, no se incluyan todavía ejercicios suficientes.

1.1 Sintaxis

Definición 1.1 (Alfabeto) Un alfabeto es un *conjunto de símbolos*.

Ejemplo 1.2 Los dos siguientes conjuntos pueden considerarse alfabetos:

$$A_1 = \{\$, s, 5\} \qquad A_2 = \{p_1, p_2, p_3, p_4, \ldots\}$$

A_1 es un alfabeto finito, que consta de 3 símbolos. Por contra, A_2 puede aceptarse como un alfabeto infinito (si se considera cada p_j como un único carácter, distinto del resto).

Definición 1.3 (Expresión) Una expresión es una *secuencia finita* de símbolos.

Por *'expresión sobre el alfabeto A'* se entenderá una expresión compuesta exclusivamente de símbolos del alfabeto A. A los símbolos también se les denomina *caracteres* y a las expresiones, *cadenas*.

Ejemplo 1.4 Cada una de las cinco secuencias siguientes es una expresión sobre el alfabeto A_1:

$$5 \qquad ss \qquad \$\$s55s \qquad ss5 \qquad 5555ss\$\$$$

Observe que en una expresión se pueden repetir símbolos, pero no incluir símbolos ajenos: la cadena '$55ss4s$', que incluye el carácter '4', no contenido en A_1, no es una expresión sobre A_1.

> *S*obre un alfabeto finito, como A_1, ¿cuántas expresiones distintas pueden formarse?, ¿alguna de estas expresiones puede ser infinita?.

Observe que la definición de expresión exige que cada secuencia sea *finita*: no es una expresión $'5555\ldots'$, donde se espera que detrás de cada 5 siempre aparezca otro. Sin embargo, no se fija una longitud máxima: por muy grande que sea una expresión siempre puede construirse *otra* mayor adjuntando un nuevo carácter.

Así, puede generarse *un número infinito* de cadenas incluso con un alfabeto finito, como p. ej. $A_3 = \{a\}$ con un solo símbolo: $a, aa, aaa, aaaa, aaaaa, \ldots$ Al conjunto (siempre infinito) de *todas* las expresiones que pueden construirse sobre un alfabeto A se le suele notar como A^*.

Definición 1.5 (Lenguaje) Un lenguaje es un conjunto de expresiones. Más concretamente, un lenguaje sobre el alfabeto A es cualquier subconjunto de A^*.

Dado un alfabeto A, el conjunto total de cadenas A^* es un lenguaje sobre A, pero también lo es cualquier subconjunto suyo $L \subset A^*$ (vacío, finito o infinito).

Ejemplo 1.6 Sobre $A = \{\$, s, 5\}$ se pueden definir, entre otros, los lenguajes

$$L_1 = \{5, \$\$, ss, 55\} \qquad \text{ó} \qquad L_2 = \{s, ss, sss, \ldots\}$$

L_2 es un lenguaje con un número infinito de cadenas. Para definir este tipo de lenguajes se requiere precisar las propiedades que distinguen sus expresiones. Por ejemplo, para los lenguajes

$$L_3 = \{s, s\$, s5, \$s, 5s, ss, \ldots\} \qquad L_4 = \{s5, 5s, s\$5, \$s5, 5\$s, \ldots\}$$

L_3: 'todas las cadenas formadas al menos por una s' ; o L_4: 'todas las cadenas que incluyen igual número de símbolos s que de 5'.

1.1.1 El lenguaje de la Lógica de Proposiciones

El alfabeto de la Lógica de Proposiciones debe proporcionar los símbolos necesarios para representar proposiciones sobre el mundo. Como el número de proposiciones que pueden manejarse en un mismo razonamiento no está limitado, debe proveer un número infinito de letras proposicionales.

Definición 1.7 (Alfabeto de la Lógica de Proposiciones) El alfabeto A de la Lógica de Proposiciones consta de los siguientes elementos:

1. infinitas letras proposicionales: $p_0, p_1, p_2, p_3 \ldots$
2. símbolos lógicos: constantes $(\bot, \top)$, conectiva monaria $(\neg)$ y conectivas binarias $(\wedge, \vee, \rightarrow, \leftrightarrow)$
3. dos símbolos auxiliares de puntuación: paréntesis izquierdo'(' y derecho ')'

Así, $A = \{p_0, p_1, \ldots, \bot, \top, \neg, \wedge, \vee, \rightarrow, \leftrightarrow, (,)\}$

En las exposiciones teóricas, el número de letras proposicionales que se consideran simultáneamente es pequeño (por ejemplo, de p_0 a p_8). En estos caso se suelen notar informalmente con las últimas letras del alfabeto latino: $\{p, q, r, s, t, \ldots\}$. Seguiremos esta notación informal a partir de este punto.

Lectura y representaciones alternativas En la siguiente tabla se adjunta el nombre usual de cada conectiva y su lectura. También se añaden algunas formas alternativas de representación en otros textos. Ciertos lenguajes de programación pueden reservar símbolos distintos para estas mismas conectivas.

$\bot$,$\top$	'falso', 'verdadero'		$false, true$
$\neg p$	'no p'	negación	$\sim p$
$p \wedge q$	'p y q'	conjunción	$p\&q$
$p \vee q$	'p o q'	disyunción	
$p \rightarrow q$	'si p entonces q'	condicional	$p \supset q$, $p \Rightarrow q$
$p \leftrightarrow q$	'p si y sólo si q'	bicondicional	$p \equiv q$, $p \Leftrightarrow q$

Ejemplo 1.8 Las siguientes secuencias son expresiones sobre el alfabeto fijado:

(	$(pq$	$(\rightarrow \neg p)$	$(p \vee \wedge p)$	$(r \rightarrow\leftrightarrow p \vee q\neg)$
p	$(\neg p)$	$(p \rightarrow q)$	$((p \vee p) \wedge \bot)$	$((r \leftrightarrow \top) \wedge (r \rightarrow (\neg q)))$

Para nuestros fines, algunas de estas cadenas no son útiles. En este ejemplo, sólo aceptaríamos las de la segunda fila. A las expresiones aceptables se las denominará *fórmulas* o *expresiones bien formadas*. El conjunto (infinito) que incluye todas estas fórmulas (y nada más) es el *Lenguaje de la Lógica de Proposiciones*, que notaremos como *Form*.

Como introducción intuitiva le proponemos que estudie el proceso de generación de la fórmula (correcta, aceptable) $((p \wedge \top) \vee (\neg p))$. Se representa gráficamente en la siguiente figura, que debe recorrerse de abajo arriba.

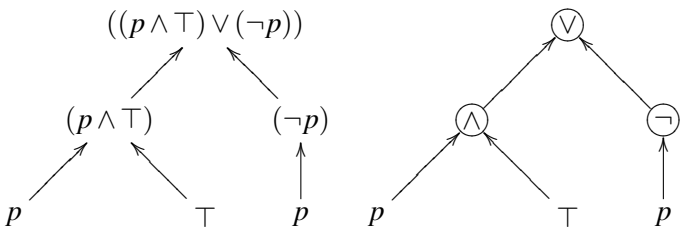


Figura 1.1: Árboles sintácticos: presentación

Observe que, partiendo de bloques mínimos, se van construyendo expresiones más complejas. Éstas, a su vez, siempre pueden ser reutilizadas como nuevos bloques componentes:

Los bloques mínimos aceptables, las fórmulas atómicas, están compuestas por un único caracter del alfabeto. Pero no cualquiera: sólo letra proposicional o constante.

Definición 1.9 (Fórmula atómica) Una fórmula atómica es una expresión compuesta exclusivamente por una letra proposicional, la constante $\bot$ o la constante $\top$.

El conjunto de todas las fórmulas, el lenguaje buscado, podría definirse (en primera aproximación) como el conjunto X que verificase:

1. todas las fórmulas atómicas pertenecen a X
2. si la expresión $\varphi \in X$ entonces $(\neg\varphi) \in X$
3. si las expresiones $\varphi, \psi \in X$ entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in X$

Un conjunto así garantiza que *todas* las fórmulas pertenecen al mismo. Pero puede además constar de otros elementos. En particular, el conjunto A^* formado por todas las cadenas generables a partir del alfabeto, verifica esas tres propiedades. Pero también las verifica el conjunto *'A^* menos la cadena* $(\rightarrow \wedge)$'.

Intuitivamente, si sólo se partiera de las fórmulas atómicas y se produjeran todas las negaciones y conexiones binarias posibles, debiera obtenerse un conjunto que contiene fórmulas y sólo fórmulas. Y estaría incluido en cada uno de los dos conjuntos antes citados (y, por tanto, en su intersección).

Definición 1.10 (Lenguaje de la Lógica de Proposiciones) El conjunto *Form*, que contiene todas las expresiones aceptables en nuestro lenguaje (y sólo éstas), es el menor de los conjuntos X que verifica:

1. todas las fórmulas atómicas pertenecen a X
2. si la expresión $\varphi \in X$ entonces $(\neg\varphi) \in X$
3. si las expresiones $\varphi, \psi \in X$ entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in X$

Esta familia de conjuntos X no es vacía: pertenece a la misma al menos el conjunto A^*. La intersección de todos ellos produce un conjunto que verifica también las tres propiedades requeridas. Y es el menor de todos, está incluido en cada uno. En particular, es el conjunto que incluye todas las expresiones que pueden generarse con estas tres reglas y exclusivamente tales expresiones.

Ejemplo 1.11 Las siguientes cuatro líneas muestran tanto fórmulas atómicas como compuestas con 1, 2 y 3 conectivas, respectivamente:

- $p, q, r, \bot, \top$
- $(\neg p), (\neg\bot), (\neg\top), (p \wedge q), (p \rightarrow \bot)$
- $(\neg(\neg p)), (\neg(\neg\top)), ((p \wedge q) \vee r), (p \rightarrow (\neg\bot))$
- $((\neg(p \wedge q)) \rightarrow p), ((p \rightarrow r) \rightarrow (\neg\bot))$

Trate de explicitar los pasos (las reglas utilizadas), desde las fórmulas atómicas, hasta la generaración de una fórmula del ejemplo. Puede resultarle útil considerar una representación como la de figura 1.2. A este tipo de gráficos se les denomina *árboles sintácticos*. Posponemos su definición formal.

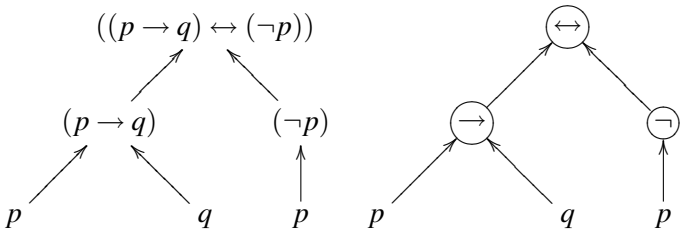


Figura 1.2: Árbol sintáctico: $((p \to q) \leftrightarrow (\neg p))$

Metalenguaje

A lo largo de estas notas encontrará expresiones como '$(\varphi \leftrightarrow \psi)$'. Ésta es una abreviatura de fórmula: la que se compone uniendo por un bicondicional dos fórmulas cualesquiera, denotadas como φ y ψ. Puesto que φ y ψ no forman parte del alfabeto, esta expresión *no es una fórmula* del lenguaje sino una frase para hablar *acerca del* lenguaje. A este lenguaje superior, utilizado para analizar otro, se le denomina *metalenguaje*. En concreto, en estos apuntes se utilizará el castellano más algunos símbolos adicionales como metalenguaje.

Algunas de las expresiones del metalenguaje serán meras abreviaturas de fórmulas, o esquemas de fórmulas: por '$(\varphi * \psi)$' se entenderá la fórmula construida por cualesquiera dos otras y una de las conectivas binarias. Observe que, como φ y ψ pueden ser cualesquiera fórmulas, no se descarta que puedan particularizarse en la misma fórmula. Otras expresiones del metalenguaje abreviarán frases más complejas, ya no fórmulas, expresando propiedades o relaciones entre fórmulas; por ejemplo, '$\phi \equiv \psi$' se leerá como *'las fórmulas ϕ y ψ son equivalentes'*.

En este texto se utilizarán letras griegas minúsculas para denotar fórmulas y letras griegas mayúsculas para denotar conjuntos de fórmulas. La siguiente tabla facilita su lectura:

A	α	alfa	I	ι	iota	P	ρ	ro
B	β	beta	K	κ	kappa	Σ	σ, ς	sigma
Γ	γ	gamma	Λ	λ	lambda	T	τ	tau
Δ	δ	delta	M	μ	mu	Υ	υ	ípsilon
E	ε	épsilon	N	ν	nu	Φ	ϕ, φ	fi
Z	ζ	dseta	Ξ	ξ	xi	X	χ	ji
H	η	eta	O	o	omicron	Ψ	ψ	psi
Θ	θ, ϑ	zeta	Π	π, ϖ	pi	Ω	ω	omega

Tabla 1.1: Alfabeto griego

1.1.2 Sobre la estructura inductiva del lenguaje

En la definición 1.10 de lenguaje, con tres reglas de composición se ha definido un conjunto de fórmulas, infinito pero muy estructurado. Este conjunto, por definición, tiene una estructura inductiva: nuevas fórmulas se construyen a partir de otras utilizando alguna de las cinco opciones de generación, alguna de las cinco conectivas fijadas.

Sobre un conjunto con esta estructura, las demostraciones y definiciones se producen por métodos inductivos y recursivos respectivamente. Así, en muy pocas líneas, se puede fijar una definición aplicable sobre cualquier fórmula, por muy compleja que sea. O demostrar que todas y cada una de las infinitas fórmulas verifican cierta propiedad.

Demostraciones inductivas

El lector puede estar familiarizado con el Principio de Inducción aritmético, donde sólo se requiere un paso inductivo: el que lleva de un número a su sucesor. Así, para demostrar una propiedad de los números naturales, basta demostrar que el primer número la verifica y que dado un número cualquiera también la verifica su sucesor.

Sobre el conjunto de fórmulas existen varias opciones de generación de fórmulas más complejas. Es preciso disponer de un retocado Principio de Inducción, que fije tanto el caso base como cada una de las alternativas inductivas.

Definición 1.12 (Principio de inducción estructural) Para demostrar que toda fórmula $\varphi \in \mathit{Form}$ tiene la propiedad P basta demostrar que:

1. Caso base: toda fórmula atómica tiene la propiedad P
2. Pasos inductivos:
   (a) si la expresión φ tiene la propiedad P, entonces $(\neg\varphi)$ tiene la propiedad P
   (b) si las expresiones φ y ψ tienen la propiedad P, entonces $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$ tienen la propiedad P

Sea $\mathcal{P}$ el conjunto de expresiones de A^* que tienen la propiedad P. Si se verifican los puntos 1 y 2, $\mathcal{P}$ contiene todas las fórmulas atómicas, la negación de cualquier expresión que esté en $\mathcal{P}$ y la composición binaria de cualesquiera dos expresiones pertenecientes a $\mathcal{P}$.

Así, el conjunto $\mathcal{P}$ es uno de aquellos conjuntos X que verificaban las propiedades exigidas en la definición 1.10 del Lenguaje de la Lógica de Proposiciones. Es decir, $\mathit{Form} \subset \mathcal{P}$: todas las fórmulas verifican la propiedad P (además de, quizá, otras expresiones).

Ejercicio 1.13 Demuestre, mediante inducción, que toda fórmula tiene un número par de paréntesis.

Entre otras propiedades demostrables, resaltaremos una en este punto: si se facilita una expresión, supuesto que sea una fórmula, sólo existe un medio de descomponerla sintácticamente.

Teorema 1.14 (Análisis sintáctico único) Cada fórmula proposicional φ pertenece a una y sólo una de las siguientes categorias:

1. φ es atómica
2. φ es de la forma $(\neg\psi)$, para ψ fórmula única
3. φ es de la forma $(\psi * \chi)$, para una determinada conectiva y para ψ y χ fórmulas únicas

La descomposición de una fórmula no atómica φ en una o dos componentes produce sus *subfórmulas inmediatas*. La conectiva que se aplicaba sobre ella o ellas es la *conectiva principal* de φ.

Ejemplo 1.15 La fórmula

$$((p \to (\neg q)) \wedge (p \vee r))$$

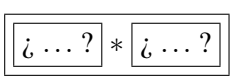
es una conjunción. Es decir, es de la forma $(\psi \wedge \chi)$ con

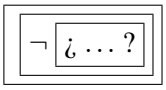
$$\psi = (p \to (\neg q)), \qquad \chi = (p \vee r)$$

Su conectiva principal es $\wedge$ y sus subfórmulas inmediatas son esas ψ y χ.

Ejercicio 1.16 Trate de encontrar un algoritmo que determine la conectiva principal de cualquier fórmula, contabilizando los paréntesis abiertos y cerrados.

Observe que, de acuerdo al teorema 1.14 previo, cualquier fórmula (por muy extensa que sea) presenta la siguiente estructura:





p

Definiciones recursivas

En el ejercicio 1.13 se utilizaba el concepto intuitivo de *número de paréntesis*. Hay que suponer que estamos refiriéndonos a una función del conjunto de fórmulas en el conjunto de números naturales $num_{par}(\varphi) = n$. ¿Cómo puede definirse rigurosamente esta función? Observe que un proceso como 'cuente usted ...' no nos sirve como definición.

Si las fórmulas hablasen, una fórmula conjuntiva como $(\varphi \wedge \psi)$ diría: 'yo sólo sé que tengo 2 paréntesis más que mis componentes inmediatas'. De esta forma, el número que se quiere calcular supone una operación sobre el número que 'aportan' las componentes. Y sobre el que aportan las componentes de las componentes de éstas, recursivamente hasta llegar al caso base. Para las fórmulas atómicas su aportación debe estar fijada.

Más precisamente, la función $num_{par} : Form \mapsto N$ buscada se define como:

$$num_{par}(\varphi) = \begin{cases} 2 + num_{par}(\psi) + num_{par}(\chi) & ,\varphi = (\psi * \chi) \\ 2 + num_{par}(\psi) & ,\varphi = (\neg\psi) \\ 0 & ,\varphi \text{ atómica} \end{cases}$$

Ejemplo 1.17 La función $num_{\leftrightarrow} : Form \mapsto N$ hace corresponder a cada fórmula el número total de sus conectivas bicondicionales. Se define como:

$$num_{\leftrightarrow}(\varphi) = \begin{cases} 1 + num_{\leftrightarrow}(\psi) + num_{\leftrightarrow}(\chi) & ,\varphi = (\psi * \chi,) \text{ con } * = \leftrightarrow \\ 0 + num_{\leftrightarrow}(\psi) + num_{\leftrightarrow}(\chi) & ,\varphi = (\psi * \chi), \text{ con } * \neq \leftrightarrow \\ 0 + num_{\leftrightarrow}(\psi) & ,\varphi = (\neg\psi) \\ 0 & ,\varphi \text{ atómica} \end{cases}$$

Observe que, aunque de forma compacta, se fija la imagen tanto para las fórmulas atómicas como para cada una de las 5 conectivas. En algunos casos, la imagen para alguna de estas alternativas puede coincidir.

En el ejemplo 1.17, el número de bicondicionales es una función $num_{\leftrightarrow} : Form \mapsto N$, del conjunto de fórmulas en el conjunto de números naturales. En general, las definiciones sobre fórmulas resultarán ser una función de *Form* en algún conjunto, aunque no necesariamente sobre N.

Ejemplo 1.18 Definiremos una función *color*, de *Form* en el conjunto *Semáforo={verde,ámbar,rojo}*, que abreviaremos como conjunto $S = \{v, a, r\}$.

El primer objetivo es fijar el *color* de cualquier fórmula atómica. Definamos una función $color_{at}$ sólo desde el conjunto de fórmulas atómicas sobre el conjunto *Semáforo*:

$$\begin{array}{ccc} color_{at}(p_0) = v & color_{at}(p_1) = a & color_{at}(p_2) = r \\ color_{at}(p_3) = v & color_{at}(p_4) = a & color_{at}(p_5) = r \\ \multicolumn{3}{c}{\ldots} \\ color_{at}(p_{0+3j}) = v & color_{at}(p_{1+3j}) = a & color_{at}(p_{2+3j}) = r \end{array}$$

con $color_{at}(\top) = v$, $color_{at}(\bot) = r$. En este punto, pueden empezarse a valorar los nodos terminales (inferiores) de una figura como la 1.3. Y continuar la valoración hacia el nodo raíz.

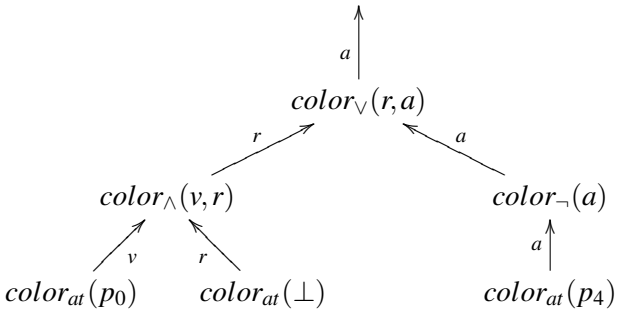


Figura 1.3: Los colores de las fórmulas

Para ello es preciso definir qué color 'sale' de un nodo $\neg$ (negación), según el color que 'entre'. Y qué color 'sale' de cada conjuntiva binaria, según qué par de colores entren. Para el ejemplo de la figura: $color_{\neg}(a) = a$, $color_{\wedge}(v,r) = r$, $color_{\vee}(r,a) = a$. Así,

$$color(\ ((p_0 \wedge \bot) \vee p_4)\ ) = a$$

Observe que, para definir la función $color : Form \mapsto S$ se requiere definir las funciones auxiliares:

$color_{at} : Form\_Atom \mapsto S$ $\quad$ $color_{\neg} : S \mapsto S$ $\quad$ $color_{*} : S^2 \mapsto S$ para cada conectiva binaria

Teorema 1.19 (Principio de Recursión Estructural) Dadas las funciones

$f_{at} : Form\_Atom \mapsto X$ $\quad$ $f_{\neg} : X \mapsto X$ $\quad$ $f_{*} : X^2 \mapsto X$ para cada conectiva binaria

existe una única función $F : Form \mapsto X$ tal que:

$$F(\varphi) = \begin{cases} f_*(\ F(\psi), F(\chi)\ ) & \varphi = \psi * \chi \\ f_\neg(\ F(\psi)\ ) & \varphi = \neg\psi \\ f_{at}(\varphi) & \varphi \in Form\_Atom \end{cases}$$

El Principio de Recursión estructural garantiza que, para una determinada elección de las funciones previas, la función resultante es única y está bien definida.

1.1.3 Derivación de conceptos sintácticos

Con ayuda del Principio de Recursión Estructural se pueden definir otros conceptos sintácticos. En este apartado abordaremos la definición de rango, número de símbolos, apariciones de un determinado símbolo, la definición de árbol sintáctico y la de conjunto de subfórmulas de una fórmula dada.

Rango

La función $rango : Form \mapsto N$, intuitivamente, proporciona la longitud de la mayor rama del árbol sintáctico de la fórmula. Se define:

$$rango(\varphi) = \begin{cases} 1 + max(rango(\psi), rango(\chi)) & , \varphi = (\psi * \chi) \\ 1 + rango(\psi) & , \varphi = (\neg\psi) \\ 0 & , \varphi \text{ atómica} \end{cases}$$

Dejamos al lector la definición de otros conceptos similares, como *número de símbolos* o *número de apariciones de conectivas* (o de algunas, en particular).

Ejercicio 1.20 Construya, exclusivamente con 7 conectivas binarias, una fórmula con el mayor rango posible y otra con el menor.

Dada una fórmula, reflexione sobre cómo su rango, número de símbolos o apariciones de conectivas se acotan entre sí.

El lector puede encontrar en la literatura demostraciones inductivas sobre el lenguaje que no utilizan el Principio de Inducción Estructural. Son demostraciones que emplean el conocido Principio de Inducción Completa sobre números naturales. En este caso, el número natural asignado a cada fórmula es su rango o su número de símbolos:

Teorema 1.21 (Principio de Inducción Completa) Si para cualquier fórmula φ se demuestra que:

todas las fórmulas con rango (o número de símbolos) menor que φ *tienen la propiedad P*

entonces todas las fórmulas tienen la propiedad *P*.

La demostración de que ambos principios de inducción son equivalentes no se aborda en este texto.

Árboles sintácticos

Se ha estado utilizando el concepto de árbol sintáctico sin una definición. En la figura 1.4 se representan, respectivamente, las definiciones de $árbol(\varphi * \psi)$, $árbol(\neg\psi)$ y $árbol(\varphi)$ para φ atómica.

Puede precisar y formalizar más esta definición si utiliza como conjunto imagen el de las representaciones matemáticas del concepto de árbol, o el de alguna estructura computacional que lo represente.

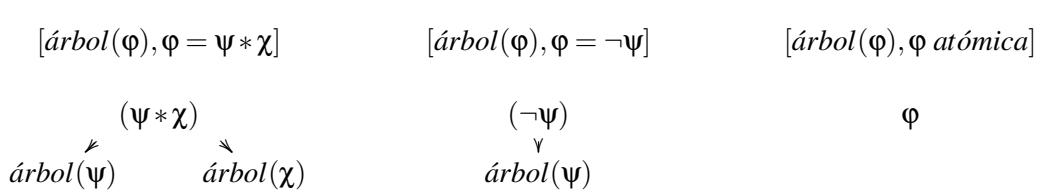


Figura 1.4: Definición recursiva de árbol sintáctico

Subfórmulas

Definición 1.22 (Subfórmulas) Dada una fórmula φ, el conjunto de *todas* sus subfórmulas se define recursivamente como:

$$subform(\varphi) = \begin{cases} \{\varphi\} \cup subform(\psi) \cup subform(\chi) & ,\varphi = (\psi * \chi) \\ \{\varphi\} \cup subform(\psi) & ,\varphi = (\neg\psi) \\ \{\varphi\} & ,\varphi \text{ atómica} \end{cases}$$

Observe que $subform : Form \mapsto \mathcal{P}(Form)$ es una función del conjunto de fórmulas en el conjunto de subconjuntos de fórmulas.

Ejemplo 1.23 El conjunto de subfórmulas de

$$\varphi = ((p \wedge r) \rightarrow (q \vee (\neg t))$$

es

$$subform(\phi) = \{((p \wedge r) \rightarrow (q \vee (\neg t)), (p \wedge r), (q \vee (\neg t)), p, r, q, (\neg t), t\}$$

Sustitución uniforme

La sustitución uniforme permite escribir una fórmula a partir de otra. Será una operación sintáctica de extraordinaria utilidad en las secciones siguientes. Intuitivamente, se puede visualizar contemplando los árboles de la figura 1.5.

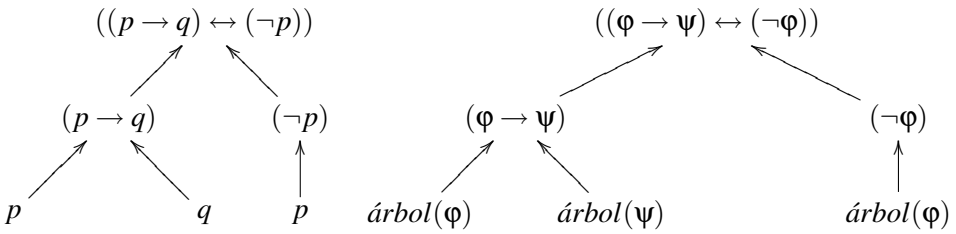


Figura 1.5: Sustitución uniforme $[p/\varphi, q/\psi]$ sobre un árbol sintáctico

Ejemplo 1.24 Considere la fórmula

$$\chi := ((p \rightarrow q) \vee p)$$

y sustituya todas la apariciones de:

$$p, \text{ por la fórmula } \varphi := (r \wedge s) \qquad q, \text{ por la fórmula } \psi := (t \rightarrow u)$$

el resultado es la fórmula

$$(\underbrace{(r \wedge s)}_{\varphi} \rightarrow \underbrace{(t \rightarrow u)}_{\psi}) \vee \underbrace{(r \wedge s)}_{\varphi}$$

donde las marcas de subrayado se han añadido ocasionalmente para facilitar la correspondencia.

Cuidado Observe que la sustitución se produce uniformemente, por igual, en todas y cada una de las apariciones de la letra proposicional sustituida.

Sobre la misma fórmula χ de partida, se podían haber sustituido todas las instancias de:

$$p, \text{ por la fórmula } \varphi := q \qquad q, \text{ por la fórmula } \psi := (q \to \neg p)$$

el resultado sería entonces la fórmula

$$(\underbrace{q}_{\varphi} \to \underbrace{(q \to \neg p)}_{\psi}) \vee \underbrace{q}_{\varphi})$$

Cuidado Observe que la sustitución no vuelve a aplicarse sobre instancias nuevamente aparecidas de la letra proposicional. En $[p/q, q/(q \to (\neg p))]$ se produce las sustituciones atómicas simultáneamente: *no se 'convierten' primero* las p en q y, luego, todas las q (las primitivas y las recién aparecidas) en $(q \to (\neg p))$.

Algo más formalmente, dada una fórmula φ su sustitución $(\varphi)^\sigma$ es una fórmula. Y esta función $()^\sigma : \mathit{Form} \mapsto \mathit{Form}$ se puede definir recursivamente.

Definición 1.25 (Instancia, por sustitución, de una fórmula) Considere una fórmula φ y una función $\sigma_{atóm} : \mathit{Form\_Atóm} \mapsto \mathit{Form}$ que define la sustitución para cada fórmula atómica . Entonces, la sustitución uniforme $(\varphi)^\sigma$ se define como:

$$(\varphi)^\sigma = \begin{cases} (\ (\psi)^\sigma * (\chi)^\sigma\ ) & ,\varphi = \psi * \chi \\ (\ \neg(\psi)^\sigma\ ) & ,\varphi = \neg\psi \\ \sigma_{atóm}(\varphi) & ,\varphi \text{ atómica} \end{cases}$$

De la sustitución atómica $\sigma_{atóm}$ se requiere que $\sigma_{atóm}(\bot) = \bot$ y que $\sigma_{atóm}(\top) = \top$. Cada $\sigma_{atóm}$ fija una única sustitución uniforme $()^\sigma$. La fórmula $(\varphi)^\sigma$ es la instancia de φ por la sustitución σ.

Así, si una sustitución se aplica sobre una fórmula, por ejemplo, conjuntiva, el resultado será la conjunción de las transformadas. Si se aplica sobre una negación, el resultado es la negación de la transformada. Cuando se llega a una letra proposicional, se cambia ésta por su fórmula sustituyente.

Ejemplo 1.26 Sea $\sigma_{atóm} : \mathit{Form\_Atóm} \mapsto \mathit{Form}$ tal que:

$$\begin{array}{ll} \sigma_{atóm}(\bot) = \bot & \sigma_{atóm}(\top) = \top \\ \sigma_{atóm}(p_0) = (p_1 \vee p_0) & \sigma_{atóm}(p_1) = p_3 \\ \sigma_{atóm}(p_k) = p_k \text{ para toda letra proposicional } p_k \text{ con } k \neq 0,1 & \end{array}$$

Entonces está definida la transformada $(\varphi)^\sigma$ de cualquier fórmula φ, por ejemplo $\varphi := (p_0 \to \neg p_1)$

$$\begin{array}{lllllll} (\varphi)^\sigma & = & (p_0 \to \neg p_1)^\sigma & = & ((p_0)^\sigma \to (\neg p_1)^\sigma) & = & \\ & = & ((p_0)^\sigma \to (\neg(p_1)^\sigma)) & = & (\sigma_{atóm}(p_0) \to (\neg\sigma_{atóm}(p_1))) & = & ((p_1 \vee p_0) \to (\neg p_3)) \end{array}$$

Ejercicio 1.27 Con la sustitución definida en el ejemplo previo, calcule:

$$((p_0 \wedge p_1) \to (\bot \vee p_3))^\sigma$$

Cuando en una sustitución atómica, como $\sigma_{atóm} = [p/(q \wedge r), q/(t \vee q)]$, no se facilite la sustitución para todas (las infinitas) letras proposicionales, se entenderá que $\sigma_{atóm}(p_k) = p_k$ para esas letras.

Aunque no fuera así, suelen precisarse sólo las sustituciones atómicas de las letras incluidas en las fórmulas analizadas en ese momento. Si dos sustituciones coinciden sobre esas letras (aunque difieran en alguna de las infinitas restantes), producen las mismas transformaciones sobre esas fórmulas consideradas.

1.1.4 Eliminación de paréntesis

Considere el árbol sintáctico de la figura (fig. 1.6). Cada nodo es una de las subfórmulas utilizadas en su generación. Y una fórmula no es más que la expresión lineal de esta estructura.

Con todos sus paréntesis (y correctamente situados) a cada fórmula le corresponde un único árbol sintáctico. Y a cada árbol una única fórmula. Si se omiten los paréntesis, a una fórmula le pueden corresponder varios árboles: resulta ambigua. Por ejemplo,

$$p \wedge q \vee \neg p$$

puede tener por conectiva principal una conjunción o una negación. Trate de dibujar sus posibles árboles sintácticos.

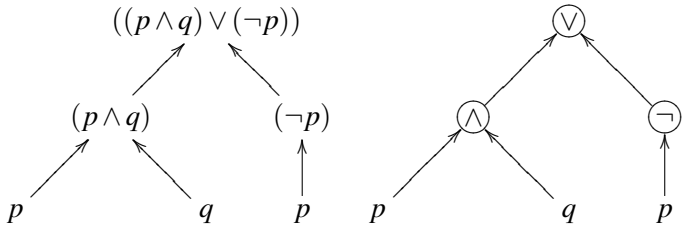


Figura 1.6: Árboles y paréntesis

Además de los paréntesis (cuya anidación replica la del arbol) existen otras dos maneras de linealizar un árbol sintáctico: utilizando notación prefija o conviniendo un orden de precedencia entre conectivas.

Notación prefija

Una operación aritmética binaria como $(3+5)$ puede representarse así (en notación infija) o $(+\ 3\ 5)$ (en notación prefija). También se puede considerar la notación posfija.

Así, una fórmula como

$$((p \wedge q) \vee (\neg p))$$

se puede representar en notación prefija como

$$\vee \overbrace{\wedge \underline{p}\,\underline{q}}\ \overbrace{\neg \underline{p}}$$

Donde las llaves y líneas se han añadido para facilitar la lectura. Observe que corresponde a un recorrido del árbol en profundidad. Y que la fórmula, ya sin paréntesis, no resulta ambigua.

A la notación prefija también se la conoce como *notación polaca*.

Precedencia

Para interpretar una expresión aritmética como $3 \cdot 5 + 7 \cdot 4$, ambigua, se suele suponer que existe un convenio entre el escritor y el lector que deshace esta ambiguedad. El convenio más generalizado permite interpretar esa expresión como $((3 \cdot 5) + (7 \cdot 4))$. Se suele explicar informalmente este convenio fijando que, en caso de ambiguedad, 'las multiplicaciones se consideran antes que las sumas'.

Entre los operadores lógicos, en este mismo tono informal, se suelen considerar primero las negaciones, luego conjunciones y disyunciones, y después condicionales y bicondicionales. Como quiera que es un convenio, trate de determinar con exactitud qué precedencia utiliza el autor del texto que esté leyendo.

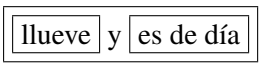
1.2 Semántica

1.2.1 Introducción

Los sistemas lógicos no son meros ejercicios formales. Se utilizan para *representar* declaraciones sobre el mundo y *operar* sobre estas representaciones.

Representación de declaraciones Cada declaración, cada expresión declarativa, enuncia un estado de cosas. 'LLueve' es una expresión declarativa. Una orden o una pregunta no lo son. De una expresión declarativa se puede juzgar cuánto de verdadera es.

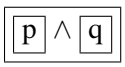
La sintaxis del lenguaje natural, con el que nos comunicamos diariamente, permite construir expresiones declarativas a partir de otras. Así, en una frase como 'llueve y es de día' se distinguen 3 expresiones: 'llueve', 'es de día' y 'llueve-y-es de día'.



Los sistemas lógicos proporcionan un lenguaje formal. Y sobre este lenguaje se intenta representar las declaraciones del lenguaje natural. Como el lenguaje natural es mucho más rico en matices, excepciones, ambigüedades y contextos, este proceso de representación no es fácil. Ciertamente, algunos lenguajes formales son más expresivos que otros, permiten captar más detalles del lenguaje natural. Si estos detalles son significativos para describir un nuevo tipo de razonamiento, bienvenidos. Si no, podíamos habernos quedado con el lenguaje formal previo, más rígido.

La lógica de proposiciones facilita un lenguaje muy pobre. Sin embargo, hay procesos de razonamiento utilizados diariamente que pueden ser formalizados, captados convenientemente, sobre este lenguaje. Para formalizar otros procesos habrá que definir nuevos sistemas lógicos.

En el lenguaje proposicional, el ejemplo previo sobre la lluvia y el estado diurno, se representaría como:



Observe que, de nuevo, se pueden considerar 3 fórmulas: p, q y $p \wedge q$.

Valores de verdad de una declaración Volvamos al lenguaje natural, a las 3 declaraciones detectadas en 'llueve y es de día'. Sobre cada una de ellas se puede considerar su grado de verdad, pero no de forma totalmente independiente. Una vez que se ha fijado el valor de verdad de las componentes, el de la expresión compuesta está determinado: es función de los anteriores. Para este ejemplo, si sólo una componente fuese verdadera, el uso habitual de la conjunción 'y' nos obliga a aceptar que la frase compuesta no lo es. Otra cosa sería si estuvieran unidas por una disyunción 'o'.

La lógica de proposiciones trata de captar estas dependencias entre valores de verdad: una dependencia distinta para cada conectiva, que debe precisar qué valor de verdad tiene la expresión compuesta para cada combinación de valores de las componentes.

Los posibles valores de las componentes hay que fijarlos. Si se admite que una declaración puede ser 'más bien verdadera', o 'un 80% falsa', la función asociada a cada conectiva debiera definirse para todos estos casos. Se estaría entonces en el marco de lógicas polivaluadas o lógicas borrosas.

En todos los sistemas lógicos abordados este curso se considerarán sólo dos valores de verdad. Así, una expresión es verdadera o falsa, necesariamente sólo una de las dos opciones: si se afirma que no es una de ellas, entonces tiene el otro valor de verdad.

Sobre la utilidad del formalismo A un lector técnico puede (sólo puede) que no le motive un ejemplo sobre lluvia, luz y olor a campo. Considere este otro:

- Si (<encendido> y <configurado> y <conectado>) entonces <accedo servidor>
- Si <luce piloto> entonces <encendido>
- Si <icono parpadea> entonces <conectado>
- <luce piloto> e <icono parpadea> y no <accedo servidor>

Consta de cuatro expresiones declarativas compuestas, con componentes comunes. Que se hayan enunciado no supone que se acepten como verdaderas, todas o algunas. En algunos textos se podría encontrar, previo a su enunciado, la frase 'suponga que son ciertas todas las siguientes expresiones'. En otras situaciones, sí que al enunciarlas se acepta implícitamente que se suponen todas ellas verdaderas. No será nuestro caso.

Considere el valor de cada componente. Si se opta por pensar que <luce piloto> es verdad, lo aceptaremos en las cuatro expresiones. Igualmente se puede 'tomar partido' ante el resto de las componentes básicas. Se tendrá entonces una *interpretación* común del estado de las cosas. Para esa interpretación, resultará que cada una de las cuatro declaraciones compuestas tendrán definido su valor de verdad, (función del de las componentes y de las conectivas usadas). En este punto es posible plantearse algunas cuestiones:

- *Satisfacibilidad:* efectivamente, de todas las interpretaciones posibles, alguna hay que consigue que esas cuatro declaraciones compuestas resulten simultáneamente verdaderas. No obstante, si se añade como quinta declaración <configurado>, no se encontrará interpretación alguna que haga verdaderas a estas cinco declaraciones: resultará un conjunto insatisfacible de expresiones.
- *Consecuencia:* en todas las interpretaciones en que las cuatro declaraciones sean verdaderas (en todas, todas), resultará que la expresión 'no <configurado>' se evaluaría como verdadera. Esta expresión es consecuencia lógica de las cuatro anteriores. Vuelva a leer las cuatro declaraciones del ejemplo, suponiendo que efectivamente son verdaderas, y trate de llegar a esta conclusión.
- *Equivalencia:* si se compara la declaración 'Si <luce piloto> entonces <encendido>' con esta otra 'Si no <encendido> entonces no <luce piloto>', resultará que presentan una relación curiosa: toda interpretación que hace a una de ellas verdadera también lo hace a la otra, y lo mismo ocurre con las interpretaciones que las hacen falsas. Son dos formas distintas de expresar lo mismo.
- *Validez:* es posible construir expresiones que sean verdaderas en toda interpretación, análogamente a cómo, en el lenguaje de la aritmética, $x+3+(-x)=3$ no importa cuál sea el valor de x. En nuestro ejemplo, sería posible contruir una única expresión de este estilo con las cuatro declaraciones y una quinta: 'no <configurado>'.

Como se puede intuir, todas estas cuestiones están interrelacionadas. Para expresar formalmente esta interrelación es necesario definir cada uno de los conceptos utilizados. Todo ello se produce sobre la semántica de la lógica de proposiciones: la manera precisa de evaluar una expresión en función de los valores de sus componentes.

1.2.2 Valores de verdad de fórmulas atómicas

Sólo se considerarán dos posibles valores de verdad: verdadero o falso. Se denotarán preferentemente como 1 (verdadero) y 0 (falso). Alternativamente, se utilizará la notación v (verdadero) y f (falso).

Definición 1.28 (Asignación) Una asignación $\nu_{atom} : Form\_Atom \mapsto \{0,1\}$ es una función del conjunto de fórmulas atómicas en el conjunto de valores de verdad, con $\nu_{atom}(\bot) = 0$ y $\nu_{atom}(\top) = 1$.

Ejemplo 1.29 Para fijar ideas, si nuestro alfabeto contuviera sólo 3 letras proposicionales, una asignación haría corresponder a cada una de estas letras su valor de verdad: bien 0 (falso), bien 1 (verdadero). En cuanto a las constantes, toda asignación debe hacer corresponder 0 a $\bot$ y 1 a $\top$.

En este caso, la siguiente correspondencia ν_{atom} es una asignación:

$$\nu_{atom}(p) = 0, \nu_{atom}(q) = 1, \nu_{atom}(r) = 0; \nu_{atom}(\bot) = 0, \nu_{atom}(\top) = 1$$

Se pueden definir, en este conjunto de 3 letras, hasta 8 asignaciones distintas. En la tabla (tabl. 1.2) se enumeran todas: cada línea es una asignación distinta. La inferior fija que las 3 proposiciones son falsas y la superior que son verdaderas. Si se consideran las filas como codificación en binario de un número, es fácil generarlas todas.

En general, para n variables, hay 2^n asignaciones distintas.

		p	q	r
7	$\rightsquigarrow$	1	1	1
6	$\rightsquigarrow$	1	1	0
5	$\rightsquigarrow$	1	0	1
4	$\rightsquigarrow$	1	0	0
3	$\rightsquigarrow$	0	1	1
2	$\rightsquigarrow$	0	1	0
1	$\rightsquigarrow$	0	0	1
0	$\rightsquigarrow$	0	0	0

Tabla 1.2: 2^3 asignaciones distintas sobre 3 letras proposicionales

1.2.3 Semántica de las conectivas

El valor de verdad de una fórmula compuesta debe depender sólo del valor de sus subfórmulas inmediatas y de la conectiva que las une. En el caso más simple, $(\neg p)$ será verdadera si p es falsa y será falsa si p es verdadera.

La tabla (tabl. 1.3) reune la función asociada a cada conectiva binaria, en definitiva, la función que la caracteriza y la distingue de otras. Así, $(p \rightarrow q)$ tendrá como valor de verdad 0 si p es verdadera y q falsa.

Observe que $\varphi = (q \rightarrow p)$ tendrá como valor 1 si p es verdadera y q falsa. En la tabla, p se refiere a la subfórmula izquierda y q a la derecha y lo que se expresa es 'si la subfórmula izquierda es falsa (0) y la derecha verdadera (1), entonces la fórmula es verdadera (1). Exactamente lo que se fijó para φ.

Lo que distingue una conectiva de otra es exclusivamente su función. Si en otro texto se define una conectiva '&' con, exactamente, la misma función que $\wedge$, considere que se trata del mismo concepto de

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

Tabla 1.3: Semántica de las conectivas binarias

conjunción, denotado de otra forma. Sobre estas cuatro conectivas binarias conviene resaltar algunos detalles:

- *Conjunción ($\wedge$):* formalmente, sólo es *verdadera en un caso*, cuando todas las subfórmulas inmediatas lo son. Representa el concepto de conjunción del lenguaje natural.

- *Disyunción ($\vee$):* formalmente, sólo es *falsa en un caso*, cuando todas las subfórmulas inmediatas lo son. Representa el concepto de disyunción no exclusiva del lenguaje natural: cuando se acepta una cosa, la otra o ambas. Una frase como 'vienes o te quedas' utiliza una disyunción exclusiva que representa más fielmente otra conectiva.

- *Condicional ($\rightarrow$):* en expresiones condicionales, a la subfórmula izquierda se le denomina *antecedente* y a la derecha, *consecuente*. Un condicional sólo es *falso en un caso*, cuando el antecedente es verdadero y el consecuente es falso. Capta, en parte, el uso que se hace de la implicación en lenguaje natural: efectivamente, si el antecedente es verdadero y el consecuente (lo implicado) no lo es, no se admite que exista, que sea verdadera, esa relación de implicación.

  Sin embargo, la implicación en lenguaje natural suele llevar asociada una causalidad física: 'si lo sueltas, se cae'. Hay algo en la estructura interna del antecedente o en lo que expresa sobre el mundo que obliga al consecuente a ser verdad, cuando el antecedente lo es. Nada de eso es formalmente expresable sobre un condicional proposicional.

  No se suelen establecer relaciones de implicación, en lenguaje natural, usando antecedentes que son falsos: 'si 8 es un número primo entonces ...' Nuestro condicional ha resaltado, muy útilmente, un único caso en que es falso a costa de agrupar los otros tres. En la evaluación formal de expresiones, puede ocurrir que el antecedente de un condicional resulte falso, entonces se admitirá, se aceptará como verdadera la relación condicional expresada, puesto que no se está en el (único) caso en que es falsa.

  Todas estas reflexiones sólo tienen que ver con la capacidad de expresar implicaciones naturales con condicionales en este formalismo. Si uno se mantiene sólo dentro del formalismo, el condicional es la función que lo define, como en cualquier otra conectiva. Observe que, automáticamente,

  - si el antecedente es falso ya se puede afirmar que el condicional es verdadero (sin evaluar el consecuente),
  - y, si el consecuente es verdadero, también lo es el condicional (sin necesidad de evaluar el antecedente).

- *Bicondicional ($\leftrightarrow$):* si se admite que es verdad que 'si un número es divisible por cuatro entonces es divisible por dos', se está aceptando una relación condicional entre antecedente y consecuente (en ese sentido). Esta aceptación, direccional, no afirma (ni niega) que se verifique

la relación en el otro sentido. Cuando se enuncia un bicondicional, se enuncia una relación condicional en ambos sentidos: 'un número es múltiplo de 10 (si y sólo si, bicondicional) termina en 0'.

Un bicondicional es falso en dos interpretaciones: las que, respectivamente, harían falso el condicional en un sentido u otro.

1.2.4 Valores de verdad de fórmulas complejas

No sólo las fórmulas atómicas tienen asignado un valor de verdad. Cualquier fórmula, por muy compleja que sea, debe poder evaluarse. De hecho, la función que define las conectivas ya facilita evaluar fórmulas con, a lo sumo, una conectiva.

Si el número de fórmulas fuera finito y pequeño, se podría establecer 'a mano' una correspondencia entre fórmulas y valores de verdad. No todas las correspondencias serían aceptables. En concreto, se descartaría las correspondencias que evaluaran como verdadera tanto a una fórmula como a su negación. Las correspondencias aceptables se denominan *interpretaciones* y deben verificar ciertas propiedades.

Definición 1.30 (Interpretación) Una interpretación $\nu : \mathit{Form} \mapsto \{0,1\}$ es una función que asigna a cada fórmula un valor de verdad y que verifica las siguientes restricciones:

1. $\nu(\bot) = 0$, $\nu(\top) = 1$
2. $\nu(\neg\varphi) = \neg\nu(\varphi)$
3. $\nu(\varphi * \psi) = \nu(\varphi) * \nu(\psi)$, para cada uno de los operadores binarios

La última propiedad requiere que el valor asignado a una fórmula compleja coincida con el que calcula la conectiva cuando se aplica sobre los valores de las dos subfórmulas.

La figura (fig. 1.7) representa el árbol sintáctico de una fórmula. Se ha fijado una asignación de partida $\nu_{atom}(p) = 1$, $\nu_{atom}(q) = 0$, que se propaga hacia arriba hasta producir el valor de la fórmula. En todo nodo, cada conectiva calcula exactamente la función que la definía.

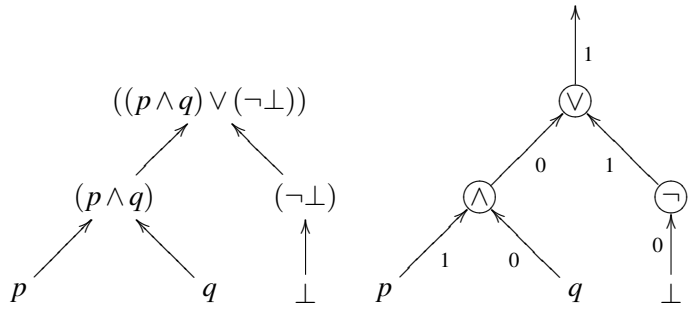


Figura 1.7: Interpretación inducida por una asignación

Parece, gráficamente, que partiendo de una asignación sobre las fórmulas atómicas queda determinado un valor para cada fórmula: una interpretación. Efectivamente, este proceso se puede formalizar recursivamente. Y el principio de recursión estructural garantiza que la función obtenida es única (para cada asignación de partida) y está bien definida.

Definición 1.31 (Interpretación, por extensión de una asignación) :

$$\nu(\varphi) = \begin{cases} \nu_*(\varphi) = (\psi) * \nu(\chi) & ,\varphi = (\psi * \chi) \\ \nu_\neg(\varphi) = \neg\nu(\psi) & ,\varphi = (\neg\psi) \\ \nu_{atom}(\varphi) & ,\varphi \text{ atómica} \end{cases}$$

Resta sólo un pequeño detalle formal. Si el alfabeto consta de 100 letras proposicionales (o de infinitas), una asignación ν debe facilitar el valor de verdad de todas ellas. Suponga que se está trabajando, en ese momento, sobre una fórmula φ (o fórmulas) que no contiene más que 10 letras proposicionales distintas. Existen, entonces, muchas asignaciones que coinciden en los valores sobre esas 10 letras pero difieren en el resto. Escoja cualquiera de ellas para calcular el valor de φ, todas producirán el mismo. Depende sólo de la asignación sobre esas 10 letras, no de la asignación sobre el resto.

Teorema 1.32 Si $\nu_1(p_k) = \nu_2(p_k)$ para todas las variables p_k que aparecen en φ, entonces $\nu_1(\varphi) = \nu_2(\varphi)$

Este resultado permite que, cuando se fija una asignación, sólo se haga sobre las letras que aparecen en las fórmulas analizadas, sin precisar el valor de otras letras del alfabeto no utilizadas.

1.2.5 Tablas de verdad

Si vuelve a observar la figura (fig. 1.7) apreciará que, lo único necesario para interpretar la fórmula, es fijar una asignación. Y también que, por contener sólo dos letras proposicionales, existen 4 asignaciones distintas.

Así, si el problema es interpretar la fórmula, basta una asignación. Pero si interesa conocer cómo se comporta globalmente la fórmula habrá que estudiarla frente a toda asignación posible. La tabla (tabl. 1.4) es una enumeración completa del valor de la fórmula para cada asignación distinta. A este tipo de tablas se le denomina *tabla de verdad* de la fórmula.

p	q	$((p \wedge q) \vee (\neg\bot))$
1	1	1
1	0	1
0	1	1
0	0	1

Tabla 1.4: Tabla de verdad de $((p \wedge q) \vee (\neg\bot))$

Para este ejemplo, sólo cuando se consideran todas las interpretaciones se confirma una propiedad de esta fórmula: es invariablemente verdadera bajo toda interpretación. Para confirmar propiedades como ésta resultará inevitable calcular íntegramente la tabla (hasta disponer de otros métodos). Otras cuestiones se resolverán considerando sólo algunas interpretaciones determinadas.

Considere una fórmula casi idéntica a la del ejemplo previo:$((p \wedge q) \vee (\neg p))$. Tiene el mismo árbol que el de la figura (fig. 1.7), sustituyendo $\bot$ por p. En este nuevo árbol, p aparece en dos posiciones iniciales. Sintácticamente se requiere así. Semánticamente, ambas apariciones están relacionadas: bajo una misma asignación no se puede considerar que una aparición de p es verdadera al tiempo que la otra es falsa.

La tabla de verdad de esta nueva fórmula se presenta en (tabl. 1.5). Cuando la fórmula es compleja suele ser útil ir calculando la tabla de verdad de algunas subfórmulas como pasos intermedios. En concreto, en esta tabla, la última columna se obtiene por disyunción de las dos auxiliares previas.

p	q	$(p \wedge q)$	$(\neg p)$	$((p \wedge q) \vee (\neg p))$
1	1	1	0	1
1	0	0	0	0
0	1	0	1	1
0	0	0	1	1

Tabla 1.5: Tabla de verdad de $((p \wedge q) \vee (\neg p))$

Es quizá más efectivo rellenar la tabla de verdad con el árbol en mente: de abajo arriba, se calcula el resultado de cada nodo-conectiva que se visita y se escriben los valores de salida en la columna debajo de la conectiva. Un ejemplo de este proceso se muestra en la tabla (tabl. 1.6). Los valores que resultan debajo de la conectiva principal (en negrita) son los últimos que se calculan y corresponden a la fórmula total.

p	q	$(p \wedge q)$	$\vee$	$\neg p$
1	1	1	**1**	0
1	0	0	**0**	0
0	1	0	**1**	1
0	0	0	**1**	1

Tabla 1.6: Tabla de verdad, compacta, de $((p \wedge q) \vee (\neg p))$

1.2.6 Tautologías y contradicciones

Ya se ha visto un ejemplo de fórmula que es verdadera en toda interpretación, en toda línea de la tabla de verdad. También pueden definirse fórmulas que sean falsas en toda interpretación. Y la tabla (tabl. 1.6) facilita un ejemplo de fórmula tanto con valores 1 como 0, según fuera la interpretación.

Definición 1.33 (Tautologías y contradicciones) A una fórmula verdadera para toda interpretación se le denomina *tautología*. A una fórmula falsa para toda interpretación se le denomina *contradicción*.

Las tautologías y contradicciones son dos presentaciones de la misma realidad. Resultarán extremadamente importantes en el estudio de la lógica proposicional. A las fórmulas que no son ni tautología ni contradicción se las suele denominar contingentes.

1.3 Conceptos semánticos básicos

1.3.1 Satisfacibilidad

La satisfacibilidad es la potencialidad de ser satisfecho. En Lógica, una interpretación satisface una o varias fórmulas cuando éstas se evalúan como verdaderas en esa interpretación.

Definición 1.34 (Satisfacción) Una interpretación ν satisface una fórmula φ si $\nu(\varphi) = 1$. Una interpretación ν satisface un conjunto de fórmulas $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ si $\nu(\varphi_k) = 1$ para toda fórmula φ_k de Φ.

Gráficamente, sobre la tabla de verdad, cualquier 'línea' (interpretación) donde una fórmula se evalúa como 1 *satisface* esa fórmula. Una interpretación satisface a un conjunto de fórmulas si *todas ellas* presentan valor 1 *en esa misma línea*.

La satisfacción de una fórmula coincide con la satisfacción del conjunto de fórmulas que sólo la contiene a ella. Cuando una interpretación satisface a una fórmula (o a un conjunto) se dice que es un *modelo* de esa fórmula (o conjunto).

El concepto de satisfacción requiere: (1) disponer de una interpretación y (2) de un conjunto de fórmulas. Entonces se puede decidir si esa interpretación satisface o no a ese conjunto. La satisfacibilidad es la posibilidad de ser satisfecho por alguna interpretación. No se facilita la interpretación, sólo la fórmula o fórmulas, y se requiere encontrar al menos una interpretación que las satisfaga.

p	q	r	$(p \wedge q) \wedge \neg(q \vee r)$	$(p \wedge q) \rightarrow (q \wedge r)$	$(p \wedge q) \rightarrow (q \vee r)$
1	1	1	0	1	1
1	1	0	0	0	1
1	0	1	0	1	1
1	0	0	0	1	1
0	1	1	0	1	1
0	1	0	0	1	1
0	0	1	0	1	1
0	0	0	0	1	1

Tabla 1.7: Una fórmula insatisfacible y dos satisfacibles

Definición 1.35 (Satisfacibilidad) Una fórmula φ es satisfacible si existe alguna interpretación ν tal que $\nu(\varphi) = 1$. Un conjunto de fórmulas $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ es satisfacible si existe alguna interpretación ν tal que $\nu(\varphi_k) = 1$, para toda φ_k en Φ.

De nuevo coloquialmente, basta que exista una línea (al menos) donde se satisfaga (simultáneamente) ese conjunto de fórmulas, para afirmar que es satisfacible. Si un conjunto (o una fórmula) no es satisfacible se denominará *insatisfacible*. Las fórmulas insatisfacibles también se denominan *contradicciones*.

Ejemplo 1.36 Observe la tabla de verdad (tabl. 1.7). La primera fórmula (por la izquierda) es insatisfacible, una contradicción. Las dos restantes son satisfacibles. De estas dos fórmulas satisfacibles, una resulta ser verdadera en toda interpretación y la otra no.

Ejemplo 1.37 Observe la tabla de verdad (tabl. 1.8). Ese conjunto de 3 fórmulas es satisfacible. Existe una línea (al menos) donde todas las fórmulas tienen el valor 1. La tabla (tabl. 1.9) corresponde a un conjunto de fórmulas insatisfacible.

Propiedades

El conjunto de fórmulas de la tabla (tabl. 1.8) se satisface simultáneamente en 5 líneas. Si se eliminase una de las fórmulas, el conjunto resultante se satisfaría en un número igual o mayor de líneas. Por contra, si se añadiese una fórmula cualquiera, el conjunto resultante se satisfaría en un número igual o menor de líneas.

En la generalización de este estudio, los siguientes resultados serán especialmente útiles. Se los adjuntamos en un único teorema, de nuevo sin demostración:

p	q	r	$p \to (q \vee r)$	$(p \wedge q) \vee r$	$r \to (r \vee p)$	
1	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\boxed{\surd}$
1	1	0	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\boxed{\surd}$
1	0	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\boxed{\surd}$
1	0	0	0	0	1	
0	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\boxed{\surd}$
0	1	0	1	0	1	
0	0	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\boxed{\surd}$
0	0	0	1	0	1	

Tabla 1.8: El conjunto $\Gamma = \{p \to (q \vee r), (p \wedge q) \vee r, q \to (r \vee p)\}$ es satisfacible

Teorema 1.38 Sea $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ un conjunto de fórmulas:

- Si Φ es satisfacible
  - y se elimina una de sus fórmulas, entonces el conjunto resultante es satisfacible
  - y se le añade una tautología, el conjunto resultante es satisfacible
  - y se le añade una contradicción, el conjunto resultante es insatisfacible
- Si Φ es insatisfacible
  - y se añade una fórmula cualquiera, entonces el conjunto resultante es insatisfacible
  - y se elimina de entre sus fórmulas una tautología (si la hubiera), el conjunto resultante es insatisfacible

Si a un conjunto satisfacible se le añade una fórmula satisfacible no se puede asegurar que el conjunto resultante sea satisfacible. Tampoco se puede asegurar nada si de un conjunto insatisfacible se elimina una fórmula cualquiera (no tautológica).

p	q	r	$p \to (q \vee r)$	$(p \wedge q)$	$\neg(r \vee p)$
1	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	0
1	1	0	$\cdots 1 \cdots$	$\cdots 1 \cdots$	0
1	0	1	$\cdots 1 \cdots$	0	0
1	0	0	0	0	0
0	1	1	$\cdots 1 \cdots$	0	0
0	1	0	$\cdots 1 \cdots$	0	1
0	0	1	$\cdots 1 \cdots$	0	0
0	0	0	$\cdots 1 \cdots$	0	0

Tabla 1.9: El conjunto $\Omega = \{(p \to (q \vee r), (p \wedge q), \neg(r \vee p)\}$ es insatisfacible

Procedimiento de decisión

El método más directo (aunque más costoso) para decidir la satisfacibilidad de una fórmula (o de un conjunto) consiste en recorrer todas las interpretaciones de la tabla de verdad, hasta producir:

- un resultado afirmativo: es satisfacible; para ello, *basta encontrar la primera* interpretación satisfactoria.

- o un resultado negativo: no es satisfacible; para ello es preciso recorrer *todas* las interpretaciones posibles.

Si en un conjunto de fórmulas aparecen n letras proposicionales, el número de interpretaciones distintas es 2^n. Es decir, si $n = 30$, sería preciso comprobar una tabla de verdad con más de mil millones de entradas.

'Afortunadamente', en lógica proposicional, el número de interpretaciones distintas (aunque intratable por exponencial) es finito. En lógica de primer orden se pierde esta cualidad: existen infinitas interpretaciones distintas para una fórmula.

Satisfacibilidad y conjunción

Existe una relación estrecha entre la satisfacibilidad de un conjunto de fórmulas y la satisfacibilidad de una única fórmula: la que se construye por conjunción de todas las fórmulas del conjunto.

Teorema 1.39 Sea $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ un conjunto de fórmulas:

- $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ es satisfacible si y sólo si la fórmula $\varphi_1 \wedge \ldots \wedge \varphi_n$ es satisfacible

- $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ es insatisfacible si y sólo si la fórmula $\varphi_1 \wedge \ldots \wedge \varphi_n$ es una contradicción

Observe que los resultados se aseguran de forma bicondicional: si Φ es satisfacible entonces la conjunción lo es; y si la conjunción es satisfacible, entonces el conjunto de las fórmulas conectadas por esas conjunciones es satisfacible. Cualquier interpretación que satisfaga al conjunto satisfará a la conjunción (y viceversa).

1.3.2 Validez

Una fórmula válida es aquélla que es verdadera frente a cualquier interpretación. Las tautologías son fórmulas válidas. La satisfacibilidad divide en dos al conjunto de fórmulas: en insatisfacibles (contradicciones) y satisfacibles. Este último conjunto también se divide en dos: fórmulas tautológicas y fórmulas contingentes. Observe que:

- si niega una fórmula insatisfacible, la fórmula resultante es una tautología

- si niega una tautología, la fórmula resultante es insatisfacible

- si niega una fórmula contingente, la fórmula resultante es contigente

- si niega una fórmula satisfacible, la fórmula resultante puede ser satisfacible o no serlo; tan sólo se puede afirmar que no será tautología

Notación Para expresar que una fórmula φ es válida se utilizará la notación $\models \varphi$.

Procedimiento de decisión

Para decidir la validez de una fórmula, de nuevo, el procedimiento semántico extensivo requiere recorrer toda la tabla de verdad. Los resultados negativos se pueden obtener más rápidamente: basta encontrar la primera interpretación que no satisface la fórmula. Pero los resultados positivos requieren una comprobación completa.

De los resultados del apartado anterior volvemos a resaltar que:

> *una fórmula* φ *es válida si y sólo si* $\neg\varphi$ *es insatisfacible*

Por lo tanto:

> *cualquier método de decisión de la (in)satisfacibilidad permite decidir la validez, y viceversa*

Preservación por sustitución

La sustitución uniforme (1.25) es una operación sintáctica que permite, a partir de una fórmula y de una sustitución, obtener otra: su instancia por esa sustitución dada. Es una operación extremadamente útil, ya que preserva la validez.

Teorema 1.40 Si φ es una fórmula válida entonces su instancia por sustitución $(\varphi)^\sigma$ es una fórmula válida, para cualquier sustitución σ.

Ejemplo 1.41 Compruebe que la fórmula φ:

$$((p \wedge q) \rightarrow (q \vee r))$$

es una tautología. Fijemos la sustitución

$$\sigma(p) = p, \quad \sigma(q) = (p \wedge r), \quad \sigma(r) = (\neg r)$$

entonces $(\varphi)^\sigma$ es una tautología:

$$((p \wedge (p \wedge r)) \rightarrow ((p \wedge r) \vee (\neg r)))$$

1.3.3 Consecuencia

Observe la tabla (tabl. 1.10). La última fórmula verifica una interesante relación respecto al conjunto de las fórmulas previas:

> "en *todas* (todas, todas) las líneas en que esas fórmulas coinciden en ser verdaderas, la última *también lo es* (y quizá en alguna más)".

Esta es una descripción informal de la relación 'ser consecuencia lógica de'. En este ejemplo, $(q \vee r)$ es consecuencia lógica del conjunto de fórmulas $\{((\neg p) \vee q), (p \vee r)\}$, a las que denominaremos premisas o hipótesis.

Muy coloquialmente: 'en cualquier estado de cosas en que las hipótesis, todas, sean verdaderas, la consecuencia (para merecer tal nombre) no puede dejar de serlo'. Observe que la consecuencia *puede* ser cierta en más líneas, pero *debe* ser cierta en las líneas verdaderas comunes de las hipótesis.

p	q	r	$\neg p \vee q$	$p \vee r$		$q \vee r$	
1	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright \triangleright \cdots$	1	$\surd$
1	1	0	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright \triangleright \cdots$	1	$\surd$
1	0	1		1		1	
1	0	0		1			
0	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright \triangleright \cdots$	1	$\surd$
0	1	0	1			1	
0	0	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright \triangleright \cdots$	1	$\surd$
0	0	0	1				

Tabla 1.10: Consecuencia. $\{(\neg p \vee q), (p \vee r)\} \models (q \vee r)$

Definición 1.42 (Consecuencia) Una fórmula ψ es consecuencia de un conjunto $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ de fórmulas si toda interpretación que satisface a Φ también satisface a ψ.

Notación Para representar que ψ es consecuencia lógica de $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ se suele emplear la notación $\Phi \models \psi$, ó $\{\varphi_1, \ldots, \varphi_n\} \models \psi$. Cuando no se cumple la relación de consecuencia se denota tachando el símbolo: $\Phi \not\models \psi$.

Es también usual omitir las llaves del conjunto: $\varphi_1, \ldots, \varphi_n \models \psi$. En particular, $\varphi \models \psi$ denota que 'la fórmula ψ es consecuencia lógica de la fórmula φ'.

p	q	r	$\neg p \wedge r$	$\neg(p \leftrightarrow q)$		$p \vee q$	
1	1	1	0	0		1	
1	1	0	0	0		1	
1	0	1	0	1		1	
1	0	0	0	1		1	
0	1	1	$\cdots 1 \cdots$	1	$\cdots \triangleright \triangleright \cdots$	$\cdots 1 \cdots$	$\surd$
0	1	0	0	1		1	
0	0	1	1	0		0	
0	0	0	0	0		0	

Tabla 1.11: Consecuencia. $\{\neg p \wedge r, \neg(p \leftrightarrow q)\} \models (p \vee q)$

La definición anterior se puede presentar, semiformalmente, como:

'$\Phi \models \psi$ si

para toda interpretación ν: [(ν satisface Φ) $\rightarrow$ (ν satisface ψ)] '

Es decir, es preciso comprobar, una a una, que toda interpretación ν verifica el condicional entre corchetes. En la tabla (tabl. 1.11), de las 8 interpretaciones posibles, 7 verifican el condicional porque hacen falso el antecedente (ninguna de ellas satisface a toda fórmula del conjunto Φ de premisas). Y la interpretación restante también verifica el condicional porque hace verdadero el antecedente y verdadero el consecuente.

Observe que para que ese condicional no se satisfaga debe existir (al menos) una interpretación (una línea) que satisfaga todas las premisas de Φ pero no satisfaga a ψ.

Procedimiento de decisión

En Lógica de Proposiciones, donde el número de interpretaciones distintas es finito, la relación de consecuencia lógica se puede decidir mediante el siguiente procedimiento:

1. construya la tabla de verdad común de todas las fórmulas (considerando todas las variables distintas)
2. señale las líneas verdaderas de la primera hipótesis (φ_1), de la segunda (φ_2), ..., de la enésima φ_n
3. determine la intersección de estos conjuntos de líneas
4. valore la consecuencia (ψ) sólo en esas líneas comunes (el valor de la consecuencia en el resto es irrelevante): si es verdadera en todas ellas, entonces ψ es consecuencia del conjunto $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$. Es decir, $\{\varphi_1, \varphi_2, \ldots, \varphi_n\} \models \psi$

Propiedades de interés

La relación de consecuencia, entre una fórmula y un conjunto de fórmulas, presenta las siguientes propiedades:

- Reflexividad: $\varphi \models \varphi$; es decir, toda fórmula es consecuencia lógica de sí misma
- Transitividad: si $\{\varphi_1, \ldots, \varphi_n\} \models \psi$ y $\psi \models \chi$, entonces $\{\varphi_1, \ldots, \varphi_n\} \models \chi$
- Monotonía: sea $\Phi = \{\varphi_1, \ldots, \varphi_n\}$; si $\Phi \models \psi$ y $\Phi \subset \Omega$, entonces $\Omega \models \psi$; es decir, si de un conjunto de premisas se sigue una consecuencia ψ, la incorporación de nuevas premisas a este conjunto no altera este hecho: ψ sigue siendo consecuencia de este conjunto ampliado.

Consecuencia de un conjunto insatisfacible

El último punto del apartado anterior puede resultar chocante: si se añaden nuevas premisas a un conjunto, es posible que produzca un conjunto insatisfacible. ¿La definición de consecuencia permite que una fórmula sea consecuencia de un conjunto insatisfacible?

En la definición (def. 1.42) de consecuencia, no existe restricción alguna sobre el conjunto Φ de premisas. De hecho, en el análisis que se hace del ejemplo de la tabla (tabl.1.11), se acepta que $\Phi \models \psi$ (para las fórmulas de esa tabla) porque 7 interpretaciones no satisfacen Φ (hacen falso el antecedente de la definición) y la restante satisface tanto Φ como ψ.

Con el mismo argumento empleado allí, si fueran 8 (todas) las interpretaciones que no satisficieran Φ, el condicional de la definición

> para toda interpretación ν: [(ν satisface Φ) $\rightarrow$ (ν satisface ψ)] '

no se haría falso en ningún caso. Y, por tanto, se admitiría que $\Phi \models \psi$.

Observe que el condicional requerido es verdadero para toda interpretación, en toda línea, porque ninguna consigue hacer falso el condicional, porque ninguna interpretación satisface Φ. Y todo esto, fuera quien fuera la consecuencia propuesta ψ.

> **Importante:** *Absolutamente cualquier fórmula verifica que es consecuencia lógica de un conjunto de fórmulas insatisfacible.*

Observe que, cuando se dice 'cualquier fórmula', ésto incluye a una fórmula cualquiera, pero también a su negación. Es decir:

$$\{p, \neg p\} \models p \vee q \qquad \text{, pero también,} \qquad \{p, \neg p\} \models \neg(p \vee q)$$

Si el conjunto de hipótesis es satisfacible ésto nunca ocurre: si una fórmula es consecuencia lógica de ese conjunto, su negación no lo es.

Importante: $\Phi \models \psi$ *y* $\Phi \models \neg\psi$ *si y sólo si* Φ *es insatisfacible*

Para desarrollar cualquier teoría es importante garantizar que el conjunto de premisas inicial es satisfacible.

Consecuencia, validez y satisfacibilidad

Ya se fijó la relación entre validez y satisfacibilidad: 'una fórmula φ es válida si y sólo si $\neg\varphi$ es insatisfacible. También existe una dependencia formal entre estos dos conceptos y el concepto de consencuencia.

Consecuencia y validez Observe la Tabla 1.12. Obviando la última columna, corresponde a la comprobación semántica de que r es consecuencia lógica de las tres fórmulas previas: $\{(p \vee q), (p \rightarrow r), (q \rightarrow r)\} \models r$.

Si esto es así (si se verifica la consecuencia lógica), no puede dejar de ser cierto lo siguiente:

> 'la fórmula condicional que se construye con la conjunción de todas las hipótesis como antecedente, y la consecuencia como consecuente resulta siempre ser una tautología'

p	q	r	$p \vee q$	$p \rightarrow r$	$q \rightarrow r$		r		$[(p \vee q) \wedge (p \rightarrow r) \wedge (q \wedge r)] \rightarrow r$
1	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright$	1	$\boxed{\surd}$	1
1	1	0	1						1
1	0	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright$	1	$\boxed{\surd}$	1
1	0	0	1		1				1
0	1	1	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots 1 \cdots$	$\cdots \triangleright$	1	$\boxed{\surd}$	1
0	1	0	1	1					1
0	0	1		1	1		1		1
0	0	0		1	1				1

Tabla 1.12: Si $\{\varphi_1, \ldots, \varphi_n\} \models \psi$ entonces $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi$ es tautología

Puede comprobar que el condicional construido 'no puede dejar de ser verdadero' en línea alguna:

1. sólo sería falso si su antecedente fuera verdadero y su consecuente fuera falso.
2. su antecedente es verdadero exclusivamente en todas las líneas en que coinciden las hipótesis en ser verdaderas
3. en esas líneas en concreto, el consecuente 'no puede dejar de ser verdadero' (puesto que es consecuencia lógica de las hipótesis con las que se ha construido el antecedente).

4. en el resto de las líneas, alguna hipótesis es falsa; luego, el antecedente es falso (y el condicional global analizado 'no corre el riesgo de ser falso').

Si	$\{\varphi_1,\ldots,\varphi_n\} \models \psi$	
entonces	$(\varphi_1 \wedge \ldots \wedge \varphi_n) \rightarrow \psi$	es tautología

Pero, además, se puede demostrar el resultado recíproco

Si	$(\varphi_1 \wedge \ldots \wedge \varphi_n) \rightarrow \psi$	es tautología
entonces	$\{\varphi_1,\ldots,\varphi_n\} \models \psi$	

Por lo tanto, en general

$\{\varphi_1,\ldots,\varphi_n\} \models \psi$ si y sólo si $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi$ es tautología.

Consecuencia y satisfacibilidad

Existe una interdependencia formal entre consecuencia e insatisfacibilidad que trataremos de justificar de la forma más intuitiva posible:

1. Si $\{\varphi_1,\ldots,\varphi_n\} \models \psi$ entonces el conjunto $\{\varphi_1,\ldots,\varphi_n,\neg\psi\}$ es insatisfacible

2. Si el conjunto $\{\varphi_1,\ldots,\varphi_n,\neg\psi\}$ es insatisfacible entonces $\{\varphi_1,\ldots,\varphi_n\} \models \psi$

Recuerde que un conjunto es insatisfacible si y sólo si la conjunción de sus fórmulas es insatisfacible. Así, el resultado anterior se podía haber escrito:

1. Si $\{\varphi_1,\ldots,\varphi_n\} \models \psi$ entonces la fórmula $\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi$ es insatisfacible

2. Si la fórmula $\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi$ es insatisfacible entonces $\{\varphi_1,\ldots,\varphi_n\} \models \psi$

De consecuencia a insatisfacibilidad Observe la Tabla 1.13. Se supone, de partida, que se verifica la relación de consecuencia: $\{\varphi_1,\varphi_2,\varphi_3\} \models \psi$, donde $\psi = r$. Puede comprobarse que efectivamente ocurre.

Las fórmulas del conjunto $\{\varphi_1,\varphi_2,\varphi_3\}$ coinciden en ser verdad en 3 líneas y *en todas ellas* es verdad r (puesto que es consecuencia). La fórmula $\neg r$ garantiza que es *falsa en todas ellas*. Luego el conjunto $\{\varphi_1,\varphi_2,\varphi_3,\neg r\}$ es insatisfacible.

El lector puede fácilmente generalizar este ejemplo: si una fórmula es consecuencia de un conjunto, la negación de esa fórmula garantiza que será falsa en todas las líneas en que se satisfacía el conjunto.

Si el conjunto de partida fuera insatisfacible se mantiene igualmente la propiedad: cualquier fórmula es consecuencia suya, y la inclusión de su negación en ese conjunto no lo convierte en satisfacible.Luego, se obtiene el resutado:

si $\{\varphi_1,\ldots,\varphi_n\} \models \psi$ entonces, $\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi$ es insatisfacible

p	q	r	$p\vee q$	$p\to r$	$q\to r$		r		$[(p\vee q)\wedge(p\to r)\wedge(q\wedge r)]$	$\neg r$
1	1	1	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots\triangleright$	1	$\boxed{\surd}$	$\cdots 1\cdots$	$\cdots 0\cdots$
1	1	0	1							
1	0	1	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots\triangleright$	1	$\boxed{\surd}$	$\cdots 1\cdots$	$\cdots 0\cdots$
1	0	0	1		1					
0	1	1	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots 1\cdots$	$\cdots\triangleright$	1	$\boxed{\surd}$	$\cdots 1\cdots$	$\cdots 0\cdots$
0	1	0	1	1						
0	0	1		1	1		1			0
0	0	0		1	1					

Tabla 1.13: Si $\{\varphi_1,\ldots,\varphi_n\}\models\psi$ entonces $\varphi_1\wedge\ldots\wedge\varphi_n\wedge\neg\psi$ es insatisfacible

De insatisfacibilidad a consecuencia En el resultado anterior, la fórmula ψ que hay que negar y añadir a las hipótesis está perfectamente definida: la que se afirma como consecuencia lógica de ellas. Suponga que quiere 'convencerse intuitivamente' del resultado recíproco:

si $\{\varphi_1,\ldots,\varphi_n,\neg\psi\}$ es insatisfacible, entonces $\{\varphi_1,\ldots,\varphi_n\}\models\psi$

En la expresión formal del conjunto se ha señalado una fórmula, pero realmente, en cada caso particular, lo único que se conoce es 'estas 10 fórmulas son insatisfacibles'. La pregunta ahora es ¿hay alguna de ellas que sea consecuencia lógica del resto? La respuesta sería: *cualquiera de ellas (negada) es consecuencia lógica del resto*.

Sobre la Tabla 1.14, compruebe que $\{\varphi_1,\varphi_2,\varphi_3\}$ es insatisfacible. Entonces, debe resultar (puede comprobarlo sobre la tabla) que:

1. $\{\varphi_2,\varphi_3\}\models\neg\varphi_1$
2. $\{\varphi_1,\varphi_3\}\models\neg\varphi_2$
3. $\{\varphi_1,\varphi_2\}\models\neg\varphi_3$

Observe que $\{\varphi_2,\varphi_3\}$ y $\{\varphi_1,\varphi_3\}$ resultan ser satisfacibles, mientras $\{\varphi_1,\varphi_2\}$ es insatisfacible.

Recapitulando hasta el momento: se parte de que un conjunto es insatisfacible, se extrae una fórmula, se niega, y se pretende demostrar que es consecuencia del resto. Al extraer esa fórmula del conjunto insatisfacible, éste (el conjunto resultante) puede seguir siendo insatisfacible o ya no. Consideremos la demostración buscada para estos dos casos.

p	q	r	$\varphi_1: p\to(q\wedge r)$	$\varphi_2: p\wedge\neg q$	$\varphi_3:\neg p\vee\neg q\vee\neg r$	$\neg\varphi_1$	$\neg\varphi_2$	$\neg\varphi_3$
1	1	1	1				1	1
1	1	0			1	1	1	
1	0	1		1	1	1		
1	0	0		1	1	1		
0	1	1	1		1		1	
0	1	0	1		1		1	
0	0	1	1		1		1	
0	0	0	1		1		1	

Tabla 1.14: Si $\varphi_1\wedge\ldots\wedge\varphi_n$ insatisfacible entonces $\{\varphi_1,\ldots,\varphi_{j-1},\varphi_{j+1},\ldots,\varphi_n\}\models\neg\varphi_j$

Si un conjunto es insatisfacible y suprimo una de las fórmulas φ_j el conjunto restante puede resultar satisfacible o no:

1. si resulta satisfacible: la fórmula φ_j 'taponaba (era falsa en)' *todas* las líneas en que ahora resultan coincidir las restantes; luego $\neg\varphi_j$ resulta verdadera en todas esas líneas. Así que $\neg\varphi_j$ es consecuencia del conjunto restante.

2. si resulta insatisfacible: cualquier fórmula (incluida $\neg\varphi_j$) es consecuencia de este conjunto resultante.

Combinando este resultado con el recíproco anterior se obtiene:

$$\boxed{\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi \text{ es insatisfacible} \qquad \text{si y sólo sí} \qquad \{\varphi_1, \ldots, \varphi_n\} \models \psi}$$

Resultados equivalentes Aprovechando la relación existente entre satisfacibilidad y validez, este mismo resultado podría reescribirse como:

$$\boxed{\neg(\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi) \text{ es válida} \qquad \text{si y sólo sí} \qquad \{\varphi_1, \ldots, \varphi_n\} \models \psi}$$

En la subsección anterior ya se fijó un resultado similar:

$$\boxed{\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \psi \text{ es válida} \qquad \text{si y sólo sí} \qquad \{\varphi_1, \ldots, \varphi_n\} \models \psi}$$

Las dos expresiones válidas de estos resultados son equivalentes: su tabla de verdad es exactamente la misma (la de una tautología). Cada una es válida si y sólo si la otra lo es. La equivalencia entre expresiones es el próximo objeto de estudio.

1.3.4 Equivalencia

Definición 1.43 (Fórmulas equivalentes) Dos fórmulas, φ y ψ, son equivalentes si $\varphi \models \psi$ y $\psi \models \varphi$.

Es decir, se requiere que una fórmula sea consecuencia lógica de la otra (y viceversa). Para fijar ideas, si φ se satisface en 5 líneas de la tabla de verdad, cualquier consecuencia suya ψ se satisfará en esas 5 líneas y quizá en alguna más. Pero se exige además, que también la primera sea consecuencia de la segunda.

Estas dos restricciones sólo se cumplen si ambas fórmulas son verdaderas exclusivamente en las mismas 5 líneas. Así, se podía haber partido de esta otra definición.

Definición 1.44 (Fórmulas equivalentes) Dos fórmulas, φ y ψ, son equivalentes si $\nu(\psi) = \nu(\varphi)$ para toda interpretación ν

Notación Escribiremos $\varphi \equiv \psi$ cuando ambas fórmulas sean equivalentes y $\varphi \not\equiv \psi$ cuando no lo sean.

Sobre la tabla de verdad, dos fórmulas equivalentes tienen exactamente los mismos valores de verdad sobre cada línea. Muy coloquialmente, son dos formas sintácticamente distintas de expresar 'lo mismo' (puesto que semánticamente son indistinguibles).

Propiedades A cualquier relación binaria que es reflexiva, simétrica y transitiva se le denomina relación de equivalencia. La que nos ocupa, ('tener la misma tabla de verdad que'), que representamos con el símbolo $\equiv$, tiene esas propiedades:

- Reflexividad: $\varphi \equiv \psi$
- Simetría: si $\varphi \equiv \psi$ entonces $\psi \equiv \varphi$
- Transitividad: si $\varphi \equiv \psi$ y $\psi \equiv \chi$ entonces $\varphi \equiv \chi$

Recuerde que una relación de equivalencia produce una partición en clases de equivalencia. La tabla (tabl. 1.15) muestra algunas fórmulas con dos letras proposicionales, agrupando las fórmulas equivalentes en la misma clase.

		C_0	C_1			C_8	C_{11}		C_{15}	
p	q	$p \wedge \neg p$	$\neg(p \vee q)$	$\neg p \wedge \neg q$	$\neg(\neg q \to p)$	$p \wedge q$	$p \to q$	$\neg p \vee q$	$q \vee \neg q$	$p \to p \vee q$
1	1	0	0	0	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	1	1	1	1
0	0	0	1	1	1	0	1	1	1	1

Tabla 1.15: Algunas fórmulas de dos variables, agrupadas por clases de equivalencia

Para fórmulas con dos letras proposicionales, sólo hay 16 clases de equivalencia distintas: $C_0, \ldots, C_{15}$. En nuestra notación, el subíndice k de la clase C_k resulta de leer las columnas como si codificaran números binarios (con el dígito más significativo arriba). Es fácil comprobar que las fórmulas con 3 letras proposicionales admiten 8 interpretaciones distintas, y por tanto, existen 2^8 clases de equivalencia distintas, desde C_0 a C_{255}.

Conjuntos completos de conectivas

Cualquier fórmula que contenga sólo dos letras proposicionales, por muy compleja que sea, pertenece a una (y sólo una) de las 16 clases de equivalencia posibles. Es equivalente, entonces, a cualquier fórmula de esa clase.

La fórmula más simple de la clase C_8 es $(p \wedge q)$. De igual forma, si se hubieran definido 16 conectivas binarias, la fórmula más simple de cada clase sería de la forma $(p * q)$, donde $*$ representa cualquiera de estas 16 conectivas. La tabla (tabl. 1.16) muestra las 16 clases de equivalencia y un posible símbolo para cada hipotética conectiva binaria.

Observe las clases C_7 y C_8. Presentan valores de verdad complementarios. De hecho, para toda clase existe otra complementaria. Se pueden buscar por simetría respecto al eje que separa C_8 y C_7.

		$\pi_\bot$	$\downarrow$	$\not\leftarrow$	$\pi_{\neg p}$	$\not\rightarrow$	$\pi_{\neg q}$	$\oplus$	$\uparrow$	$\wedge$	$\leftrightarrow$	π_q	$\to$	π_p	$\leftarrow$	$\vee$	$\pi_\top$
p	q	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Tabla 1.16: Todas las clases de equivalencia sobre fórmulas con dos variables

Puesto que nuestro alfabeto contiene sólo 4 conectivas binarias, ¿es menos expresivo que otro que contuviera las 16 posibles? No. Cualquier expresión de la forma $(\varphi \uparrow \psi)$ puede escribirse equivalentemente como $\neg(\varphi \wedge \psi)$. O una fórmula como $(\varphi \, \pi_p \, \psi)$ se puede escribir como $(\varphi \wedge \psi) \vee (\varphi \wedge \neg\psi)$.

Sentaremos, sin demostración, que cualquier fórmula que contuviera alguna de las 12 restantes conectivas puede expresarse equivalentemente como una fórmula sólo con las 4 primitivas del alfabeto (más la negación). Evidentemente, a costa de aumentar la longitud de las fórmulas.

Es más, algunas conectivas binarias primitivas pueden escribirse en términos de otra (más la negación). A un conjunto de conectivas sobre las que se puede representar cualquier otra se le denomina *completo*.

Teorema 1.45 Los conjuntos de conectivas $\{\neg,\wedge\}$, $\{\neg,\vee\}$, $\{\neg,\rightarrow\}$ son conjuntos completos de conectivas. Los únicos conjuntos completos de conectivas con una sola conectiva son $\{\uparrow\}$ y $\{\downarrow\}$.

Si el lector consulta textos diversos podrá observar que en el alfabeto no siempre se utilizan todas las conectivas empleadas en estas notas. Algunos sistemas lógicos sólo reconocen como conectivas a las de uno de estos sistemas completos. Las otras conectivas se definen como abreviaturas metalingüísticas de las primitivas. Este enfoque acorta las demostraciones inductivas y las definiciones por recursión.

Cálculo de equivalencias

Equivalencias básicas La tabla 1.17 muestra algunas de las equivalencias básicas de la lógica de proposiciones. Puede utilizar cualquiera de estos pares para comprobar su dominio de las tablas de verdad: salvo erratas, deben producir tablas iguales.

$\neg\neg p$	$\equiv$	p	doble negación			
$\neg\bot$	$\equiv$	$\top$		$\neg\top$	$\equiv$	$\bot$
$p\wedge\top$	$\equiv$	p		$p\vee\bot$	$\equiv$	p
$p\wedge\bot$	$\equiv$	$\bot$		$p\vee\top$	$\equiv$	$\top$
$p\wedge p$	$\equiv$	p	idempotencia	$p\vee p$	$\equiv$	p
$p\wedge q$	$\equiv$	$q\wedge p$	conmutatividad	$p\vee q$	$\equiv$	$q\vee p$
$p\wedge(q\wedge r)$	$\equiv$	$(p\wedge q)\wedge r$	asociatividad	$p\vee(q\vee r)$	$\equiv$	$(p\vee q)\vee r$
$p\wedge(p\vee q)$	$\equiv$	p	absorción	$p\vee(p\wedge q)$	$\equiv$	p
$p\wedge(q\vee r)$	$\equiv$	$(p\wedge q)\vee(p\wedge r)$	distributividad	$p\vee(q\wedge r)$	$\equiv$	$(p\vee q)\wedge(p\vee r)$
$p\wedge q$	$\equiv$	$\neg(\neg p\vee\neg q)$	De Morgan	$(p\vee q)$	$\equiv$	$\neg(\neg p\wedge\neg q)$
$p\leftrightarrow q$	$\equiv$	$(p\rightarrow q)\wedge(q\rightarrow p)$		$p\leftrightarrow q$	$\equiv$	$(p\wedge q)\vee(\neg p\wedge\neg q)$
$p\rightarrow q$	$\equiv$	$(\neg p\vee q)$		$p\rightarrow q$	$\equiv$	$\neg q\rightarrow\neg p$

Tabla 1.17: Equivalencias básicas

Sustitución y reemplazo El siguiente resultado relaciona la equivalencia entre dos fórmulas con la validez de una única fórmula.

Teorema 1.46 Dadas dos fórmulas, φ y ψ, son equivalentes si y sólo si la fórmula $\varphi\leftrightarrow\psi$ es una tautología.

Observe que la implicación entre resultados se produce en ambos sentidos (si y sólo si): si se conoce que son equivalentes se puede afirmar que esa única fórmula bicondicional es válida. Y, si se facilita una fórmula bicondicional (y además se asegura que es válida) entonces sus subfórmulas izquierda y derecha, por separado, resultan ser equivalentes.

Teorema 1.47 Dadas dos fórmulas equivalentes, $\varphi \equiv \psi$, si se produce la misma sustitución uniforme σ en ambas, sus instancias por sustitución resultan ser equivalentes: $\varphi^\sigma \equiv \psi^\sigma$.

Puesto que la sustitución uniforme preserva la validez, si $\varphi \leftrightarrow \psi$ es válida también lo será su instancia por sustitución $(\varphi \leftrightarrow \psi)^\sigma$, es decir, $\varphi^\sigma \leftrightarrow \psi^\sigma$. Luego, de la validez de este bicondicional se sigue $\varphi^\sigma \equiv \psi^\sigma$.

> **Importante:** *En la tabla de equivalencias básicas, éstas están expresadas en términos de letras proposicionales. Pero se pueden sustituir uniformemente, preservando la equivalencia: es decir, que si $p \to q \equiv \neg p \vee q$ también se verifica que $\varphi \to \psi \equiv \neg\varphi \vee \psi$, para todo par de fórmulas.*

Ejemplo 1.48 Partamos de la equivalencia

$$p \to q \equiv \neg p \vee q$$

y de la sustitución

$$\sigma(p) = (t \wedge s) \qquad \sigma(q) = (p \vee v)$$

Se obtiene entonces un par de expresiones equivalentes entre sí (no necesariamente con las de partida):

$$(t \wedge s) \to (p \vee v) \equiv \neg(t \wedge s) \vee (p \vee v)$$

Recuerde que la sustitución uniforme supone cambiar *cada* aparición de una letra proposicional por su fórmula sustituta. Es posible definir otro tipo de sustitución que no requiera cambiar cada aparición de la cadena sustituida. Para diferenciar esta operación de la sustitución uniforme, la denominaremos *reemplazo*.

Definición 1.49 (Reemplazo) Sea $\varphi \equiv \psi$, y X una fórmula donde φ puede aparecer varias veces como subfórmula. Si se reemplaza en X la subformula φ por ψ (sobre una o varias apariciones, no necesariamente en todas) la fórmula Y resultante es equivalente a la X de partida.

Muy intuitivamente: si en un circuito electrónico aparece un componente en varias posiciones y se reemplaza en una (o varias) por un componente equivalente (quizá con otro diseño interno, pero con la misma relación entrada-salida), el circuito global resultante es equivalente al circuito global de partida.

La sustitución uniforme preserva la equivalencia porque se produce en toda aparición del elemento sustituido (en ambas fórmulas equivalentes). No requiere que el sustituido sea equivalente al sustituyente.

El reemplazo preserva la equivalencia porque el elemento sustituyente es equivalente al sustituido. Se produce sobre una fórmula para generar una equivalente. No requiere que se produzca en toda aparición del sustituido.

Ejemplo 1.50 Partamos de la fórmula

$$X = ((p \to q) \to ((p \to q) \vee p)$$

y de la equivalencia

$$(p \to q) \equiv (\neg q \to \neg p)$$

entonces, la fórmula

$$Y = ((p \to q) \to ((\neg q \to \neg p) \vee p)$$

verifica que

$$X \equiv Y$$

Formas normales

Forma normal disyuntiva En una tabla de verdad donde sólo se consideren las variables p, q y r, la fórmula $\varphi = \neg p \wedge q \wedge r$ es verdadera en una única línea:

p	q	r	$\neg p \wedge q \wedge r$
0	1	1	1

Si se considerase otra variable adicional s, la misma fórmula sería verdadera en dos líneas:

p	q	r	s	$\neg p \wedge q \wedge r$
0	1	1	1	1
0	1	1	0	1

Situémonos en el primer caso: la conjunción utiliza todas las letras proposicionales consideradas (negadas o no). Una conjunción similar, como $\psi = p \wedge q \wedge \neg r$ también será verdadera en otra única línea. Así, la disyunción de estas conjunciones será verdadera exactamente en dos interpretaciones:

p	q	r	φ	ψ	$(\varphi \vee \psi)$
1	1	0	0	1	1
0	1	1	1	0	1

Los resultados anteriores sugieren que cualquier fórmula puede escribirse equivalentemente como una disyunción de estas conjunciones. Por ejemplo, de la fórmula γ de la tabla (tabl. 1.18) no se facilita su expresión sintáctica: podría ser cualquiera de las expresiones equivalentes que producen esa tabla de verdad. Una de esas expresiones equivalentes será la disyunción de las tres conjunciones señaladas en la tabla: la *forma normal disyuntiva* de γ.

p	q	r	γ	$\bigvee$
1	1	1	0	
1	1	0	1	$(p \wedge q \wedge \neg r)$
1	0	1	0	
1	0	0	1	$(p \wedge \neg q \wedge \neg r)$
0	1	1	0	
0	1	0	0	
0	0	1	1	$(\neg p \wedge \neg q \wedge r)$
0	0	0	0	

Tabla 1.18: Forma normal disyuntiva de una fórmula

Para ser más precisos, a esta forma normal disyuntiva se le denomina *completa*: cada letra proposicional aparece una vez y sólo una. Toda fórmula puede ser reescrita equivalentemente a esta forma *salvo las contradicciones*.

Para poder expresar contradicciones es preciso relajar la completitud, permitiendo que algunas letras falten e incluso que algunas se repitan. Observe que una forma disyuntiva con una sola conjunción como $(p \wedge \neg p \wedge q)$ ya expresa una contradicción.

Definición 1.51 (Forma normal disyuntiva) Una fórmula está en forma normal disyuntiva si es de la forma $\varphi_1 \vee \ldots \vee \varphi_n$, donde cada φ_k es una conjunción de literales.

Se denomina literal a una letra proposicional o a su negación.

Una forma normal disyuntiva es una contradicción si y sólo si cada una de sus conjunciones incluye una letra negada y no negada.

Forma normal conjuntiva Observe la Tabla (tabl. 1.19). Las fórmulas ψ_1 y ψ_2 sólo son *falsas* en una única línea cada una. Luego, su *conjunción* sólo es falsa en esas dos líneas. Es una fórmula expresada en forma normal conjuntiva completa respecto al conjunto de letras proposicionales $\{p,q,r\}$

Toda fórmula es expresable, equivalentemente, en esta forma completa, *salvo las tautologías*. De forma dual a la exposición anterior, fijaremos una definición de forma normal conjuntiva que incluya a las completas y a otras (en particular, a las que permiten expresar tautologías). Así, toda fórmula será expresable en forma normal conjuntiva.

p	q	r	$\varphi_1 : \neg p \wedge q \wedge r$	$\varphi_2 : p \wedge q \wedge \neg r$	$\varphi_1 \vee \varphi_2$	$\psi_1 : p \vee \neg q \vee \neg r$	$\psi_2 : \neg p \vee \neg q \vee r$	$\psi_1 \wedge \psi_2$
1	1	1						
1	1	0		1	1		0	0
1	0	1						
1	0	0						
0	1	1	1		1	0		0
0	1	0						
0	0	1						
0	0	0						

Tabla 1.19: Formas normales conjuntivas y disyuntivas

Definición 1.52 (Forma normal conjuntiva) Una fórmula está en forma normal conjuntiva si es de la forma $\varphi_1 \wedge \ldots \wedge \varphi_n$, donde cada φ_k es una disyunción de literales.

Una forma normal conjuntiva es una tautología si y sólo si cada una de sus disyunciones incluye una letra negada y no negada.

Forma clausulada

Considere una forma normal conjuntiva

$$(\varphi_1) \wedge \ldots \wedge (\varphi_n)$$

A cada una de las disyunciones φ_k se le denomina *cláusula*. Como la conjunción es conmutativa, resulta que se obtiene una expresión equivalente si se varía el orden de las cláusulas. Puesto que este orden es irrelevante, se puede escribir la expresión como un conjunto de cláusulas:

$$\{(\varphi_1), \ldots, (\varphi_n)\}$$

Recuerde que el concepto de conjunto no supone un orden entre sus elementos. Obviamente, para que esta reescritura mantenga las propiedades de la forma original recordaremos que entre los elementos de este conjunto (sus cláusulas) se aplicaba una conjunción.

De forma análoga, si nos centramos en una única cláusula

$$\varphi_k = (\varphi_k^1 \vee \ldots \vee \varphi_k^m)$$

se puede también escribir como un conjunto de literales (recordando que estaban unidos por una disyunción). Así, una fórmula cualquiera se puede 'almacenar en un sistema' como un conjunto de cláusulas, es decir, como un conjunto de conjuntos de literales:

$$\{(\varphi_1), \ldots, (\varphi_n)\} = \{\{\varphi_1^1, \ldots, \varphi_1^r\}, \ldots, \{\varphi_n^1, \ldots, \varphi_n^t\}\}$$

A esta forma de expresión se le denomina *forma clausulada* de una fórmula. También se utiliza este término, en algunos textos, como sinónimo de forma normal conjuntiva.

Ejemplo 1.53 Sea la fórmula

$$(p \wedge q \wedge r) \rightarrow (s \vee t)$$

Entonces, su forma normal conjuntiva se obtiene:

$$\begin{array}{lcll} ((p \wedge q) \wedge r) \rightarrow (s \vee t) & \equiv & \neg((p \wedge q) \wedge r) \vee (s \vee t) & \text{ya que } X \rightarrow Y \equiv \neg X \vee Y \\ & \equiv & (\neg(p \wedge q) \vee \neg r) \vee (s \vee t) & \text{ya que } \neg(X \wedge Y) \equiv (\neg X \vee \neg Y) \\ & \equiv & ((\neg p \vee \neg q) \vee \neg r) \vee (s \vee t) & \text{ya que } \neg(X \wedge Y) \equiv (\neg X \vee \neg Y) \\ & \equiv & (\neg p \vee \neg q \vee \neg r \vee s \vee t) & \text{asociatividad de } \vee \text{ (abuso de notación)} \end{array}$$

Para la fórmula de partida, su forma normal conjuntiva sólo contiene una cláusula, con cinco literales. Su forma clausulada se puede escribir como:

$$\{\{\neg p, \neg q, \neg r, s, t\}\}$$

Puede comprobar que la forma normal conjuntiva de $p \rightarrow (s \wedge q)$ es $(\neg p \vee s) \wedge (\neg p \vee q)$. Y su forma clausulada

$$\{\{\neg p, s\}, \{\neg p, q\}\}$$

En el ejemplo previo, a partir de una fórmula, se obtiene finalmente otra equivalente. Este resultado se calcula, sin necesidad de utilizar tablas de verdad: la primera fórmula es equivalente a una segunda (y se justifica por qué), ésta segunda a una tercera, y así sigue una cadena de equivalencias hasta llegar a la forma deseada. La transitividad de la equivalencia garantiza que la primera es equivalente a la última (en realidad, que todas son equivalentes: este cálculo 'no se sale' de la clase de equivalencia de la fórmula inicial).

El cálculo (manual) de fórmulas equivalentes es una habilidad operacional necesaria. Se facilitarán ejercicios sobre este punto. Estas notas se están ciñendo a resaltar los conceptos básicos fundamentales, que se resumen en el próximo apartado.

Equivalencia, consecuencia, validez y satisfacibilidad

El concepto de equivalencia tiene conexiones formales con los otros conceptos semánticos expuestos: consecuencia, validez e insatisfacibilidad.

Todos los resultados que siguen han ido apareciendo en la exposición:

$$\begin{array}{lll} \varphi \equiv \psi & \text{si y sólo si} & \varphi \models \psi \text{ y } \psi \models \varphi \\ & \text{si y sólo si} & (\varphi \rightarrow \psi) \text{ es una tautología y } (\psi \rightarrow \varphi) \text{ es una tautología} \\ & \text{si y sólo si} & (\varphi \leftrightarrow \psi) \text{ es una tautología} \\ & \text{si y sólo si} & (\neg(\varphi \leftrightarrow \psi)) \text{ es insatisfacible.} \end{array}$$

1.4 Sistemas deductivos

Los sistemas deductivos son de naturaleza sintáctica, combinatoria. En general, constan de un conjunto quizá vacío de fórmulas especialmente señaladas, denominadas axiomas y de un conjunto determinado de reglas de inferencia. El juego formal consiste en construir una secuencia finita de pasos, de aplicaciones de esas reglas, que llamaremos demostración o prueba. El interés de estas construcciones reside en su conexión con el concepto semántico de consecuencia (o su expresión como insatisfacibilidad).

Si un sistema es correcto, consistente o coherente, cada fórmula derivada desde un conjunto de premisas es consecuencia de ellas. Y si un sistema es completo, cualquier relación de consecuencia existente se puede llegar a explicitar como una derivación, como una demostración.

En estos apuntes no se ha abordado el análisis formal de estas propiedades. Están más bien enfocados a facilitar el uso de los sistemas descritos, garantizando su buen comportamiento. Le recomendamos que se familiarice con los sistemas deductivos propuestos porque todos ellos se ampliarán para el caso de la Lógica de Primer Orden.

1.4.1 Deducción natural

Los sistemas de deducción natural, sistema de tipo Gentzen, son quizá los que más directamente reflejan nuestra propia manera de construir argumentos. Constan de varias reglas de inferencia y de ningún axioma. Y permiten hacer suposiciones adicionales, de abrir líneas argumentales adicionales que, cuando se cierran, aportan algo al flujo argumental principal.

Puesto que se pueden aplicar varias reglas distintas, su implementación es más complicada que la de los otros dos sistemas abordados. No obstante, cuando se conoce la fórmula que se pretende demostrar, su estructura y la de las premisas casi determina alguna de las secuencias de reglas necesarias. El resto es creatividad (humana o de silicio).

Conjunciones

Introducción de la conjunción *Si se aceptan como verdaderas* las dos siguientes frases: (p) *'hoy es viernes'*, (q) *'son las cinco'*, el uso habitual de la conjunción obliga a aceptar como verdadera la frase única ($p \wedge q$) *'hoy es viernes y son las cinco'*.

$$\frac{\begin{array}{c}\text{'hoy es viernes'}\\ \text{'son las cinco'}\end{array}}{\text{'hoy es viernes y son las cinco'}} \qquad \frac{\begin{array}{c}p\\ q\end{array}}{p \wedge q}$$

Observe que se genera una sentencia final a partir de dos previas, con una forma determinada. Esta forma garantiza que la frase construida es consecuencia de las anteriores.

$$\frac{\begin{array}{c}(p \rightarrow q)\\ (r \vee \neg p)\end{array}}{(p \rightarrow q) \wedge (r \vee \neg p)} \qquad \frac{\begin{array}{c}\varphi\\ \psi\end{array}}{\varphi \wedge \psi}$$

Eliminación de la conjunción Veamos otro proceso distinto de generación, esta vez partiendo de una única fórmula previa:

$$\frac{(p \rightarrow q) \wedge (r \vee \neg p)}{(p \rightarrow q)} \qquad \frac{\varphi \wedge \psi}{\varphi}$$

O también:

$$\frac{(p \rightarrow q) \wedge (r \vee \neg p)}{(r \vee \neg p)} \qquad \frac{\varphi \wedge \psi}{\psi}$$

Compruebe en qué líneas de la tabla de verdad son verdaderas las fórmulas previas: en todas ellas (y quizá en alguna más) será verdadera la fórmula generada. Esta opción siempre es posible, aunque

costosa. A partir de este punto se describe una alternativa sintáctica, basada en cálculo: respete estrictamente la forma de generación de nuevas fórmulas y resultarán consecuencia de las de partida.

Hasta el momento se han propuesto 3 de estas reglas sintácticas. Nos referiremos a ellas como: introducción de la conjunción ($\wedge I$), eliminación derecha ($\wedge \mathcal{E}_d$) y eliminación izquierda ($\wedge \mathcal{E}_i$) de la conjunción.

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \wedge I \qquad \frac{\varphi \wedge \psi}{\varphi} \wedge \mathcal{E}_d \qquad \frac{\varphi \wedge \psi}{\psi} \wedge \mathcal{E}_i$$

En la figura (fig. 1.8) se utilizan estas reglas para demostrar que de $p \wedge (q \wedge r)$ se deduce $p \wedge r$. La expresión de partida (que suponemos verdadera) se denomina *premisa*. A partir de ella se deducen tanto p (línea 2) como $(q \wedge r)$ (línea 3) por eliminación de la conjunción. En la línea 4 se deduce r por eliminación de la conjunción sobre 3: observe la explicación $\wedge \mathcal{E}_i 3$. Por último, a partir de 2 y de 4, por introducción de la conjunción se obtiene 5.

1	$p \wedge (q \wedge r)$	*premisa*
2	p	$\wedge \mathcal{E}_d 1$
3	$(q \wedge r)$	$\wedge \mathcal{E}_i 1$
4	r	$\wedge \mathcal{E}_i 3$
5	$p \wedge r$	$\wedge I 2,4$

Figura 1.8: Deducción natural. $p \wedge (q \wedge r) \vdash p \wedge r$

Los números que etiquetan cada línea del argumento permiten linealizar la deducción, que realmente tiene estructura de árbol (fig. 1.9).

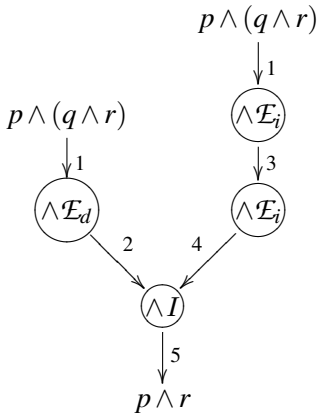


Figura 1.9: Árbol de la deducción natural. $p \wedge (q \wedge r) \vdash p \wedge r$

En este apartado aparece un nuevo símbolo $\vdash$ distinto de (aunque esperamos que relacionado con) $\models$. La relación $p \wedge q \models p$ es de carácter semántico: *'compruebe todas las interpretaciones donde ...'*. Mientras que $p \wedge q \vdash p$ es de carácter sintáctico, combinatorio: *'utilizando adecuadamente las reglas del cálculo se ha conseguido llegar a generar la fórmula p'*.

Cada regla, de las 3 propuestas, es *correcta, consistente o coherente* puesto que toda fórmula deducida ($\vdash$) es además consecuencia ($\models$) de las premisas de la regla.

Cuando se afirma que suponemos las premisas verdaderas no significa que sean tautologías. Significa que nos estamos restringiendo a las líneas de la tabla de verdad en que todas ellas coinciden en ser verdaderas. En todas esas interpretaciones, las fórmulas deducidas resultarán verdaderas.

Disyunciones

Introducción de la disyunción La introducción de la disyunción se acepta intuitivamente con facilidad: si se admite que es verdad que *'llueve'*, también debe aceptarse *'llueve o hace viento'*.

$$\frac{\text{llueve}}{\text{llueve o hace viento}}\ \vee I_d \qquad \frac{\varphi}{\varphi \vee \psi}\ \vee I_d \qquad\qquad \frac{\varphi}{\psi \vee \varphi}\ \vee I_i \qquad \frac{(p \to q)}{(r \leftrightarrow s) \vee (p \to q)}\ \vee I_i$$

La fórmula nueva (cualquiera) que se introduce disyuntivamente puede hacerlo por la derecha ($\vee I_d$) o por la izquierda ($\vee I_i$).

Eliminación de la disyunción También existe otra regla denominada 'eliminación de la disyunción'. El sistema propuesto se recuerda fácilmente: se define una regla de introducción y otra de eliminación para cada conectiva. Sin embargo, en este caso, el nombre 'eliminación de la disyunción' NO corresponde a su materialización más directa:

$$\frac{p \vee q}{p} \qquad \text{¡¡NO!!} \qquad \frac{p \vee q}{q}$$

En su lugar, considere esta introducción:

1. $(p \vee q)$: *'pago la hipoteca o pago el recibo del coche (o ambos)'*
2. $[p \ldots r]$: suponiendo que *'pago la hipoteca'*, entonces (bla,bla,bla), y entonces ..., conclusión: *'no llego a fin de mes'*.
3. $[q \ldots r]$: suponiendo que *'pago el recibo del coche'*, entonces (quiza otro bla,bla,bla), conclusión: *'no llego a fin de mes'*.

Partiendo de que es verdadera la premisa disyuntiva $(p \vee q)$, se supone (adicional y momentáneamente) p. Con esta hipótesis adicional, razonando adecuadamente, se llega a r. Cerremos este argumento opcional, es decir, dejemos de dar por supuesto p (y todo a lo que nos llevaba). Ahora supongamos alternativamente q, que vuelve a llevarnos a r. En este caso, de la disyunción $(p \vee q)$ se deduce r, con el auxilio de los dos argumentos opcionales mencionados:

$$\begin{array}{l} \varphi \vee \psi \quad \boxed{\begin{array}{c}\varphi \\ \vdots \\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}} \\ \hline \chi \end{array} \vee \mathcal{E}$$

Ejemplo 1.54 Observe la figura (fig. 1.10). Es un argumento con una única premisa (línea 1) que intenta concluir la fórmula de la línea 7. Escriba sólo ambas, dejando un hueco entremedias. Cuando (1) sea verdadera no podrán dejar de serlo (2) y (3): así que las escribimos automáticamente, por si nos son útiles. Ahora bien, (3) es una disyunción, 'presenta 2 caminos'. ¿con todo lo que ya sé, desde 1 hasta 3, puedo llegar por ambos caminos a la misma conclusión (7)?

Para intentar probarlo, supongo primero que es verdadera q (que es verdadero el primer camino). En este caso, como ya era verdadera p (línea 2) también lo será su conjunción con q. En (6) se añade disyuntivamente la fórmula que se necesitaba para llegar a la (7) buscada.

Por el otro camino, supuesto que es r verdadera, se efectúa un proceso parecido que acaba concluyendo (6) en esa caja derecha. Yo no sé qué camino es verdadero, sólo sé que (3) me garantiza que uno de ellos lo es. Y eso garantiza que (7) lo es.

La fórmula (7) está fuera del ámbito de ambas cajas: es verdadera si la premisa lo es. Sin embargo, $p \wedge q$ (línea 5, caja izquierda) se supone verdadera si lo es la hipótesis que abre la caja izquierda.

Observe que, dado el principio y el fin, sólo los pasos de las líneas 5-6 (en cada caja) requieren un esfuerzo creativo para obtener el resultado fijado.

1	$p \wedge (q \vee r)$				*premisa*
2	p				$\wedge\mathcal{E}1$
3	$(q \vee r)$				$\wedge\mathcal{E}1$
4	q	*suposic*		r	*suposic*
5	$(p \wedge q)$	$\wedge I3,4$		$(p \wedge r)$	$\wedge I3,4$
6	$(p \wedge q) \vee (p \wedge r)$	$\vee I5$		$(p \wedge q) \vee (p \wedge r)$	$\vee I5$
7	$(p \wedge q) \vee (p \wedge r)$				$\vee\mathcal{E}3$

Figura 1.10: Deducción natural. $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$

La dependencia entre expresiones del argumento anterior puede verse en el árbol de la figura (fig. 1.11).

Ejemplo 1.55 Observe el argumento de la figura (fig. 1.12). Utiliza de nuevo todas las reglas anteriores. Aquí, de nuevo, sólo hay unas pocas líneas (5-6) que requieren un poco de creatividad.

Si se fija en la premisa y la conclusión de (fig. 1.10) y (fig. 1.12) verá que están intercambiadas: cada una se deduce de la otra. Esta puede ser una definición sintáctica de equivalencia. Puesto que efectivamente las reglas usadas son correctas no hacen más que reflejar el hecho de que ambas fórmulas son semánticamente equivalentes.

Condicionales

Eliminación del condicional La eliminación del condicional ($\rightarrow \mathcal{E}$) es una de las reglas de inferencia con más tradición. Se conoce alternativamente como *Modus Ponens* y es también intuitivamente inmediata.

$$\frac{\begin{array}{c}\text{Si presiono entonces se rompe}\\ \text{Presiono}\end{array}}{\text{Se rompe}} \qquad \frac{\begin{array}{c}p \rightarrow q\\ p\end{array}}{q}$$

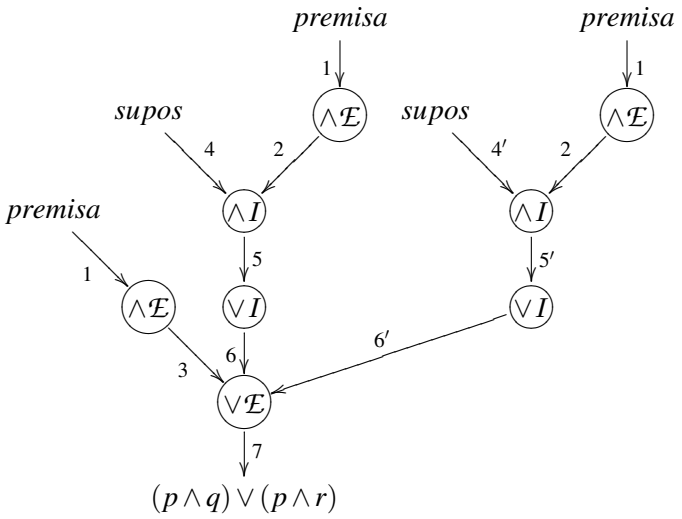


Figura 1.11: Árbol de la deducción natural. $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$

1	$(p \wedge q) \vee (p \wedge r)$			*premisa*
2	$(p \wedge q)$	*suposic*	$(p \wedge r)$	*suposic*
3	p	$\wedge\mathcal{E}2$	p	$\wedge\mathcal{E}2$
4	q	$\wedge\mathcal{E}2$	r	$\wedge\mathcal{E}2$
5	$(q \vee r)$	$\vee I4$	$(q \vee r)$	$\vee I5$
6	$p \wedge (q \vee r)$	$\wedge I3,5$	$p \wedge (q \vee r)$	$\wedge I3,5$
7	$p \wedge (q \vee r)$			$\vee\mathcal{E}1$

Figura 1.12: Deducción natural. $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$

Observe que cuando el condicional $p \rightarrow q$ se supone verdadero sólo se descarta una de las cuatro líneas de su tabla de verdad. No se sigue ni se descarta que p sea verdadero o que q lo sea: sólo que no puede ocurrir simultáneamente que p sea verdadero y q falso. Si a esta premisa se añade que otra que afirma que p es verdadero, necesariamente q debe serlo.

Introducción del condicional De nuevo, esta es otra regla que no puede aplicarse tan directamente como su nombre evoca. Corresponde a este esquema formal:

$$\dfrac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \psi\end{array}}}{\varphi \rightarrow \psi} \rightarrow I$$

Si en el curso de un argumento se aventura una hipótesis adicional (φ), lo que se deduce a partir de ella, en el ámbito de su caja, se garantiza verdadero en tanto que φ lo sea. ¿Cuándo se puede cerrar ese ámbito? En cualquier momento, mediante introducción de la implicación.

Suponga que en el punto en que quiere cerrarse se había llegado a derivar ψ. En ese punto se afirma que ψ es verdadera, siempre dependiendo de que φ lo sea. Es decir, en el argumento de esa caja se ha demostrado (quizá con ayuda de fórmulas previas a la caja), que si φ es verdadera entonces ψ no puede dejar de serlo. Entonces, el condicional $\varphi \rightarrow \psi$ es verdadero.

Observe que $\varphi \rightarrow \psi$ está ya fuera de la caja: no depende de la verdad de φ. Si φ es falso resultará verdadero. Y si φ es verdadero, en la caja se ha demostrado que no puede dejar de ser ψ verdadero. Como esta derivación puede haber utilizado fórmulas externas previas, el condicional sí depende de que sean verdaderas las premisas de la línea argumental exterior.

En resúmen, si se abre una hipótesis adicional luego no puede cerrarse con un condicional cualquiera: el consecuente del mismo será la última fórmula efectivamente derivada en la caja. Los siguientes ejemplos le ayudarán a ver la aplicación correcta de esta regla de inferencia.

1	$p \rightarrow q$	*premisa*
2	$q \rightarrow r$	*premisa*
3	p	*suposic*
4	q	$\rightarrow \mathcal{E} 1,3$
5	r	$\rightarrow \mathcal{E} 2,4$
6	$p \rightarrow r$	$\rightarrow I 3,5$

Figura 1.13: Deducción natural. $p \rightarrow q, q \rightarrow r \vdash p \rightarrow r$

Ejemplo 1.56 Considere primero el ejemplo de la figura (fig. 1.15). Como se trata de derivar finalmente el condicional $(p \wedge q) \rightarrow r$, conviene abrir una caja con la hipótesis adicional $(p \wedge q)$, el antecedente, y ver si se puede llegar a derivar internamente r, el consecuente. Y, en efecto, se consigue con ayuda de las premisas.

El argumento de (fig. 1.13) es muy parecido. Y el de (fig. 1.14) abre una caja dentro de otra. Sus cierres sucesivos, mediante introducción de la implicación, producen condicionales sucesivos.

Resulta particularmente interesante el último ejemplo (fig. 1.16): no existen premisas del argumento, tan sólo una suposición que se descarga. La fórmula resultante no depende de la verdad de

1	$(p \wedge q) \rightarrow r$	*premisa*
2	p	*suposic*
3	q	*suposic*
4	$(p \wedge q)$	$\wedge I 2,3$
5	r	$\rightarrow \mathcal{E} 1$
6	$(q \rightarrow r)$	$\rightarrow I 3,5$
7	$p \rightarrow (q \rightarrow r)$	$\rightarrow I 2,6$

Figura 1.14: Deducción natural. $(p \wedge q) \rightarrow r \vdash p \rightarrow (q \rightarrow r)$

p, ni de la de las premisas (puesto que no hay). El lector puede comprobar que dicha fórmula es una tautología: es verdadera en todo caso. Trate de derivar de forma análoga (con 2 cajas anidadas) la tautología $p \rightarrow (q \rightarrow (q \vee r))$.

1	$p \rightarrow (q \rightarrow r)$	*premisa*
2	$(p \wedge q)$	*suposic*
3	p	$\wedge \mathcal{E} 2$
4	q	$\wedge \mathcal{E} 2$
5	$(q \rightarrow r)$	$\rightarrow \mathcal{E} 1,3$
6	r	$\rightarrow \mathcal{E} 5,4$
7	$(p \wedge q) \rightarrow r$	$\rightarrow I 2,6$

Figura 1.15: Deducción natural. $p \rightarrow (q \rightarrow r) \vdash (p \wedge q) \rightarrow r$

1	p	*suposic*
2	$p \vee q$	$\vee I 1$
3	$p \rightarrow (p \vee q)$	$\rightarrow I 1,2$

Figura 1.16: Deducción natural. $\vdash p \rightarrow (p \vee q)$

Negaciones

Restan cuatro reglas para completar la descripción del sistema: introducción y eliminación de la negación, eliminación de la contradicción y eliminación de la doble negación.

Introducción de la negación Si de una suposición adicional se deriva una contradicción, puede cerrarse el ámbito de aquélla y concluir la negación de la suposición.

Eliminación de la negación Dadas dos fórmulas, donde una es negación de la otra, se puede concluir la contradicción.

$$\dfrac{\begin{array}{|c|}\hline \varphi \\ \vdots \\ \bot \\ \hline \end{array}}{\neg\varphi}\,\neg I \qquad\qquad \dfrac{\varphi \quad \neg\varphi}{\bot}\,\neg\mathcal{E}$$

Eliminación de la contradicción De una contradicción se puede concluir cualquier fórmula.

Eliminación de la doble negación De una fórmula doblemente negada se puede derivar dicha fórmula.

$$\dfrac{\bot}{\varphi}\,\bot\mathcal{E} \qquad\qquad \dfrac{\neg\neg\varphi}{\varphi}\,\neg\neg\mathcal{E}$$

En este punto le recomendamos que haga una recopilación, en una hoja en blanco, de las reglas de inferencia propuestas. Preste atención a las reglas que requieren abrir una o varias cajas. Si no se quiere que la derivación dependa finalmente de estas suposiciones, deberá cerrarlas. Cada una de estas reglas permite cerrar ese ámbito de una forma determinada. En todo caso, en cualquiera de estos ámbitos puede utilizarse una fórmula anterior, siempre que pertenezca a ese ámbito o a uno que le englobe (es decir, que no haya sido cerrado).

Reglas derivadas

Cuando se utilizan frecuentemente las reglas propuestas se advierte que hay ciertos patrones que se repiten. Observe el argumento de la figura (fig. 1.17). Esta derivación se puede producir en cualquier punto de un argumento, no se precisa que las dos primeras fórmulas sean las premisas.

De hecho, si desde un cierto punto del argumento se pueden utilizar dos fórmulas como la 1 y la 2 (estén donde estén, mientras sean formalmente accesibles), ya se sabe que se puede derivar la fórmula 6, repitiendo exactamente esos pasos.

1	$p \to q$	*premisa*
2	$\neg q$	*premisa*
3	p	*suposic*
4	q	$\to\mathcal{E}3,1$
5	$\bot$	$\neg\mathcal{E}2,4$
6	$\neg p$	$\neg I3,5$

Figura 1.17: Deducción natural. $p \to q, \neg q \vdash \neg p$

Con el mismo argumento (fig. 1.17) se puede derivar $\varphi \to \psi, \neg\psi \vdash \neg\varphi$, para dos fórmulas φ y ψ cualesquiera.

Como los pasos que llevan de 1 y 2 a 6 son conocidos y repetibles (y, de hecho, muy frecuentes), se suelen obviar los pasos intermedios. Se justifica entonces la derivación diciendo que se ha utilizado

una determinada regla derivada, a la que se le asigna un nombre. En concreto, la regla del ejemplo propuesto se suele denominar Modus Tollens.

$$\begin{array}{ll} 1 & (p \vee r) \rightarrow (q \wedge s) \\ 2 & \neg(q \wedge s) \\ 3 & \neg(p \vee r) \qquad\qquad MT1,2 \end{array}$$

Para fijar ideas, puede contemplar las reglas derivadas como lo hace con las subrutinas en programación.

1.4.2 Resolución

Introducción

Estrategia deductiva por refutación Ya se resaltó que existía una estrecha relación entre consecuencia y satisfacibilidad. De tal forma que, decidir si ψ es consecuencia de un conjunto de fórmulas $\Phi = \{\phi_1, \ldots, \phi_n\}$ se puede reducir a otro problema: decidir si Φ y $\neg\psi$ pueden ser simultáneamente verdaderos.

$$\phi_1, \ldots, \phi_n \models \psi \quad \text{ si y sólo si } \quad \Phi \cup \{\neg\psi\} = \{\phi_1, \ldots, \phi_n, \neg\psi\} \text{ es insatisfacible}$$

Así, *si desea comprobar que* una fórmula es consecuencia de otras, niéguela e incorporéla a esas otras. Si resulta insatisfacible este nuevo conjunto, efectivamente existía aquella relación de consecuencia.

Constatación sintáctica de la insatisfacibilidad En general resulta difícil decidir si varias fórmulas complejas pueden (simultáneamente) satisfacerse por alguna interpretación. La operación que se propone facilita este proceso: si el conjunto de partida $\Phi \cup \{\neg\psi\}$ era insatisfacible, en algún momento del cálculo se evidencia claramente.

Para ello, la resolución amplía este conjunto: va añadiéndole sucesivamente nuevas fórmulas (menos complejas). Pero todas estas fórmulas se construyen a partir de las que hay en ese momento en el conjunto. Y lo que es más importante: *este proceso no altera la satisfacibilidad*. Si el conjunto de partida era satisfacible, ninguna ampliación por este método produce un conjunto ampliado insatisfacible. Y si era insatisfacible, esta propiedad se mantiene en todo conjunto ampliado por este método.

¿Cómo se detecta la insatisfacibilidad? Si el conjunto inicial lo es, el proceso mencionado acaba aportando una nueva fórmula que sólo aparece en este caso: la cláusula vacía. De hecho, aparece si y sólo si el conjunto inicial $\Phi \cup \{\neg\psi\}$ es insatisfacible.

Requisitos formales del proceso El cálculo que se menciona arriba se define sobre fórmulas en forma normal conjuntiva. En otro caso 'no sabe cómo operar'. Tanto las premisas como la negación de la supuesta consecuencia deben reescribirse equivalentemente en esta forma. Existe un procedimiento que garantiza la expresión equivalente en esta forma de cualquier fórmula proposicional.

A partir de la forma normal conjuntiva se puede denotar una fórmula de manera más compacta, sin conectivas. Esta notación se conoce como forma clausulada y es realmente la que se maneja en este apartado.

Forma clausulada

Definición 1.57 (Literal) Un literal es una fórmula atómica o la negación de una fórmula atómica.

Es decir, son literales cada una de las siguientes seis expresiones: p, q, $\neg r$, $\neg p$, $\bot$, $\neg\bot$. Y no son literales las expresiones: $\neg\neg p$, $r \vee q$, $\neg(p \wedge q)$.

A cada literal l le corresponde un literal complementario l^c. El complementario de un literal positivo como p es $\neg p$. Y el de un literal negativo como $\neg p$ es p.

Forma normal conjuntiva Una fórmula como la siguiente está en forma normal conjuntiva:

$$(p \vee \neg q) \wedge (\neg r \vee \neg q \vee r \vee p) \wedge (r \vee q)$$

Es decir, *es una conjunción de fórmulas, cada una de las cuales es una disyunción de literales*. A toda fórmula proposicional se le puede hacer corresponder una fórmula normal conjuntiva equivalente. Para ello basta:

1. eliminar los bicondicionales, si los hubiera, mediante el reemplazo:

$$X \leftrightarrow Y \equiv (X \rightarrow Y) \wedge (Y \rightarrow X)$$

2. eliminar los condicionales, si los hubiera, mediante el reemplazo:

$$X \rightarrow Y \equiv \neg X \vee Y$$

3. introducir todas las negaciones hasta que afecten a fórmulas atómicas:

$$\neg(X \wedge Y) \equiv (\neg X \vee \neg Y) \qquad \neg(X \vee Y) \equiv (\neg X \wedge \neg Y)$$

4. eliminar las dobles negaciones:

$$\neg\neg X \equiv X$$

5. reubicar correctamente las conjunciones y disyunciones:

$$X \wedge (Y \vee Z) \equiv (X \wedge Y) \vee (X \wedge Z) \qquad X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z)$$

La demostración de que es un proceso correcto requiere precisar un poco más. De momento es suficiente si le permite calcular, a mano, la fórmula normal conjuntiva equivalente a una dada.

Ejemplo 1.58 Una fórmula normal conjuntiva equivalente a $q \leftrightarrow (s \vee t)$ puede calcularse como sigue:

$$\begin{array}{lcll}
q \leftrightarrow (s \vee t) & \equiv & (q \rightarrow (s \vee t)) \wedge ((s \vee t) \rightarrow q) & X \leftrightarrow Y \equiv (X \rightarrow Y) \wedge (Y \rightarrow X) \\
& \equiv & (\neg q \vee (s \vee t)) \wedge (\neg(s \vee t) \vee q) & X \rightarrow Y \equiv \neg X \vee Y \\
& \equiv & (\neg q \vee s \vee t) \wedge (\neg(s \vee t) \vee q) & \text{asociatividad } \vee\text{, abuso notación} \\
& \equiv & (\neg q \vee s \vee t) \wedge ((\neg s \wedge \neg t) \vee q) & \neg(X \vee Y) \equiv \neg X \wedge \neg Y \\
& \equiv & (\neg q \vee s \vee t) \wedge ((\neg s \vee q) \wedge (\neg t \vee q)) & (X \wedge Y) \vee Z \equiv (X \vee Z) \wedge (Y \vee Z) \\
& \equiv & (\neg q \vee s \vee t) \wedge (\neg s \vee q) \wedge (\neg t \vee q) & \text{asociatividad } \wedge\text{, abuso notación}
\end{array}$$

En el penúltimo paso, la aplicación de la distributividad no corresponde exactamente al esquema fijado en el punto 5. Se ha utilizado implícitamente la permutación conmutativa $X \vee Y \equiv Y \vee X$.

Ejemplo 1.59 Puede comprobar que una fórmula normal conjuntiva equivalente a

$$(p \rightarrow (t \vee p)) \wedge (\neg p \rightarrow (p \vee q))$$

resulta ser

$$(\neg p \vee t \vee p) \wedge (p \vee p \vee q)$$

que puede simplificarse equivalentemente como:

$$((\neg p \vee p) \vee t) \wedge ((p \vee p) \vee q) \equiv (\top \vee t) \wedge (p \vee q) \equiv \top \wedge (p \vee q) \equiv (p \vee q)$$

Forma clausulada El valor de verdad de una fórmula como $(\neg p \vee q \vee r \vee \neg p)$ (dada una interpretación) no depende de la posición de los literales, que pueden permutarse. Tampoco del correcto emplazamiento de paréntesis interiores, para determinar exactamente qué dos subfórmulas une cada disyunción. Es más, también es independiente de la aparición repetida de algunos literales en la misma cláusula.

Todo esto permite que este tipo de fórmulas se puedan denotar como conjuntos de literales. Así, la fórmula mencionada puede expresarse como el conjunto $\{\neg p, q, r\}$. Recuerde que un conjunto no supone secuencia alguna entre sus elementos y que tampoco es significativa la enumeración repetida de un elemento: ese elemento pertenece o no pertenece al conjunto, sin más.

Definición 1.60 (Forma clausulada) Una cláusula C es un conjunto de literales $C = \{l_1, \ldots, l_n\}$. Implícitamente se suponen disyuntivamente unidos.

Una fórmula en forma clausulada está escrita como un conjunto de cláusulas $\{C_1, \ldots, C_k\}$, implícitamente unidas por conjunciones.

Toda fórmula en forma normal conjuntiva es fácilmente expresable en forma clausulada. Por ejemplo,

$$(p \vee q \vee p) \wedge (r \vee \neg p \vee \neg t) \text{ se puede escribir como } \{\{p, q\}, \{r, \neg p, \neg t\}\}$$

Se puede obtener una notación un poco más compacta si se denota $\neg p$ como $\bar{p}$ y simplemente se yuxtaponen los literales de cada cláusula. Así, la fórmula anterior se escribiría como: $\{pq, r\bar{p}\bar{t}\}$

El camino descrito permite representar una fórmula cualquiera en notación clausulada: como conjunto de cláusulas. En términos de implementación de estos sistemas deductivos, como listas de cláusulas (es decir, como listas de listas de literales).

Definición 1.61 (Satisfacibilidad de cláusulas) La satisfacibilidad de las cláusulas se define como:

- Una cláusula $C = \{l_1, \ldots, l_n\}$ es satisfacible si y sólo si la fórmula $(l_1 \vee \ldots \vee l_n)$ es satisfacible.
- La cláusula vacía $\{\}$ es insatisfacible.
- Un conjunto de cláusulas $\{C_1, \ldots, C_k\}$ es satisfacible si y sólo si existe una misma interpretación que satisface a cada cláusula del conjunto.

Recuerde que una cláusula es un disyunción implícita. Es, por lo tanto satisfacible si existe un literal que se hace verdadero bajo cierta asignación. En la cláusula vacía no existe ningún literal que pueda hacerla verdadera, luego es insatisfacible. Denotaremos a la cláusula vacía como $\{\}$ o como $\square$.

Para satisfacer un conjunto de cláusulas basta encontrar una interpretación que satisface al menos un literal en cada cláusula.

Principio de Resolución

Considere dos cláusulas cualesquiera, por ejemplo:

$$\{p, \neg q, r\} \qquad \{s, \neg t, r\}$$

Tratemos de encontrar interpretaciones que satisfagan ambas cláusulas. Para ello, recuerde que los literales en cada una están implícitamente unidos por disyunciones. Así, en la primera, basta que p sea verdadera para que lo sea toda la cláusula. Entonces, fijar que p y s son verdaderas produce la satisfacción de ambas cláusulas (con independencia de los valores del resto de las variables). El mismo resultado se produce si se fija que q es falsa y s es verdadera. O simplemente si se asigna el valor verdadero a r, que aparece en ambas cláusulas.

Considere ahora dos cláusulas con una particularidad especial. Por ejemplo, la que presentan

$$\{p, \neg q, r\} \quad \text{y} \quad \{q, \neg t\}$$

En ambas aparece la misma letra proposicional q, negada en una y no negada en la otra. Es decir, para cierto literal l de una cláusula, la otra contiene el literal complementario l^c.

Tratemos, de nuevo, de encontrar interpretaciones que las satisfagan. En este caso no basta con asignar un cierto valor a q, aunque la letra aparezca repetida. Si se afirma que es falsa, se satisface sin más la primera cláusula (pero no la segunda). Si se afirma que es verdadera, sólo se asegura la satisfacción de la segunda. Es decir, para un cierto valor de q se satisface sólo una cláusula: debe existir al menos otro literal (distinto de q y de $\neg q$) que produzca la satisfacción de la otra cláusula. En este ejemplo, p debe ser verdadera o r, o t debe ser falsa.

Esta simple consideración produce unos excelentes resultados. Observe la cláusula $\{p, r, \neg t\}$. Se ha obtenido por unión de las dos anteriores salvo los literales q y $\neg q$. Cualquier interpretación que satisficiera a las dos cláusulas generatrices satisface a la cláusula generada.

Definición 1.62 (Principio de Resolución) Sean X e Y cláusulas tales que $l \in X$ y $l^c \in Y$. Se denomina *Resolución* a la regla:

$$\frac{X \qquad Y}{(X \cup Y) - \{l, l^c\}}$$

Las cláusulas X e Y se *resuelven* sobre l. A la cláusula resultante se la denomina *resolvente*.

Teorema 1.63 El Principio de Resolución preserva la satisfacibilidad. Esto es, si una asignación satisface un conjunto de cláusulas, también satisface cualquier resolvente de dos de ellas.

Para fijar ideas, suponga que se parte de un conjunto satisfacible de 6 cláusulas y se genera (si es posible) la resultante de dos de ellas. Entonces las 7 cláusulas (la generada y las 6 previas) se siguen satisfaciendo por las mismas interpretaciones que las originales. De nuevo, si de estas 7 cláusulas se vuelven a resolver dos, las 8 cláusulas resultantes siguen siendo satisfechas por las mismas interpretaciones originales.

Cuidado En las cláusulas $\{\neg p, q\}$ y $\{p, \neg q\}$ se puede escoger el par de literales sobre el que efectuar la resolución. Una resolución sobre p produciría $\{\neg q, q\}$ y una resolución sobre q produciría $\{\neg p, p\}$. Puede comprobarse que una interpretación como $\nu(p) = 0, \nu(q) = 0$ satisface tanto las dos cláusulas originales como sus resolventes.

Sin embargo, una resolución *simultánea* sobre p y q produciría la cláusula vacía, que no se satisface para la interpretación que satisfacía a las cláusulas generatrices. De hecho, éste es un uso incorrecto de la regla de Resolución, que definía el resolvente sobre *un único literal*.

A partir de un conjunto de cláusulas, una derivación por resolución es una secuencia finita de cláusulas, donde cada una de ellas es bien una de la cláusulas de partida o bien la resolvente de dos previas.

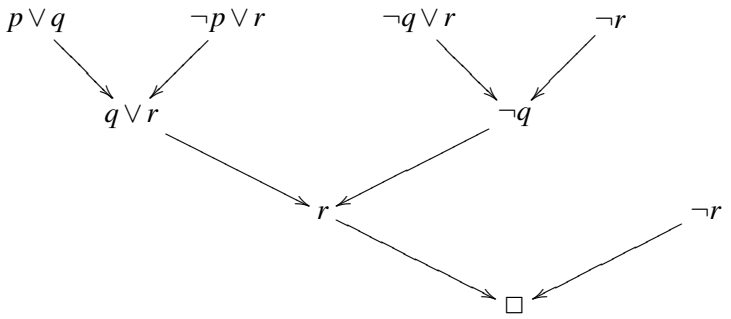


Figura 1.18: Árbol de la resolución $\{pq, \bar{p}r, \bar{q}r, \bar{r}\}$

1	$\{p,q\}$	
2	$\{\neg p,r\}$	
3	$\{\neg q,r\}$	
4	$\{\neg r\}$	
5	$\{q,r\}$	$(1,2)$
6	$\{\neg q\}$	$(3,4)$
7	$\{r\}$	$(6,5)$
8	$\{\}$	$(4,7)$

Figura 1.19: Una derivación de la cláusula vacía

Teorema 1.64 (Consistencia y Completud de la Resolución) Un conjunto no vacío de cláusulas es insatisfacible si y sólo si existe a partir del mismo una derivación por resolución de la cláusula vacía.

Como de todos los sistemas deductivos, la consistencia garantiza la corrección de las reglas: 'que todo lo que se va obteniendo es consecuencia de lo anterior'. Algo más complicado generalmente de demostrar es la completud de un sistema: 'dame cualquier consecuencia de un conjunto de premisas y encontraré una derivación de la misma en este sistema'.

Cuando se aborda indirectamente el concepto de consecuencia, via insatisfacibilidad, la consistencia y completud toman una forma como la del teorema previo. La mención a la cláusula vacía es específica de este sistema. De los dos sentidos del bicondicional 'si y sólo si' trate de ver cuál se refiere a la consistencia ('lo que hago, lo hago correctamente') y cuál a la completud ('todo lo que es necesario ser hecho, se puede hacer').

Ejemplo 1.65 Se quiere comprobar si $\psi := r$ es consecuencia lógica de Φ, donde $\Phi = \{p \vee q, p \rightarrow r, q \rightarrow r\}$. O, equivalentemente, si $\Phi \cup \{\neg\psi\}$ es insatisfacible. Puesto que el sistema es completo, si efectivamente este conjunto es insatisfacible, existirá una derivación de la cláusula vacía a partir del mismo.

Escribamos en forma clausulada cada premisa y la negación de la supuesta conclusión. Cada una de estas fórmulas se expresa como una única cláusula. Así, el conjunto inicial de cláusulas es

$$\{\{p,q\},\{\neg p,r\},\{\neg q,r\},\{\neg r\}\}$$

El árbol de la figura (fig. 1.18) muestra el proceso de resolución. Observe que una de las cláusulas $(\neg r)$ se ha utilizado dos veces. Una linealización de este proceso se encuentra en la figura (fig. 1.19).

Ejemplo 1.66 Consideremos un conjunto $\Phi = \{\phi_1, \phi_2, \phi_3\}$ con las siguientes fórmulas:

1. $\phi_1 := p \rightarrow (q \wedge r)$
2. $\phi_2 := \neg(s \vee t)$
3. $\phi_3 := q \leftrightarrow (s \vee t)$

Se quiere comprobar si $\psi := \neg p$ es consecuencia lógica de Φ. O, equivalentemente, si $\Phi \cup \{\neg\psi\}$ es insatisfacible. Puesto que el sistema es completo, si efectivamente este conjunto es insatisfacible, existirá una derivación de la cláusula vacía a partir del mismo.

Escribamos en forma clausulada cada premisa y la negación de la supuesta consecuencia:

1. $\phi_1 := \{\{\neg p,q\},\{\neg p,r\}\}$
2. $\phi_2 := \{\{\neg s\},\{\neg t\}\}$
3. $\phi_3 := \{\{\neg q,s,t\},\{\neg s,q\},\{\neg t,q\}\}$
4. $\neg\psi := \{\{p\}\}$

Ahora en este conjunto de fórmulas algunas producen más de una cláusula. Entre todas las fórmulas se han producido 8 cláusulas. Recuerde que las cláusulas están implícitamente unidas entre sí por conjunciones. En definitiva, las cuatro fórmulas serán simultáneamente satisfacibles si y sólo si lo son las 8 cláusulas (por separado, sin importar qué fórmulas las produjo).

En la figura (fig. 1.20) se puede encontrar una derivación que confirma que el conjunto inicial era insatisfacible.

Cláusulas de Horn

Considere la fórmula $(p \wedge q \wedge r) \rightarrow s$. En ciertas aplicaciones, a este tipo de fórmulas se les denomina *reglas*. Con varias de estas reglas puede tratar de modelizar una máquina expendedora o el comportamiento de un programa.

Supuesta verdadera una regla como $(p \wedge q \wedge r) \rightarrow s$, si se verifican además las tres fórmulas p, q y r, no puede dejar de ser verdadera s.

Escribamos ahora esta regla en forma normal conjuntiva o en forma clausulada:

$$(p \wedge q \wedge r) \rightarrow s \;\equiv\; \neg(p \wedge q \wedge r) \vee s \;\equiv\; (\neg p \vee \neg q \vee \neg r \vee s)$$

Observe que las fórmulas atómicas del antecedente aparecen como literales negativos y las del consecuente como literales positivos. Recorramos el camino inverso desde esta otra fórmula, con dos literales positivos:

$$(\neg a_1 \vee \neg a_2 \vee \neg a_3 \vee c_1 \vee c_2) \;\equiv\; \neg(a_1 \wedge a_2 \wedge a_3) \vee (c_1 \vee c_2) \;\equiv\; (a_1 \wedge a_2 \wedge a_3) \rightarrow (c_1 \vee c_2)$$

1	$\{\neg p, q\}$	
2	$\{\neg p, r\}$	
3	$\{\neg s\}$	
4	$\{\neg t\}$	
5	$\{\neg q, s, t\}$	
6	$\{\neg s, q\}$	
7	$\{\neg t, q\}$	
8	$\{p\}$	
9	$\{q\}$	$(8,1)$
10	$\{r\}$	$(8,2)$
11	$\{s,t\}$	$(9,5)$
12	$\{t\}$	$(11,3)$
13	$\{\}$	$(12,4)$

Figura 1.20: Una derivación de la cláusula vacía

Definición 1.67 (Cláusulas de Horn) Una cláusula de Horn es una cláusula con, a lo sumo, un literal positivo.

Ejemplo de cláusulas de Horn son:

1. $\{\neg p, \neg q, \neg r, s\}$
2. $\{\neg p, \neg q\}$
3. $\{s\}$

A las del primer tipo se le conoce como *reglas* y a las del último como *hechos*. Las del segundo tipo son equivalentes a un condicional como $(p \wedge q) \rightarrow \bot$.

Analizaremos más en detalle las cláusulas de Horn cuando se aborde la resolución en Lógica de Primer Orden. Baste en este momento saber que resultan especialmente útiles no sólo porque facilitan una representación legible e intuitiva de ciertos sistemas sino porque la resolución con cláusulas de Horn facilita la labor computacional.

1.4.3 Tablas semánticas

Las tablas semánticas también se denominan tablas analíticas y más comúnmente *tableaux* (tableau, en singular).

Introducción

Estrategia deductiva por refutación Las tablas semánticas proporcionan un medio sintáctico de investigar la satisfacibilidad de un conjunto. Todo lo que se mencionó en (1.4.2) sobre la relación entre consecuencia y satisfacibilidad es aplicable en este punto.

Así, *si desea comprobar que* una fórmula es consecuencia de otras, niéguela e incorporéla a esas otras. Si resulta insatisfacible este nuevo conjunto, efectivamente existía aquella relación de consecuencia.

Constatación sintáctica de la insatisfacibilidad De nuevo, si el conjunto de partida $\Phi \cup \{\neg\psi\}$ era insatisfacible, en algún momento del cálculo se evidencia claramente.

Intuitivamente, las fórmulas del conjunto inicial se estructuran como árbol. En particular como un árbol muy lineal, con una sola rama. Existen dos tipos de reglas de inferencia y ambas añaden nodos al árbol: una, linealmente, y otra produciendo una bifurcación binaria. La satisfacibilidad de un árbol se puede comprobar en cualquier momento, considerando la satisfacibilidad de cada rama. Existen indicadores sintácticos que lo explicitan.

De nuevo, la clave de estas operaciones consiste en la preservación de la satisfacibilidad. Cada aplicación de una regla amplía el árbol sin modificar la satisfacibilidad. Si el árbol original (las fórmulas de partida) eran satisfacibles así resultará el árbol ampliado en todo momento. Y si eran insatisfacibles, se llegará a constatar sintácticamente en algún momento.

Notación uniforme

Considere la fórmula $(p \wedge \neg q)$. Una interpretación la satisface si y sólo si satisface a sus componentes conjuntivos. Es decir,

$$(p \wedge \neg q) \text{ es satisfacible si y sólo si } \{p, \neg q\} \text{ es satisfacible}$$

La fórmula propuesta era evidentemente conjuntiva. Otra fórmula como $\neg(\neg p \vee q)$ se puede fácilmente reescribir de forma equivalente como una fórmula conjuntiva. En particular como $(\neg\neg p \wedge \neg q)$. Y se satisfaría, de nuevo, si y sólo si se satisfacen simultáneamente sus componentes.

Dualmente, una fórmula disyuntiva como $(p \vee \neg q)$ se satisface si y sólo si se satisface alguna de sus dos fórmulas inmediatas. Otras fórmulas, no explícitamente disyuntivas como $(p \rightarrow \neg q)$ también se puede reescribir equivalentemente como $(\neg p \vee \neg q)$.

Vamos a fijar el desarrollo que se persigue con esta introducción. Observe la tabla en (fig. 1.21). Cada fórmula α de la izquierda es equivalente a la conjunción de sus componentes α_1 y α_2. Y cada fórmula β, a la disyunción de sus componentes β_1, β_2.

Tanto en un caso como en otro, las componentes son subfórmulas de la fórmula principal. Y a partir sólo de ellas y de sus negaciones se construye una conjunción o una disyunción equivalente a la fórmula dada.

α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \vee Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \rightarrow Y)$	X	$\neg Y$	$X \rightarrow Y$	$\neg X$	Y
$\neg(X \leftarrow Y)$	$\neg X$	Y	$X \leftarrow Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\rightarrow Y$	X	$\neg Y$	$\neg(X \not\rightarrow Y)$	$\neg X$	Y
$X \not\leftarrow Y$	$\neg X$	Y	$\neg(X \not\leftarrow Y)$	X	$\neg Y$

Figura 1.21: Notación uniforme

En las tres primeras líneas se utilizan las conectivas básicas que se fijaron en el alfabeto. Las cinco líneas restantes utilizan otras conectivas binarias que se podían haber incorporado igualmente al alfabeto. Si se asume esta notación, incluso utilizando todas estas conectivas, cada fórmulas es simplemente de tipo α o de tipo β. Y, recursivamente, cada una de sus subfórmulas es de uno de estos dos tipos.

Ni el bicondicional ni su negación (la disyunción exclusiva) se pueden escribir como una conjunción o disyunción de sus subfórmulas (negadas o no). Por tanto, no se considerarán conectivas primarias del lenguaje sino abreviaturas.

Tableaux: definición

Ejemplo 1.68 Observe la figura (fig. 1.22) considerando que inicialmente sólo consta del nodo 1. Esto es así porque el conjunto de fórmulas iniciales analizado consta de una única fórmula: $\neg(p \rightarrow (q \wedge r))$. Luego se construye un árbol A que consta de ese único nodo.

Puede comprobarse que la fórmula analizada es de tipo α, en concreto $\neg(X \rightarrow Y)$. Expandamos el árbol A con dos nodos más: α_1 (nodo 2) y α_2 (nodo 3). El resultado es otro árbol A': una rama compuesta por tres fórmulas que son satisfacibles por las mismas interpretaciones.

La fórmula del nodo 3 puede aún expandirse. Es una fórmula de tipo β (disyuntiva), y por tanto para su satisfación basta que una de las dos componentes se satisfaga. Representemos ese hecho bifurcando el árbol en ese punto terminal. El resultado es el árbol A'' final que se observa en la figura (fig. 1.22).

Con las consideraciones anteriores, la fórmula inicial se satisface si y sólo si se satisfacen todas las de la rama 1-4 o las de la rama 1-5. Cuando se recorren, tanto una como otra, no hay señales evidentes de que tales fórmulas no puedan ser satisfechas a la vez. De hecho, para que la rama 1-4 se satisfaga basta considerar q falsa y p verdadero. Y en la rama 1-5, r falso y p verdadero.

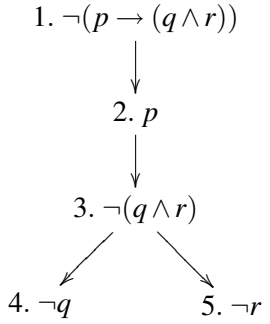


Figura 1.22: Tableau de $\neg(p \rightarrow (q \wedge r))$

Ejemplo 1.69 La figura (fig. 1.23) inicialmente constaba sólo de los nodos 1 y 2. Se pretende analizar la satisfacibilidad del conjunto formado por esas dos fórmulas.

Nos situaremos en el nodo 2, por ser el extremo actual de ese árbol. Y aplicaremos allí la expansión de una de esas dos fórmulas. En concreto, escogemos expandir la fórmula 2 (podía haber sido la 1). La fórmula 2 es de tipo α y produce los nodos 3 y 4.

Nos situamos en el nodo 4 y expandimos otra fórmula. En este caso, la 1, que es de tipo β. Como la satisfacción de esta fórmula puede venir por un lado o por otro, se produce la bifurcación de los nodos 5 y 6. En este punto, las fórmulas iniciales son (simultáneamente) satisfacibles si y sólo si lo son todas las fórmulas de la rama 1-5 o todas las fórmulas de la rama 1-6.

Recorriendo las fórmulas de la rama 1-5 se observa que una de las fórmulas es p mientras otra es $\neg p$. Luego la satisfacibilidad simultánea de todas las fórmulas de esa rama queda descartada (se marca su extremo). Parafraseando a Bogart, 'siempre nos quedará París', que en este caso es la otra rama: el conjunto inicial es satisfacible si y sólo si lo es esta rama.

Como aún se pueden expandir más fórmulas, nos situamos en 6 y lo hacemos. En concreto se expande la propia fórmula 6, que produce los nodos 7 y 8 por ser de tipo conjuntivo. De nuevo, recorriendo esta rama se encuentra uno con r y $\neg r$ y se pierde, por último, la esperanza, que es lo último que se pierde.

El conjunto formado por las dos fórmulas iniciales no es satisfacible.

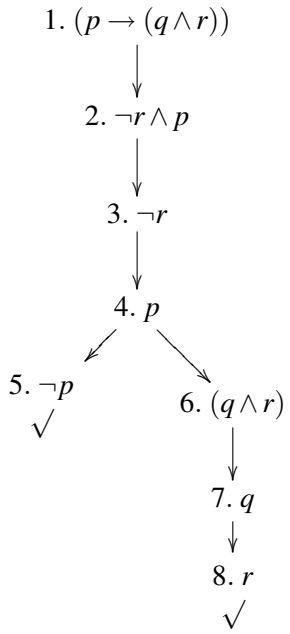


Figura 1.23: Tableau de $\{p \to (q \wedge r), \neg r \wedge p\}$

Definición 1.70 (Tableau de un conjunto de fórmulas) Sea $\{\phi_1, \ldots, \phi_n\}$ un conjunto de fórmulas. Entonces el árbol formado por la única rama $\phi_1 \ldots \phi_n$ es un tableau A para este conjunto.

Si A es un tableau para ese conjunto y A' resulta de aplicar a A alguna de las reglas de expansión de tableaux, entonces A' es un tableau para ese conjunto.

Falta precisar cuáles son las reglas de expansión, dónde y cómo se puede expandir un árbol y si existe un final determinado para este proceso.

Las reglas de expansión son las que se muestran en la tabla (tabl. 2.5). Dado un árbol, se escoge una rama cualquiera del mismo y una fórmula de esa rama que pueda expandirse, que no sea un literal. La expansión se produce en el terminal de esa rama: una bifurcación si la fórmula escogida era disyuntiva o la secuencia de dos nodos si era conjuntiva.

Definición 1.71 (Tableau cerrado) Una rama se dice cerrada si ocurren en ella tanto una fórmula X como una fórmula $\neg X$, o si ocurre la fórmula $\bot$. Está atómicamente cerrado si la fórmula X es atómica o si ocurre $\bot$.

Un árbol se ha cerrado si se han cerrado todas sus ramas. Cuando todos los cierres son atómicos se dice que el árbol se ha cerrado de forma atómica.

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

Tabla 1.20: Reglas de expansión de un tableau

Observe que las reglas de expansión son no deterministas. Se puede escoger la rama y la fórmula siguientes que se van a expandir. Esta opción obliga a plantearse si existen estrategias de elección, heurísticas, más eficientes que otras. En efecto, por regla general, trate de expandir todas las fórmulas α primero.

Previa a esta preocupación por la complejidad, conviene fijar si el proceso para en algún momento. No hay nada en las definiciones que impida expandir la misma fórmula α una y otra vez sobre una rama, puesto que no se elimina. Afortunadamente, para la lógica proposicional abordada, basta utilizar una única vez cada fórmula para garantizar el cierre del tableau (si es que debe cerrarse). No ocurría así en la Resolución.

El sistema descrito es consistente y completo en los mismo términos con que se enunció para Resolución: un conjunto es insatisfacible si y sólo si existe un tableau cerrado del mismo.

Bibliografía complementaria

Entre la bibliografía inicial en castellano, sin ánimo de ser exhaustivos, puede consultar [Badesa et al. 98], [Garrido 95] o [Deaño 93] como primeras lecturas. Son textos enfocados hacia alumnos de filosofía pero le facilitarán una buena compresión de la semántica y de la formalización del lenguaje natural. Más orientados al uso de la lógica en computación, puede considerar [Cuena 85] ó [Kowalski 86]

En [Huth y Ryan 2000], [Ben-Ari 90] ó [Burris 98] puede encontrar una buena introducción a la lógica de proposiciones, desde una perspectiva aplicada. Una presentación más formal puede encontrarse en [Dalen 97] o en [Mendelson 97].

Los sistemas de deducción natural se encuentran claramente descritos en [Broda et al., 94] ó [Huth y Ryan 2000]. En [Fitting 96] se describe la implementación de sistemas basados en resolución o en tableaux, con especial énfasis en éstos.

Actividades y evaluación

El alumno dispone de ejemplos y actividades en el grupo de tutorización telemática del curso, así como exámenes resueltos de años pasados.

Capítulo 2

LÓGICA DE PREDICADOS DE PRIMER ORDEN

Resumen

Este capítulo extiende el lenguaje de la Lógica de Proposiciones mediante la introducción de los cuantificadores ('todos los ...') y la explicitación de propiedades y de relaciones entre términos ('sujetos relacionados'). La sintaxis es ahora más compleja pero mucho más expresiva.

Acorde con esta ampliación sintáctica, tanto los objetos matemáticos necesarios para interpretar una expresión como su proceso de evaluación son ahora más complejos. Es preciso ahora estar pendiente de más detalles, en particular de la posición relativa de las variables respecto a los cuantificadores (si existen) que las referencian.

Como contrapartida, se dispone de un lenguaje y de una semántica sobre la que se está edificando (con algunas limitaciones) gran parte de la matemática y de las teorías formalizadas. Desde el punto de vista computacional, la lógica de predicados es la base de los procesos de representación y de razonamiento. A partir de aquí han derivado otros sistemas particulares, generalmente por razones de eficiencia.

Todos los conceptos semánticos básicos del capítulo anterior se pueden reformular en éste. Desgraciadamente aquí, incluso para la fórmula más sencilla existen infinitas representaciones posibles. Resultan ahora (salvo casos particulares) inabordables los procedimientos de decisión que requerían un recorrido exhaustivo por todas las interpretaciones. Los sistemas deductivos se convierten en la única opción general posible, siempre que se demuestren correctos y completos.

Objetivos

El uso correcto de los diversos lenguajes de primer orden es el primer objetivo. No obstante, es difícil manipular con soltura uno de estos lenguajes sin tener clara su semántica. Así, estos dos objetivos básicos se realimentan en su proceso de consolidación.

El siguiente objetivo debiera ser la compresión y uso de (al menos) los sistemas basados en Resolución, ampliamente utilizados en Computación. Adicionalmente, los sistemas basados en Tableaux se están imponiendo como una eficiente alternativa de implementación.

En el estado actual de estos apuntes aún no se han abordado las estrategias de implementación.

Metodología

Trabaje con lenguajes de primer orden progresivamente más complejos. Primero, sólo cuantificadores y predicados monádicos (luego diádicos, etc.) Introduzca posteriormente el uso de funciones 'para referirse a ciertos sujetos', es decir, como términos (aparte de las variables y constantes). Por último, introduzca la igualdad en su lenguaje.

En cada una de estas etapas, procure cuidar la correcta generación e interpretación de fórmulas (así como de sus árboles sintácticos). Y ponga un especial cuidado en encontrar tanto interpretaciones que satisfagan sus expresiones como interpretaciones que no las satisfagan (si es posible).

Por último, en los sistemas basados en Resolución y Tableaux recuerde, de la lógica de proposiciones, que facilitan la identificación de conjuntos de fórmulas insatisfacibles. Ahora, en estos procesos es importante no introducir espúreamente la insatisfacibilidad por una incorrecta manipulación de los pasos intermedios. En particular, tenga cuidado con los procesos que requieren una particularización de un caso general: si no se siguen las restricciones sobre la elección de esos sujetos particulares (constantes, términos unificables ...) puede afirmarse como insatisfacible un conjunto de fórmulas que no lo era. O perderse indefinidamente en intentos vanos de confirmación, incluso aunque sí pueda llegarse sintácticamente a la misma.

2.1 Sintaxis

2.1.1 Lenguajes de primer orden

Alfabetos

Todos los lenguajes de Lógica de Primer Orden utilizan un conjunto común de símbolos. Además de este conjunto, cada lenguaje utiliza algunos símbolos propios. Los símbolos propios de cada lenguaje determinan qué constantes, funciones y relaciones considera.

Definición 2.1 (Alfabeto) El alfabeto de un lenguaje de Primer Orden incluye:

- símbolos comunes:
  - variables: $Var = \{x_1, x_2, x_3 \ldots\}$
  - conectivas: $\{\bot, \top, \neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$
  - cuantificadores: $\{\forall, \exists\}$
  - símbolos de puntuación: paréntesis y comas
  - (símbolo de igualdad: $\{\approx\}$)
- símbolos propios:
  - su conjunto de constantes: $\mathcal{C} = \{c_1, c_2, \ldots\}$
  - su conjunto de funciones: $\mathcal{F} = \{f_1, f_2, \ldots\}$
  - su conjunto de relaciones: $\mathcal{R} = \{R_1, R_2, \ldots\}$

Notación Se denomina A al conjunto de símbolos comunes, S al de símbolos propios y A_S al alfabeto resultante. Cada elección de S determina un lenguaje: se denotará como $L(S)$ ó $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$.

Los conjuntos $\mathcal{C}$, $\mathcal{F}$ y $\mathcal{R}$ se escogen independientemente: un lenguaje puede o no contener constantes, o funciones o relaciones. Un lenguaje sin relaciones propias debe, al menos, utilizar la relación de igualdad.

Toda función y toda relación tienen asignado un número n. Una función o relación *n-ádica* o *n-aria* se aplica sobre una *n-tupla* de términos: $R(t_1,\ldots,t_n)$.

No todos los lenguajes utilizan la relación de igualdad. Por su especial tratamiento semántico convenía excluirla del conjunto $\mathcal{R}$. En los lenguajes con igualdad, se utilizará excepcionalmente la notación infija $(t_1 \approx t_2)$ en vez de la prefija $\approx (t_1,t_2)$.

Al cuantificador $\forall$ (léase 'para todo') se le denomina *universal* y al cuantificador $\exists$ ('existe'), *existencial*.

En las exposiciones teóricas, cuando son muy pocos los símbolos requeridos, se utilizan como constantes las letras iniciales del alfabeto latino $\{a,b,c,d,\ldots\}$, las letras finales como variables $\{\ldots,u,v,w,x,y,z\}$ y letras intermedias $\{f,g,h,\ldots\}$ como funciones. Para las relaciones se usarán letras mayúsculas.

Ejemplo 2.2 Para representar y analizar el conjunto de números naturales se requieren, al menos, una constante y una función. La constante fija el primer número (el que no es sucesor de otro) y la función proporciona el sucesor de cada número.

Para desarrollar una teoría sobre grafos basta un lenguaje con una única relación binaria (la relación entre nodos). En principio, no se precisa de símbolos constantes ni de funciones.

Las ecuaciones son relaciones de igualdad entre términos: $t_1 \approx t_2$. La lógica ecuacional estudia este fragmento de la lógica de primer orden, donde la única relación precisa es la de igualdad. Los términos se construyen a partir de variables, constantes y funciones.

Lenguajes

Suponga fijado un cierto alfabeto A_S. Todas las definiciones y resultados que siguen se restringen a ese alfabeto. Así, los términos y fórmulas serán expresiones sobre este alfabeto; y las constantes, funciones y relaciones requeridas en su definición deben pertenecer a este alfabeto. En las escasas ocasiones en que se consideren varios alfabetos, se avisará explícitamente.

Definición 2.3 (Término) Un término es una expresión obtenida por aplicación de las siguientes reglas:

1. cada constante c es un término
2. cada variable x es un término
3. si f es una función n-aria y $t_1,\ldots,t_n$ son términos, entonces $f(t_1,\ldots,t_n)$ es un término

Notación El conjunto de todos los términos se denotará como $Term$.

Definición 2.4 (Fórmula atómica) Una fórmula atómica es una expresión de la forma:

$R(t_1,\ldots,t_n)$, donde R es un símbolo relacional n-ario y $t_1,\ldots,t_n$ son términos

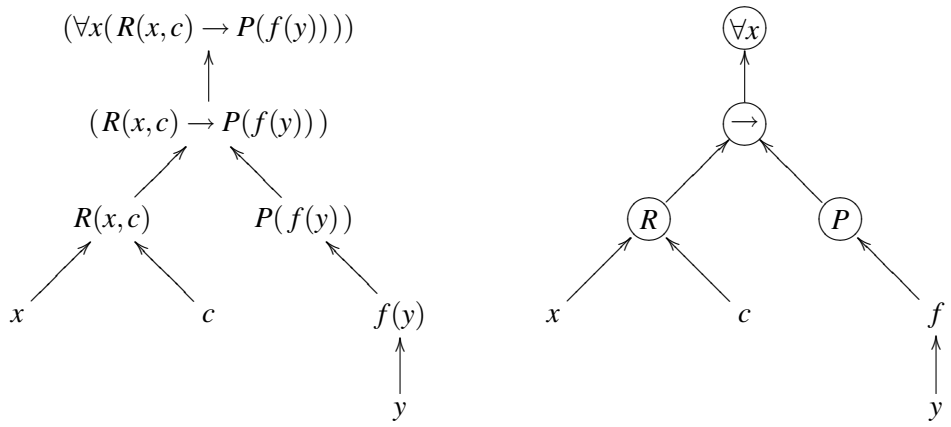
En los lenguajes con igualdad también es una fórmula atómica $(t_1 \approx t_2)$, donde t_1 y t_2 son términos. Asimismo en los lenguajes que utilicen los símbolos $\bot$ y $\top$ (que se pueden entender como conectivas 0-arias), ambas fórmulas serán fórmulas atómicas.

Notación Al conjunto de todas las fórmulas atómicas se le denominará $Atom$.

Definición 2.5 (Fórmula) Una fórmula es una expresión obtenida por aplicación de las siguientes reglas:

1. toda fórmula atómica es una fórmula
2. si ϕ es una fórmula entonces $(\neg\phi)$ es una fórmula
3. si ϕ y ψ son fórmulas entonces $(\phi * \psi)$ es una fórmula, para toda conectiva binaria $*$
4. si ϕ es un fórmula y x una variable entonces $(\forall x\phi)$ y $(\exists x\phi)$ son fórmulas

Notación El conjunto de todas las fórmulas se denotará como *Form*.



(a) árbol sintáctico (b) árbol, más esquemático

Figura 2.1: Fórmula de Primer Orden: árbol sintáctico

Ejemplo 2.6 En la figura (fig. 2.1) se representa el árbol sintáctico de la fórmula $(\forall x(R(x,c) \rightarrow P(f(y))$. Se han marcado flechas ascendentes para resaltar el proceso de generación:

- x (variable) y c (constante) son términos; si R es un símbolo relacional diádico, $R(x,c)$ es una fórmula atómica.
- y es un término; si f es un símbolo monádico de función, $f(y)$ es un término; si P es un símbolo relacional monádico, $P(f(y))$ es una fórmula atómica.
- el condicional de dos fórmulas es una fórmula: $(R(x,c) \rightarrow P(f(y)))$
- la cuantificación de una fórmula es una fórmula: $(\forall x(R(x,c) \rightarrow P(f(y))))$

En la subfigura derecha se han enmarcado con un círculo los nodos que representan una fórmula. El resto de los nodos representan términos.

Ejemplo 2.7 Todas las expresiones siguientes son términos, no fórmulas: x, z, c, $f(x)$, $f(c)$, $g(x,y)$, $g(x,c)$, $g(f(y),x)$, $h(x,c,g(x,y),f(z))$. Trate de expresar el árbol sintáctico de este último término.

Los términos son los 'sujetos', los 'individuos' citados en nuestras frases formales. Sobre ellos no se podrá afirmar si son verdaderos o falsos, sólo tendrá sentido determinar 'quiénes son'.

Ejemplo 2.8 Todas las expresiones siguientes son fórmulas:

- $P(x)$, $Q(y)$, $(P(x) \to (\neg Q(y)))$, $((P(c) \wedge R(x,y)) \vee Q(y))$
- $(P(f(x)) \to (\neg Q(f(c))))$, $R(g(x,f(y)),f(c))$
- $\forall x(P(x) \to (\neg Q(y)))$, $\forall x(P(x) \to \exists y(\neg Q(y)))$, $\exists y \forall x R(g(x,f(y)),f(c))$

Asegúrese de que puede expresar el árbol sintáctico de cualquiera de estas fórmulas. Los anidamientos relativos de los paréntesis facilitan la determinación de los nodos hijo.

2.1.2 Inducción y recursión

Cada lenguaje de primer orden tiene una estructura inductiva. Ya se consideró esta estructura en el estudio de los lenguajes proposicionales. Recuerde que permitía enunciar definiciones de manera concisa, recursiva, sobre el conjunto infinito de fórmulas. También facilitaba un mecanismo de prueba, por inducción estructural, de propiedades de estas fórmulas. Estrictamente, en los lenguajes de primer orden hay dos conjuntos inductivos: el de términos y el de fórmulas.

Definición 2.9 (Principio de inducción estructural) Para demostrar que todos los términos de un lenguaje de primer orden tienen la propiedad P basta demostrar que:

1. toda variable tiene la propiedad P
2. toda constante del alfabeto tiene la propiedad P
3. si los términos $t_1,\ldots,t_n$ tienen la propiedad P y f es una función n-aria del alfabeto, entonces $f(t_1,\ldots,t_n)$ tiene la propiedad P

Para demostrar que todas las fórmulas tienen la propiedad P basta demostrar que:

1. toda fórmula atómica tiene la propiedad P
2. si la fórmula ϕ tiene la propiedad P, entonces $(\neg\phi)$ tiene la propiedad P
3. si las fórmulas ϕ y ψ tienen la propiedad P, entonces $(\phi * \psi)$ tiene la propiedad P
4. si la fórmula ϕ tiene la propiedad P y x es una variable, entonces tanto $(\forall x\phi)$ como $(\exists x\phi)$ tienen la propiedad P

El Principio de Inducción estructural puede demostrarse a partir del conocido Principio de Induccion sobre números naturales: aplicándolo sobre la longitud de cada expresión, sobre su número de símbolos. Entre las propiedades de interés que pueden demostrarse inductivamente resaltaremos la unicidad en la descomposición sintáctica de un término y de una fórmula.

Teorema 2.10 (Análisis sintáctico único) Cada término pertenece a una y sólo una de las siguientes categorias:

1. es una variable
2. es una constante de A_S
3. es de la forma $f(t_1,\ldots,t_n)$ para una función n-aria f de A_S y términos $t_1,\ldots,t_n)$ unívocamente determinados

Cada fórmula ϕ pertenece a una y sólo una de las siguientes categorias:

1. ϕ es atómica
2. ϕ es de la forma $(\neg\psi)$, para una ψ determinada
3. ϕ es de la forma $(\psi * \chi)$, para ψ , χ y conectiva binaria determinadas
4. ϕ es de la forma $(\forall x\psi)$, o de la forma $(\exists x\psi)$

   para cuantificador, variable y fórmula ψ determinados

Este resultado permite escribir *el* árbol de un término o de una fórmula, garantizando que tendrá estructura de árbol y que a cada fórmula le corresponde exactamente uno.

Observe que en el ejemplo (ej. 2.6), para generar una fórmula, se partía de varios nodos inconexos y se ascendía, intentando construir un árbol. Ahora, de arriba abajo, el proceso de análisis garantiza que, si la expresión es un término o una fórmula, se produce un árbol.

2.1.3 Subfórmulas

Definición 2.11 (Subfórmulas) Dada una fórmula ϕ, el conjunto de todas sus subfórmulas se define recursivamente como:

$$subform(\phi) = \begin{cases} \{\phi\} & ,\phi \text{ atómica} \\ \{\phi\} \cup subform(\psi) & ,\phi = (\neg\psi) \\ \{\phi\} \cup subform(\psi) \cup subform(\chi) & ,\phi = (\psi * \chi) \\ \{\phi\} \cup subform(\psi) & ,\phi = (\forall x\psi) \\ \{\phi\} \cup subform(\psi) & ,\phi = (\exists x\psi) \end{cases}$$

Las subfórmulas de una fórmula dada son todas las que aparecen en su árbol sintáctico, incluida ella misma. Observe que no todos los nodos de este árbol representan fórmulas: cada fórmula atómica tiene por subárbol el árbol de todos sus términos.

Ejemplo 2.12 En la figura (fig. 2.2), la fórmula analizada a la izquierda, tiene 4 subfórmulas: todas las de sus nodos salvo los términos x, c, $f(y)$, y. La fórmula analizada a la derecha tiene 5 subfórmulas, todas las de sus nodos salvo los términos: x, y, y.

Esta definición (def. 2.11) es un ejemplo de definición recursiva sobre el conjunto de fórmulas. En particular, define una función $subform : Form \mapsto \mathcal{P}(Form)$ de fórmulas en subconjuntos de fórmulas.

Observe que la definición de $subform(\phi)$ requiere 5 funciones previas que facilitan la imagen de ϕ según ϕ se encuentre en una de los 5 categorías en que puede estar. En realidad, son más las categorías: todas la conexiones binarias se han condensado en una línea y las dos opciones de fórmulas atómicas también. Una función recursiva que, por ejemplo, calcule el número de conjunciones de una fórmula necesita diferenciar en dos la línea única dedicada a conectivas binarias.

Ya se ha garantizado que una fórmula pertence a una y sólo una de esas categorías. Además, el Principio de Recursión estructural garantiza que, para una elección dada de esas funciones previas en cada categoría, la función $f : Form \mapsto X$ está bien definida y es única. Y esto, sea cual sea el conjunto X. Puede intentar definir recursivamente una función del conjunto de fórmulas *Form* sobre un conjunto de 2 equipos de fútbol o sobre los 3 colores de un semáforo.

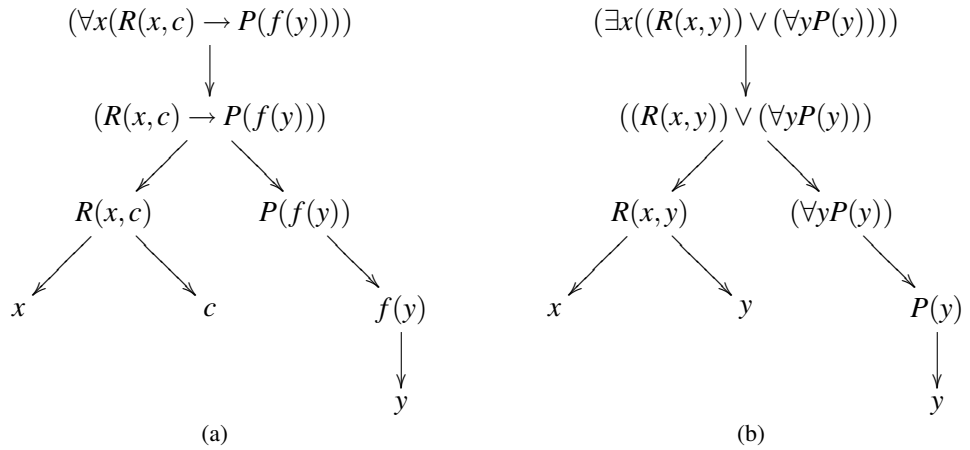


Figura 2.2: Dos fórmulas de Primer Orden: subfórmulas

Ejemplo 2.13 La función $numsubf : Form \mapsto N$ calcula el número de subfórmulas de una dada:

$$numsubf(\phi) = \begin{cases} 1 & ,\phi \text{ atómica} \\ 1 + numsubf(\psi) & ,\phi = (\neg\psi) \mid (\forall x\psi) \mid (\exists\psi) \\ 1 + numsubf(\psi) + numsubf(\chi) & ,\phi = (\psi * \chi) \end{cases}$$

2.1.4 Eliminación de paréntesis

La sintaxis descrita, con todo el rigor necesario, permite:

1. evitar un lenguaje ambiguo, donde una fórmula se pueda descomponer en más de una forma distinta
2. comprobar la correcta definición de conceptos
3. describir procedimientos para el cálculo efectivo de estas definiciones (subfórmulas, variables libres, etc.) y de otras, relativas a la interpretación de las fórmulas.

Todo esto es aún más crítico si se pretenden diseñar sistemas automáticos que utilicen las lógicas descritas. Como contrapartida, el exceso de paréntesis dificulta la comprensión de un lector, de un 'agente lógico humano'. Los convenios de precedencia, como los fijados para la lógica proposicional, permiten eliminar paréntesis sin caer en la ambigüedad: en caso de duda, el convenio es todo lo que se necesita para deshacerla.

1. Se prescindirá de los paréntesis externos de la fórmula. También de los que rodean a una expresión cuantificada $(\forall xP(x))$ y a una fórmula negada $(\neg P(x))$. Asimismo se prescinde de los paréntesis que delimitan los términos de una relación $R(x,y,z)$. Se podría prescindir de los que delimitan los términos de una función $g(x,y)$, pero preferimos conservarlos.
2. En caso de duda, se aplicarán primero los cuantificadores y las negaciones, después las conjunciones y disyunciones, y luego los condicionales y bicondicionales. No se establece precedencia entre conjunciones y disyunciones o entre condicionales y bicondicionales.

3. Gracias a la propiedad asociativa de la conjunción, en una expresión ambigua $\phi_1 \wedge \phi_2 \wedge \phi_3$ no importará (a efectos semánticos) si se ha evaluado como $(\phi_1 \wedge \phi_2) \wedge \phi_3$ o como $\phi_1 \wedge (\phi_2 \wedge \phi_3)$. Luego, en una conjunción de n fórmulas se eliminarán los paréntesis. Lo mismo ocurre para la disyunción.

Ejemplo 2.14 Con este convenio, una expresión como

- $\forall xPx \rightarrow Qx$ es una abreviatura de la fórmula $((\forall xP(x)) \rightarrow Q(x))$ y no de $(\forall x(P(x) \rightarrow Q(x)))$
- $\neg Px \vee Qy$ es una abreviatura de la fórmula $((\neg P(x)) \vee Q(y))$ y no de $(\neg(Px \vee Q(y)))$
- $\forall xPx \wedge Qx \wedge Rxyz \rightarrow Sxy$ es una abreviatura de la fórmula $(((((\forall xP(x)) \wedge Q(x)) \wedge R(x,y,z)) \rightarrow S(x,y)))$

2.1.5 Variables libres

No es difícil definir una función que produzca todas las variables de una fórmula dada. Para ello basta definir el conjunto de las variables de una fórmula en función de los conjuntos de variables de sus fórmulas componentes. Al llegar a las fórmulas atómicas, sus variables son las que hayan aparecido en sus términos.

Ejemplo 2.15 La función $var_t : Term \mapsto \mathcal{P}(Var)$ calcula el conjunto de variables que aparecen en un término:

$$var_t(t) = \begin{cases} \emptyset & ,t = c \\ \{x\} & ,t = x \\ var_t(t_1) \cup \ldots \cup var_t(t_n) & ,t = f(t_1, \ldots, t_n) \end{cases}$$

La función $var : Form \mapsto \mathcal{P}(Var)$ calcula el conjunto de variables que aparecen en una fórmula:

$$var(\phi) = \begin{cases} var_t(t_1) \cup var_t(t_2) & \phi = (t_1 \approx t_2) \\ var_t(t_1) \cup \ldots \cup var_t(t_n) & \phi = R(t_1, \ldots, t_n) \\ var(\psi) & \phi = (\neg\psi) \mid (\forall x\psi) \mid (\exists x\psi) \\ var(\psi) \cup var(\chi) & \phi = (\psi * \chi) \end{cases}$$

Ámbito de un cuantificador Si una fórmula es de la forma $(\forall x\phi)$ o $(\exists x\phi)$ se dice que ϕ es el *ámbito de ese cuantificador*. En la figura (fig. 2.3a), toda la fórmula condicional que ocurre como subárbol de $\forall x$ determina el ámbito de este cuantificador. En la figura (fig. 2.3b) hay dos cuantificadores: el ámbito de $\forall y$ es la fórmula atómica Py, y está incluido en el ámbito de $\exists x$.

Los ámbitos de los cuantificadores (sus respectivos subárboles) no se solapan: se anidan, como en el ejemplo de la figura (fig. 2.3b) o son disjuntos, como en $((\forall xPx) \vee (\forall yQy))$.

Apariciones libres y ligadas Todas las apariciones de una variable x, en el ámbito de un cuantificador para esa variable, $(\forall x\phi)$ o $(\exists x\phi)$, se denominan *ligadas*. Así, en una fórmula sin cuantificadores ninguna variable será ligada. Y todo cuantificador liga, a lo sumo, las apariciones de una variable en su ámbito.

En la figura (fig. 2.3b) hay 5 fórmulas distintas. Veámos cuál es el comportamiento de las variables en esas fórmulas:

- en las fórmulas atómicas $R(x,y)$ y $P(y)$ ninguna aparición de variable es ligada

- en la fórmula $(\forall y P y)$ la aparición de y es ligada
- en la fórmula disyuntiva $(R(x,y) \vee (\forall y P y))$ sólo es ligada la última aparición de la variable y
- en la fórmula $(\exists x(R(x,y) \vee (\forall y P y)))$ todas las apariciones son ligadas salvo la de la primera y

Observe que a cada (sub)fórmula le corresponde un nodo del árbol sintáctico. Para determinar si una variable está ligada *en esa (sub)fórmula* basta ascender desde el nodo de esa variable hasta el nodo de esa (sub)fórmula: si se encuentra en esa rama un cuantificador aplicable a esa variable, entonces está ligada en esa (sub)fórmula.

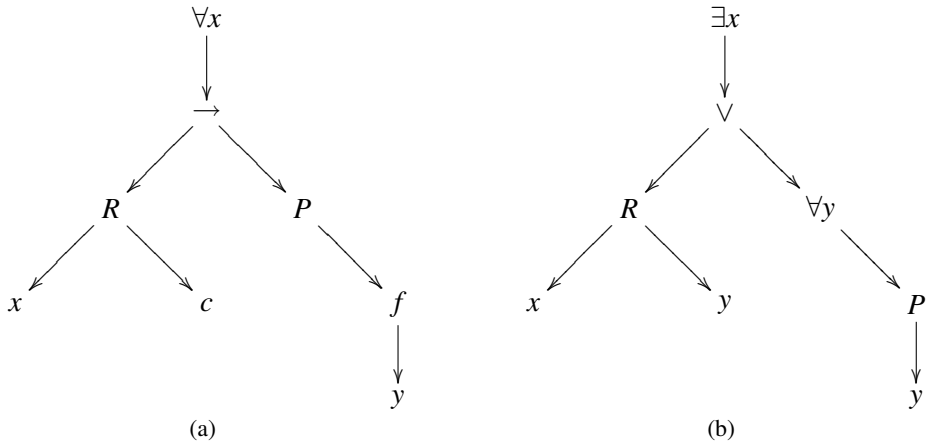


Figura 2.3: Ámbitos y variables libres

Si la aparición de una variable no es ligada, se la denomina *libre*. Una aparición es ligada o libre; pero, como se observa en la fórmula raíz de 2.3, una variable puede tener apariciones libres y ligadas en una misma fórmula:

$$(\exists x(R(x,\underline{y}) \vee (\forall y P \underline{y})))$$

Una variable se denomina libre en una fórmula si *todas sus apariciones* en esa fórmula son libres. En la fórmula precedente ni x ni y son variables libres.

Definición 2.16 (Variables libre) El conjunto de variables libres de una fórmula se define como la siguiente función, $libres : Form \mapsto \mathcal{P}(Var)$, de fórmulas en conjuntos de variables:

$$libres(\phi) = \begin{cases} var(t_1) \cup var(t_2) & \phi = (t_1 \approx t_2) \\ var(t_1) \cup \ldots \cup var(t_n) & \phi = R(t_1, \ldots, t_n) \\ libres(\psi) & \phi = (\neg\psi) \\ libres(\psi) \cup libres(\chi) & \phi = (\psi * \chi) \\ libres(\psi) - \{x\} & \phi = (\forall x\psi) \mid (\exists x\psi) \end{cases}$$

Definición 2.17 (Sentencia) Una fórmula sin variables libres se denomina sentencia.

En una sentencia toda variable está en el ámbito de un cuantificador para esa variable.

2.1.6 Sustituciones

Una fórmula de primer orden tiene siempre estructura de árbol. El desarrollo de este árbol, desde la fórmula inicial, llega necesariamente a las subfórmulas elementales: las subfórmulas atómicas. Un ejemplo de fórmula atómica es $Q(x,a)$ ó $Q(x,g(a))$; en general: un predicado n-ario aplicado a n términos.

Cada uno de estos términos (de estos 'sujetos' del predicado) admite asimismo un desarrollo en árbol. Un término puede ser simplemente una variable (y), o una constante (b) o una función k-aria aplicada a k términos: $f(a,x,g(b,x))$. Observe que uno de los tres términos sobre los que se aplica la función f es, a su vez, otra función binaria g aplicada sobre dos términos.

> La operación sintáctica que se propone en esta sección se aplicará sólo sobre términos. Más específicamente, sustituirá términos que sean *variables* por otros términos.

El lector ya ha utilizado sustituciones y reemplazos de fórmulas, no de términos. Por ejemplo, cuando reemplaza un condicional por la fórmula equivalente '$\neg$ *antecedente* $\vee$ *consecuente*'. En cada caso, este proceso de sustitución debía ajustarse a unas normas, a unas restricciones. Como contrapartida, se garantizaba que el resultado obtenido seguía manteniendo cierta propiedad determinada, si ésta existía en la fórmula inicial.

La sustitución de variables por términos también se ajustará a ciertas restricciones: las que se necesiten para que sea útil. En particular, la utilidad básica consistirá en descubrir, por manipulación sintáctica, si un mismo predicado (aplicado a términos distintos) se está refiriendo, implícitamente, 'a los mismos elementos del universo'.

Sustituciones de variables

Definición 2.18 (Sustituciones) Una sustitución σ_v es una función $\sigma_v : Var \mapsto Term$, del conjunto de *variables* en el conjunto de términos.

Observe que esta definición depende del lenguaje $L(\mathcal{C},\mathcal{F},\mathcal{R})$ empleado. Si no existen constantes ni funciones en el lenguaje, los únicos términos posibles son los formados por una variable. En este caso, se trata de sustituir una variable por otra.

Ejemplo 2.19 Considere un lenguaje $L(\mathcal{C},\mathcal{F},\mathcal{R})$ tal que el conjunto de constantes del lenguaje es $\mathcal{C} = \{a,b\}$ y el conjunto de funciones es $\mathcal{F} = \{g\}$ (donde g es binaria).

Una sustitución σ_v fija, para cada variable, qué término de este lenguaje le corresponde. Por ejemplo:

$$\ldots\ \sigma_v(v) = a, \quad \sigma_v(w) = a, \quad \sigma_v(x) = y, \quad \sigma_v(y) = g(x,z), \quad \sigma_v(z) = g(g(b,y),g(a,g(z,w))),\ \ldots$$

Los puntos suspensivos anteriores responden al hecho de que en todo lenguaje de primer orden (salvo que se indique lo contrario) el número de variables es infinito. La sustitución σ_v asigna a cada una de ellas un término.

Ejemplo 2.20 La sustitución del ejemplo anterior no estaba bien definida: no se especificaba la imagen de cada variable (salvo de unas pocas). La forma más sencilla de fijar la imagen de las restantes es precisar que 'no se modifican'. Es decir, que $\sigma_v(u) = u$, que cada una de estas variables se sustituye por sí misma.

En este caso, basta enumerar las variables que 'sí se modifican'. Una sustitución como la del ejemplo anterior se denotará como:

$$\sigma_v = [v/a,\ w/a,\ x/y,\ y/g(x,z),\ z/g(g(b,y),g(a,g(z,w)))]$$

entendiendo que, para el resto de variables, $\sigma_v(u) = u$.

Para este ejemplo, tan sólo 5 variables se sustituyen por un término distinto de sí mismas. Cuando el número de estas variables sea finito se dirá que la sustitución tiene un *soporte finito*.

Sustituciones en términos

Dado un término como $f(x,a)$ y una sustitución σ_v tal que $\sigma_v(x) = b$, el resultado de aplicar tal sustitución es $f(b,a)$. Es decir, dado un término 'de entrada', una sustitución determina un único término 'de salida'. Se puede ampliar cada función σ_v (de variables en términos) a otra función σ_t (de términos en términos).

Definición 2.21 (Sustituciones: extensión a términos) Sea $\sigma_v : Var \mapsto Term$ una sustitución, se define entonces recursivamente una función $\sigma_t : Term \mapsto Term$ tal que:

- $x\sigma_t = \sigma_v(x)$, para cada variable x del lenguaje
- $c\sigma_t = c$, para cada constante c del lenguaje
- $[f(t_1,\ldots,t_n)]\sigma_t = f(t_1\sigma_t,\ldots,t_n\sigma_t)$ para cada función n-aria f del lenguaje.

Notación Escribiremos (término)σ o, más adelante, (fórmula)σ, para designar la imagen de la sustitución, en lugar de σ(término) ó σ(fórmula). En particular, escribiremos $x\sigma$ para referirnos al término imagen de la variable x.

Desafortunadamente, cada texto utiliza una notación particular para denotar sustituciones. Así, en unos se usa (término)σ y en otros σ(término). Además, la sustitución de la variable x por el término t se puede encontrar como: $[x/t]$ ó $[t/x]$ ó $(x \leftarrow t)$ ó σ^x_t, entre otras notaciones.

Ejemplo 2.22 Sea la sustitución sobre variables $\sigma_v = [x/f(y,a),\ y/z]$. Entonces, para cada uno de los cuatro siguientes términos su imagen es:

- $w\sigma_t = w$, donde w es una variable distinta de x e y
- $a\sigma_t = a$, donde a es una constante
- $(h(a,x,w))\sigma_t = h(a\sigma_t, x\sigma_t, w\sigma_t) = h(a, f(x\sigma_v, a), w\sigma_v) = h(a, f(y,a), w)$
- $(h(a,f(x,y),w)\sigma_t = h(a\sigma_t, (f(x,y))\sigma_t, w\sigma_t) = h(a, f(x\sigma_v, y\sigma_v), w\sigma_v) = h(a, f(f(y,a),z), w)$

Trate de visualizar, de dos maneras, una sustitución sobre términos sobre el árbol de *un término* (aún no de una fórmula). Primero, de forma recursiva, desde la raíz a los nodos hoja. Alternativamente, efectuando la sustitución sobre las nodos hoja y propagándola hasta el nodo raíz.

Composición de sustituciones

Los modernos procesadores de texto permiten (casi) simular este proceso de sustitución. Dado un texto como $f(x,b,g(x,c))$ se puede requerir la sustitución *en todas las apariciones* de la variable x por la cadena $g(x,b)$. El resultado sería $f(g(x,b),b,g(g(x,b),c))$. Observe que:

- la sustitución afecta a todas las apariciones de x en el texto original,
- no se modifican las nuevas apariciones de x (si las hubiera) introducidas por la sustitución.

Una única sustitución que afecta varias variables Si el texto inicial hubiera sido $f(x,b,g(x,z))$ se podía haber requerido *simultáneamente* la sustitución de las x por $g(z,b)$ y la de las z por a. Se habría obtenido entonces la cadena $f(g(z,b),b,g(g(z,b),a))$.

Esta operación no se encuentra generalmente en un procesador de texto. Supone aplicar una única sustitución σ_t, pero σ_t asigna a cada variable el término que debe sustituirla. Y la sustitución de cada variable se produce simultáneamente, sin interrelación entre ellas. No se sustituyen primero las x y luego, sobre ese resultado, se produce la sustitución de las z (las originales y las que aparecen tras sustituir x). Compruebe, sobre la cadena ejemplo, que este proceso conduce a otro resultado.

Una sustitución tras otra Si se dispone de dos sustituciones σ_1 y σ_2 se pueden *componer*. Dada una cadena de entrada, se puede aplicar una de ellas (p.ej.,σ_1), sustituyendo simultáneamente todas las variables de la cadena por el término asignado por σ_1. Después, sobre el resultado final, se producen las sustituciones fijadas por σ_2.

Definición 2.23 (Composición de sustituciones) Si σ_1 y σ_2 son sustituciones sobre términos, la composición $\tau = \sigma_1\sigma_2$ es otra composición sobre términos definida como:

$$t\tau = t[\sigma_1\sigma_2] = (t\sigma_1)[\sigma_2] = ((t\sigma_1)\sigma_2)$$

donde t es un término.

Es decir, sustituya en t primero (y simultáneamente) cada variables por su término sustituyente fijado en σ_1. Y, sobre la cadena resultante, sustituya simúltáneamente cada variable (aunque hayan aparecido tras la sustitución anterior) por su término en σ_2. El resultado es óbviamente otro término, al que denominaremos $t\sigma_1\sigma_2$.

En la definición se ha denominado τ a esta nueva sustitución para resaltar que efectivamente es una única sustitución, aunque construida a partir de dos previas. Generalmente, a esta nueva sustitución se le denomina simplemente '$\sigma_1\sigma_2$', nombre que explicita su génesis. El término correspondiente a t por esta única sustitución (compuesta a partir de las anteriores) es $t\tau$ ó $t[\sigma_1\sigma_2]$.

Observe que, con nuestra notación postfija para composiciones, en $x\sigma_1\sigma_2$ es σ_1 la que se aplica primero (la más próxima a la cadena original). Ésta misma composición, en notación prefija, se escribiría como $\sigma_2\sigma_1 x$.

Ejemplo 2.24 Dadas las sustituciones $\sigma_1 = [x/f(z,a),\ y/w]$, $\sigma_2 = [x/b,\ z/g(w)]$, aplicando la definición previa:

- $x[\sigma_1\sigma_2] = (x\sigma_1)[\sigma_2] = f(z,a)[\sigma_2] = f(z\sigma_2, a\sigma_2) = f(g(w),a)$
- $y[\sigma_1\sigma_2] = (y\sigma_1)[\sigma_2] = w[\sigma_2] = w\sigma_2 = w$
- $z[\sigma_1\sigma_2] = (z\sigma_1)[\sigma_2] = z[\sigma_2] = z\sigma_2 = g(w)$
- $w[\sigma_1\sigma_2] = (w\sigma_1)[\sigma_2] = w[\sigma_2] = w\sigma_2 = w$
- $h(a,y,x)[\sigma_1\sigma_2] = ((h(a,y,x))\sigma_1)[\sigma_2] = h(a,w,f(z,a))[\sigma_2] = h(a\sigma_2, w\sigma_2, (f(z,a))\sigma_2) = h(a,w,f(g(w),a))$

De los resultados previos se sigue que la sustitución $\sigma_1\sigma_2$ es:

$$\sigma_1\sigma_2 = [x/f(g(w),a),\ y/w,\ z/g(w)]$$

En general:

1. cualquier variable, como w, que no se sustituya ni en σ_1 ni en σ_2 tampoco se verá alterada por $\sigma_1\sigma_2$. Es decir, $w\sigma_1\sigma_2 = w$

2. todas las variables sustituidas en σ_1 resultarán sustituidas en $\sigma_1\sigma_2$. Observe cómo se llega tanto a $x\sigma_1\sigma_2 = f(g(w),a)$ como a $y\sigma_1\sigma_2 = w$

3. todas las variables sustituidas en σ_2 que no lo fueran en σ_1, también serán sustituidas en $\sigma_1\sigma_2$. Observe cómo se llega a $z\sigma_1\sigma_2 = g(w)$

Obviamente, una vez que se dispone de la definición de esta nueva sustitución $\tau = \sigma_1\sigma_2$

$$\tau = \sigma_1\sigma_2 = [x/f(g(w),a),\ y/w,\ z/g(w)]$$

se puede aplicar directamente sobre cualquier término:

$$h(a,y,x)[\sigma_1\sigma_2] = (h(a,y,x))\tau = h(a\tau, y\tau, x\tau) = h(a,w,f(g(w),a))$$

Proposición 2.25 Sean σ_1 y σ_2 dos sustituciones con soporte finito, tales que $\sigma_1 = [x_1/t_1,\ldots,x_n/t_n]$ y $\sigma_2 = [y_1/u_1,\ldots,y_k/u_k]$. Entonces, la sustitución compuesta $\sigma_1\sigma_2$ tiene soporte finito, y viene definida por:

$$\sigma_1 = [x_1/(t_1\sigma_2),\ldots,x_n/(t_n\sigma_2),\ z_1/(z_1\sigma_2),\ldots,z_m/(z_m\sigma_2)]$$

donde $x_1,\ldots,x_n$ son todas las variables modificadas por la primera sustitución y $z_1,\ldots,z_m$ son sólo algunas de las variables $y_1,\ldots,y_k$, en particular, aquellas variables 'nuevas', no incluidas entre las $x_1,\ldots,x_n$.

En el ejemplo previo, las variables modificadas por la primera sustitución eran x e y. Y las variables sólo modificadas por la segunda se limitaban a z. Observe qué términos se asignan a cada una.

Proposición 2.26 La composición de sustituciones es asociativa:

$$(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$$

> En general, la composición no es conmutativa. Es decir, habitualmente $\sigma_1\sigma_2$ producirá una sustitución distinta a $\sigma_2\sigma_1$ (salvo excepciones).

Ejemplo 2.27 Sean las sustituciones:

$$\sigma_1 = [x/f(y),\ y/w,], \sigma_2 = [x/g(w),\ z/b], \sigma_3 = [y/b,\ w/f(c),\ v/w]$$

entonces:

$$\sigma_1\sigma_2 = [x/f(y),\ y/w,\ z/b]$$

y

$$(\sigma_1\sigma_2)\sigma_3 = [x/f(b),\ y/f(c),\ z/b,\ w/f(c),\ v/w]$$

Sustituciones en fórmulas

Cuando se produce la sustitución de las variables de un término el resultado es otro término. Esto nos llevó a extender el concepto de sustitución hasta una función de términos en términos. De igual forma, cuando se sustituyen las variables de una fórmula, el resultado es otra fórmula. De ahí, que la simple sustitución definida inicialmente pueda ser extendida como función de fórmulas en fórmulas.

Ahora bien, una vez que somos capaces de calcular cualquier sustitución, nos limitaremos a utilizar sólo 'unas pocas'. Aquellas que nos garanticen 'un buen comportamiento'. En particular:

1. tan sólo sustituiremos las apariciones libres de las variables
2. las apariciones de las variables que aporte cada término sustituyente deben resultar libres en la fórmula final

Este camino formal tiene una recompensa valiosa:

> Las sustituciones de variables en fórmulas, con las restricciones mencionadas, producen nuevas fórmulas que son tan satisfacibles (positiva o negativamente) como las de partida.

Este resultado será fundamental para decidir relaciones de consecuencia via insatisfacibilidad. En concreto, esta es la estrategia que se utiliza en sistemas deductivos tales como los basados en Resolución o los basados en Tablas Semánticas.

Sustituir sólo en apariciones libres Las definiciones siguientes tienen por objetivo incluir esta restricción en la propia definición de 'sustitución en una fórmula'.

Definición 2.28 Sea σ_t una sustitución sobre términos. Se puede entonces extender, recursivamente, a una función σ de fórmulas en fórmulas:

1. $(Q(t_1,\ldots,t_n))\sigma = Q(t_1\sigma_t,\ldots,t_n\sigma_t)$
2. $\top\sigma = \top, \quad \bot\sigma = \bot,$
3. $(\neg X)\sigma = \neg(X\sigma)$
4. $(X * Y)\sigma = (X\sigma * Y\sigma)$, para toda conectiva binaria $*$
5. $(\forall xX)\sigma = \forall x(X\sigma_x), \quad (\exists xX)\sigma = \exists x(X\sigma_x)$

Donde σ_x es una sustitución sobre fórmulas igual a σ salvo el hecho de que no modifica la variable x.

Ejemplo 2.29 Sea la sustitución $\sigma = [x/f(y),\ y/b]$. Entonces

- $(\forall x(Qx \to Rxy))\sigma = \forall x((Qx \to Rxy)\sigma_x) = \forall x((Qx)\sigma_x \to (Rxy)\sigma_x) = \forall x(Qx \to Rxb)$
- $(Qx \to \forall xRxy)\sigma = (Qx)\sigma \to (\forall xRxy)\sigma = Q(f(y)) \to \forall x((Rxy)\sigma_x) = Q(f(y)) \to \forall xRxb$
- $(\forall x(Qx \to \forall yRxy))\sigma = \forall x((Qx \to \forall yRxy)\sigma_x) = \forall x((Qx)\sigma_x \to (\forall y(Rxy))\sigma_x) = \forall x(Qx \to \forall y((Rxy)\sigma_{xy})) = \forall x(Qx \to \forall yRxy)$

Observe que se ha incorporado, en la propia definición de sustitución sobre fórmulas, la primera de las restricciones: dada una sustitución, aplíquese sólo sobre las apariciones libres de las variables.

Notación Suponga una fórmula ϕ. Resaltaremos, cuando interese, que existen apariciones libres de, p.ej., las variables x e y, mediante la notación $\phi(x,y)$.

Se denotará como $\phi(x/t_1, y/t_2)$ la fórmula *resultado* de aplicar una sustitución a ϕ que sustituye las x libres por t_1 y las y libres por t_2.

Usualmente, la notación $\phi(x,y)$ no requiere que estas dos sean las únicas variables con apariciones libres. A veces no es preciso enumerar todas, cuando lo que realmente es significativo para la demostración o el desarrollo es que al menos esas dos tengan apariciones libres.

En una fórmula del tipo $\forall x\phi$ no existen variables x libres. Así, ninguna sustitución, tal y como se ha definido, modificará las x. Por otro lado, las apariciones de x que estuvieran ligadas por el cuantificador resultan apariciones libres en la subfórmula ϕ (si sólo se considera ésta, sin el cuantificador previo).

Más adelante, dada una fórmula como $\forall x\phi$ y una sustitución, se le pedirá que la efectúe sobre ϕ y observará en los ejemplos que sí se modifican las x. Observe, en esos casos, que se requiere la sustitución sobre la fórmula ϕ y no sobre la fórmula $\forall x\phi$.

Ejercicio 2.30 Aplique la sustitución $\sigma = [x/f(y),\ y/w]$ sobre las fórmulas:

- $Px \rightarrow Qy$
- $Px \rightarrow \forall yQy$
- $\exists xPx \rightarrow \forall yQy$

Producir sólo apariciones libres La utilidad de esta restricción (como de la anterior) se entenderá cuando se explique la semántica de las fórmulas de Primer Orden.

Definición 2.31 (Sustitución libre para una fórmula) Una sustitución se denomina libre para una fórmula cuando todas las apariciones de variables introducidas por la sustitución en esa fórmula resultan libres.

Ejemplo 2.32 La sustitución $[z/f(x)]$ no es libre para la fórmula

$$Rz \wedge \exists x(Px \rightarrow \exists yRxyz)$$

porque algunas de las apariciones de la variable x que introduce ocurren en el ámbito de un cuantificador $\exists x$.

2.2 Semántica

2.2.1 Introducción

La interpretación en Lógica de Proposiciones Para decidir si una fórmula proposicional es verdadera se requiere interpretarla. La interpretación se produce sobre un objeto matemático: sobre una *asignación*, sobre una función del conjunto de letras proposicionales en $\{0,1\}$.

Basta una asignación para que, de cualquier fórmula, se decida su valor de verdad. Es decir, basta una asignación para que se compruebe si esa interpretación de la fórmula la satisface. La definición recursiva de satisfacción facilita esta decisión, por muy compleja que sea la fórmula.

Es más, sobre una misma asignación se pueden interpretar todas las fórmulas de un conjunto de fórmulas dado. Se puede apreciar así su distinto 'comportamiento' (su valor de verdad) frente a un mismo 'estado de cosas' (una misma interpretación). Bastaba escribir la tabla de verdad conjunta, para todas las letras proposicionales que aparecían en esas fórmulas.

La interpretación en Lógica de Primer Orden Los lenguajes de primer orden tienen un alfabeto más expresivo: constantes, funciones, relaciones,... Para interpretar cualquiera de sus fórmula es preciso un objeto matemático más complejo. De hecho, tanto más complejo cuántos más símbolos propios tenga el lenguaje. Para no perdernos, cuando la notación sea más farragosa, conviene ofrecer una descripción coloquial de este 'objeto matemático':

1. Escoja primero un conjunto $\mathcal{U}$ no vacío, cualquiera.
2. Por cada predicado monádico, como $P(x)$, debe escoger un subconjunto de $\mathcal{U}$. Por cada predicado diádico, como $R(x,y)$ debe escoger una relación binaria en $\mathcal{U}$: un conjunto de pares de elementos de $\mathcal{U}$. En general, por cada predicado n-ádico, un conjunto de n-tuplas de elementos de $\mathcal{U}$.
3. Por cada símbolo constante en la fórmula debe escoger un elemento de $\mathcal{U}$.
4. Por cada símbolo funcional, como $g(x,z)$, en la fórmula debe escoger una función sobre $\mathcal{U}$ con el mismo número de argumentos: 2, en el caso de g.

A una construcción como ésta se la denominará *estructura*. Por ejemplo, si una fórmula no contiene constantes, ni funciones y todos sus predicados son monádicos, una estructura adecuada es simplemente un conjunto y varios de sus subconjuntos.

Cualquier sentencia se puede interpretar sobre una estructura adecuada a esa fórmula. Si existen apariciones de variables libres, la interpretación va a depender de 'quiénes se suponen que son esas variables sobre la estructura'. En este caso es preciso establecer una correspondencia entre variables y elementos de $\mathcal{U}$, que denominaremos *asignación*.

Relaciones y funciones sobre un universo

Sea $\mathcal{U}$ un conjunto no vacío. El conjunto $\mathcal{U}^n$ es el conjunto de todas las n-tuplas de $\mathcal{U}$. Una relación n-aria R sobre $\mathcal{U}$ es un subconjunto de $\mathcal{U}^n$.

Ejemplo 2.33 Sea $\mathcal{U} = \{a,b,c\}$. Entonces,

$$\mathcal{U}^2 = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c),(c,a),(c,b),(c,c)\}$$

y

$$\mathcal{U}^4 = \{(a,a,a,a),(a,a,a,b),(a,a,a,c),(a,a,a,d),\ldots,(c,c,c,b),(c,c,c,c)\}$$

Una relación binaria R es un subconjunto de $\mathcal{U}^2$. Posibles relaciones binarias son: $R_1 = \emptyset$ (sin pares de $\mathcal{U}^2$) o $R_2 = \mathcal{U}^2$ (con todos los pares). O bien,

$$R_3 = \{(a,a),(a,c),(b,a),(b,b),(c,c)\}$$

La relación R_3 se puede representar como una matriz, como una tabla (tabl. 2.1a). Si se fija en el par $(a,c) \in R_3$ sobre esta tabla verá que las filas representan el primer elemento del par y las columnas el segundo. Existen $2^{3\times 3}$ relaciones R_k binarias distintas sobre un conjunto de 3 elementos.

También pueden representarse gráficamente como grafos: dibuje los nodos a, b y c; si $(a,c) \in R_3$ trace una flecha desde el nodo a hasta el c. Observe que alguna de estas flechas pueden ir de un nodo a sí mismo.

(a) $R_3 \subset \mathcal{U}^2$

R_3	a	b	c
a	x		x
b	x	x	
c			x

(b) $f : \mathcal{U} \mapsto \mathcal{U}$

R_f	a	b	c
a	x		
b	x		
c		x	

Tabla 2.1: Relaciones y funciones

Una función $f : \mathcal{U}^n \mapsto \mathcal{U}$ hace correspoder a cada n-tupla de su dominio $\mathcal{U}^n$ un elemento de $\mathcal{U}$.

Observe la relación binaria $R_f \subset \mathcal{U}^2$ de la tabla (tabl. 2.1b). Determina una función monaria $f : \mathcal{U} \mapsto \mathcal{U}$. Es una función porque 'cada línea de la tabla contiene una marca y sólo una'. En general, las funciones n-arias son especiales relaciones $(n+1)$-arias.

2.2.2 Interpretaciones

Dada una fórmula ϕ, una estructura adecuada a ella será una construcción matemática sobre la que se pueda interpretar ϕ: un conjunto $\mathcal{U}$ y la 'elección de un representante' para cada símbolo propio empleado en ϕ (constantes,funcionales,relacionales). En realidad, se ofrecerá una interpretación para cada símbolo propio del lenguaje, así se podrán interpretar varias fórmulas sobre una misma estructura aún cuando alguna utilice un símbolo propio que otra no usaba.

Definición 2.34 (Estructura) Una estructura adecuada al lenguaje $L(\mathcal{R}\mathcal{F}\mathcal{C})$ es un par $\langle \mathcal{U}, I \rangle$ tal que:

1. $\mathcal{U}$ es un conjunto no vacío, denominado *dominio* o *universo*
2. I es una función sobre el conjunto de símbolos propios de $S = \mathcal{R} \cup \mathcal{F} \cup \mathcal{C}$ que hace corresponder:
   (a) a cada símbolo relacional n-ario $R \in S$, una relación n-aria sobre $\mathcal{U}$
   (b) a cada símbolo funcional n-ario $f \in S$, una función n-aria sobre $\mathcal{U}$
   (c) a cada constante $c \in S$, un elemento de $\mathcal{U}$

Ejemplo 2.35 Suponga un lenguaje $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$ con una relación binaria R, una función monaria f y una constante c. Las dos fórmulas siguientes pertenecen a ese lenguaje:

$$\forall x R x c \quad \text{y} \quad \exists x R f(x) c \rightarrow \forall z R z z$$

Para decidir su valor de verdad sobre una misma estructura, primero es preciso escoger un universo y elegir qué relación sobre el mismo representa a R, qué función a f y qué elemento a c.

Restrinjámonos a un universo $\mathcal{U} = \{1,2,3\}$. Existen $2^{3\times 3}$ relaciones binarias distintas. Y 3^3 funciones monarias distintas. Y 3 elecciones de elemento representante de c. Salvo error de cálculo, hay 27×2^9 interpretaciones distintas sobre este universo (adecuadas a ese conjunto de símbolos propios). El número es lo de menos. Escojamos una de ellas:

$$R^I = \{(1,2),(2,1),(2,3)\}, \qquad f^I = \{(1,1),(2,3),(3,3)\}, \qquad c^I = 2$$

Es decir, sobre este universo, '2 está relacionado por R con 1, la imagen por f de 2 es $3 = f(2)$ y el elemento 2 representa a c. Cualquier fórmula sobre este lenguaje se podrá interpretar en esta estructura.

> Las fórmulas de alfabeto A_S se pueden interpretar sobre un número infinito de estructuras: puede escoger cualquier universo (finito o infinito) y, sobre él, cualquier combinación de elecciones de subconjuntos, relaciones, funciones y elementos (constantes).

2.2.3 Asignaciones

Definición 2.36 (Asignación) Una asignación A sobre una estructura $\langle \mathcal{U}, I \rangle$ es una función $A : Var \mapsto \mathcal{U}$: hace corresponder a cada variable x del alfabeto con un elemento del universo $\mathcal{U}$.

Al elemento de $\mathcal{U}$ imagen de x por A se le denotará preferentemente como x^A, en vez de $A(x)$.

Definición 2.37 Suponga que $\langle \mathcal{U}, I \rangle$ es una estructura adecuada a un lenguaje $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$. Y que A es una asignación sobre esta estructura. Entonces, se puede ampliar el concepto de asignación para que a cada término t del lenguaje le corresponda un único elemento $t^{I,A}$ del universo U:

- $c^{I,A} = c^I$
- $xt^{I,A} = x^A$
- $f(t_1, \ldots, t_n)^{I,A} = f^I(t_1^{I,A}, \ldots, t_n^{I,A})$

Los dos primeros casos son inmediatos. A cada constante le corresponde su elemento por la interpretación I (sea cual sea la asignación). Y a cada variable le corresponde su elemento por la asignación A (sea cual sea la interpretación).

El último caso se ocupa de términos como $g(f(x), y, c)$, donde f es una función monaria del lenguaje y g una función ternaria del lenguaje. Por ejemplo, suponga un universo de 3 elementos donde $c^I = 2$, $x^A = 1$, $y^A = 1$. Además, a la función f de la fórmula le representa una función tal que $f^I(1) = 3$. Entonces se trata de determinar qué termino es $g^I(3, 1, 2)$, que debe ser un elemento del universo.

Definición 2.38 (Asignación variante de otra) Suponga que sobre una estructura se ha fijado una asignación A, que hace corresponder a cada variable de la fórmula con un elemento del universo. Otra asignación A_x es una variante en x de A si coincide con A en la asignación de toda variable excepto para la variable x.

2.2.4 Satisfacción

Definición 2.39 (Satisfacción de una fórmula) Sea $\langle \mathcal{U}, I \rangle$ una estructura adecuada al lenguaje $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$ y A una asignación. A cada fórmula ϕ de $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$ se le hace corresponder un valor de verdad $\phi^{I,A}$ como sigue:

1. Fórmulas atómicas
   - $\bot^{I,A} = 0$, $\top^{I,A} = 1$
   - $(t_1 \approx t_2)^{I,A} = 1$ si y sólo si $t_1^A = t_2^A$
   - $[R(t_1, \ldots, t_n)]^{I,A} = 1$ si y sólo si $(t_1^A, \ldots, t_n^A) \in R^I$

2. $[\neg\psi]^{I,A} = \neg[\psi^{I,A}]$
3. $[\psi * \chi]^{I,A} = \psi^{I,A} * \chi^{I,A}$
4. $[(\forall x\psi)]^{I,A} = 1$ si y sólo si ψ^{I,A_x} para toda asignación A_x variante en x respecto a A
5. $[(\exists x\psi)]^{I,A} = 1$ si y sólo si ψ^{I,A_x} para alguna asignación A_x variante en x respecto a A

2.2.5 Ejemplos de interpretación

Sólo con predicados

Monádicos Los ejemplos de este apartado son básicos. Consideran fórmulas con, a lo sumo, 2 predicados monádicos y 3 constantes. No contienen funciones ni el símbolo de igualdad. En concreto, son fórmulas de un lenguaje $L(\mathcal{R}, \mathcal{F}, \mathcal{C})$ con:

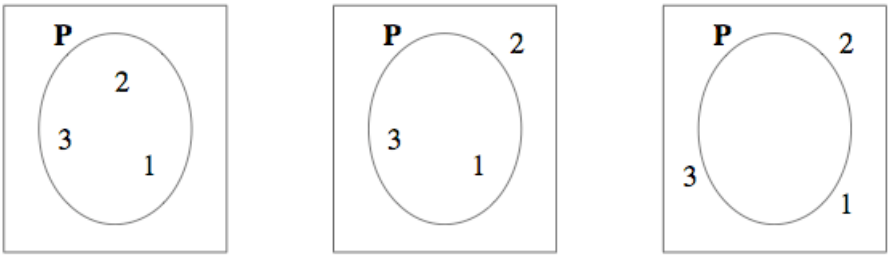
$$\mathcal{R} = \{P, Q\}, \quad \mathcal{F} = \emptyset, \quad \mathcal{C} = \{a, b, c\}$$

donde todos los símbolos de $\mathcal{R}$ son monádicos. Para cada una de estas fórmulas se escogerán (1) un universo $\mathcal{U}$, (2) una interpretación I adecuada y (3) una asignación A, sobre los que decidir el valor de verdad de la fórmula en ese caso.

Cada interpretación de este lenguaje debe fijar: (1) qué subconjuntos del universo son P^I y Q^I y (2) qué elementos del universo son a^I, b^I y c^I.

En las fórmulas que no contengan todos estos símbolos, basta especificar la interpretación sólo para los símbolos que aparecen. Lo mismo ocurre con las asignaciones A: basta precisar qué elemento $u = x^A$ del universo representa a cada variable x que aparece en la fórmula.

> *Salvo en los ejemplos iniciales, en vez de P^I, se denotará símplemente como* **P**, *en negrita, al subconjunto representante del predicado P en la fórmula. Lo mismo se hará con las constantes.*



(a) $P = \mathcal{U}$ (b) $P \neq \mathcal{U}, P \neq \emptyset$ (c) $P = \emptyset$

Figura 2.4: Tres estructuras sobre el mismo universo

Ejemplo 2.40 (*Pc*) El proceso de interpretación que se propone se enuncia coloquialmente como: "yo no sé qué propiedad formaliza P, pero en este universo de n elementos marco arbitrariamente que estos m tienen esa propiedad; díme quién consideras que es c en este universo y comprobaremos si se encuentra entre los m marcados".

Sobre la figura (fig. 2.4b), si c es el elemento 1, entonces la fórmula Pc es verdadera. Por contra, si se considera que 2 representa a c, entonces es falsa. Observe que se ha utilizado una interpretación I adecuada que facilita 'quién es P' en ese universo y 'quién es c'.

Más formalmente, siempre se requiere un universo $\mathcal{U}$, una interpretación I adecuada y una correspondencia A entre variables y elementos. En este caso, el valor de verdad de Pc resultará independiente de la asignación A: no hay variables en la fórmula. Por eso, se suele omitir:

$$\langle \mathcal{U} = \{1,2,3\}; P^I = \{1,3\}; c^I = 1 \rangle$$

Entonces, aplicando la definición de satisfacción de una fórmula:

$$(Pc)^{I,A} = 1 \quad \text{si y sólo si} \quad c^{I,A} \in P^I \quad \text{si y sólo si} \quad c^I \in P^I \quad \text{si y sólo si} \quad 1 \in \{1,3\}$$

Luego Pc resulta verdadera sobre esta estructura. Sobre toda estructura en que Pc sea verdadera su negación $\neg Pc$ es falsa. Y sobre toda estructura en que Pc sea falsa, su negación $\neg Pc$ es verdadera.

Una fórmula como Pc podría representar la frase: *"Antonio es rubio"* o *"El número 0 es par"*. Y una fórmula como $\neg Pc$: *"Antonio no es rubio"* o *"El número 0 no es par"*; o tambien: *"Antonio es no-rubio"* o *"El número 0 es no-par"*.

Ejemplo 2.41 (Px) Observe la figura (fig. 2.4b). La fórmula Px es verdadera sobre la siguiente estructura y asignación:

$$\langle \mathcal{U} = \{1,2,3\};\ P^I = \{1,3\} \rangle, \text{ con } A(x) = 3$$

Efectivamente, aplicando la definición de satisfacción de una fórmula:

$$(Px)^{I,A} = 1 \quad \text{si y sólo si} \quad x^{I,A} \in P^I \quad \text{si y sólo si} \quad x^A \in P^I \quad \text{si y sólo si} \quad 3 \in \{1,3\}$$

Sin embargo, sobre esta misma estructura, si se utiliza otra asignación A'

$$\langle \mathcal{U} = \{1,2,3\};\ P^I = \{1,3\} \rangle, \text{ con } A'(x) = 2$$

entonces la fórmula Px resulta falsa. Observe que sólo variando la asignación sobre x, en la misma estructura, se consigue que la fórmula sea bien verdadera, bien falsa.

Hay dos casos extremos donde lo anterior no se produce. En estructuras como (fig. 2.4a), donde $\mathbf{P} = \mathcal{U}$, ninguna asignación consigue que la fórmula Px sea falsa. Y en (fig. 2.4c), donde $\mathbf{P} = \emptyset$, ninguna asignación consigue que la fórmula Px sea verdadera.

Una fórmula como ésta puede representar la frase *"x es rubio"* ó *"x es par"*. Su valor de verdad depende de la asignación de x.

Ejemplo 2.42 ($\forall xPx$) Una fórmula como ésta puede representar la frase *"todos (los elementos del universo) son rubios"* o *"todos (los elementos del universo) son pares"*. Es sólo verdadera en las estructuras en que P sea igual a todo el universo (fig. 2.4a) y falsa en otros casos (b y c).

Observe que su valor de verdad no depende de la asignación. De hecho, así se valora el cuantificador: si toda asignación de la variable x hace verdadera Px entonces $\forall xPx$ es verdadero. En (fig. 2.4a) las tres asignaciones posibles $A(x) = 1$, $A'(x) = 2$ y $A''(x) = 3$ hacen verdadera Px sobre esa estructura; y por tanto, $\forall xPx$ es verdadera sobre esa estructura.

($\neg\forall xPx$) La fórmula $\neg\forall xPx$ es verdadera donde $\forall xPx$ es falsa: (fig. 2.4b-c). Podría representar las frases *"no todos son rubios"* o *"no todos son pares"*.

$(\forall x \neg Px)$ No confunda la fórmula anterior con $\forall x \neg Px$: *"todos son no-rubios"*, *"todos son no-pares"*. O, más bien, *"ninguno es rubio"*, *"ninguno es par"*. Esta fórmula sólo es cierta en (fig. 2.4c), donde todos los elementos verifican $\neg P$, es decir, donde todos están en $\bar{\mathbf{P}}$, complementario de $\mathbf{P}$.

Ejemplo 2.43 ($\exists xPx$) Una fórmula como ésta puede representar la frase *"alguno (elemento del universo) es rubio"* o *"hay alguien (elemento del universo) que es par"*. Es sólo verdadera en las estructuras en que P sea distinto del vacío (fig. 2.4a-b) y falsa en otro caso (c).

Observe que su valor de verdad no depende de la asignación. De hecho, así se valora el cuantificador: si al menos una asignación de la variable x hace verdadera Px entonces $\exists xPx$ es verdadero. En (fig. 2.4b), dos de las asignaciones posibles $A(x) = 1$, $A'(x) = 3$ hacen verdadera Px sobre esa estructura; y por tanto, $\exists xPx$ es verdadera sobre la estructura de (fig. 2.4b).

$(\neg\exists xPx)$ La fórmula $\neg\exists xPx$ es verdadera donde $\exists xPx$ es falsa: (fig. 2.4c). Podría representar las frases *"no existe alguien rubio"* o *"ningún número es par"*.

$(\exists x \neg Px)$ No confunda la fórmula anterior con $\exists x \neg Px$: *"alguien es no-rubio"*, *"alguien es no-par"*. Para que esta fórmula sea verdadera basta que exista un elemento que tenga la propiedad $\neg P$, que esté 'fuera del conjunto P'.

Ejercicio 2.44 Interprete los siguientes pares de fórmulas sobre la misma estructura:

$$\begin{array}{ccc} \neg\forall xPx & , & \exists x\neg Px \\ \neg\exists xPx & , & \forall x\neg Px \\ \forall xPx & , & \neg\exists x\neg Px \\ \exists xPx & , & \neg\forall x\neg Px \end{array}$$

¿Qué distingue a esos pares de fórmulas de los siguientes?

$$\begin{array}{ccc} Pa & , & Pc \\ Px & , & Pc \\ Px & , & \exists xPx \\ Pc & , & \exists yPy \end{array}$$

Ejemplo 2.45 ($\exists x(Px \wedge Qx)$) Esta fórmula es verdadera en las estructuras donde al menos un elemento pertenece a P^I y (ese mismo elemento) pertenece también a Q^I. Es decir, cuando la intersección de ambos conjuntos no sea vacía. Efectivamente, en ese caso, hay una asignación de x que verifica $(Px \wedge Qx)$, y por lo tanto $\exists x(Px \wedge Qx)$ es verdadera.

Sobre universos finitos, puede ser útil la siguiente aproximación. Considere su evaluación como un proceso iterativo, como un 'bucle':

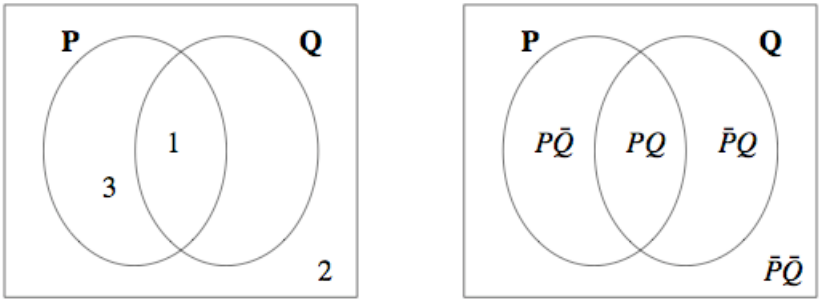
$$\left[\begin{array}{l} \exists x \\ \qquad (Px \wedge Qx) \\ \\ \end{array}\right.$$

En cada iteración (para cada asignación), la variable x del bucle es un elemento distinto del universo. Suponga que en la primera iteración es $x^A = 1$. Entonces, en el cuerpo del bucle se pregunta si '$(P1 \wedge Q1)$'; en la siguiente iteración '$(P2 \wedge Q2)$', ..., hasta que finalizan los elementos del universo. En realidad, se abandona el proceso (contestando afirmativamente) tan pronto como se verifica la fórmula para un elemento. Sólo es preciso recorrer todo el universo para ofrecer una respuesta negativa. Obviamente, para universos infinitos hay que armarse con algo de paciencia.

Observe la figura (fig. 2.5a). La fórmula analizada es verdadera en ella. De hecho, basta que exista (al menos) un elemento en la región PQ, con independencia de los elementos que hubiera en las otras 3 regiones.

$\exists x(\neg Px \wedge Qx)$ Esta fórmula es falsa en (fig. 2.5a). Requiere que un mismo elemento no verifique P pero si Q. Es decir, debería haber (al menos) un elemento en la región $\bar{P}Q$.

$\exists x(Px \vee Qx)$ Esta fórmula es verdadera en (fig. 2.5a). Hay dos elementos (1 y 3) que cuando se les 'pregunta', en ese hipotético bucle, si verifican P o Q (o ambos) responde afirmativamente. De hecho, podían estar situado en cualquiera de las 3 regiones que comprende la unión $P \cup Q$, que no debe ser vacía.



(a) Estructura (b) Esquema de estructura

Figura 2.5: Estructuras con dos predicados monádicos

Ejemplo 2.46 **($(\exists xPx) \wedge (\exists xQx)$)** Para interpretar esta fórmula es fundamental percatarse de que hay dos ámbitos disjuntos cuantificados. Siguiendo con la imagen del bucle, correspondería al siguiente proceso:

$$\left[\begin{array}{l} \exists x \\ \quad Px \end{array}\right] \wedge \left[\begin{array}{l} \exists x \\ \quad Qx \end{array}\right]$$

Se 'pregunta' primero, a todos los elementos del universo, si tienen la propiedad P. Si se encuentra un elemento (una asignación de la variable x del bucle) que lo verifique, entonces se sale de este ámbito dando un valor verdadero a la subfórmula '$(\exists xPx)$'. Otro tanto se hace con la subfórmula '$(\exists xQx)$'. Y se evalúa su conjunción.

En la evaluación de $\exists x(Px \wedge Qx)$, las dos variables del ámbito de $\exists x$ referencian *al mismo elemento*. En $(\exists xPx) \wedge (\exists xQx)$ no existe esa relación entre esas variables, de ámbitos distintos. 'Alguien es rubio y alguien es alto' se hace verdadera aunque el rubio (3) no sea el mismo que el alto (1). Observe que una evaluación equivalente se obtiene para la fórmula $(\exists xPx) \wedge (\exists yQy)$

En lógica de proposiciones, cuando se repetía una letra como p dentro de la misma fórmula representaba 'la misma p', la misma proposición. En lógica de primer orden las variables (sólo las variables) puede que no tengan esa restricción: dependen del ámbito del cuantificador en que se encuentren.

Ejemplo 2.47 ($\exists x \exists y(Px \wedge Qy)$) Para poder evaluar esta fórmula primero analizaremos una previa: $\exists y(Px \wedge Qy)$. Volviendo a la imagen del bucle, la variable y va a recorrer todas las posibles asignaciones, luego no hace falta precisar una en concreto para ella. Sin embargo, la variable libre x va a adoptar el 'valor', el elemento (único) que se le asigne:

$$\left[\begin{array}{l} \exists y \\ \quad (Px \wedge Qy) \\ \\ \end{array}\right.$$

Sobre la figura (fig. 2.5a), con la asignación $A(x) = 3$ la fórmula es verdadera. Observe que, dentro de este bucle, se pregunta primero si '$(P3 \wedge Q1)$', después si '$(P3 \wedge Q2)$', si '$(P3 \wedge Q3)$'. La primera asignación de y ($A(y) = 1$) satisface el cuerpo del bucle, la fórmula cuantificada existencialmente. Luego hace verdadera la fórmula existencial.

$\exists x \exists y(Px \wedge Qy)$ Los ámbitos de los cuantificadores o se anidan o son disjuntos, sintácticamente nunca se solapan. Esta es una fórmula del tipo $\exists x \phi$, donde ϕ puede informalmente verse como un bucle anidado. La fórmula global será verdadera si alguna asignación de x hace verdadera ϕ. En concreto, como se ha visto antes, para $A(x) = 3$ (o para $A(x) = 1$) se satisface la fórmula existencial interna.

Ejemplo 2.48 ($\forall x(Px \rightarrow Qx)$) Observe la figura (fig. 2.5b). En tres de las cuatro regiones marcadas puede haber elementos sin que esta fórmula sea falsa:

$$\left[\begin{array}{l} \forall x \\ \quad (Px \rightarrow Qx) \\ \\ \end{array}\right.$$

De este 'bucle' informal sólo se sale afirmativamente si *todas* las asignaciones de x verifican la subfórmula de su ámbito. Basta que haya un elemento en $P\bar{Q}$, que haga verdadero el antecedente y falso el consecuente, para que se evalúe como falso todo el bucle.

Observe que una estructura en que P no tenga elementos también satisface esta fórmula. No se requiere que P tenga elementos, pero, si los tiene, deben estar en la región PQ. Es decir, la fórmula se hace verdadera donde $P \subset Q$ es verdadero. Esta fórmula puede leerse 'todos los P son Q'.

Ejercicio 2.49 Evalúe sobre la misma estructura los siguientes pares de fórmulas:

$$\begin{array}{ccc} \forall x(Px \rightarrow Qx) & , & \forall x(\neg Px \vee Qx) \\ \forall x(\neg Px \vee Qx) & , & \forall x \neg(Px \wedge \neg Qx) \\ \forall x \neg(Px \wedge \neg Qx) & , & \neg \exists x(Px \wedge \neg Qx) \end{array}$$

Diádicos Las fórmulas de este apartado utilizan, a lo sumo, un par de predicados diádicos (R y S), un par de predicados monádicos (P y Q) y un par de constantes (a y b).

Ejemplo 2.50 (*Rxy*) Esta fórmula, con ambas variables libres, es verdadera sobre la siguiente estructura y asignación:

$$\langle \mathcal{U} = \{1,2,3\};\ R^I = \{(1,3),(2,2)\}\rangle,\ \text{con } A(x) = 1, A(y) = 3$$

Gráficamente, sobre la tabla (tabl. 2.2a), supone verificar que si se entra por la fila 1 y la columna 3, la casilla pertenece a la relación.

(a) $R \subset \mathcal{U}^2$

R	1	2	3
1			x
2		x	
3			

(b) $S \subset \mathcal{U}^2$

S	1	2	3
1		x	x
2		x	
3	x	x	

Tabla 2.2: Dos relaciones sobre el mismo universo

Ejemplo 2.51 ($\exists y Rxy$) Esta fórmula, con la variable libre x, es verdadera sobre la siguiente estructura y asignación:

$$\langle \mathcal{U} = \{1,2,3\};\ R^I = \{(1,3),(2,2)\}\rangle,\ \text{con } A(x) = 1$$

Puesto que existe una asignación de y que satisface $R1y$. Gráficamente, sobre la tabla (tabl. 2.2a), supone verificar que si se entra por la fila 1 para alguna columna y, la casilla pertenece a la relación.

Ejemplo 2.52 ($\forall x \exists y Rxy$) Esta fórmula ya no requiere asignaciones: todas las variables están ligadas. Informalmente se puede leer como 'para toda fila x existe una columna y tal que la casilla (x,y) pertenece a la relación.

Sobre la tabla (tabl. 2.2), es falsa en (a), si $R^I = R$ pero es verdadera en (b), si $R^I = S$.

$\exists y \forall x Rxy$ Existe una columna y tal que, para toda fila x sobre esa columna, la casilla pertenece a la relación. Sobre la tabla (tabl. 2.2), es falsa en (a), pero verdadera en (b) para $y^A = 2$. Observe que esta fórmula es la que se obtiene de la última analizada permutando los cuantificadores: no son equivalentes, no es lo mismo decir 'todo el mundo quiere a alguien' que 'alguien es querido por todo el mundo'.

Puede comprobar que la fórmula $\exists x \forall y Rxy$ no se satisface sobre ninguna de las dos relaciones de la tabla (tabl. 2.2).

Ejemplo 2.53 ($\forall x \forall y(Rxy \to Sxy)$) Interprete $R^I = R$ y $S^I = S$ de la tabla (tabl. 2.2). Este 'doble bucle anidado' es verdadero si en todas sus 3×3 iteraciones el condicional no se hace falso. Para ello basta que nunca el antecedente sea verdadero y el consecuente falso: que ninguna casilla marcada de R deje de estar marcada en S. Es decir, que $R \subset S$, el conjunto de pares R sea un subconjunto del de S.

Compruebe que, si esta fórmula es verdadera sobre una estructura, no pueden dejar de ser verdaderas $\exists x \forall y(Rxy \to Sxy)$ y $\forall x \exists y(Rxy \to Sxy)$, menos exigentes.

Ejemplo 2.54 ($\exists x(\forall y Rxy \wedge Px)$) Esta expresión podría formalizar una sentencia como 'existe alguien que quiere a todo el mundo y que es rubio'. En una estructura como:

$$\langle \mathcal{U} = \{1,2,3\};\ R^I = \{(1,3),(2,1),(2,2),(2,3)\},\ P^I = \{(2\}\rangle$$

efectivamente existe alguien (el 2) que verifica esta fórmula. Observe que si 'alguien quiere a todo el mundo' no puede dejar de 'quererse a sí mismo'.

La interpretación de predicados poliádicos $U(x_1,\ldots,x_n)$ es gráfica e intuitivamente menos inmediata. También lo es la interpretación sobre universos infinitos o con muchos elementos. Es preciso utilizar sistemáticamente la definición formal de satisfacción. Esperamos que los ejemplos previos hayan facilitado el uso del formalismo.

Funciones e igualdad

En esta sección se completan los posibles símbolos propios que pueden utilizarse en un lenguaje de primer orden.

Ejemplo 2.55 ($\forall xRxf(c)$) Esta fórmula podría representar la sentencia 'todo el mundo quiere a la madre de Juan'. Gráficamente, considere la estructura de la tabla (tabl. 2.2b), donde $c^I = 3$ y la función $f^I : \mathcal{U} \mapsto \mathcal{U}$ es tal que $f^I(3) = 2$. La fórmula es verdadera en esa estructura.

$\forall xRf(x)x$ Esta fórmula podría ser una versión simbólica de la expresión 'mi mamá me mima': para toda persona, su madre quiere a esa persona. Considere la relación de esta fórmula con $\forall x\exists yRyx$. ¿Cuál de ellas no puede dejar de verificarse donde se verifica la otra?

Ejemplo 2.56 ($Pc \wedge \forall x(Px \rightarrow x \approx c)$) Esta fórmula es una conjunción: sólo es verdadera cuando se satisfagan ambas subfórmulas. Así, c^I debe pertenecer a P^I, y además el condicional no debe ser falso para ninguna asignación de x. Es decir, nunca puede darse que x tenga la propiedad P sin que coincida que esa x es c. En pocas palabras: c tiene la propiedad P y es el único del universo que la tiene.

Observe que el predicado diádico $\approx$ no se interpreta arbitrariamente como cualquier relación binaria sobre el universo. Una interpretación *normal* requiere que $t_1 \approx t_2$ sea verdad cuando el elemento que representa a t_1 sea el mismo que el que representa a t_2: $t_1^I = t_2^I$.

Ejemplo 2.57 ($\forall x\forall y\forall z g(g(x,y),z) \approx g(x,g(y,z))$) Esta fórmula sólo es verdadera sobre estructuras donde la función binaria g sea asociativa. Por poner un ejemplo con universo infinito, considere el conjunto de los números naturales donde g^I es la función suma.

2.2.6 Conceptos semánticos básicos

En las notas sobre semántica proposicional se utilizaron profusamente las tablas de verdad para fijar los conceptos de consecuencia lógica, validez y equivalencia. Para aprovechar estas mismas intuiciones basta considerar que ahora, para una fórmula cualquiera, tiene algo parecido a una 'tabla de verdad con infinitas líneas (interpretaciones) distintas'.

Todos los esquemas utilizados en lógica de proposiciones siguen siendo válidos: así, una fórmula será consecuencia de otra si es verdadera en todas las líneas en que ésta lo es (y quizá en alguna más). Y un conjunto de fórmulas será satisfacible si existe al menos una 'línea' (una interpretación-asignación) donde coincidan en ser verdaderas.

Definición 2.58 (Satisfacibilidad) Una fórmula es satisfacible si existe algún universo, interpretación y asignación donde sea verdadera. Un conjunto de fórmulas es satisfacible si existe algún universo, interpretación y asignación donde coincidan todas en ser verdaderas.

Definición 2.59 (Validez) Una fórmula ϕ es verdadera en un universo, con interpretación I y asignación A, si se satisface $\phi^{I,A}$ en el mismo. Una fórmula es válida si se satisface para todo universo, toda interpretación y asignación.

Una fórmula es válida si y sólo si su negación es insatisfacible. Un conjunto de fórmulas es satisfacible si y sólo si la fórmula conjunción de todas ellas es satisfacible.

Definición 2.60 (Consecuencia) Una fórmula ϕ es consecuencia lógica de un conjunto de fórmulas Φ si en toda estructura $\langle \mathcal{U}, I \rangle$ y asignación en que todas las fórmulas de Φ sean verdaderas también lo es ϕ. Se expresará entonces como $\Phi \models \phi$.

Es decir, si las fórmulas de Φ sólo fueran simultáneamente verdaderas en 3 casos, ϕ debería serlo en esos 3 casos (y opcionalmente puede serlo en alguno más). Si el conjunto Φ es insatisfacible cualquier fórmula es consecuencia lógica suya.

Definición 2.61 (Equivalencia) Dos fórmulas ϕ y ψ son equivalentes si $\phi \models \psi$ y $\psi \models \phi$.

2.3 Deducción Natural

Los sistemas deductivos en Lógica Proposicional eran (casi) un lujo. Y lo eran, en tanto que teóricamente mediante tablas de verdad podían decidirse las cuestiones semánticas básicas sobre fórmulas dadas. Otro tema, tanto en los métodos semánticos como en los sintácticos, era su complejidad temporal.

Las consideraciones siguientes tratan de presentar las sutilezas que deben considerar los sistemas deductivos en Lógica de Predicados, sean éstos del tipo que sean.

2.3.1 Consideraciones previas

Decida cuáles de estas consecuencias admite intuitivamente. Los párrafos siguientes confirmarán su intuición.

1. ¿ $\forall x Px \models Pa$? *(todos los elementos tienen la propiedad P, luego el elemento a la tiene)*
2. ¿ $\exists x Px \models Pa$? *(algún elemento tiene la propiedad P, luego el elemento a la tiene)*
3. ¿ $Pa \models \forall x Px$? *(el elemento a tiene la propiedad P, luego todos la tienen)*
4. ¿ $Pa \models \exists x Px$? *(el elemento a tiene la propiedad P, luego alguno la tiene)*

No son consecuencia ... La segunda y tercera propuestas no se verifican. Entre las interpretaciones que satisfacen $\exists x Px$, efectivamente, algunas satisfacen Pa, pero no todas. Para fijar ideas, en toda situación en que 'alguien es rubio' no tiene por qué satisfacerse de forma general que 'Juan es rubio'. Formalmente, $\exists x Px \not\models Pa$.

Tampoco en toda situación en que 'Juan es rubio' debe satisfacerse que 'todos sean rubios'. Puede que existan situaciones donde se satisfagan ambas, pero la primera no garantiza generalmente la satisfacción de la segunda: $Pa \not\models \forall x Px$.

Para negar una relación de consecuencia $\phi_1, \ldots, \phi_n \models \psi$ basta mostrar un contrajemplo, una interpretación que satisface a todas las fórmulas de $\{\phi_1, \ldots, \phi_n\}$ pero no a ψ. En nuestro caso:

$$2.\ \exists x Px \not\models Pa \qquad I = \langle U = \{1,2\}, \text{ con } P^I = \{1\}, a^I = 2 \rangle$$

$$3.\ Pa \not\models \forall x Px \qquad I = \langle U = \{1,2\}, \text{ con } P^I = \{1\}, a^I = 1 \rangle$$

Es importante insistir en que $\exists x Px \not\models Pa$. No siempre que 'alguien tiene la propiedad P' se puede asegurar que 'precisamente este determinado elemento la tiene'.

Son consecuencia ... Sin embargo, sí se puede afirmar:

$$1.\ \forall xPx \models Pa \qquad\qquad 4.\ Pa \models \exists xPx$$

1. Siempre que todos los elementos del universo, todas las asignaciones posibles de la variable x, resultan pertenecer a P, no se va a poder escoger un elemento representante de la constante 'a' que no pertenezca a P.

4. Siempre que ha sido posible encontrar un representante de 'a' en el universo tal que pertenece a P, basta asignar ese elemento a x para que se satisfaga Px y por tanto $\exists xPx$. Al fin y al cabo, esta última fórmula se satisface si hay alguna asignación de x que satisface Px.

Ejemplo 2.62 Considere la fórmula universalmente cuantificada:

$$\forall y(Qy \rightarrow Ryy)$$

Una de las (infinitas) interpretaciones que la satisfacen es:

$$I = \langle U = \{1,2,3\},\ \text{con } Q^I = \{1,2\}, R^I = \{(1,1),(1,3),(2,2)\}\rangle$$

Esto es así porque, cuando se considera que y es el elemento 1, efectivamente:

$$'Q1 \rightarrow R11'\text{ , más formalmente, } 1 \in Q^I \rightarrow (1,1) \in R^I$$

Y porque resulta verdadera cualquier otra asignación de y. Compruebe que son verdaderas tanto $'Q2 \rightarrow R22'$ como $'Q3 \rightarrow R33'$.

Compruebe que en todos los modelos en que es verdadera la fórmula $\forall y(Qy \rightarrow Ryy)$ también lo es una fórmula como $Qa \rightarrow Raa$.

Propiedades preservadas La pregunta inicial se podía haber planteado sobre fórmulas más complejas, como las de la segunda columna de esta tabla:

1.	¿ $\forall xPx \models Pa$?	¿ $\forall x\big(Px \rightarrow (Qz \vee \exists yRyx)\big) \models \big(Pa \rightarrow (Qz \vee \exists yRya)\big)$?	¿ $\forall x\varphi \models \varphi(x/a)$?
2.	¿ $\exists xPx \models Pa$?	¿ $\exists x\big(Px \rightarrow (Qz \vee \exists yRyx)\big) \models \big(Pa \rightarrow (Qz \vee \exists yRya)\big)$?	¿ $\exists x\varphi \models \varphi(x/a)$?
3.	¿ $Pa \models \forall xPx$?	¿ $\big(Pa \rightarrow (Qz \vee \exists yRya)\big) \models \forall x\big(Px \rightarrow (Qz \vee \exists yRyx)\big)$?	¿ $\varphi(x/a) \models \forall x\varphi$?
4.	¿ $Pa \models \exists xPx$?	¿ $\big(Pa \rightarrow (Qz \vee \exists yRya)\big) \models \exists x\big(Px \rightarrow (Qz \vee \exists yRyx)\big)$?	¿ $\varphi(x/a) \models \exists x\varphi$?

Para cada caso, para cada línea, la respuesta es la misma que la que se adelantó para las fórmulas iniciales más sencillas. Es decir, dada una fórmula existencial (línea 2) no resultará consecuencia lógica de ella la que se produzca eliminando el cuantificador y 'particularizando' (sustituyendo) la variable x por una constante a. Esto es lo que expresa como $\varphi(x/a)$.

En la tercera línea, una expresión como $\varphi(x/a) \models \exists x\varphi$ se interpreta como que 'la hipótesis tiene la misma forma que la de la supuesta conclusión, si a ésta se le hubiera eliminado el cuantificador y sustituido las x por a.'

Todos los sistemas deductivos que siguen hacen uso adecuado de estas particularizaciones, que toman la forma sintáctica de una sustitución. Todas las restricciones sobre las mismas (qué puede ser sustituido y qué sustituyente) garantizarán en cada caso que se mantiene el buen comportamiento semántico de la expresión obtenida.

Por 'buen comportamiento semántico' entendemos que, a veces, se querrá garantizar que la expresión resultante sea consecuencia de la inicial: que todo modelo de la primera lo sea de la segunda. O, a veces, algo menos fuerte, que sean ambas satisfacibles o ambas insatisfacibles.

Observe que esto último sólo requiere que si hay un modelo que satisface una de las fórmulas entonces hay un modelo (quizá distinto) que satisface a la otra. Por ejemplo, si bien $\exists xPx \not\models Pa$, sin embargo son igualmente satisfacibles. Basta que se muestre un sólo modelo de una para (rectificándolo si es necesario) obtener un modelo de la otra.

2.3.2 Cuantificadores universales

Ya se ha expuesto un sistema de deducción natural para la Lógica de Proposiciones. Con aquellas reglas de inferencia se puede, por ejemplo, demostrar (véase fig. 2.6a):

$$p \to q, q \to r \vdash p \to r$$

Sin ampliar aquellas reglas de inferencia puede demostrarse (véase fig. 2.6b):

$$\exists x Px \to \forall y Qy, \forall y Qy \to \forall z Sz \vdash \exists x Px \to \forall z Sz$$

Observe este último esquema deductivo: se obtiene por sustitución uniforme de cada letra proposicional por una fórmula de primer orden.

1	$p \to q$	*premisa*
2	$q \to r$	*premisa*
3	p	*suposic*
4	q	$\to\mathcal{E}1,3$
5	r	$\to\mathcal{E}2,4$
6	$p \to r$	$\to I3,5$

(a) Fórmulas proposic.

1	$\exists x Px \to \forall y Qy$	*premisa*
2	$\forall y Qy \to \forall z Sz$	*premisa*
3	$\exists x Px$	*suposic*
4	$\forall y Qy$	$\to\mathcal{E}1,3$
5	$\forall z Sz$	$\to\mathcal{E}2,4$
6	$\exists x Px \to \forall z Sz$	$\to I3,5$

(b) Instancias de fórm. proposic.

Figura 2.6: Inferencia sobre instancias de fórmulas proposicionales

No obstante, con las reglas expuestas hasta el momento no se puede deducir un secuente correcto como:

$$\frac{\forall x(Px \to Qx) \quad \forall x(Qx \to Sx)}{\forall x(Px \to Sx)}$$

Es preciso, por tanto, añadir nuevas reglas de inferencia para (cuando se requiera):

- 'abrir' las fórmulas cuantificadas (eliminar cuantificadores)
- aplicar las reglas de inferencia proposicionales
- 'cerrar' las fórmulas resultantes (introducir cuantificadores)

Existen, respectivamente, una regla de introducción y otra de eliminación para cada uno de los cuantificadores.

Eliminación

La eliminación de un cuantificador universal es intuitiva y no requiere demasiadas cautelas:

$$\frac{\forall x \varphi}{\varphi[x/t]} \forall x\,\mathcal{E}$$

Es decir, de una fórmula universal $\forall x\varphi$ se puede derivar en cualquier momento una particularización: la que resulta de instanciar todas las apariciones libres de x en φ por un término cualquiera t. Tan sólo se requiere que el término t sea libre para x en φ.

Ejemplo 2.63 Por aplicación directa de esta regla, sin complicaciones, se deriva:

$$\frac{\forall x(Qx \wedge Rxa)}{Qg(y,b) \wedge Rg(y,b)a}\,\forall x\mathcal{E}$$

donde el término $t = g(y,b)$ ha sustituido a todas las apariciones libres de x en esa subfórmula $(Qx \wedge Rxa)$.

Ejemplo 2.64 Por aplicación de esta regla se deriva:

$$\frac{\forall x(Qx \wedge \exists xRxx)}{Qa \wedge \exists xRxx}\,\forall x\mathcal{E}$$

donde el término $t = a$ ha sustituido a todas las apariciones libres de x en esa subfórmula $(Qx \wedge \exists xRxx)$.

Una fórmula como $\forall x(Qx \wedge \exists xRxx)$ conviene escribirla en su forma equivalente $\forall x(Qx \wedge \exists yRyy)$. No obstante, sintácticamente puede ocurrir, y la instanciación (por sustitución) sólo debe afectar a todas las apariciones libres de x tras prescindir del cuantificador $\forall x$.

Ejemplo 2.65 Por aplicación *incorrecta* de esta regla se podría derivar:

$$\text{¡¡incorrecto!!} \qquad \frac{\forall x(Qx \wedge \exists yRxy)}{Qy \wedge \exists yRyy}\,\forall x\mathcal{E}$$

donde el término $t = y$ ha sustituido a todas las apariciones libres de x en esa fórmula $(Qx \wedge \exists yRxy)$, pero *no era un término libre para x en esa fórmula*.

El término $t = y$ con el que se sustituye a x no es libre para x en esa fórmula porque produce incorrectamente una aparición ligada de y.

Ejemplo 2.66 Un ejemplo algo más completo:

1	$\forall y(Py \to Qy)$	*premisa*
2	Pa	*premisa*
3	$Pa \to Qa$	$\forall y\mathcal{E} : 1$
4	Qa	$\to\mathcal{E} : 2,3$

Observe que Pa es una de las premisas. Para poder concluir Qa es preciso instanciar la fórmula universal de (1) con la sustitución [y/a]. Cualquier otra instanciación, de las muchas posibles, no permite concluir Qa.

Introducción

El ejemplo más citado de generalización se encuentra en la geometría euclídea. En ella hay varias fórmulas que se aceptan como premisas. Y un argumento podría comenzar y acabar así: "*Sea un triangulo a cualquiera* [...varios pasos deductivos...] *por lo tanto, en a sus ángulos suman 180 grados, así que* en todo triángulo *sus ángulos suman 180 grados*".

La corrección del argumento reside en que a es un triángulo *cualquiera*: es decir, en ninguna de las premisas (o suposiciones previas no cerradas) aparece ese término a. Ninguna fórmula previa fija propiedades especiales para ese triángulo a.

Por otro lado, toda fórmula universal previa se puede particularizar para ese a (como para cualquier otro término). Trabajando, inferiendo, a partir de estas propiedades no específicas de a se puede

llegar a una determinada expresión. La generalización del resultado se produce sustituyendo todas las apariciones de a por una variable (por ejemplo, x) y ligando esta variable, anteponiendo $\forall x$.

$$\dfrac{\boxed{\begin{array}{l} a \\ \quad \vdots \\ \quad \varphi[x/a] \end{array}}}{\forall x \varphi}\ \forall x\, I$$

Ejemplo 2.67 Un breve ejemplo que incluye eliminación e introducción de cuantificadores universales:

	1	$\forall x Px$	*premisa*
a	2	Pa	$\forall y \mathcal{E} : 1$
	3	$\forall y Py$	$\forall x\, I : 2$

El sentido de la caja, del marco que rodea a la fórmula (2) es: 'a partir de este punto vamos a hablar de un término a, no utilizado antes, sin propiedades específicas, con afán de generalizar los resultados'. Observe que la caja sólo marca el ámbito donde se utilizará el término a: no se abre porque se haga una suposición que haya que eliminar posteriormente. De hecho, la fórmula (2) no es una de estas suposiciones adicionales, sino la particularización de una propiedad general para este término a.

En este ejemplo, el resultado que se quería conseguir para ese a se consigue en un sólo paso: a tiene la propiedad P. Como Pa se puede obtener a partir de Py mediante la sustitución $[y/a]$, se ajusta al esquema propuesto para la regla, y se puede generalizar como $\forall y Py$.

Ejemplo 2.68 Otro ejemplo breve que incluye tanto eliminación como introducción de universales:

	1	$\forall x(Px \to Qx)$	*premisa*
	2	$\forall y(Qy \to Sy)$	*premisa*
a	3	$Pa \to Qa$	$\forall x \mathcal{E} : 1$
	4	$Qa \to Sa$	$\forall y \mathcal{E} : 2$
	5	Pa	*suposic*
	6	Qa	$\to \mathcal{E} : 3,5$
	7	Sa	$\to \mathcal{E} : 4,6$
	8	$Pa \to Sa$	$\to I : 5,7$
	9	$\forall x(Px \to Sx)$	$\forall x\, I : 8$

2.3.3 Cuantificadores existenciales

Introducción

La introducción del cuantificador existencial también es directa e intuitiva:

$$\dfrac{\varphi[x/a]}{\exists x \varphi}\ \exists x\, I$$

Si un elemento a tiene cierta propiedad, se puede afirmar que existe algún elemento que la tiene.

Ejemplo 2.69

1	$\forall x(Px \wedge Qx)$	*premisa*
2	$Pa \wedge Qa$	$\forall x\mathcal{E} : 1$
3	Pa	$\wedge\mathcal{E} : 2$
4	$Pa \vee Sa$	$\vee I : 3$
5	$\exists x(Px \vee Sx)$	$\exists x I : 4$

Eliminación

La eliminación del cuantificador existencial no se puede realizar directamente: ya se ha visto que aunque se conozca que 'alguien tiene cierta propiedad' no se puede derivar que 'tal elemento determinado la tiene'. De hecho, la regla de eliminación del existencial tiene esta otra forma (no muy intuitiva):

$$\dfrac{\exists x\varphi x \qquad \boxed{\begin{array}{ll} a & \varphi[x/a] \\ & \vdots \\ & \psi \end{array}}}{\psi}\ \exists x\mathcal{E}$$

Es decir, si se conoce que 'alguien tiene cierta propiedad' se abre una caja *con la suposición* de que es un determinado elemento, por ejemplo a. Esta constante no debe aparecer en ninguna fórmula previa (fuera de la caja). Tampoco se exportará posteriormente fuera de ese ámbito. De hecho, en ψ no debe aparecer este término.

Ejemplo 2.70

	1	$\exists x(Px \wedge Qx)$	*premisa*
a	2	$Pa \wedge Qa$	*suposic*
	3	Pa	$\wedge\mathcal{E} : 2$
	4	$\exists xPx$	$\exists x I : 3$
	5	$\exists xPx$	$\exists x\mathcal{E} : 1,2$

2.4 Tablas semánticas

2.4.1 Fórmulas proposicionales

Recordemos las propiedades básicas de los tableaux para fórmulas proposicionales:

1. se utilizan para decidir la satisfacibilidad de un conjunto de fórmulas; indirectamente, para decidir la relación de consecuencia entre una fórmula y un conjunto: niegue aquélla e incorpórela al conjunto inicial analizado

2. se construye un primer árbol, con una sola rama, que consta de tantos nodos como fórmulas haya en el conjunto inicial

3. las ramas se pueden bifurcar o ampliar linealmente; los nodos añadidos son subfórmulas adecuadas (negadas o no) de una fórmula en esa rama

4. una rama es satisfacible si lo es el conjunto de todas sus fórmulas; obviamente, si entre ellas se encuentran tanto una fórmula como su negación, la rama es insatisfacible

5. un árbol es satisfacible si lo es alguna de sus ramas

6. el arbol inicial es tan satisfacible como los sucesivos árboles ampliados; así, si se detecta que alguno de ellos es insatisfacible, también lo era el conjunto inicial de fórmulas

Ejemplo 2.71 Observe el árbol de la figura (fig. 2.7). Muestra que, de la hipótesis $p \rightarrow (q \wedge r)$, es consecuencia lógica $\neg r \rightarrow \neg p$. Más apropiadamente, que esto segundo se deduce de aquello primero. Pero en este sistema (consistente) todo lo que se deduce es consecuencia.

[nodos 1-2] El árbol inicial lo constituyen el nodo 1 (única hipótesis) y el nodo 2 (negación de la supuesta conclusión). Situados en el nodo más bajo de esta rama (el 2), tratamos de ampliarla descomponiendo adecuadamente alguna de las fórmulas de esa rama. Estratégicamente, se recomienda descomponer primero las fórmulas α (conjuntivas) y después las β (disyuntivas).

[nodos 3-5] Optamos por utilizar la fórmula 2 (α, conjuntiva): da lugar a una expansión lineal de esa rama (nodos 3 y 4). Y seguimos expandiendo: estamos en el nodo 4 y podemos expandir esa rama utilizando cualquiera de las fórmulas de la misma. Entre las diversas opciones, se opta por utilizar la propia fórmula 4: la 'doble negación' se puede expandir como 'sin negación' (nodo 5).

[nodos 6-7] En el nodo 5 se decide expandir la fórmula 1 (β, disyuntiva), que bifurca la rama (nodos 6 y 7). La rama del nodo 6 se puede cerrar inmediatamente: entre las fórmulas de esa rama (1-6) hay tanto una fórmula (la 6) como su negación (en 5).

[nodos 8-9] En la rama del nodo 7 no se observa aún este tipo de contradicción. Pero todavía se pueden expandir fórmulas de esa rama; en concreto, la propia fórmula 7 (de tipo α). Se obtienen los nodos 8 y 9. Y el nodo 9 es la negación de la fórmula del nodo 3, en esa misma rama desde la raíz.

El árbol se ha cerrado. Luego el conjunto de partida (φ en nodo 1, $\neg\psi$ en nodo 2) era insatisfacible. Así que se ha demostrado que $\varphi \vdash \psi$, es decir, que $\varphi \models \psi$.

Ejemplo 2.72 El árbol de la figura (fig. 2.8) es igual al del ejemplo previo (fig. 2.7). Compare las fórmulas de uno y otro: todas las fórmulas del segundo ejemplo se obtienen, por sustitución uniforme, de las del primero. Observe cómo se produce el cierre del segundo ejemplo: donde antes se cerraban en fórmulas atómicas, ahora se pueden cerrar en sus fórmulas sustituyentes.

El árbol de la figura (fig. 2.9) es ya una primera incursión en lógica de predicados. Sin embargo, no se van a necesitar aquí reglas adicionales: también es una instancia por sustitución del primer árbol considerado. Compruebe qué fórmula ha sustituido a qué proposición.

2.4.2 Notación uniforme

En lógica de predicados, no todas las relaciones de consecuencia entre fórmulas tienen raíz puramente conectiva (proposicional). Por eso se requieren nuevas reglas de inferencia. En concreto, en este sistema, se añadirán dos nuevas reglas:

1. una, para manipular fórmulas universalmente cuantificadas (tipo γ)

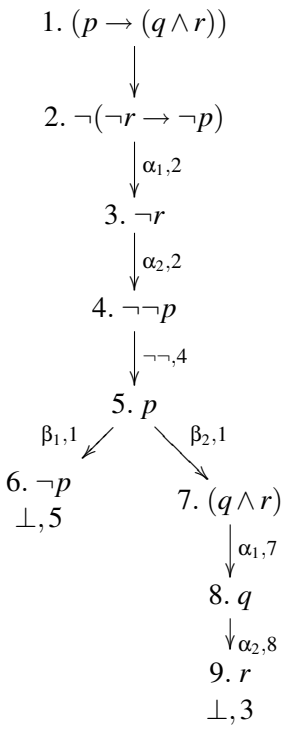


Figura 2.7: Tableau que confirma que $p \to (q \wedge r) \vdash \neg r \to \neg p$

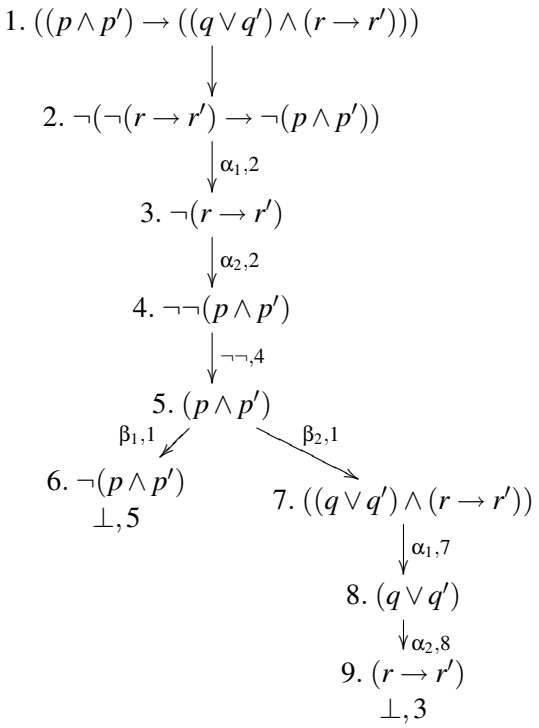


Figura 2.8: $(p \wedge p') \to ((q \vee q') \wedge (r \to r')) \vdash \neg(r \to r') \to \neg(p \wedge p')$

2. otra, para manipular fórmulas existencialmente cuantificadas (tipo δ)

Así, toda fórmula en este sistema es (intrínsecamente) conjuntiva (α), disyuntiva (β), universal (γ), existencial (δ) o una doble negación (ó $\bot$ ó $\top$).

La tabla (tabl. 2.4) resume las fórmulas que se consideran intrínsecamente universales (γ) y existenciales (δ), junto a su expansión. Todas ellas producen una expansión del árbol en un sólo nodo. No producen, por tanto, bifurcación del árbol.

Ejemplo 2.73 Considere la fórmula:

$$\neg\forall x(\forall z\neg(Pz \to Qx) \to \exists wRwa)$$

es de tipo existencial (δ). Su subfórmula inmediata:

$$(\forall z\neg(Pz \to Qx) \to \exists wRwa)$$

es de tipo (β). Sus subfórmulas inmediatas:

$$(\forall z\neg(Pz \to Qx), \qquad \exists wRwa$$

son, respectivamente, (γ) y (δ). Y la subfórmula:

$$\neg(Pz \to Qx)$$

es de tipo (α).

2.4.3 Reglas de expansión γ y δ

En ambos tipos de fórmulas la expansión aporta un sólo nodo. En concreto, la subfórmula inmediata: la fórmula que resta cuando se omite el cuantificador principal. Más precisamente, una instancia por sustitución de esta subfórmula. El párrafo siguiente expone cuáles pueden ser las cadenas sustituyentes. Más adelante se determinará qué se debe sustituir y con qué restricciones.

Parámetros Cada lenguaje de primer orden fija sus propias constantes y funciones. Si se pretende que el lenguaje sirva, por ejemplo, para razonar sobre números naturales, debe incluir una constante (que se asignará al 0) y una función (la función sucesor).

Las demostraciones sobre fórmulas de un lenguaje suelen requerir, como herramienta, el uso de constantes auxiliares. A estas constantes adicionales se les denomina 'parámetros'. Así, las demostraciones *sobre* fórmulas de un cierto lenguaje L se describen *usando* fórmulas del lenguaje L^{par}: extensión del L por adición de estas constantes.

Las cadenas sustituyentes en la expansión serán, según los casos, parámetros o términos cerrados: términos de L^{par} que no incluyen variables.

Regla de expansión δ Las fórmulas δ son del tipo $\exists x\phi$ ó $\neg\forall x\phi$. Su expansión es un único nodo de la forma $\phi(x/p)$ ó $\neg\phi(x/p)$ (respectivamente), donde todas las apariciones libres en ϕ de la variable del cuantificador se han sustituido por el mismo parámetro p. Este parámetro, esta constante auxiliar, *debe ser nueva* en el árbol: no puede figurar en ninguna fórmula previa (realmente, basta que sea nueva en la rama).

De una fórmula como $\exists xPx$ no tiene por qué ser consecuencia Pa. Sin embargo, si $\exists xPx$ es satisfacible, también lo será Pa. Y estamos trabajando en un sistema que trata de decidir la satisfacibilidad de un conjunto de fórmulas.

Cada instanciación debe hacerse sobre una constante nueva. De lo contrario, esta constante tendría unas propiedades (fijadas en otras fórmulas, donde aparece) que pueden modificar (innecesariamente) la decisión final sobre la satisfabilidad del conjunto.

Este cuidado se mantiene en toda manipulación de fórmulas de primer orden. Si no se considera, se puede llegar a afirmar (erróneamente) que 'siempre que hay alguien que tiene la propiedad P y hay alguien que tiene la propiedad Q, entonces alguien tiene ambas propiedades'. Un razonamiento erróneo (en deducción natural) sería: 'puesto que alguien tiene la propiedad P, llamémosle a, luego Pa. Puesto que alguien tienen la propiedad Q, llamémosle a, luego Qa. Entonces, $Pa \wedge Qa$. Luego alguien tiene (a la vez) la propiedad P y la Q'.

Regla de expansión γ Las fórmulas γ son del tipo $\forall x\phi$ ó $\neg\exists x\phi$. Su expansión es un único nodo de la forma $\phi(x/t)$ ó $\neg\phi(x/t)$ (respectivamente), donde todas las apariciones libres de la variable del cuantificador se han sustituido por el mismo término t. Este término debe ser cerrado: no debe incluir variables, sólo constantes y funciones de L o constantes auxiliares.

La particularización de una fórmula universal no requiere de ningún cuidado especial: lo que es verdadero para todo elemento del universo lo será para el que represente al término t.

Todas las reglas de expansión se recogen en la tabla (tabl. 2.5).

2.4.4 Ejemplos

Ejemplo 2.74 El árbol de la figura (fig. 2.10) muestra que:

$$\forall xPx \vee \exists yQy \vdash \exists y\forall x(Px \vee Qy)$$

- [nodos 1-2]Respectivamente, hipótesis y negación de la supuesta conclusión.
- [nodos 3-4] Expansión (β) de la fórmula 1. Estratégicamente, siempre es preferible expandir primero las fórmulas proposicionales (α y β), luego las existenciales (δ) y finalmente las universales (γ) para intentar cerrar.
- [nodo 5] Expansión de la fórmula 2 (universal, γ), porque no existía ninguna fórmula existencial en la rama 1-3. Se puede instanciar y por cualquier parámetro.
- [nodo 6] Expansión de la fórmula 5 (existencial, δ). El parámetro de la instanciación debe ser nuevo: b.
- [nodos 7-8]Expansión de la fórmula 6 (α)
- [nodo 9]Aprovechando que ya existe $\neg Pb$ en el nodo 8, expandiremos la fórmula 3 (universal). Como se puede escoger cualquier término, elegiremos instanciarla para b, cerrando esta rama.
- [nodos 10-14] Se procede de manera análoga (indicada en el árbol). Tan sólo merece reseñar la elección de parámetros. El de la línea 10 debía ser un parámetro nuevo en la rama. Se ha escogido incluso nuevo en el árbol (c).
- [nodos 11-14]El parámetro de la línea 11 proviene de una sustitución universal: podía ser cualquiera. Como ya existía Qc previamente, se escoge asimismo Qc. El parámetro de la línea 12 debía ser nuevo: d. La elección correcta de la instancia universal en la línea 11 acaba favoreciendo el cierre de esta última rama en 14.

Ejemplo 2.75 La tabla (2.11) confirma que

$$\forall x\exists y(Rxy \rightarrow Qy), \forall x\forall yRxy \vdash \exists zQz$$

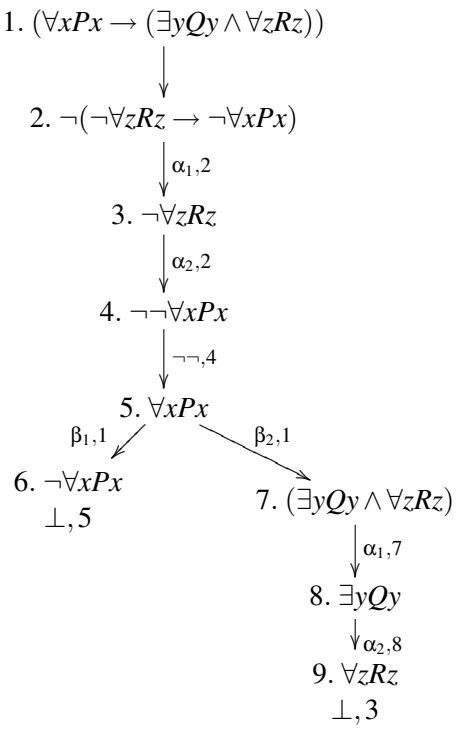


Figura 2.9: $\forall xPx \to (\exists yQy \wedge \forall zRz) \vdash \neg\forall zRz \to \neg\forall xPx$

conjuntivas			disyuntivas		
α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \vee Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \to Y)$	X	$\neg Y$	$X \to Y$	$\neg X$	Y
$\neg(X \leftarrow Y)$	$\neg X$	Y	$X \leftarrow Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\to Y$	X	$\neg Y$	$\neg(X \not\to Y)$	$\neg X$	Y
$X \not\leftarrow Y$	$\neg X$	Y	$\neg(X \not\leftarrow Y)$	X	$\neg Y$

Tabla 2.3: Notación uniforme, conectivas binarias

universales		existenciales	
γ	$\gamma(t)$	δ	$\delta(a)$
$\forall xX$	$X(t)$	$\exists xX$	$X(a)$
$\neg\exists xX$	$X(t)$	$\neg\forall xX$	$X(a)$

Tabla 2.4: Notación uniforme, fórmulas cuantificadas

conectivas monarias:	$\dfrac{\neg\neg X}{X}$ $\quad$ $\dfrac{\neg\top}{\bot}$ $\quad$ $\dfrac{\neg\bot}{\top}$	
conectivas binarias:	$\dfrac{\alpha}{\alpha_1}$ $\quad$ $\dfrac{\beta}{\beta_1 \mid \beta_2}$	
	α_2	
cuantificador universal:	$\dfrac{\gamma}{\gamma(t)}$	(para cualquier término t cerrado de L^{par})
cuantificador existencial:	$\dfrac{\delta}{\delta(p)}$	(para cualquier parámetro p nuevo)

Tabla 2.5: Reglas de expansión de un tableau

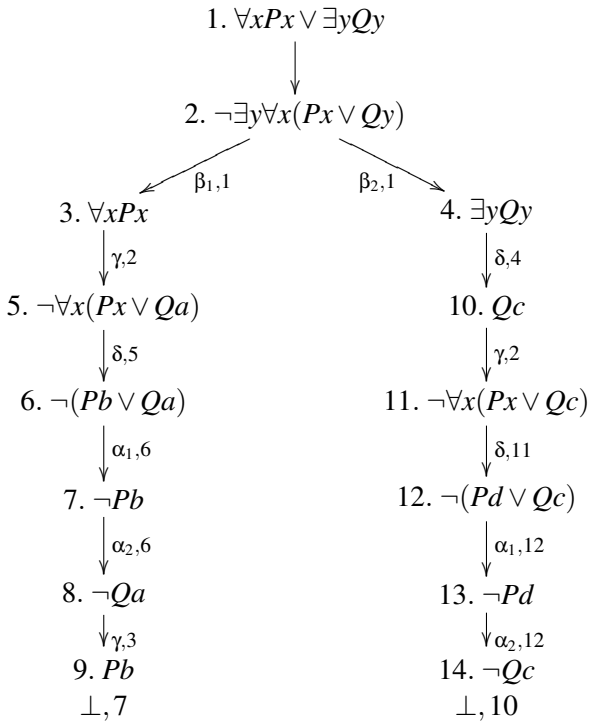


Figura 2.10: Tableau que confirma que $\forall xPx \vee \exists yQy \vdash \exists y\forall x(Px \vee Qy)$

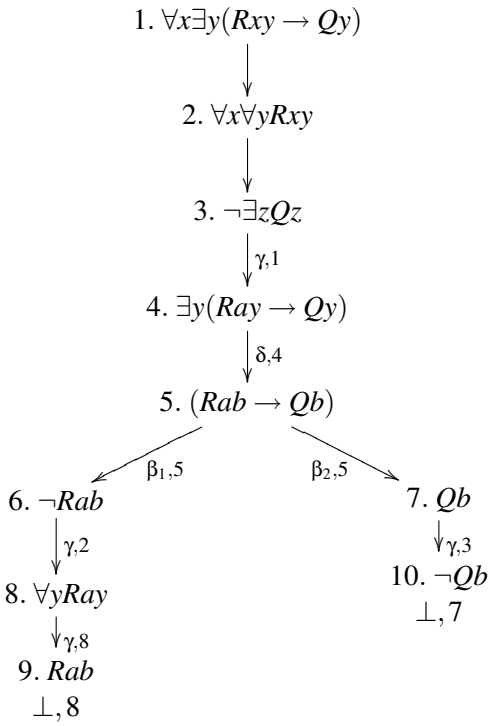


Figura 2.11: Tableau que confirma que $\forall x\exists y(Rxy \to Qy), \forall x\forall yRxy \vdash \exists zQz$

2.5 Resolución

2.5.1 Forma prenexa

Toda fórmula de primer orden admite infinitas fórmulas equivalentes: fórmulas satisfacibles exactamente por los mismos modelos. Dos fórmulas equivalentes son dos expresiones sintácticas distintas que 'significan lo mismo' (sobre esta semántica).

> El objetivo de esta sección se resume en pocas palabras: "dada una fórmula cualquiera, encontrar una fórmula equivalente tal que todos sus cuantificadores estén situados *al principio* de la fórmula."

Ejemplo 2.76 El lector aprenderá, a lo largo de esta sección, a calcular que, dada la fórmula:

$$(\forall xPx \wedge \exists yQy) \to \forall t\exists wRtw$$

cualquiera de las siguientes fórmulas son equivalentes a ella (y por tanto, entre sí):

$$\exists x\forall y\forall t\exists w((Px \wedge Qy) \to Rtw)$$

$$\forall y\exists x\forall t\exists w((Px \wedge Qy) \to Rtw)$$

$$\forall t\exists x\forall y\exists w(\neg(Px \wedge Qy) \vee Rtw)$$

Sin embargo, entre otras muchas, no será equivalente a las anteriores:

$$\exists w\exists x\forall y\forall t(\neg(Px \wedge Qy) \vee Rtw)$$

ni ninguna otra donde $\exists w$ aparezca antes de $\forall t$.

Lo primero que hay que fijar es que esto es posible: efectivamente, se demuestra que para toda fórmula existen fórmulas prenexas equivalentes. Inmediatamente después la pregunta es cómo: ¿existe un procedimiento que permita calcular una fórmula prenexa equivalente a una fórmula dada?. El desarrollo de esta sección responde esta pregunta.

Equivalencias básicas El Teorema de Reemplazo permite calcular sintácticamente fórmulas equivalentes entre sí: reemplace, en la fórmula inicial, una de sus subfórmulas por otra equivalente y obtendrá una fórmula global final equivalente a la fórmula inicial.

Este teorema comienza a ser útil cuando se dispone de algunas equivalencias elementales, que permiten sustituir unas subfórmulas por otras. Para nuestro objetivo, entre otras, será imprescindible utilizar las equivalencias mostradas en la tabla (tabl. 2.6).

Siempre que la función ψ no contenga apariciones libres de la variable x:

$$\begin{array}{llll} \forall x\phi \wedge \psi & \equiv & \forall x(\phi \wedge \psi) & \qquad \psi \wedge \forall x\phi \equiv \forall x(\psi \wedge \phi) \\ \forall x\phi \vee \psi & \equiv & \forall x(\phi \vee \psi) & \qquad \psi \vee \forall x\phi \equiv \forall x(\psi \vee \phi) \\ & & & \\ \exists x\phi \wedge \psi & \equiv & \exists x(\phi \wedge \psi) & \qquad \psi \wedge \exists x\phi \equiv \exists x(\psi \wedge \phi) \\ \exists x\phi \vee \psi & \equiv & \exists x(\phi \vee \psi) & \qquad \psi \vee \exists x\phi \equiv \exists x(\psi \vee \phi) \end{array}$$

Tabla 2.6: Equivalencias de interés para alcanzar la forma prenexa

A estas equivalencias habrá que añadir otras que se utilizarán para:

- introducir negaciones: $\neg\forall x\phi \equiv \exists x\neg\phi, \quad \neg\exists x\phi \equiv \forall x\neg\phi,$ (más las equivalencias de De Morgan)
- renombrar variables: $\forall x\phi \equiv \forall y\phi(x/y), \quad \exists x\phi \equiv \exists y\phi(x/y)$
- permutar cuantificadores del mismo tipo: $\forall x\forall y\phi \equiv \forall y\forall x\phi, \quad \exists x\exists y\phi \equiv \exists y\exists x\phi$

Ejemplo 2.77 Dada la fórmula $\forall xPx \vee \exists yQy$, si se pretende 'desplazar el cuantificador $\forall x$', se observa que es de la forma $\forall x\phi \vee \psi$ donde ψ no tiene apariciones libres de x. Luego:

$$\forall xPx \vee \exists yQy \equiv \forall x(Px \vee \exists yQy)$$

Alternativamente, la fórmula inicial era también de la forma $\psi \vee \exists y\phi$ donde en ψ no hay apariciones libres de y (del cuantificador que se desea desplazar). Luego, también:

$$\forall xPx \vee \exists yQy \equiv \exists y(\forall xPx \vee Qy)$$

Siguiendo con este último caso, la subfórmula $(\forall xPx \vee Qy)$ es de la forma $\forall x\phi \vee \psi$, donde ψ no contiene x libres, luego $(\forall xPx \vee Qy) \equiv \forall x(Px \vee Qy)$. Y por tanto:

$$\forall xPx \vee \exists yQy \equiv \exists y(\forall xPx \vee Qy) \equiv \exists y\forall x(Px \vee Qy)$$

Alternativamente se podía haber obtenido:

$$\forall xPx \vee \exists yQy \equiv \forall x(Px \vee \exists yQy) \equiv \forall x\exists y(Px \vee Qy)$$

Le recomendamos que interprete este sencillo ejemplo sobre algún universo pequeño (de 2 o 3 elementos). Ajuste la interpretación hasta que satisfaga a una de las fórmulas. Compruebe que todas las demás fórmulas equivalentes se satisfacen. Si tiene dificultades para interpretar alguna de las fórmulas convendría, en este punto, que repase las notas sobre semántica de primer orden.

Ejemplo 2.78 Considere la fórmula $\exists xPx \rightarrow \exists xQx$. Puesto que es una fórmula condicional:

$$\exists xPx \rightarrow \exists xQx \equiv \neg\exists xPx \vee \exists xQx \equiv \forall x\neg Px \vee \exists xQx \equiv \forall x(\neg Px \vee \exists xQx)$$

Este último paso se justifica porque la fórmula era de la forma $\forall x\phi \vee \psi$, donde en ψ no había x libres. Conviene que lo contraste sobre una interpretación sencilla.

No obstante, llegados a este punto no se puede conseguir la forma prenexa final. En la subfórmula $(\neg Px \vee \exists xQx)$ no se puede 'desplazar' de forma equivalente el cuantificador $\exists x$ puesto que la 'otra fórmula' $(\neg Px)$ sí contiene apariciones libres de la variable del cuantificador:

$$(\neg Px \vee \exists xQx) \not\equiv \exists x(\neg Px \vee Qx)$$

Basta ver que $\langle U = \{1,2\}$, con $P = \{1\}, Q = \emptyset\rangle$ satisface la fórmula derecha: efectivamente existe un elemento (2) que satisface $(\neg Px \vee Qx)$. Esta misma interpretación, con la asignación $x^A = 1$ para la variable libre x sigue satisfaciendo la fórmula derecha, puesto que no hay variables libres en ellas. Pero ya no satisface la fórmula la izquierda.

Sin embargo, puesto que $\exists xQx \equiv \exists yQy$, aún se puede obtener:

$$(\neg Px \vee \exists xQx) \equiv (\neg Px \vee \exists yQy) \equiv \exists y(\neg Px \vee Qy)$$

Y se consigue la fórmula prenexa equivalente a la inicial:

$$(\exists xPx \rightarrow \exists xQx) \equiv (\neg\exists xPx \vee \exists xQx) \equiv (\forall x\neg Px \vee \exists xQx) \equiv \forall x(\neg Px \vee \exists xQx) \equiv \forall x(\neg Px \vee \exists yQy) \equiv \forall x\exists y(\neg Px \vee Qy)$$

En este último ejemplo siga la pista del cuantificador existencial del antecedente: acaba produciendo un cuantificador universal en la cabecera de la fórmula prenex. Intrínsecamente, su significado no era existencial sino universal puesto que estaba en el ámbito de una negación. Este comportamiento es generalizable y produce nuevas relaciones de equivalencia:

Siempre que la función ψ no contenga apariciones libres de la variable x:

$\forall x\phi \rightarrow \psi$	$\equiv$	$\exists x(\phi \rightarrow \psi)$	$\psi \rightarrow \forall x\phi$	$\equiv$	$\forall x(\psi \rightarrow \phi)$
$\exists x\phi \rightarrow \psi$	$\equiv$	$\forall x(\phi \rightarrow \psi)$	$\psi \rightarrow \exists x\phi$	$\equiv$	$\exists x(\psi \rightarrow \phi)$

Tabla 2.7: Otras equivalencias de interés para alcanzar la forma prenexa

Ejemplo 2.79 Los inconvenientes mostrados en el ejemplo previo se pueden evitar desde el principio. Basta renombrar variables inicialmente. Así, la fórmula:

$$\forall y(\exists xRxy \rightarrow \neg(\exists xSxy \vee \forall yPy))$$

puede reescribirse equivalentemente como:

$$\forall z(\exists xRxz \rightarrow \neg(\exists wSwz \vee \forall yPy))$$

donde cada cuantificador afecta a una variable distinta. A partir de aquí, uno de los posibles caminos hacia la forma prenexa es:

1.	$\forall y(\exists x Rxy \rightarrow \neg(\exists x Sxy \vee \forall y Py))$	$\equiv$	$\forall z(\exists x Rxz \rightarrow \neg(\exists w Swz \vee \forall y Py))$	[renombrar variables]
2.		$\equiv$	$\forall z(\neg\exists x Rxz \vee \neg(\exists w Swz \vee \forall y Py))$	[eliminar $\rightarrow$]
3.		$\equiv$	$\forall z(\forall x\neg Rxz \vee \neg(\exists w Swz \vee \forall y Py))$	[introducir $\neg$]
4.		$\equiv$	$\forall z(\forall x\neg Rxz \vee (\neg\exists w Swz \wedge \neg\forall y Py))$	[introducir $\neg$]
6.		$\equiv$	$\forall z(\forall x\neg Rxz \vee (\forall w\neg Swz \wedge \exists y\neg Py))$	[introducir $\neg$]
7.		$\equiv$	$\forall z(\forall x\neg Rxz \vee \exists y(\forall w\neg Swz \wedge \neg Py))$	[desplaz. cuantif.]
8.		$\equiv$	$\forall z(\forall x\neg Rxz \vee \exists y\forall w(\neg Swz \wedge \neg Py))$	[desplaz. cuantif.]
9.		$\equiv$	$\forall z\exists y(\forall x\neg Rxz \vee \forall w(\neg Swz \wedge \neg Py))$	[desplaz. cuantif.]
10.		$\equiv$	$\forall z\exists y\forall x(\neg Rxz \vee \forall w(\neg Swz \wedge \neg Py))$	[desplaz. cuantif.]
11.		$\equiv$	$\forall z\exists y\forall x\forall w(\neg Rxz \vee (\neg Swz \wedge \neg Py))$	[desplaz. cuantif.]

Los cuantificadores de una fórmula prenexa constituyen la cabecera de la fórmula. La subfórmula restante (sin cuantificadores ya) se denomina *matriz* de la fórmula.

En este punto, el lector puede volver al ejemplo (2.76). Trate de obtener las fórmulas prenexas allí expuestas e intente justificar por qué no se podían desplazar los cuantificadores en cualquier orden.

2.5.2 Funciones de Skolem

En la sección (secc. 2.4.3) ya se presentó el concepto de *parámetro*: constantes auxiliares que se introducían en el curso de una demostración. Facilitaban las demostraciones sobre las fórmulas de un cierto lenguaje aunque no eran constantes de ese lenguaje.

Las derivaciones por resolución requieren introducir no sólo constantes auxiliares sino también funciones auxiliares, que se denominan constantes y funciones de Skolem. Su elección adecuada en cada caso suele denominarse 'skolemización' ó 'eskolemización' de una fórmula. Se utilizan para eliminar cuantificadores existenciales a costa de introducir tales elementos auxiliares.

> El objetivo de esta sección es: " dada una fórmula que puede incluir cuantificadores existenciales, generar otra fórmula sin tales cuantificadores pero que sea *tan satisfacible como la anterior*".
>
> Observe que el proceso que se va a describir no genera necesariamente una fórmula equivalente a la anterior. Es decir, no en *todos* los modelos donde la inicial se satisfaga lo hará la resultante. Sin embargo, si existe algún modelo donde la inicial se satisface existirá otro (quizá distinto) que satisface la resultante.

Ejemplo 2.80 Dada una fórmula como $\exists x\forall y Rxy$, su 'eskolemización' produce $\forall y Ray$. Si la fórmula con el existencial era satisfacible en algún modelo es intuitivo es porque existía una asignación para x en ese modelo tal que $\forall y Rxy$ se satistacía. Utilizando ese mismo elemento del universo para interpretar la constante a se consigue que $\forall y Ray$ sea también satisfacible.

Obviamente, sobre ese mismo modelo, si no se interpreta adecuadamente la constante se satisfará la fórmula existencial pero no la otra. De hecho,

$$\forall y Ray \models \exists x\forall y Rxy \text{ pero } \exists x\forall y Rxy \not\models \forall y Ray$$

La reflexión anterior sólo nos sirve para resaltar que no son expresiones equivalentes. Quedémonos con la propiedad que efectivamente conserva este proceso:

$$Satisfacible(\exists x\forall y Rxy) \text{ si sólo si } Satisfacible(\forall y Ray)$$

El camino total en que estamos inmersos puede describirse así: "para confirmar que una fórmula es consecuencia de otras, niéguela e inclúyala entre éstas; escriba todas estas fórmulas en forma clausulada y determine si este conjunto es insatisfacible por Resolución."

La eskolemización forma parte del proceso de escritura de una fórmula en su forma clausulada. Luego la forma clausulada no es equivalente a la fórmula inicial, sino tan sólo igualmente satisfacible. Pero eso es justo lo que requiere un sistema deductivo que trata de detectar sintácticamente la insatisfacibilidad.

La explicación del proceso de eskolemización se simplifica bastante si se considera que las fórmulas están en forma prenexa. No obstante, no es un paso previo necesario.

Ejemplo 2.81 Una fórmula como $\exists x \forall y Qyx$ podría expresar 'existe alguien (al menos uno) tal que todo el mundo le quiere'. Con esta misma lectura, $\forall y \exists x Qyx$ se entendería como 'todo el mundo tiene alguien (al menos uno) al que quiere'.

En la primera fórmula, puesto que hay alguien a quien todo el mundo quiere, llamémosle a y escribamos $\forall y Qya$ 'todo el mundo quiere a a'. Recuerde que estas dos fórmulas no son equivalentes pero sí igualmente satisfacibles.

En el segundo caso no se puede elegir una constante como receptor universal del cariño de todos. De hecho en $\forall y \exists x Qyx$ cada elemento debe apreciar al menos a uno, que puede coincidir o diferir del apreciado por otro elemento. La fórmula $\forall y Qyf(y)$ es más restrictiva: ahora cada elemento y se relaciona sólo con uno, su $f(y)$, aunque aún se puede coincidir en el afecto a un mismo elemento. Volviendo al formalismo puro y duro: $\forall y \exists x Qyx$ es satisfacible si y sólo si $\forall y Qyf(y)$ lo es.

En general, dada una fórmula prenexa, para eliminar un existencial es preciso considerar cuántos cuantificadores universales la preceden. La variable del existencial, en la matriz, se debe sustituir por una función *nueva* de las variables de los cuantificadores universales precedentes.

Ejemplo 2.82 Al eliminar (en el proceso de eskolemización) los existenciales de las siguientes fórmulas se obtiene:

1. $\forall x \forall y \exists z (Qzx \vee Syz) \rightsquigarrow \forall x \forall y (Qf(xy)x \vee Syf(xy))$
2. $\forall x \forall y \exists z \forall w (Rzxw \vee Syz) \rightsquigarrow \forall x \forall y \forall w (Rf(xy)xw \vee Syf(xy))$
3. $\forall x \forall y \exists z \exists t \forall w (Rzxw \vee Tyzt) \rightsquigarrow \forall x \forall y \forall w (Rf(xy)xw \vee Tyf(xy)g(xy))$
4. $\forall x \exists t \forall y \exists z \forall w (Rzxw \vee Tyzt) \rightsquigarrow \forall x \forall y \forall w (Rf(xy)xw \vee Tyf(xy)h(x))$
5. $\exists t \forall x \forall y \exists z \forall w (Rzxw \vee Tyzt) \rightsquigarrow \forall x \forall y \forall w (Rf(xy)xw \vee Tyf(xy)a)$
6. $\exists t \exists z \forall x \forall y \forall w (Rzxw \vee Tyzt) \rightsquigarrow \forall x \forall y \forall w (Rbxw \vee Tyba)$

Las constantes de Skolem se pueden interpretar como una función de Skolem de cero argumentos; es decir, se producen cuando no hay ningún cuantificador universal delante del existencial eliminado.

2.5.3 Forma clausulada

Dada una fórmula inicial ϕ cualquiera se puede obtener una expresión prenexa equivalente ϕ_{prenex}. Si incluyera existenciales, mediante constantes y funciones de Skolem, se puede obtener una fórmula prenexa sin ellos ϕ^{sko}_{prenex}, que resulta tan satisfacible como las anteriores. Supuesta esta fórmula ϕ^{sko}_{prenex},

el objetivo de esta sección es reescribir equivalentemente la matriz de esta fórmula hasta expresarla como una conjunción de cláusulas.

Si se han seguido los pasos sugeridos en las secciones previas, sobre la fórmula original:

1. se independizaron los ámbitos, renombrando las variables utilizadas en más de un cuantificador
2. se eliminaron bicondicionales (reescribiéndolos como condicionales) y condicionales (como 'no antecedente ó consecuente')
3. se introdujeron las negaciones (cambiando el tipo de cuantificador y utilizando las leyes de De Morgan)
4. se desplazaron sucesivamente todos los cuantificadores a la cabeza; *estratégicamente conviene desplazar al inicio los existenciales primero, si es posible*
5. se eliminaron los existenciales mediante el uso de constantes y funciones de Skolem (constantes y funciones nuevas en la fórmula para cada existencial eliminado)

En este punto se tiene una fórmula matriz, con sus variables universalmente cuantificadas en la cabecera, donde las negaciones no afectan más que a fórmulas atómicas (están 'pegadas directamente a los predicados') y donde las únicas conectivas binarias son conjunciones y disyunciones. Basta entonces aplicar adecuadamente la distributividad entre ellas para reescribir la matriz como una conjunción de cláusulas.

Un ejemplo de cláusula es: $(Pa \vee Rxf(x) \vee \neg Qby)$ donde las fórmulas atómicas (negadas o no) están unidas por disyunciones. Si se denomina *literal* a una fórmula atómica o a su negación, una cláusula C se puede definir como una disyunción de literales: $C = (l_1, \ldots, l_n)$. La matriz debe tener la forma de una conjunción de tales cláusulas: $C_1 \wedge \ldots \wedge C_k$.

Ejemplo 2.83 Los pasos que se han enumerado se pueden apreciar en el siguiente desarrollo:

$$\begin{array}{rcl@{\qquad}rcl}
 & \exists x\forall y Rxy \rightarrow (\exists x Px \wedge \forall y Qy) & & & \equiv & \exists x\forall y Rxy \rightarrow (\exists z Pz \wedge \forall w Qw) \\
\equiv & \neg(\exists x\forall y Rxy) \vee (\exists z Pz \wedge \forall w Qw) & & \equiv & (\forall x\neg\forall y Rxy) \vee (\exists z Pz \wedge \forall w Qw) \\
\equiv & (\forall x\exists y\neg Rxy) \vee (\exists z Pz \wedge \forall w Qw) & & \equiv & \forall x\exists y(\neg Rxy) \vee \exists z\forall w(Pz \wedge Qw) \\
\equiv & \exists z\forall x\exists y\forall w(\neg Rxy \vee (Pz \wedge Qw)) & & \rightsquigarrow & \forall x\forall w(\neg Rxf(x) \vee (Pa \wedge Qw)) \\
\equiv & \forall x\forall w((\neg Rxf(x) \vee Pa) \wedge (\neg Rxf(x) \vee Qw)) & & &
\end{array}$$

La matriz de esta última fórmula está efectivamente en forma normal conjuntiva: consta de dos cláusulas unidas por una conjunción. Ambas cláusulas comparten algunas variables en común (en concreto, x). En este punto siempre se puede (y se debe, ya que será útil) evitar conexión entre cláusulas:

$$\forall x\forall w((\neg Rxf(x) \vee Pa) \wedge (\neg Rxf(x) \vee Qw))$$

es equivalente a

$$\forall x((\neg Rxf(x) \vee Pa) \wedge \forall w(\neg Rxf(x) \vee Qw))$$

utilizando, en el otro sentido, la equivalencia que permitía desplazar cuantificadores a la cabeza, y

$$(\forall x(\neg Rxf(x) \vee Pa) \wedge \forall x\forall w(\neg Rxf(x) \vee Qw))$$

puesto que $\forall x(\phi \wedge \psi) \equiv (\forall x\phi \wedge \forall x\psi)$ distributividad que presenta el universal sólo frente a la conjunción (así como el existencial frente a la disyunción). Renombrando variables:

$$(\forall x(\neg Rxf(x) \vee Pa) \wedge \forall t\forall w(\neg Rtf(t) \vee Qw))$$

y volviendo a colocar los cuantificadores en cabeza:

$$\forall x \forall t \forall w((\neg Rxf(x) \vee Pa) \wedge (\neg Rtf(t) \vee Qw))$$

En resúmen: *llegados a la forma normal conjuntiva, a cláusulas distintas se les puede (y debe) independizar sus variables comunes, renombrándolas*.

Lo que resta es un casi un mero cambio notacional: prescinda de los cuantificadores y simplemente muestre el conjunto de cláusulas de la fórmula:

$$\{(\neg Rxf(x) \vee Pa), (\neg Rtf(t) \vee Qw)\}$$

o, incluso, defina cada cláusula como el conjunto de sus literales:

$$\{\{\neg Rxf(x), Pa\}, \{\neg Rtf(t), Qw\}\}$$

2.5.4 Unificación

Introducción

Recordemos brevemente el Principio de Resolución para lógica de proposiciones. Suponga que una interpretación satisface las siguientes dos cláusulas:

$$(p \vee \neg r \vee t) \qquad (r \vee \neg q)$$

tales que una contiene un literal (r) y la otra su complementario ($\neg r$). Sea cual sea el valor asignado a r esta interpretación sólo garantiza la satisfacción de una de las dos cláusula. En la otra, debe ser verdadero algún literal para que la cláusula lo sea. Esto garantiza que, en una cláusula que los contenga a todos (salvo r y $\neg r$), al menos uno de estos literales será verdadero frente a la misma interpretación. Es decir, que *toda* interpretación que satisface:

$$(p \vee \neg r \vee t) \qquad (r \vee \neg q)$$

también satisface

$$(p \vee t \vee \neg q)$$

No es difícil aceptar el mismo principio sobre cláusulas con predicados sin variables. Toda interpretación que satisface:

$$(Pa \vee \neg Rba \vee Tcb) \qquad (Rba \vee \neg Qe)$$

también satisface

$$(Pa \vee Tcb \vee \neg Qe)$$

Este principio no se verifica si los alguno de los términos del literal sobre el que se resuelve fueran distintos: si en vez de $\neg Rba$ y Rba aparecieran $\neg Rba$ y Rca. Otra cosa sería si se cuenta, respectivamente en cada cláusula, con $\neg Rxa$ y Rba. Por ejemplo, si se parte de las cláusulas:

$$(Pa \vee \neg Rxa \vee Tcx) \qquad (Rba \vee \neg Qe)$$

Sobre ellas no es aplicable directamente el principio de resolución previo; no obstante, si son satisfacibles ambas también lo serán las siguientes dos cláusulas:

$$(Pa \vee \neg Rba \vee Tcb) \qquad (Rba \vee \neg Qe)$$

obtenidas de las anteriores por instanciación (por sustitución) de la variable x por el término b. Y ya sí se podrá aplicar el principio de resolución, generando la cláusula:

$$(Pa \vee Tcb \vee \neg Qe)$$

Al proceso previo de 'instanciación' se le denomina *unificación*. Se produce utilizando sustituciones. El objetivo es que en dos referencias a un mismo predicado sus términos (los 'sujetos' del predicado') lleguen a ser respectivamente idénticos en ambas referencias.

Respetada cierta condición, este proceso de sustitución no altera la satisfacibilidad. Si dos cláusulas eran satisfacibles, también lo serán las cláusulas resultantes de estas sustituciones. Además, si lo que se ha unificado son literales complementarios (un predicado en una cláusula y su negación en la otra), se puede aplicar el Principio de Resolución. Éste nos facilita adicionalmente una tercera cláusula a partir de estas dos, tan satisfacible como ellas.

Puesto que ninguna de las acciones que se ejecutan (unificación y resolución) alteran la satisfacibilidad: si se parte de un conjunto de cláusulas satisfacible nunca se obtendrá la cláusula vacía (señal sintáctica inequívoca de contradicciones en el conjunto analizado).

Para fijar ideas, si se parte de 10 cláusulas y se utilizan la unificación y resolución adecuadamente, se va ampliando este conjunto de cláusulas garantizando que es tan satisfacible como el de partida. Si el proceso de ampliación incluye la cláusula vacía, se puede ya afirmar que el conjunto inicial de cláusulas (o cualquiera de sus sucesivas expansiones) es insatisfacible.

Después de esta declaración de intenciones y de rumbo, 'sólo' faltan adjuntar las definiciones y algoritmos que las materializan.

Unificadores

Considere las siguientes dos fórmulas atómicas:

$$P(x, f(g(c)), x) \qquad P(b, f(y), b)$$

donde el *mismo predicado* ternario P está relacionando, respectivamente, ternas de términos distintos. Si las dos ternas fueran idénticas, estas dos fórmulas ya estarían unificadas. Y si el predicado en una y otra no fuera el mismo, no serían unificables.

Observe también que las definiciones sobre unificación, sin más, no requieren contar con un predicado $P()$ y su negación $\neg P()$. Se centran sólo en los términos que hay 'dentro' del predicado. Será posteriormente, cuando se aplique Resolución, cuando se escogerán pares tales que uno sólo esté además negado.

Ejemplo 2.84 Para unificar las fórmulas mencionadas:

$$P(x, f(g(c)), x) \qquad P(b, f(y), b)$$

se puede intentar inicialmente unificar los primeros términos. Como uno de ellos es una variable, la sustitución $\sigma_1 = x/b$ servirá:

$$P(b, f(g(c)), b) \qquad P(b, f(y), b)$$

Los segundos términos empiezan igual en ambos predicados. Cuando comienzan a diferir, aparece una variable y en uno y otro subtérmino en el otro. Basta ejecutar la sustitución $\sigma_2 = y/g(c)$. Observe que no bastaría con la sustición y/c: la variable debe sustituirse por el subtérmino máximo en que no coinciden. El resultado final de este proceso es:

$$P(b, f(g(c)), b) \qquad P(b, f(g(c)), b)$$

que se obtiene de las cláusulas originales por aplicación sucesiva de σ_1 y σ_2. Una sustitución total, única, que produce este resultado es $\sigma_3 = \sigma_1\sigma_2 = \{x/b, y/g(c)\}$

La unificación de todos los pares de términos correspondientes de un par de predicados se puede descomponer en pasos. Cada uno de ellos considera sucesivamente la unificación de *un único par de términos* correspondientes. Hasta que se diga lo contrario, nos limitaremos a estudiar este problema.

Ejemplo 2.85 Si se precisara unificar los términos $t_1 = f(g(c,h(x)))$ y $t_2 = f(g(c,y))$ se deberían recorrer ambos desde la izquierda hasta que se detecte la posición en que comienzan a diferir. En este caso difieren en los subtérminos $h(x)$ en t_1 e y en t_2. Es decir, el árbol de t_1 y t_2 es idéntico salvo en estos dos subárboles respectivamente. A estos subtérminos de desacuerdo se les denominará subtérminos críticos.

En este caso, como uno de los subtérminos críticos es una variable, basta efectuar la sustitución de esta variable por el otro subtérmino: $y/h(x)$.

No es posible unificar si ninguno de los dos subtérminos es una variable, sino constantes o funciones. Por ejemplo: en $t_1 = f(g(c,h(x)))$ y $t_2 = f(g(c,b))$ ó en $t_1 = f(g(c,h(x)))$ y $t_2 = f(g(c,g(x)))$.

Ejemplo 2.86 Al intentar unificar $t_1 = f(x,h(g(x))$ y $t_2 = f(g(y),h(z))$ se observa que el primer par de subtérminos críticos es x en t_1 y $g(y)$ en t_2. Si se consigue unificar ambos aún habrá que unificar los subtérminos críticos z y $g(x)$.

Para unificar el primer par de subtérminos críticos basta ejecutar la sustitución $\sigma_1 = x/g(y)$. Se obtiene entonces $t_1 = f(g(y),h(g(g(y))))$ y $t_2 = f(g(y),h(z))$. Luego, los siguientes subtérminos críticos unificables ya no son los que se detectaron: ahora son z en t_2 y $g(g(y))$ en t_1. Basta ejecutar la sustitución $\sigma_2 = z/g(g(y))$ para obtener: $t_1 = t_2 = f(g(y),h(g(g(y))))$.

Una sola sustitución σ que unifica ambos términos se calcula por composición de las anteriores: $\sigma = \sigma_2\sigma_1$. Y es tal que $t_1\sigma = t_2\sigma$.

Unificación de un par de términos Para un único par t_1 y t_2 de términos (de argumentos de un predicado), se puede proceder a su unificación mediante el siguiente algoritmo:

Sea σ inicialmente la sustitución identidad (la que sustituye cada variable por sí misma);

*W*HILE $t_1\sigma \neq t_2\sigma$ *DO*:

(principio)

determine un par de subtérminos críticos s_1 y s_2 entre $t_1\sigma$ y $t_2\sigma$

IF ni s_1 ni s_2 es una variable

THEN "no unificables"

ELSE

sea *var* la variable s_1 o s_2 (si ambas lo son, escoja una)

sea *term* el otro término (s_1 o s_2) no asignado a *var*

IF la variable *var* aparece en *term*

THEN "no unificables"

ELSE recalcule σ componiéndola con la sustitución *var/term*; es decir: $\sigma := \sigma\{var/term\}$

(fin)

En el pseudocódigo anterior, "no unificables" supone una salida del bucle, sin éxito. Puede deberse a que ninguno de los subtérminos críticos sea una variable, caso del par '$h(f(x))$' y '$f(a)$' ó del par 'b' y '$g(w)$'. O puede deberse a que se intenta unificar pares como 'x' y '$f(x)$' ó 'z' y 'g(h(z,a))', donde la variable sustituida aparece en el término que debe sustituirla.

En este último caso, la unificación de 'x' y de '$f(x)$', que se puede interpretar como la solución de $x = f(x)$, no es generalmente garantizable. A la comprobación de que la variable no aparece en el término sustituyente se la denominará 'comprobación de aparición' (occurs check, en inglés).

Ejemplo 2.87 Tratemos de unificar los términos

$$t_1 = f(x, g(x,a), z) \qquad t_2 = f(h(z), y, g(b,y))$$

El primer par de subtérminos críticos es x y $h(z)$. Como uno de ellos es una variable (x) y ésta no ocurre en el otro término: se efectúa la sustitución $\sigma_1 = x/h(z)$. Entonces:

$$t_1\sigma = f(h(z), g(h(z),a), z) \qquad t_2\sigma = f(h(z), y, g(b,y))$$

Todavía $\sigma = \sigma_1$ es tal que $t_1\sigma \neq t_2\sigma$, luego se continúa ejecutando el bucle. El siguiente par de subtérminos críticos es y y $g(h(z),a)$. Igual que en el caso anterior, no hay inconveniente: se efectúa la sustitución $\sigma_2 = y/g(h(z),a)$. Entonces:

$$t_1\sigma = f(h(z), g(h(z),a), z) \qquad t_2\sigma = f(h(z), g(h(z),a), g(b, g(h(z),a)))$$

Como para este $\sigma = \sigma_1\sigma_2$ aún se verifica $t_1\sigma \neq t_2\sigma$, se continúa la ejecución. El siguiente par de subtérminos críticos es:

$$z \qquad g(h(z),a)$$

donde la variable z ocurre en el término $g(h(z),a)$, luego no resultan unificables.

Si el algoritmo propuesto finaliza con éxito no sólo se obtiene la unificación de ambos términos, sino que se garantiza además que se han unificado con el unificador más general posible.

Para introducir este último concepto observe que $f(x)$ y $f(y)$ se pueden unificar mediante la sustitución $\sigma = \{x/a, y/a\}$, o también mediante $\tau = \{x/h(z), y/h(z)\}$, o también mediante $\mu = \{x/y\}$. El unificador μ es el unificador más general porque 'instancia, particulariza lo mínimo necesario para que ambos términos coincidan'. Particularizar ambos términos a la misma constante a también produce la unificación, pero es una unificación más fuerte que la necesaria.

En concreto, si se unifican ambos términos mediante μ se obtiene: $t_1\mu = t_2\mu = f(y)$. Desde aquí aún se puede obtener el mismo resultado que hubiera supuesto la aplicación directa de σ: $t_1\sigma = t_2\sigma = f(a)$. Basta aplicar la sustitución $\eta\{y/a\}$ sobre $t_1\mu = t_2\mu = f(y)$. O la sustitución $\mu\eta$ sobre los términos originales.

Más formalmente, dadas dos sustituciones σ_1 y σ_2 se dice que σ_2 es más general que σ_1 si existe otra sustitución τ tal que $\sigma_1 = \sigma_2\tau$.

Unificación de literales Ya se ha precisado cómo unificar, si es posible, dos términos t y t' correspondientes. Volvamos al problema de unificar $P(t_0, t_1, \ldots, t_n)$ y $P(t'_0, t'_1, \ldots, t'_n)$. Comencemos por unificar un primer par de términos correspondientes (p.ej. t_0 y t'_0), encontrando su unificador más general. Entonces:

$$[P(t_0,t_1,\ldots,t_n)]\sigma_0 = P(t_0\sigma_0, t_1\sigma_0, \ldots, t_n\sigma_0), \quad [P(t'_0,t'_1,\ldots,t'_n)]\sigma_0 = P(t'_0\sigma_0, t'_1\sigma_0, \ldots, t'_n\sigma_0) \quad \text{con } t_0\sigma_0 = t'_0\sigma_0$$

Es decir, las sustituciones de σ_0 se aplican también a los restantes términos. Si se desea unificar el par de términos en segunda posición, éstos son ahora $t_1\sigma_0$ y $t'_1\sigma_0$. Si se encuentra un unificador σ_1 para ellos

$$[P(t_0,t_1,\ldots,t_n)]\sigma_0\sigma_1, \qquad [P(t'_0,t'_1,\ldots,t'_n)]\sigma_0\sigma_1$$

producen:

$$P(t_0\sigma_0\sigma_1,t_1\sigma_0\sigma_1,\ldots,t_n\sigma_0\sigma_1), \qquad P(t'_0\sigma_0\sigma_1,t'_1\sigma_0\sigma_1,\ldots,t'_n\sigma_0\sigma_1)$$

donde ya:

$$(t_0\sigma_0)\sigma_1 = (t'_0\sigma_0)\sigma_1 \quad \text{y} \quad (t_1\sigma_0)\sigma_1 = (t'_1\sigma_0)\sigma_1$$

El proceso se repite hasta obtener la composición $\sigma_0\sigma_1\ldots\sigma_n$ necesaria.

Ejemplo 2.88 Tratemos de unificar los dos siguientes literales:

$$\begin{array}{llll} P(& x, & f(w), & g(f(x),b) \quad) \\ P(& h(y), & y, & g(f(h(f(t))),t) \quad) \end{array}$$

Para el primer par de términos correspondientes se obtiene $\sigma_0 = \{x/h(y)\}$. Aplicando esta sustitución sobre ambos literales resulta:

$$\begin{array}{llll} P(& h(y), & f(w), & g(f(h(y)),b) \quad) \\ P(& h(y), & y, & g(f(h(f(t))),t) \quad) \end{array}$$

Para el segundo par de términos se obtiene $\sigma_1 = \{y/f(w)\}$. Su aplicación produce:

$$\begin{array}{llll} P(& h(f(w)), & f(w), & g(f(h(f(w))),b) \quad) \\ P(& h(f(w)), & f(w), & g(f(h(f(t))),t) \quad) \end{array}$$

Para unificar el tercer par de términos se necesita primero que las w se sustituyan por t (para el primer argumento) y luego que las t se sustituyan por b. El resultado de la composición de ambas (el resultado del algoritmo previo que unificaba este par) es $\sigma_2 = \{w/t\}\{t/b\} = \{w/b,t/b\}$. Aplicando σ_2:

$$\begin{array}{llll} P(& h(f(w)), & f(w), & g(f(h(f(b))),b) \quad) \\ P(& h(f(w)), & f(w), & g(f(h(f(b))),b) \quad) \end{array}$$

Es decir,

$$[P(x,f(w),g(f(x),b))]\sigma_0\sigma_1\sigma_2 = [P(h(y),y,g(f(h(f(t))),t))]\sigma_0\sigma_1\sigma_2$$

donde

$$\sigma = \sigma_0\sigma_1\sigma_2 = \{x/h(y)\}\{y/f(w)\}\{w/b,t/b\} = \{x/h(f(b)),y/f(b),w/b,t/b\}$$

2.5.5 Resolución

La resolución, como en el caso proposicional, se produce sobre dos cláusulas que contengan, respectivamente, un literal y su complementario. Por ejemplo, en:

$$P(x,f(y)) \vee Q(x) \qquad \text{y} \qquad \neg R(z) \vee \neg P(a,z)$$

se puede intentar la resolución sobre los predicado $P()$ y $\neg P()$; pero esto requiere unificar sus términos:

$$P(a,f(y)) \vee Q(a) \qquad \text{y} \qquad \neg R(f(y)) \vee \neg P(a,f(y))$$

El resultado de la resolución es entonces la cláusula:

$$Q(a) \vee \neg R(f(y))$$

Este principio, junto a los desarrollos previos, facilita la ejecución de los siguientes pasos:

1. Dado un conjunto de fórmulas $\Phi = \{\phi_1, \ldots, \phi_k\}$, se quiere comprobar si una fórmula ψ es consecuencia de ellas. Nieguela y verifique si el conjunto $\Phi \cup \{\neg\psi\}$ es insatisfacible.

2. Para ello, en este sistema, debe escribir cada fórmula de este conjunto en forma clausulada: $\phi_m \rightsquigarrow (C_1 \wedge \ldots \wedge C_r)$. Recuerde que, si se han introducido constantes o funciones de Skolem, ambas expresiones no son equivalentes. No obstante, ϕ_m será satisfacible si y sólo si lo son todas sus r cláusulas. Luego, el conjunto de fórmulas analizado será satisfacible si y sólo si lo son todas y cada una de las cláusulas que se han producido.

3. Busque dos cláusulas que contengan literales complementarios. Su unificación, si es posible, produce dos nuevas cláusulas tan satisfacibles como las previas. La resolución de éstas produce una nueva cláusula tan satisfacible como las anteriores (aún más, consecuencia de ellas).

4. Si se obtiene la cláusula vacía, el conjunto $\Phi \cup \{\neg\psi\}$ es insatisfacible.

Ejemplo 2.89 Comprobemos, por resolución, que:

$$\forall xPx \rightarrow (\exists yQy \wedge \forall zRz) \vdash \neg\forall zRz \rightarrow \neg\forall xPx$$

Este mismo ejemplo ya fue resuelto mediante tablas semánticas (tabl. 2.9). Ambos sistemas comprueban que:

$$\{\forall xPx \rightarrow (\exists yQy \wedge \forall zRz), \neg(\neg\forall zRz \rightarrow \neg\forall xPx)\}$$

es insatisfacible.

La forma clausulada de la primera fórmula es:

1.	$\forall xPx \rightarrow (\exists yQy \wedge \forall zRz)$	$\equiv$	$\neg\forall xPx \vee (\exists yQy \wedge \forall zRz)$	$\equiv$
2.			$\exists x\neg Px \vee (\exists yQy \wedge \forall zRz)$	$\equiv$
3.			$\exists x\exists y\forall z(\neg Px \vee (Qy \wedge Rz))$	$\rightsquigarrow$
4.			$\forall z(\neg Pa \vee (Qb \wedge Rz))$	$\equiv$
5.			$\forall z((\neg Pa \vee Qb) \wedge (\neg Pa \vee Rz))$	$\equiv$
6.	$(\neg Pa \vee Qb) \wedge (\neg Pa \vee Rz)$			

Luego la primera fórmula es tan satisfacible como este par de cláusulas. Observe que, en (3.), se podían haber sacado los cuantificadores en cualquier orden. Interesa desplazar primero los existenciales para simplificar la 'eskolemización' en (4.). Esta fórmula (4) no es equivalente a la anterior, sino igualmente satisfacible. Cada eliminación de un existencial requiere una constante distinta. En (5) se reescribre equivalentemente la matriz en forma normal conjuntiva. En (6) se omiten los cuantificadores universales, que quedan implícitos.

En una expresión como (5) siempre es posible volver a introducir los cuantificadores universales y renombrar variables de forma tal que 'cláusulas distintas no utilicen las mismas variables'.

La forma clausulada de la segunda fórmula es:

1.	$\neg(\neg\forall zRz \rightarrow \neg\forall xPx)$	$\equiv$	$\neg(\neg\neg\forall zRz \vee \neg\forall xPx)$	$\equiv$
2.			$\neg\forall zRz \wedge \forall xPx$	$\equiv$
3.			$\exists z\neg Rz \wedge \forall xPx$	$\rightsquigarrow$
4.	$\neg Rc \wedge Px$			

Observe que el resultado son dos cláusulas, no una. La fórmula inicial es tan satisfacible como estas dos cláusulas. Se ha escogido la constante 'c' porque no se había utilizado hasta el momento.

El conjunto de dos fórmulas iniciales es tan satisfacible como el conjunto de estas cuatro clausulas:

$$\{(\neg Pa \vee Qb), (\neg Pa \vee Rz), (\neg Rc), (Px)\}$$

Utilizando unificación y resolución:

1.	$(\neg Pa \vee Qb)$	
2.	$(\neg Pa \vee Rz)$	
3.	$(\neg Rc)$	
4.	(Px)	
5.	$(\neg Pa)$	$2,3\ z/c$
6.	$()$	$4,5\ x/a$

Como se obtiene la cláusula vacía, el conjunto de cláusulas es insatisfacible.

Bibliografía complementaria

Cualquiera de los textos citados en el capítulo de Lógica de Proposiciones aborda también su extensión a Lógica de Primer Orden.

Entre los textos más enfocados a su uso computacional se encuentrar [Huth y Ryan 2000], [Ben-Ari 90], [Burris 98] y muy especialmente [Fitting 96]. Los lectores más interesados por la fundamentación y uso en Matemáticas pueden consultar [Ebbinghaus et al. 96], [Dalen 97] ó especialmente [Mendelson 97].

Aunque no se citen repetidamente, existen varios compendios de interés sobre Lógica en Computación e Inteligencia Artificial, por ejemplo [Gabbay el al. 93-98] entre otros.

Actividades y evaluación

El alumno dispone de ejemplos y actividades en el grupo de tutorización telemática del curso, así como exámenes resueltos de años pasados.

Parte II

FORMALISMOS PARA PROGRAMACIÓN

Capítulo 3

PROGRAMACIÓN LÓGICA

Resumen

En este tema se presenta el paradigma de la programación lógica, basado en los procedimientos de deducción de fórmulas válidas en sistemas de axiomas. En particular, se introduce, un eficiente método de deducción aplicable en sistemas de cláusulas de Horn: la resolución SLD. Este tipo de resolución constituye la base de la semántica operacional del PROLOG, el lenguaje de programación lógica más difundido. Se analiza finalmente en qué medida este lenguaje verifica los presupuestos de la programación lógica.

Objetivos

El objetivo principal de este tema no es enseñar un lenguaje de programación lógica, sino presentar los fundamentos teóricos de este paradigma de computación y mostrar cómo los lenguajes como PROLOG se apartan de la programación lógica pura por motivos de eficiencia. El alumno debe aprender a interpretar un conjunto de cláusulas de Horn como un programa lógico, y a simular manualmente la ejecución de dicho programa aplicando el método de resolución SLD. Debe asimismo conocer el funcionamiento del motor de inferencia que implementan los intérpretes PROLOG y adquirir unos conocimientos básicos de la sintaxis de este lenguaje, de modo que sea capaz de diseñar programas sencillos y utilizar adecuadamente sus predicados más característicos (en particular, el predicado de corte). Finalmente, se busca que el alumno repare en las particularidades del paradigma de la programación lógica e identifique sus naturales campos de aplicación, en el contexto global de los paradigmas computacionales.

Metodología

En el capítulo anterior se ha mostrado cómo convertir una fórmula de la lógica de predicados en una fórmula equivalente que tiene la forma clausulada. Se ha explicado también cómo extender las técnicas de resolución al caso de la lógica de predicados, incorporando técnicas de sustitución y unificación, con el fin de obtener un procedimiento coherente y completo para dictaminar la satisfacibilidad de sus fórmulas. En este tema se muestra cómo estas técnicas sirven de base para la implementación de los denominados lenguages de programación lógica.

En el paradigma de la computación lógica, un problema se formaliza como una sentencia lógica (denominada sentencia objetivo) que ha de probarse bajo los supuestos de un sistema de axiomas,

que constituye el programa. La ejecución del programa consiste en la prueba de satisfacibilidad de la sentencia.

En este tema nos centraremos en el caso de particular interés en que los axiomas pueden expresarse como cláusulas de Horn. Veremos cómo en un sistema de cláusulas de Horn es posible demostrar la satisfacibilidad de una fórmula aplicando un eficiente método de deducción: la resolución SLD. Esta forma peculiar de resolución constituye el fundamento de la programación lógica y, concretamente, es la base de la semántica operacional del difundido lenguaje PROLOG.

En la sección 1 de este capítulo mostramos cómo una fórmula lógica puede interpretarse como un programa que utiliza como motor de inferencia el método de resolución. En la sección 2 definimos el concepto de cláusula de Horn, y otros conceptos relacionados de utilidad en el contexto de la programación lógica. En la sección 3 exponemos el método de resolución SLD, que constituye un procedimiento de deducción coherente y completo cuando se aplica en sistemas de cláusulas de Horn, e investigamos la eficiencia de sus posibles implementaciones. Dedicamos la última sección del capítulo al lenguaje PROLOG, mostrando cómo sus intérpretes implementan la resolución SLD, y llamando la atención sobre cómo en la práctica algunas de sus construcciones vulneran los presupuestos de la programación lógica pura por motivos de eficiencia.

3.1 Cómo interpretar una fórmula lógica como un programa

3.1.1 Declaración de programa versus algoritmo de solución

Según Kowalski, un algoritmo que resuelve un problema particular consta de dos componentes: la lógica o declarativa, y el control. El componente lógico especifica el conocimiento útil para la resolución del problema, y el componente de control determina la forma en que ese conocimiento puede utilizarse para resolverlo.

Ejemplo 3.1 El factorial de un número se define como:

$$Fact(n) = \begin{cases} 1 & \text{cuando } n = 0 \\ n * Fact(n-1) & \text{cuando } n > 0 \end{cases}$$

definición que constituye el componente lógico de cualquier algoritmo de cálculo de la función factorial.

Así como el componente lógico de un algoritmo es único, un programador puede usualmente elegir entre varios diseños de componentes de control alternativos en función de sus requisitos de eficiencia, plataforma de implementación, etc. En nuestro ejemplo, el cómputo del factorial puede realizarse mediante dos algoritmos diferentes:

$Entrada$	n
$Paso_1$	$Fact = 1$
$Paso_2$	$If\ n = 0\ then\ return\ Fact$
$Paso_3$	$Fact = Fact * n,\ n = n - 1,\ go\ to\ Paso_2$

y:

$Entrada$	n
$Paso_1$	$Fact = 1,\ n_1 = 0$
$Paso_2$	$If\ n_1 = n\ then\ return\ Fact$
$Paso_3$	$n_1 = n_1 + 1,\ Fact = Fact * n_1,\ go\ to\ Paso_2$

Podríamos decir que el primer algoritmo calcula el factorial "de arriba a abajo", es decir va de n a 0, aportando en cada iteración un factor $n-1$ al cómputo del factorial. El segundo algoritmo, por el contrario, calcula el factorial "de abajo a arriba", partiendo de la afirmación expresada en el componente lógico como $Fact(0) = 1$ y derivando de ésta nuevas afirmaciones sucesivamente ($Fact(1) = 1$, $Fact(2) = 2$,...) hasta encontrar la solución. Ambos algoritmos generan las mismas respuestas, pero la eficiencia y el consumo de espacio de almacenamiento que requieren difiere. Los dos algoritmos constituyen dos diferentes componentes de control para el mismo componente lógico.

En la programación algorítmica, el énfasis se pone en diseñar algoritmos que resuelvan problemas, de modo que al ejecutarlos se calcule la solución. En la programación lógica, por el contrario, el énfasis se sitúa en el problema que se pretende resolver. El programador especifica el componente lógico de los algoritmos, es decir, el conjunto de condiciones que las soluciones satisfacen. Los mecanismos genéricos de inferencia incorporados en los intérpretes de los lenguajes lógicos permiten deducir las soluciones a partir de las condiciones que satisfacen. Utilizando el lenguaje de la lógica, las condiciones que satisface el factorial de un número se expresan como:

$$Fact(0,1) \wedge \forall n \forall n_1 \forall m \forall m_1 [(n > 0) \wedge (n_1 = n-1) \wedge Fact(n_1, m_1) \wedge (m = n * m_1) \rightarrow Fact(n,m)]$$

donde la interpretación de $Fact(x,y)$ es "y es el número factorial del número x". Vemos que la fórmula anterior expresa una propiedad de la solución (si m es el factorial de n entonces se cumple que ...) sin proporcionar instrucciones explícitas para su cálculo. Sin embargo, a la vista de los algoritmos antes presentados, es evidente que el componente lógico está implícito en los procedimientos de solución. El componente lógico de un programa "insinúa" un procedimiento de cálculo. Veamos otro ejemplo.

Ejemplo 3.2 Consideremos la teoría de cadenas, donde se han definido la función de concatenación (que denotaremos con el símbolo $\cdot$), los predicados $subcadena, prefijo$ y $sufijo$, y los axiomas siguientes:

1. $\forall x\ subcadena(x,x)$
2. $\forall x \forall y \forall z\ ((subcadena(x,y) \wedge sufijo(y,z)) \rightarrow subcadena(x,z))$
3. $\forall x \forall y\ sufijo(x, y \cdot x)$
4. $\forall x \forall y \forall z\ ((subcadena(x,y) \wedge prefijo(y,z)) \rightarrow subcadena(x,z))$
5. $\forall x \forall y\ prefijo(x, x \cdot y)$

Esta teoría también puede expresarse utilizando el operador de implicación inversa $\leftarrow$, de modo que cada axioma A se escriba como: $A = \forall x_1 ... \forall x_k (B \leftarrow B_1 \wedge ... \wedge B_m)$.

1. $\forall x\ subcadena(x,x)$
2. $\forall x \forall y \forall z\ (subcadena(x,z) \leftarrow (subcadena(x,y) \wedge sufijo(y,z)))$
3. $\forall x \forall y\ sufijo(x, y \cdot x)$
4. $\forall x \forall y \forall z\ (subcadena(x,z) \leftarrow (subcadena(x,y) \wedge prefijo(y,z)))$
5. $\forall x \forall y\ prefijo(x, x \cdot y)$

Bajo esta forma, resulta intuitivo interpretar cada axioma A como un procedimiento que se lee como:

> "para computar B, es necesario computar previamente $B_1 ... B_m$"

o, dicho de otro modo,

> "si desea averiguar si se satisface la conclusión B, averigüe si se satisfacen las premisas $B_1 ... B_m$"

En términos informales, el anterior procedimiento constituye el componente de control o mecanismo de inferencia genérico de los programas lógicos. En el caso de los axiomas de nuestro último ejemplo, cabe interpretarlos como:

1. x es una subcadena de x
2. Para averiguar si x es una subcadena de z, encuentre un sufijo y de z y averigüe si x es una subcadena de y
3. x es un sufijo de $y \cdot x$
4. Para averiguar si x es una subcadena de z, encuentre un prefijo y de z y averigüe si x es una subcadena de y
5. x es un prefijo de $x \cdot y$

Ejercicio 3.3 Defina una cláusula que permita resolver el problema de ordenar los tres números distintos del conjunto $X = \{X_1, X_2, X_3\}$. La secuencia ordenada $\{X_i, X_j, X_k\}$ deberá satisfacer:

- X_i, X_j, X_k pertenecen a X;
- X_i, X_j, X_k son distintos entre sí;
- X_i es menor que X_j;
- X_j es menor que X_k.

Suponga definidas las relaciones: $pertenece(X_i, X)$, $distintos(X_i, X_j)$, $menor(X_i, X_j)$.

Algunos lenguajes de programación lógica permiten especificar ambas componentes específicas de un programa lógico: la pura lógica o declaración de axiomas que caracteriza las soluciones, y el control o procedimiento de deducción de conclusiones a partir de los axiomas. En este caso la eficiencia de los programas puede incrementarse cambiando el componente de control sin modificar el componente lógico. Sin embargo el ideal de la programación lógica es que el programador se despreocupe del procedimiento de solución, que estaría predeterminado en los entornos de programación en que los programas se ejecutan. Un programa consiste exclusivamente, en este caso ideal, en su parte lógica o declarativa, y no existe ningún modo en que el programador pueda optimizar la búsqueda de la solución. A continuación veremos como las técnicas de resolución pueden verse como un tal procedimiento genérico de solución.

3.1.2 La resolución como algoritmo para la solución de problemas

En capítulos anteriores hemos visto cómo las técnicas de resolución se aplican a sistemas de axiomas expresados en forma clausulada. Vimos también cómo cualquier fórmula lógica podía expresarse en esta forma. Habitualmente los axiomas se declaran a modo de reglas:

$$Si\ premisa1\ \ y\ \ premisa2\ \ y\ ...\ \ entonces\ conclusiones$$

Formalmente, se escribe:

$$A = \forall x_1 ... \forall x_k (B_1 \wedge ... \wedge B_m \rightarrow B)$$

donde B y B_i son los átomos que representan, respectivamente, la conclusión (o consecuente) y las correspondientes premisas (o antecedentes). En el caso degenerado en que no hay antecedentes, la fórmula se escribe: $A = \forall x_1 ... \forall x_k B$.

En un capítulo anterior vimos cómo traducir este tipo de fórmulas a una clase particular de cláusulas denominadas cláusulas de Horn, en el caso de la lógica proposicional. La extensión del concepto de cláusula de Horn al caso de la lógica de predicados es inmediata. En esta sección recordaremos cómo aplicar el método de resolución en sistemas de cláusulas de Horn para dictaminar si una fórmula G es una consecuencia lógica de un conjunto de axiomas. A continuación veremos cómo la aplicación de dicho método puede interpretarse como la ejecución de un programa lógico que averigua la satisfacibilidad de la fórmula G.

La traducción a forma normal conjuntiva de una regla, prescindiendo de los cuantificadores y eliminando el condicional, es inmediata:

$$A = \neg(B_1 \wedge ... \wedge B_m) \vee B = \neg B_1 \vee .... \vee \neg B_m \vee B$$

Podemos ahora aplicar el método de resolución para dictaminar si una fórmula escrita en forma normal conjuntiva $G = G_1 \wedge ... \wedge G_l$ es una consecuencia lógica del conjunto de axiomas: basta con añadir $\neg G$ al conjunto de axiomas, expresados en forma clausulada, y refutarla por resolución. $\neg G$ se denomina habitualmente "la cláusula objetivo".

Un conjunto de axiomas es presumiblemente satisfacible, de modo que en la refutación no tiene sentido buscar resolventes entre los axiomas; en lugar de ello buscaremos resolver la cláusula objetivo con un axioma. Como $\neg G$ se expresa como una disyunción de literales negativos, $\neg G = \neg(G_1 \wedge ... \wedge G_l) = \neg G_1 \vee ... \vee \neg G_l$, sólo existe la posibilidad de que uno de tales literales negativos, $\neg G_i$, se resuelva con el único literal positivo del axioma, B. El resolvente será por tanto una clásula que constará sólo de literales negativos:

$$\neg B_1 \vee ... \vee \neg B_m \vee \neg G_1 \vee \ldots \vee \neg G_{i-1} \vee \neg G_{i+1} \vee \ldots \vee \neg G_l$$

.

Por tanto, todos los resolventes que aparezcan a lo largo de la refutación contendrán sólo literales negativos. Eventualmente, la resolución tendrá lugar con axiomas que no tengan antecedentes, es decir, axiomas cuyas formas clausuladas consistan en un único literal positivo. Progresivamente se producirán resolventes más cortos y, finalmente, la cláusula vacía. De este modo quedará refutada la fórmula $\neg G$ y se habrá probado la satisfacibilidad de la fórmula G.

Ejemplo 3.4 Expresemos la teoría de cadenas presentada en el ejemplo 3.2 anterior en forma clausal:

1 $subcadena(x,x)$
2 $\neg\, subcadena(x,y) \vee \neg\, sufijo(y,z) \vee subcadena(x,z))$
3 $sufijo(x,y\cdot x)$
4 $\neg\, subcadena(x,y) \vee \neg\, prefijo(y,z) \vee subcadena(x,z))$
5 $prefijo(x,x\cdot y)$

Una forma de probar que la fórmula $subcadena(a\cdot b\cdot c, a\cdot a\cdot b\cdot c\cdot c)$ es una consecuencia lógica de la teoría es refutar mediante resolución la fórmula $\neg\, subcadena(a\cdot b\cdot c, a\cdot a\cdot b\cdot c\cdot c)$, según se describe en la siguiente secuencia:

6 $\neg\, substr(a\cdot b\cdot c, a\cdot a\cdot b\cdot c\cdot c)$
7 $\neg\, substr(a\cdot b\cdot c, y_1) \vee \neg\, suffix(y_1, a\cdot a\cdot b\cdot c\cdot c)$ (de 6 y 2 con $x = a\cdot b\cdot c, z = a\cdot a\cdot b\cdot c\cdot c$)
8 $\neg\, substr(a\cdot b\cdot c, a\cdot b\cdot c\cdot c)$ (resolvente de 7 y 3)
9 $\neg\, substr(a\cdot b\cdot c, y_2) \vee \neg\, prefix(y_2, a\cdot b\cdot c\cdot c)$ (resolvente de 8 y 4)
10 $\neg\, substr(a\cdot b\cdot c, a\cdot b\cdot c)$ (resolvente de 9 y 5)
11 $\square$ (resolvente de 10 y 1)

Como vimos en la sección anterior, resulta más intuitivo ver este sistema axiomático como un programa si se utiliza el operador de implicación inversa $\leftarrow$. Abreviadamente (asumiendo que todas las variables están cuantificadas de forma universal y que las comas en el antecedente denotan conjunción.), la teoría se transforma entonces en:

1. $subcadena(x,x)$
2. $subcadena(x,z) \leftarrow subcadena(x,y),\ sufijo(y,z))$
3. $sufijo(x,y\cdot x)$
4. $subcadena(x,z) \leftarrow (subcadena(x,y),\ prefijo(y,z))$
5. $prefijo(x,x\cdot y)$

Bajo esta perspectiva, la refutación de $\neg\, subcadena(a\cdot b\cdot c, a\cdot a\cdot b\cdot c\cdot c)$, prueba de que la cláusula básica $subcadena(a\cdot b\cdot c, a\cdot a\cdot b\cdot c\cdot c)$ es una consecuencia lógica de los axiomas de la teoría de cadenas, puede verse como la aplicación de un procedimiento o programa para averiguar si la cadena $a\cdot b\cdot c$ es una subcadena de la cadena $a\cdot a\cdot b\cdot c\cdot c$. Los distintos pasos del procedimiento de resolución pueden verse como subprocedimientos de un procedimiento principal.

La utilidad de este programa se ve más claramente si la fórmula bajo prueba es:

$$\exists\omega\ subcadena(\omega, a\cdot a\cdot b\cdot c\cdot c)$$

El procedimiento de prueba consiste en este caso en refutar en el sistema de axiomas la fórmula

$$\neg(\exists\omega\ subcadena(\omega, a\cdot a\cdot b\cdot c\cdot c))$$

equivalente a:

$$\forall\omega \neg\, subcadena(\omega, a\cdot a\cdot b\cdot c\cdot c)$$

Es fácil ver que el proceso de resolución produce como resultado la clásula vacía bajo la sustitución:

$$\{\omega \leftarrow a\cdot b\cdot c\}$$

El procedimiento, en este caso, no sólo sirve para probar que la fórmula $\exists\omega\ subcadena(\omega, a\cdot a\cdot b\cdot c\cdot c)$ es una consecuencia lógica de los axiomas, sino que además "computa" un valor de $\omega = a\cdot b\cdot c$ que hace cierta la fórmula $subcadena(\omega, a\cdot a\cdot b\cdot c\cdot c)$.

Ejercicio 3.5 Muestre cómo puede utilizarse el sistema de cláusulas siguiente :

1. $Fact(0,1)$

2. $\forall n \forall n_1 \forall m \forall m_1 ((n > 0) \wedge (n_1 = n - 1) \wedge Fact(n_1, m_1) \wedge (m = n * m_1) \rightarrow Fact(n, m))$
para calcular el factorial del número 3.

Idem en el caso del sistema de cláusulas solución del ejercicio 3.3 y el problema de ordenar los números 9, 2 y 5.

La refutación de fórmulas en un sistema de axiomas sirve de mecanismo de inferencia genérico de los programas lógicos. Ha de observarse, no obstante, que la conversión de un procedimiento de refutación en un programa no es del todo inmediata. El programa esbozado anteriormente es altamente no determinista, ya que el procedimiento de refutación generalmente lo es: en cada paso puede darse la opción de qué literal resolver y de qué axioma, ya que la resolución puede ser posible con más de un axioma y utilizando distintos literales. Esto significa que no se ha especificado por completo el control de ejecución, condición indispensable para que un algoritmo pueda ejecutarse de forma automática. Traducir un procedimiento de refutación a un programa exige especificar reglas que fijen las opciones anteriores. Tales reglas concretan la semántica operacional del lenguaje de programación lógica utilizado. Entre las mencionadas reglas cabe distinguir entre reglas de computación y reglas de búsqueda, que a continuación definimos.

Definición 3.6 Regla de computación: Regla que indica el literal y la cláusula objetivo con que se efectuará una resolución.

Definición 3.7 Regla de búsqueda: Regla que indica la cláusula con que se efectuará una resolución con un literal y cláusula objetivo previamente escogidos.

3.1.3 Programación lógica versus programación algorítmica

La programación lógica nace a principios de los años 70 fruto de los primeros trabajos sobre demostradores automáticos de teoremas, que culminan con la regla de resolución publicada por Robinson en 1965. En 1972, Kowalski publica las primeras ideas acerca de cómo la lógica de primer orden podría ser usada como un lenguaje de programación. Poco después Colmerauer lleva a la práctica estas ideas con la implementación del lenguaje PROLOG (PROgramming in LOGic), el primer y más difundido lenguaje de programación lógica, cuyo intérprete primitivo se desarrolló en el lenguaje ALGOL-W (Roussel, Universidad de Marsella).

El paradigma de la programación lógica se encuadra dentro del paradigma más general de la programación declarativa, cuyo enfoque es esencialmente distinto del de la convencional programación algorítmica. Frente a los lenguajes algorítmicos (tales como Modula-2, C++, Java, etc), también llamados procedimentales o imperativos, los lenguajes declarativos no sirven para especificar cómo resolver un problema, sino qué problema se desea resolver. En la programación algorítmica un programa consiste en una secuencia de instrucciones que una computadora ha de ejecutar para resolver un problema. El diseño del control de ejecución de tales instrucciones forma también parte del programa. Por el contrario, idealmente en la programación declarativa la labor del programador no consiste en codificar instrucciones explícitas, sino en especificar conocimiento y suposiciones acerca del problema. Los intérpretes de los lenguajes declarativos tienen incorporado un motor de inferencia genérico que resuelve los problemas a partir de su especificación.

Entre los lenguajes declarativos se distinguen los lenguajes funcionales y los lenguajes lógicos. Mientras que en la programación funcional el mecanismo de inferencia genérico se basa en la reducción de una expresión funcional a otra equivalente simplificada, en el paradigma de la computación lógica, un problema se formaliza como una sentencia lógica que ha de probarse bajo los supuestos de un sistema axiomático, que constituye el programa. La ejecución del programa consiste en la prueba de satisfacibilidad de la sentencia.

El paradigma lógico, independiente del modelo de máquina En los párrafos anteriores hemos presentado el paradigma de programación lógica en su forma más pura. Los programas lógicos no incluyen instrucciones explícitas de operación, considerándose que el conocimiento y las suposiciones acerca del problema, proporcionados en forma de axiomas lógicos, son suficientes para resolverlo. El conjunto de axiomas describe así la relación entre la entrada y salida del programa, y constituye la alternativa al programa convencional. El programador escribe el conjunto de axiomas, y el compilador o intérprete utilizado incorpora un motor de inferencia de resolución que, junto con las correspondientes reglas de búsqueda y computación, constituye una estructura de control única, sea cual sea el programa ejecutado. El problema que se desea resolver se formaliza como una sentencia lógica (la sentencia objetivo) que ha de probarse en el sistema axiomático que constituye el programa lógico. Su computación es un intento de resolver el problema, es decir, de probar la sentencia objetivo dados los supuestos del programa lógico o, dicho de otro modo, de ”deducir” las consecuencias del programa.

Mencionamos anteriormente que la diferencia esencial entre la programación lógica y la programación algorítmica radica en que en la programación algorítmica el programador diseña el conocimiento del problema en forma de instrucciones explícitas, y diseña también el control de la computación como parte esencial del programa. Existe, sin embargo otra diferencia clave entre ambos paradigmas de programación: los lenguajes algorítmicos sólo tienen significado con referencia al procedimiento que realizan en una máquina Von Neumann. La mayoría de los ordenadores modernos se basan en este modelo de máquina, caracterizado por un conjunto homogéneo de celdas de memoria y una unidad de procesamiento con algunas celdas locales denominadas registros. La unidad de procesamiento puede cargar datos desde la memoria a los registros, realiza operaciones lógicas y aritméticas en éstos y almacena valores en la memoria. Los lenguajes de programación se han diseñado para ”comunicar” al ordenador los procedimientos de resolución de los problemas desde la perspectiva de la ingeniería del ordenador. De este modo, un programa diseñado para una máquina Von Neumann consiste en una secuencia de instrucciones encaminadas a realizar las operaciones anteriores y un conjunto adicional de instrucciones de control. Para alguien no familiarizado con las restricciones de la ingeniería que conducen al diseño von Neuman, pensar en términos de un conjunto restringido de operaciones no resulta nada fácil. De ahí que surja la división de tareas entre el diseño de métodos de solución y la codificación o traducción de las instrucciones de los diseñadores a las instrucciones que puede ”entender” el ordenador.

Partiendo del denominado ”lenguaje máquina”, directamente comprendido por el ordenador, se han ido desarrollando formalismos y notaciones más convenientes para la expresión humana, aunque en un principio también en correspondencia bastante directa con el lenguaje subyacente de la máquina, al que los programas deben finalmente traducirse para ser ejecutados. Los lenguajes de programación lógica, por el contrario, se derivan del lenguaje abstracto de la lógica matemática, y no guardan dependencia ni relación directa con ningún modelo de máquina. La lógica proporciona un lenguaje preciso para expresar explícitamente los objetivos, conocimiento y presunciones característicos de un problema, en términos más cercanos a los que un ser humano utiliza en sus razonamientos. De este modo, no obliga al programador a pensar en términos de las operaciones de un ordenador.

El compromiso entre programación lógica y procedimental La indiscutible ventaja de dejar el control de la ejecución al completo cuidado de un intérprete tiene, no obstante una importante contrapartida. Obviamente, por muy eficiente que sea un compilador de programas lógicos, su predeterminada estructura de control nunca podrá equipararse a una estructura específicamente concebida para una cierta computación. Los investigadores en programación lógica han explorado los compromisos entre la pura lógica declarativa y las construcciones procedimentales que permiten a los programadores escribir programas de eficiencia razonable frente a la de los lenguajes procedimentales.

Dejar que un intérprete de PROLOG asuma exclusivamente el control de ejecución de un programa puede conducir a soluciones ineficientes, por ejemplo, cuando se entra en ramas de recursión infinita (hay que tener presente que el cálculo de predicados es indecidible). Como veremos en las siguientes secciones, los sistemas PROLOG proporcionan algunos mecanismos básicos de especificación de control, como la posibilidad de ordenar cláusulas y objetivos y el predicado de corte. Desafortunadamente, en sistemas actuales de PROLOG algunos programas pueden entenderse sólo pensando en términos procedimentales, al abusarse de estas características no lógicas del lenguaje. En muchas ocasiones, la semántica declarativa de un programa PROLOG (que es la semántica de un sistema de cláusulas de Horn en lógica de predicados) no coincide con su semántica operacional (es decir, la ejecución del programa no proporciona exactamente el conjunto de soluciones que define el sistema de cláusulas). En la sección 3.4.2 de este capítulo discutiremos estos aspectos del lenguaje PROLOG. En la siguiente sección exploramos primero más en profundidad la base teórica de la programación lógica.

3.2 Formalismo lógico para la representación de problemas

En esta sección recordamos el concepto de cláusula de Horn, así como otra serie de conceptos útiles en el contexto de la programación lógica, y mostramos cómo la aplicación de un procedimiento de resolución se expresa en términos de los nuevos conceptos definidos. La utilidad de las cláusulas de Horn radica en la existencia de un eficiente método de resolución que constituye un método coherente y completo de deducción cuando se aplica a cláusulas de este tipo. Este método de resolución, denominado SLD, se analiza en la sección 3.3.

Definición 3.8 Cláusula Horn: Cláusula $A \leftarrow B_1, ..., B_n$, en forma clausulada $\neg(B_1 \wedge ... \wedge B_m) \vee A = \neg B_1 \vee ... \vee \neg B_m \vee A$, con a lo sumo un literal positivo. El literal positivo, A, se denomina cabeza y los literales negativos B_i, se denominan cuerpo. Su semántica informal es: "para cada asignación de cada variable, si $B_1, ..., B_n$ son fórmulas ciertas, A es una fórmula cierta". Las cláusulas de Horn se clasifican en cláusulas de programa y cláusulas obtetivo.

Definición 3.9 Cláusula objetivo u objetivo: cláusula Horn que carece de literales positivos $\leftarrow B_1, ..., B_n$ (en forma clasular $\neg B_1 \vee ... \vee \neg B_m$). Cada B_i es un subobjetivo del objetivo principal.

Definición 3.10 Cláusula de programa: cláusula Horn que consta de un literal positivo y uno o más literales negativos.

Definición 3.11 Cláusula unitaria o hecho: cláusula Horn de programa que consta de un único literal positivo $A \leftarrow$ (en forma clasular, A). Su semántica informal es: "para cada asignación de cada variable, A es una fórmula cierta."

Definición 3.12 Procedimiento o definición de un predicado: conjunto de cláusulas de Horn de programa cuyas cabezas comparten la letra de predicado.

Definición 3.13 Programa lógico: Conjunto de procedimientos

Definición 3.14 Base de datos (en un programa lógico): Procedimiento consistente en un conjunto de hechos básicos (es decir, sin variables libres).

Definición 3.15 Respuesta correcta: Sea P un programa y G una cláusula objetivo. Una substitución θ para las variables de G es una respuesta correcta si $P \models \forall(\neg\, G\theta)$, donde $\forall$ denota el cierre universal de las variables libres de $\neg\, G\theta$. Si G carece de variables, la única respuesta correcta es la sustitución identidad. El conjunto de respuestas correctas de un programa P define su semántica declarativa $S_D(P)$.

A continuación ilustramos con un ejemplo cómo la aplicación de un método de resolución en un sistema de cláusulas de Horn se interpreta como la ejecución de un programa, y se describe en términos de los conceptos definidos.

Ejemplo 3.16 El conjunto de cláusuas de Horn:

1. $q(x,y) \leftarrow p(x,y)$ (en forma clausulada $\neg p(x,y) \vee q(x,y)$)
2. $q(x,y) \leftarrow p(x,z), q(z,y)$ (en forma clausulada $\neg p(x,z) \vee \neg q(z,y) \vee q(x,y)$)
3. $p(b,a)$
4. $p(c,a)$
5. $p(d,b)$
6. $p(e,b)$
7. $p(f,b)$
8. $p(h,g)$
9. $p(i,h)$
10. $p(j,h)$

Es un programa que tiene dos procedimientos, uno de los cuales es una base de hechos. Supongamos que se desea utilizar este programa para demostrar la validez de la fórmula $\exists y \exists z(q(y,b) \wedge q(b,z))$ y, más aún, para obtener una sustitución de las variables y, z que hace cierta la fórmula $q(y,b) \wedge q(b,z)$.

La validez de la fórmula queda demostrada refutando su negación $\neg\exists y \exists z(q(y,b) \wedge\ q(b,z)) = \forall(\neg q(y,b) \vee\ \neg q(b,z))$ en el sistema de cláusulas. Una refutación posible consiste en la siguiente secuencia:

1. Se escoge resolver $q(y,b)$ con la cláusula 1 obtiéndose $\leftarrow p(y,b), q(b,z)$ (en forma clausulada $\neg p(y,b) \vee \neg q(b,z)$)
2. Se escoge resolver $p(y,b)$ con la cláusula 5 aplicando la sustitución $\{y \leftarrow d\}$ obtiéndose $\leftarrow q(b,z)$ (en forma clausulada $\neg q(b,z)$)
3. Se toma el único literal posible para resolver con la cláusula 1, obteniéndose $\leftarrow p(b,z)$ (en forma clausulada $\neg p(b,z)$)
4. Se toma el único literal posible para resolver con la cláusula 3 aplicando la sustitución $\{z \leftarrow a\}$, obteniéndose la cláusula vacía $\square$.

De la secuencia anterior deducimos que la sustitución $\theta = \{y \leftarrow d,\ z\ \leftarrow a\}$ hace falsa la fórmula $\forall(\neg q(y,b) \vee\ \neg q(b,z))$, con lo que queda demostrada la validez de la fórmula $\exists y \exists z(q(y,b) \wedge\ q(b,z))$.

Veamos ahora que el problema resuelto equivale al problema de encontrar una respuesta correcta para la fórmula $\neg q(y,b) \vee\ \neg q(b,z)$, es decir, que la sustitución encontrada, $\theta = \{y \leftarrow d,\ z\ \leftarrow a\}$, es una respuesta correcta para las variables de la cláusula objetivo $\leftarrow p(y,b), q(b,z)$. Efectivamente, según la definición antes presentada, una substitución θ para las variables de $G = \leftarrow p(y,b), q(b,z)$ es una respuesta correcta si $P \models \forall(\neg\, G\theta)$, es decir, si $P \models \forall\ (\ \neg\ (\neg q(y,b) \vee\ \neg q(b,z)))\theta\ )$, es decir, $P \models \forall\ (\ (q(y,b) \wedge\ q(b,z))\theta\ )$. Esta fórmula es precisamente la fórmula demostrada para la sustitución $\theta = \{y \leftarrow d,\ z\ \leftarrow a\}$.

Como esta sustitución es básica, en este caso el cierre universal es prescindible, de modo que $P \models\ (q(y,b) \wedge\ q(b,z))\theta$. Pero la sustitución que conlleva un proceso de refutación por resolución no siempre es básica. A veces, la cláusula vacía puede obtenerse sin necesidad de sustituir todas las variables de la fórmula demostrada. Baste como ejemplo sencillo el cálculo de una sustitución de respuesta correcta para la fórmula $G = \neg(x = y + 13)$ en el conjunto de axiomas de la aritmética. La

sustitución $\theta = \{y \leftarrow x - 13\}$ proporciona la fórmula $\forall(\neg\, G\theta) = \forall x\ (\neg\ (\neg(x = x))) = \forall x\ (x = x)$, que es válida en el sistema de axiomas considerado.

En resumen, los sistema de programación lógica ideales usan únicamente la regla de resolución como regla de inferencia. La prueba de que una fórmula $B = \exists(B_1 \wedge ... \wedge B_n)$ es una consecuencia lógica de un programa P (es decir, si $P \vDash B$) se lleva a cabo negando la fórmula y añadiéndola al conjunto de axiomas que constituyen el programa, para averiguar si el sistema de fórmulas extendido es insatisfacible (contradictorio). Si éste es el caso se irá derivando una serie de objetivos sucesivos hasta llegarse a la cláusula vacía. Esta prueba es pues equivalente a la prueba de que existe una respuesta correcta θ para la cláusula objetivo $G = \neg\ (B_1 \wedge ... \wedge B_n)$. Baste ver que: $\forall G = \forall\ \neg(B_1 \wedge ... \wedge B_n) = \neg\exists(B_1 \wedge ... \wedge B_n) = \neg B$. Encontrar una respuesta correcta para una cláusula objetivo G es encontrar una sustitución de variables que conduce a la refutación de G en el sistema de axiomas. Si esta sustitución se encuentra, se habrá probado la validez de la fórmula $\neg G$.

3.3 Resolución SLD

En esta sección exponemos el método de resolución SLD, debido a Kowalski, que constituye un procedimiento de deducción coherente y completo cuando se aplica en sistemas de cláusulas de Horn. La denominanción SLD significa "Selección Lineal de programas Definidos", haciendo así referencia a la esencia del método, que radica en la selección sucesiva de cláusulas y literales para resolver de acuerdo a criterios especificados por reglas de computación y búsqueda. Al término de la sección se investiga cómo la eficiencia del método se ve condicionada por las reglas elegidas.

Definición 3.17 Derivación por resolución SLD: Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Una derivación por resolución SLD de $P \cup \{G\}$ se define como una secuencia de etapas de resolución entre clásulas objetivo y cláusulas de programa. La primera cláusula objetivo es $G_0 = G$. Asumiendo que se ha derivado G_i, G_{i+1} se define seleccionando un literal $A_i \in G_i$ de acuerdo a la regla de computación R, escogiendo una cláusula $C_i \in P$ tal que la cabeza de C_i pueda unificarse con A_i mediante una sustitución MGU (Unificador de Máxima generalidad) θ_i y resolviendo:

$G_i = \leftarrow A_1, ..., A_{i-1}, A_i, A_{i+1}, ..., A_n$
$C_i = A \leftarrow B_1, ..., B_k$
$A_i\theta_i = A\theta_i$
$G_{i+1} = \leftarrow (A_1, ..., A_{i-1}, B_1, ..., B_k, A_{i+1}, ..., A_n)\theta_i$

Una refutación SLD de $P \cup \{G\}$ es una derivación SLD finita de $P \cup \{G\}$ que tiene la cláusula vacía, $\square$, como último objetivo en la derivación. Si G_n es la cláusula vacía, decimos que la longitud de refutación es n.

Definición 3.18 Respuesta computada: Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Una respuesta computada θ para $P \cup \{G\}$ es la sustitución obtenida por la composición $\theta_1, ..., \theta_n$ de todas las variables de G, donde $\theta_1, ..., \theta_n$ es la secuencia de MGUs usados en una refutación SLD de $P \cup \{G\}$. Se consideran incluidas sólo las θ que sean sustituciones de las variables del programa. El conjunto de respuestas computadas de un programa define su semántica operacional $S_O(P)$.

Ejemplo 3.19 Sea el programa $P = \{p(a), q(a,b)\}$ y la cláusula objetivo: $\leftarrow p(x),\ q(x,y)$

1. Resolviendo $p(x)$ con $p(a)$, aplicando la sustitución $\theta_1 = \{x/a\}$, se obtiene como nuevo objetivo: $\leftarrow\ q(a,y)$

2. Resolviendo $q(a,y)$ con $q(a,b)$, aplicando la sustitución $\theta_2 = \{y/b\}$, se obtiene $\square$

La respuesta computada en la refutación de la cláusula objetivo en el programa P es $\{x/a, y/b\}$.

Ejercicio 3.20 Derive $\square$ por resolución SLD aplicada a la cláusula objetivo $\leftarrow q(y,b),\ q(b,z)$ en el programa del ejemplo 3.16, y obtenga la respuesta computada. Observe que la derivación equivale a la refutación de la cláusula $(\neg q(y,b) \vee\ \neg q(b,z))$.

Puede demostrarse que el método de resolución SLD constituye un procedimiento de deducción coherente y completo cuando se aplica en sistemas de cláusulas de Horn. Este resultado se enuncia en los siguientes teoremas (cuya demostración no incluímos):

Teorema 3.21 Consistencia de la resolución SLD
Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Se supone la existencia de una refutación SLD de G. Sea $\theta = \theta_1 ... \theta_n$ la secuencia de unificadores usados en la refutación y sea σ la restricción de θ a las variables de G, es decir, la respuesta computada en la refutación. Entonces, σ es una respuesta correcta para G. Como las respuestas computadas definen la semántica operacional del programa, $S_O(P)$, y las respuestas correctas la semántica declarativa, $S_D(P)$, se concluye que $S_O(P) \subseteq S_D(P)$.

Teorema 3.22 Completitud de la resolución SLD aplicada a cláusulas de Horn
Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Sea σ una respuesta correcta para G. Entonces existe una refutación SLD de G a partir de P tal que σ es la restricción de la secuencia de unificadores $\theta = \theta_1 ... \theta_n$ a las variables de G, es decir, la respuesta computada en la refutación. Del mismo modo, ya que para cada respuesta correcta existe una respuesta computada, podemos afirmar que $S_D(P) \subseteq S_O(P)$.

De los dos teoremas anteriores se deduce que $S_D(P) = S_O(P)$, es decir, la semántica declarativa de un programa lógico descrito en términos de cláusulas de Horn coincide con su semántica operacional cuando el control de ejecución consiste en la derivación SLD.

Ejemplo 3.23 Consideremos el programa

$$P = \{p(a),\ q(x,y) \leftarrow p(x),\ q(a,b)\}$$

donde el dominio de las variables x e y es $\{a,b\}$. La semántica declarativa [1] de este programa está definida por el conjunto de términos básicos más sencillos que son consecuencia lógica del conjunto de cláusulas del programa (el modelo de Herbrand), es decir: $S_D(P) = \{p(a),\ q(a,b)\}$. Para obtener la semántica operacional debemos ver cuales son las respuestas más sencillas (es decir, pertenecientes a la base de Herbrand) que pueden computarse mediante la aplicación del procedimiento de resolución SLD, obteniéndose igualmente $S_O(P) = \{p(a),\ q(a,b)\}$.

En la sección anterior señalábamos la necesidad de definir reglas de computación y reglas de búsqueda que permitiesen sistematizar los procedimientos de resolución convirtiéndolos en programas eficientes. Para visualizar la aplicación de tales reglas es útil representar las posibles derivaciones SLD de una cláusula objetivo, bajo una cierta regla de computación R, en forma de una estructura tipo árbol, denominada árbol SLD. Las reglas de búsqueda pueden verse como algoritmos de búsqueda en el árbol SLD. A continuación definimos una serie de conceptos útiles en este contexto.

[1] Nos referimos aquí a los conceptos rigurosos de semántica declarativa y semántica operacional, definidos en términos de modelos de Herbrand . En el resto del capítulo se usan según sus acepciones informales (con frecuencia utilizadas en la literatura), haciendo referencia, respectivamente, al conjunto de todas las respuestas correctas y computadas; habida cuenta de que los modelos de Herbrand no son materia de estudio obligatorio en esta asignatura.

Definición 3.24 Arbol SLD. Sea P un conjunto de cláusulas de programa, R una regla de computación y G una cláusula objetivo. Todas las posibles derivaciones SLD pueden representarse en una estructura de tipo árbol denominada árbol SLD. La raíz se etiqueta con la cláusula objetivo G. Dado un nodo n etiquetado por la cláusula objetivo G_n, se crea un nodo n_i para cada nueva cláusula objetivo G_{n_i} que puede obtenerse resolviendo el literal escogido por R con la cabeza de una cláusula en P.

Definición 3.25 Rama de éxito de un árbol SLD: Rama que conduce a una refutación

Definición 3.26 Rama de fallo de un árbol SLD: Rama que conduce a una cláusula objetivo cuyo literal seleccionado no pude unificarse con ninguna cláusula del programa.

Definición 3.27 Rama infinita de un árbol SLD: Rama correspondiente a una derivación no terminante.

Definición 3.28 Regla de búsqueda en un árbol SLD: Procedimiento de búsqueda de una refutación en un árbol SLD.

Definición 3.29 Procedimiento de refutación SLD: Algoritmo de refutación SLD junto con la especificación de una regla de computación y una regla de búsqueda.

Ejemplo 3.30 Sea el programa: $P = \{p(b), p(a), q(a,b)\}$
Los siguientes árboles SLD consisten en una única rama:
$\leftarrow p(b)$ rama de éxito
$\leftarrow p(a)$ rama de éxito
$\leftarrow q(a,a)$ rama de fallo
$\leftarrow q(a,b)$ rama de éxito
$\leftarrow q(b,a)$ rama de fallo
$\leftarrow q(b,b)$ rama de fallo

Ejemplo 3.31 En la figura 3.1 se muestra el árbol de derivación correspondiente a la cláusula objetivo $q(y,b)$, en el conjunto de cláusulas de programa del ejemplo 3.16, y donde la regla de computación consiste en escoger siempre el literal situado más a la izquierda de la cláusula objetivo. Los números que etiquetan las aristas señalan la cláusula del programa con que se aplica la resolución. Las ramas etiquetadas con 5, 6 y 7 conducen a la cláusula vacía.

Ejercicio 3.32 Construya los árboles de derivación de la cláusula objetivo $\leftarrow p(y,b), q(b,z)$ cuando la regla de computación consiste en seleccionar el último literal de la cláusula objetivo, y cuando consiste en seleccionar el primer literal de la cláusula objetivo.

Teorema 3.33 Sea P un programa y G una cláusula objetivo. Entonces, cada árbol SLD de P y G bien tiene infinitas ramas de éxito o bien cada una de sus ramas tiene el mismo número finito de ramas de éxito.

Del teorema de completitud se deduce que la resolución SLD es un procedimiento completo independientemente de la elección de la regla de computación, pero sólo indica que existe una refutación. La elección de la regla de búsqueda determinará el que la refutación se encuentre o no. Surge de este modo un compromiso entre completitud y eficiencia. Una búsqueda "en anchura" en un árbol SLD, donde se comprueban todos los nodos de cada nivel antes de adentrarse más en el árbol, garantiza que se encontrará la rama de éxito siempre que exista. Por el contrario, si se aplica una estrategia

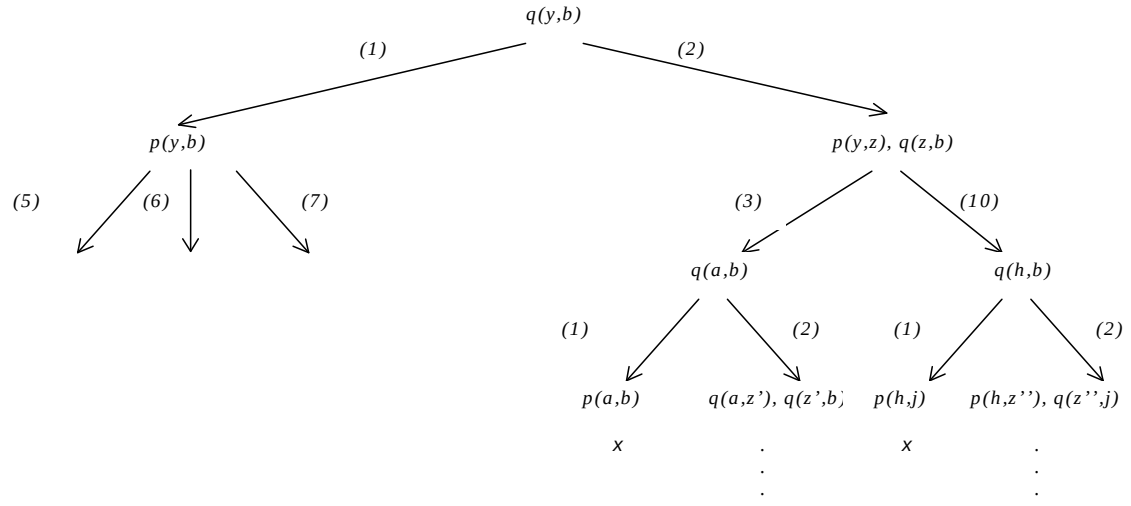


Figura 3.1: Árbol SLD del objetivo $q(y,b)$

de búsqueda en profundidad, siempre existe la posibilidad de explorar una rama no terminante. La regla de búsqueda en anchura es completa en el sentido de que siempre encuentra la respuesta correcta cuando ésta existe. Sin embargo, el tamaño del árbol que requiere almacenar en una estructura de datos crece exponencialmente con la profundidad de la búsqueda, de modo que la aplicación pura de esta estrategia no resulta una regla práctica.

3.4 El lenguaje PROLOG

El PROLOG surgió de los trabajos de Colmerauer sobre el procesamiento del lenguaje natural. Su primera implementación eficiente se debe a Warren y otros investigadores de la universidad de Edimburgo. Además de tratarse del primer y más difundido lenguaje de programación lógica, su estudio es muy interesante debido a que la traducción de cláusulas de Horn a procedimientos codificados en PROLOG es inmediata, e ilustra particularmente bien el paradigma de programación lógica expuesto en las secciones anteriores. A continuación presentamos los principios básicos de la programación en PROLOG y sus mecanismos operacionales más interesantes. Reflexionamos también sobre sus peculiaridades como lenguaje de programación, y sobre sus características no-lógicas, particularmente sobre las derivadas del uso de la controvertida primitiva de corte.

3.4.1 Definición de predicados recursivos. Reglas de computación y búsqueda

Comenzaremos por ver los sencillos cambios notacionales que introduce PROLOG con respecto a la notación matemática que hemos utilizado en los apartados anteriores. En un programa PROLOG, las variables empiezan con letras mayúsculas, los predicados y las constantes con letras minúsculas y en lugar del símbolo $\leftarrow$ se utiliza $:-$. Las cláusulas, denominadas también en PROLOG "reglas", se terminan con un punto.

Ejemplo 3.34 El ejemplo de la sección 3.16 se escribe pues en PROLOG de la forma siguiente:

1. $q(X,Y) :- p(X,Y).$
2. $q(X,Y) :- p(X,Z), q(Z,Y).$

3. $p(b,a)$. 7. $p(f,b)$.
4. $p(c,a)$. 8. $p(h,g)$.
5. $p(d,b)$. 9. $p(i,h)$.
6. $p(e,b)$. 10. $p(j,h)$.

De hecho, este programa ilustra una típica definición recursiva de predicado PROLOG. Los predicados más complejos ($q(X,Y)$) se definen en términos de predicados más sencillos ($p(X,Y)$), y las cláusulas más generales son recursivas, es decir, el predicado definido (cabeza de la cláusula) aparece en la definición (cuerpo de la cláusula). Así como en los lenguajes algorítmicos se utiliza la iteración (mediante estructuras de bucle tipo "for", "while", y "repeat/until"), en los lenguajes lógicos la recursión es el recurso principal que el programador utiliza para diseñar los predicados.

Ejemplo 3.35 Efectivamente, es fácil imaginar diferentes programas que respondan al patrón anterior. Considérese, por ejemplo, el problema de estudiar la conectividad en un grafo dirigido. Un grafo dirigido puede representarse en un programa lógico por medio de una colección de hechos. Un hecho $arco(Nodo1,Nodo2)$ estará presente en el programa si existe un arco desde el $Nodo1$ al $Nodo2$ en el grafo. Dos nodos estarán conectados si existe una serie de arcos que pueden recorrerse para llegar desde el primer nodo hasta el segundo. Es decir, la relación que representa el predicado $conectado(Nodo1,Nodo2)$, que es cierta si los nodos $Nodo1$ y $Nodo2$ están conectados, es el cierre transitivo de la relación *arco*. El predicado *conectado* se define mediante el siguiente programa:

1. $conectado(Nodo1,Nodo2) :- arco(Nodo1,Nodo2)$.
2. $conectado(Nodo1,Nodo2) :- arco(Nodo1,Enlace), conectado(Enlace,Nodo2)$.

El siguiente conjunto de hechos representaría un grafo inconexo:

3. $arco(b,a)$. 7. $arco(f,b)$.
4. $arco(c,a)$. 8. $arco(h,g)$.
5. $arco(d,b)$. 9. $arco(i,h)$.
6. $arco(e,b)$. 10. $arco(j,h)$.

Ejercicio 3.36 Defina en PROLOG el predicado *ancestro*, cierre transitivo de la relación *padre*. Proporcione asimismo una definición recursiva en PROLOG para el predicado $factorial$, partiendo de su definición como sentencia lógica presentada en la sección 3.1.1.

Utilizando el programa anterior, un intérprete de PROLOG puede responder, por ejemplo, a la pregunta "¿qué nodos Y y Z pueden conectarse a través de un camino que pase por el nodo b?". Se denominan "preguntas PROLOG" a los objetivos introducidos en un sistema PROLOG con el fin de comprobar su satisfacibilidad y/o computar las correspondientes respuestas. Normalmente, la línea de comandos en que se introducen los objetivos sometidos a prueba se identifica con el símbolo $?-$.

En el sistema de cláusulas asociado, responder a la anterior pregunta supone computar una respuesta para el objetivo $\leftarrow conectado(y,b),\ conectado(b,z)$, que se introduce en la línea de comandos del intérprete como: $?-conectado(Y,b),\ conectado(b,Z)$. La respuesta correcta obtenida en el ejemplo 3.16 de la demostración de validez de la fórmula $\exists y \exists z(q(y,b) \wedge\ q(b,z))$, $\theta = \{y \leftarrow d,\ z\ \leftarrow a\}$, es precisamente una de las respuestas que buscamos. El motor de inferencia de PROLOG computa esta respuesta aplicando el método de resolución en el sistema de cláusulas contenido en su base de datos. Como regla de computación utiliza la selección del literal situado más a la izquierda en la cláusula objetivo, y como regla de búsqueda la selección de la cláusula situada más arriba en la lista de cláusulas de cada procedimiento, de forma que realiza la computación en los siguientes pasos:

1. Escoge resolver $conectado(Y,b)$ (el subobjetivo situado más a la izquierda) con la primera cláusula en que la equiparación es posible (la cláusula 1). La nueva cláusula objetivo es: $\leftarrow arco(Y,b), conectado(b,Z)$.

2. Escoge resolver $arco(Y,b)$ (el subobjetivo situado más a la izquierda del objetivo actual) con la primera cláusula en que la equiparación es posible (la cláusula 5), aplicando la sustitución $\{Y \leftarrow d\}$. La nueva cláusula objetivo es: $\leftarrow conectado(b,Z)$

3. Toma el único subobjetivo posible, $conectado(b,Z)$, para resolver con la primera cláusula en que la equiparación es posible (la cláusula 1). La nueva cláusula objetivo es: $\leftarrow arco(b,Z)$

4. Toma el único subobjetivo posible, $arco(b,Z)$ para resolver con la única cláusula en que la equiparación es posible (la cláusula 3), aplicando la sustitución $\{Z \leftarrow a\}$, obteniéndo la cláusula vacía $\square$.

El árbol SLD correspondiente se muestra en la figura 3.2.

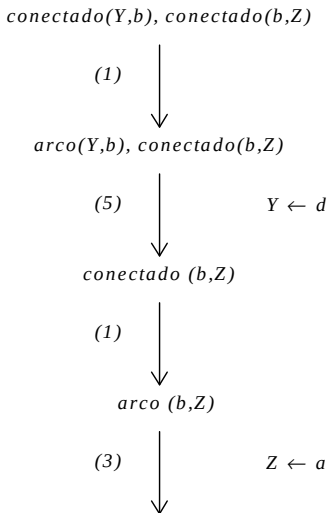


Figura 3.2: Árbol SLD del objetivo $conectado(Y,b), conectado(b,Z)$

Es la aplicación de una estrategia de búsqueda en profundidad lo que hace que los intérpretes de PROLOG sean tan eficientes. Gracias a esta estrategia, en tiempo de ejecución basta con almacenar: a) la cláusula objetivo actual, con un puntero al subobjetivo actual, b) un puntero a la cláusula del procedimiento cuya cabeza se puede unificar con el citado suobjetivo y c) el conjunto de sustituciones de variables realizadas hasta el momento. Adicionalmente, resulta útil señalar de algún modo los nodos previos del árbol SLD donde aún quedan ramas por explorar. De este modo, en caso de ocurrir un fallo, se puede retroceder directamente hasta el nodo más cercano cuyos subárboles puedan proporcionar soluciones adicionales. Tales nodos se registran en una lista y se denominan puntos de retroceso (" backtracking"). Debido a la regla de búsqueda de PROLOG, las ramas no exploradas están siempre situadas a la derecha de la rama actual. Esta estrategia es eficiente desde el punto de vista del almacenamiento, ya que no requiere guardar memoria de todos los caminos explorados, que están implícitos en los datos anteriores. Los puntos de retroceso no sólo son útiles en caso de fallo de

alguna rama, sino en los casos en que se desea computar más de una respuesta correcta, o todas las respuestas correctas para un objetivo dado.

Explorando nuevas ramas del árbol SLD, PROLOG puede computar respuestas adicionales, en particular las soluciones $\{Y \leftarrow e,\ Z \leftarrow a\}$ y $\{Y \leftarrow f,\ Z \leftarrow a\}$, como se muestra en el árbol SLD de la figura 3.3, donde los puntos de retroceso se han señalado con un asterisco.

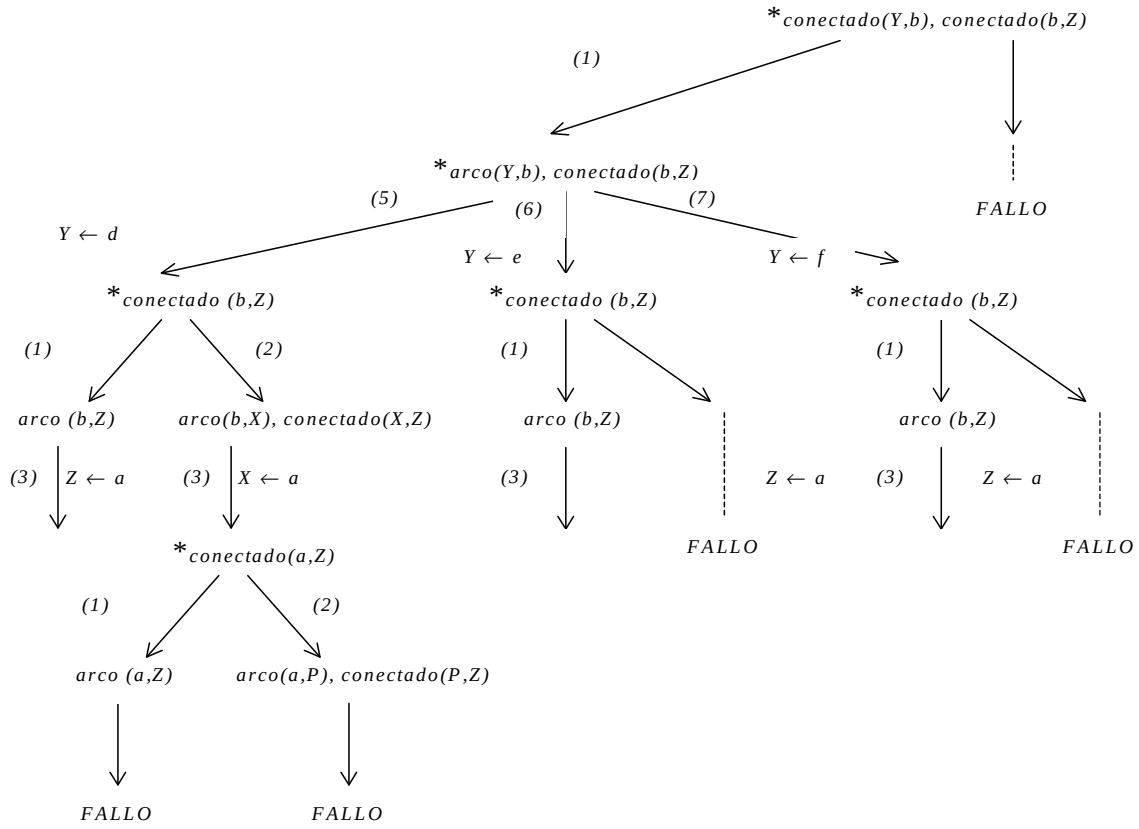


Figura 3.3: Reevaluación del objetivo $conectado(Y,b), conectado(b,Z)$

En la figura 3.4 se muestra cómo PROLOG explora el árbol en busca de nuevas soluciones, retrocediendo en cada paso al punto de retroceso más recientemente señalado.

Supongamos ahora que se formula a PROLOG la pregunta $?-conectado(X,Y)$, con el propósito de encontrar todos los pares de nodos conectados entre sí. La posibilidad de encontrar la solución radica en dar con una rama finita. PROLOG encuentra las soluciones porque su estrategia de recorrido del árbol le conduce en este caso a evaluar primero las ramas finitas. En concreto, comenzará por satisfacer repetidas veces el subobjetivo $arco(X,Y)$ hasta agotar todas las respuestas. A continuación, el retroceso le llevará a intentar satisfacer $conectado(X,Y)$ utilizando la segunda cláusula, encontrando primero un tercer nodo Z ligado mediante un arco a X, es decir, intentado satisfacer $arco(X,Z)$, etc. Sin embargo, consideremos las cláusulas y sus literales ordenados en la siguiente forma:

1. $conectado(Nodo1,Nodo2) :- conectado(Enlace,Nodo2),\ arco(Nodo1,Enlace).$
2. $conectado(Nodo1,Nodo2) :- arco(Nodo1,Nodo2).$

Al desplegar el correspondiente árbol de búsqueda de respuestas para $?-conectado(X,Y)$, nos encontramos con que es infinito. Debido a la recursión, de las ramas situadas más a la izquierda cuelga siempre un subárbol que reproduce el árbol global (véase la figura 3.5). PROLOG no sería capaz de

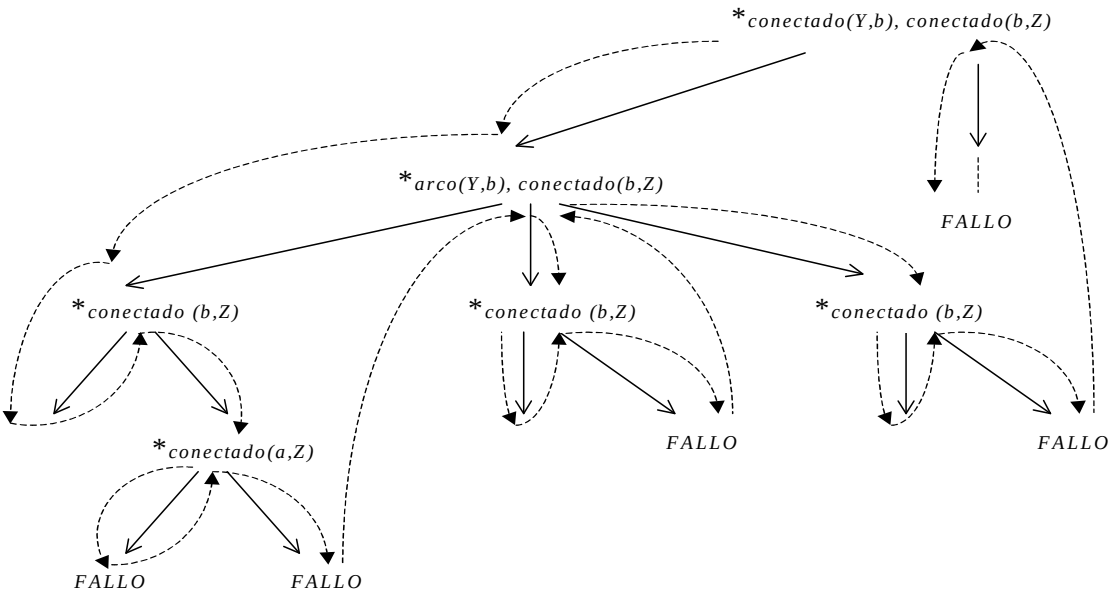


Figura 3.4: Exploración del árbol SLD del objetivo $conectado(Y,b), conectado(b,Z)$ por un intérprete PROLOG

computar ni una sola respuesta, quedándose eternamente explorando la primera rama infinita.

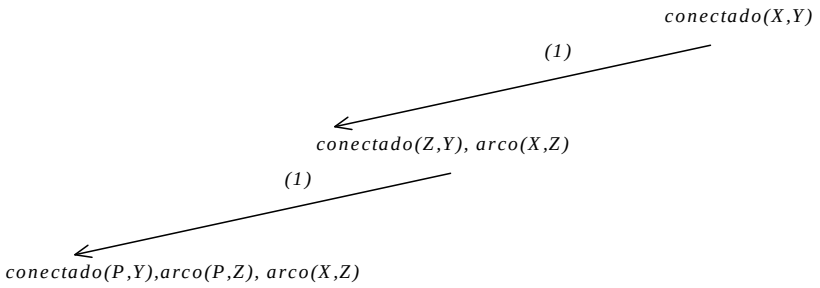


Figura 3.5: Rama infinita del árbol SLD del objetivo $conectado(X,Y)$

Vemos pues, como se anticipó en la sección anterior, que las estrategias de búsqueda en profundidad pueden conducir a computaciones no terminantes incluso en casos en que existe una respuesta correcta. Un programador de PROLOG debe ordenar cuidadosamente las cláusulas dentro de cada procedimiento, así como los literales dentro de las cláusulas, para evitar la no-terminación. Es por esta razón que es muy importante conocer las estrategias de computación y búsqueda de PROLOG y tenerlas presentes cuando se diseñan los programas. En realidad, ésta es una característica de la programación en PROLOG impropia de un paradigma puramente lógico, donde idealmente el programador puede por completo despreocuparse del control de ejecución de sus programas. En la sección 3.4.2 veremos algunas otras características no-lógicas del lenguaje PROLOG.

Ejercicio 3.37 En el caso de la primera versión del programa anterior, muestre cómo PROLOG encuentra todas las soluciones a la pregunta $?-conectado(Nodo1,Nodo2)$ explorando su árbol SLD.

¿Qué ocurre si se intercambian las cláusulas 1 y 2, conservando el orden de los literales dentro de las cláusulas?

Finalmente supóngase que se desea conocer la lista de nodos conectados a un nodo dado, por ejemplo, el nodo h. Bastaría con formular la pregunta PROLOG $?-conectado(Nodo,h)$ y solicitar todas las respuestas posibles. La tarea puede facilitarse no obstante definiendo un nuevo predicado que proporcione en una sola respuesta la lista de tales nodos. Obsérvese que la pregunta anterior podría haberse formulado en la forma $?-conectados(ListaNodos,h)$, donde el predicado *conectados* se habría definido previamente como:

$$\begin{array}{lll} conectados(Nodo1|ListaNodos,Nodo2) & :- & conectado(Nodo1,Nodo2), \\ & & conectados(ListaNodos,Nodo2). \\ conectados([],Nodo). & & \end{array}$$

En este caso el intérprete PROLOG proporcionaría todas las soluciones en la primera respuesta, con la notación característica de las listas PROLOG, donde los elementos aparecen separados por comas y encerrados entre corchetes. La notación $Nodo1|ListaNodos$, denota una lista cuyo primer elemento es $Nodo1$. Hemos introducido este último ejemplo para ilustrar la utilidad de las estructuras de tipo lista en los programas PROLOG, así como la forma en que se manipulan. La posibilidad que proporciona la recursión de definir estructuras de datos infinitas como la lista dota a los programas PROLOG de una gran potencia matemática.

Ejercicio 3.38 a) Defina un predicado PROLOG que descomponga un número en la suma de dos números pares, utilizando los predicados predefinidos $between(A,B,C)$ (pertenencia de un número a un intervalo), $+$ (suma), $=:=$(igualdad), *mod* (módulo o resto de división entera). Muestre el árbol SLD correspondiente a la computación de $descomponer(6,a,b)$.

b) Defina en PROLOG un predicado de ordenación utilizando el programa lógico del ejercicio 3.3 , haciendo uso de los predicados predefinidos PROLOG $member(X,Xs)$ (pertenecencia de un elemento a una lista), $=/=$ (distinto) y $<$ (menor). Muestre el árbol SLD correspondiente a la computación de $ordenar([2,5,9])$.

3.4.2 Usos procedimentales del PROLOG

A lo largo del capítulo hemos señalado en diversas ocasiones la necesidad de los lenguajes prácticos de alejarse del ideal de la programación lógica puramente declarativa. Así, no es posible diseñar buenos programas PROLOG ignorando los mecanismos de ejecución que implementan sus intérpretes. Una programación lógica efectiva requiere conocimiento y utilización de estos mecanismos, de este modelo de ejecución. En la sección previa veíamos cómo un programador de PROLOG se veía obligado a pensar en términos del control de ejecución de sus programas ordenando cuidadosamente cláusulas y literales con el fin de evitar ejecuciones no terminantes.

En esta sección veremos cómo es posible simular en PROLOG las construcciones iterativas de los lenguajes algorítmicos, y cómo determinadas circunstancias exigen que el programador piense en términos procedimentales e interfiera en el control de ejecución del programa utilizando predicados no lógicos, en particular, el potente predicado de corte. Este último es el único predicado que realmente afecta la estrategia de evaluación en los programas PROLOG, y su uso pone en cuestión el carácter auténticamente declarativo del lenguaje.

Programas iterativos

En la introducción de este capítulo presentamos dos algoritmos de cálculo de la función factorial. En PROLOG, dicha función se codifica naturalmente como:

$factorial(0,1)$.

$factorial(N,M) : N > 0,\ N1\ is\ N-1,\ factorial(N1,M1), .M\ is\ N*M1$.

En este lenguaje no existen construcciones iterativas, de modo que cuando se traduce un algoritmo iterativo habitualmente la iteración se sustituye por la recursión. Existe, sin embargo, una clase de programas PROLOG recursivos que guardan una relación cercana con los programas iterativos convencionales. Estos programas incluyen un tipo particular de variables denominadas "acumuladores", que hacen las veces de variables de almacenamiento de resultados intermedios, variables de las que PROLOG carece. Típicamente, estos resultados intermedios se producen como resultado de la computación de cada etapa de una iteración. Ilustramos la técnica con una versión iterativa del programa anterior:

$$
\begin{array}{ll}
factorial(N,M)\text{:-} & factorial(N,1,M).\\
factorial(N,T,M)\text{:-} & N>0,\\
 & T1\ is\ T*N,\\
 & N1\ is\ N-1,\\
 & factorial(N1,T1,M).\\
factorial(0,M,M). &
\end{array}
$$

Este programa no vulnera estrictamente los principios de la programación lógica. Sin embargo, su significado declarativo no es muy intuitivo, mientras que resulta una traducción bastante inmediata de un clásico algoritmo para el cálculo del factorial.

Predicados no lógicos de entrada/salida y cálculo aritmético

Los predicados no lógicos son predicados que carecen de significado declarativo como fórmulas lógicas y cuyo propósito principal es generar determinados "efectos laterales". En esta sección trataremos dos categorías de predicados no lógicos que no interfieren en lo esencial con el mecanismo operacional de PROLOG: los predicados de entratada y salida y los predicados de cálculo aritmético.

Predicados para la gestión de entradas y salidas. Los ejemplos más obvios de predicados no lógicos son los predicados de entrada/salida *get* y *put*. Como literales de una cláusula objetivo, estos predicados siempre se verifican (salvando el caso en que el predicado *get* detecta el final de un fichero, que produce un fallo). Su significado es procedimental: *put* pone un caracter en la pantalla y *get* lee un carácter por teclado. Ambos predicados son imprescindibles para que los sistemas PROLOG puedan interaccionar con periféricos y otros sistemas. Puesto que su uso está normalmente confinado a áreas bien definidas de los programas, y puesto que su comportamiento lógico es trivial, no interfieren realmente con la estructura lógica declarativa de los programas PROLOG.

Predicados para el cálculo aritmético La mayoría de los programas implican algún cálculo aritmético. Si bien la aritmética puede formalizarse en cálculo de predicados, lo cierto es que su formulación resulta muy poco práctica. Así por ejemplo, los números naturales pueden definirse recursivamente del siguiente modo:

$numero\_natural(0)$.

$numero\_natural(sucesor(X)) : -numero\_natural(X)$.

Todos los números naturales se obtienen así recursivamente como

$$0, sucesor(0), sucesor(sucesor(0)), sucesor(sucesor(sucesor(0)))$$

y, en general, $sucesor^n(0)$. Las operaciones de adición, multiplicación y exponenciación puede definirse asimismo de forma recursiva:

$+(0,X,X) : -numero\_natural(X).$
$+(sucesor(X),sucesor(Y),sucesor(sucesor(Z))) : -+(X,Y,Z).$
$*(0,Y,0).$
$*(sucesor(X),Y,Z) : -*(X,Y,XY),+(XY,Y,Z).$
$exp(sucesor(X),0,0).$
$exp(0,sucesor(X),sucesor(0)).$
$exp(sucesor(N),X,Y) : -exp(N,X,Z),*(Z,X,Y).$

Estas definiciones recursivas plantean varios problemas. En primer lugar, la resolución no es un método eficiente de computación numérica. En segundo lugar, la notación resulta larga y tediosa de interpretar. Todos los computadores contienen instrucciones eficientes, directamente implementadas en la máquina, para llevar a cabo operaciones sobre enteros, mientras que, utilizando los predicados anteriores, una operación tan trivial como la suma de la constante 10 a un número requiere al menos 10 unificaciones y resoluciones. La formalización axiomática de los números en coma flotante resulta ya completamente inmanejable.

PROLOG no define la aritmética de acuerdo a los axiomas y definición de tipo precedentes, sino que proporciona un conjunto de predicados predifinidos para la realización de computaciones aritméticas estándar que utilizan directamente las capacidades aritméticas subyacentes del ordenador. Estos predicados se utilizan en sentencias de sintaxis: *Resultado is Expresion*. Los predicados aritméticos se comportan de forma diferente a los predicados ordinarios, particularmente con respecto a la unificación. Un predicado aritmético se interpreta en una única dirección. Así por ejemplo, 10 *is* $X+Y$ sería una sentencia ilegal, que no produciría una secuencia de instanciaciones de X e Y a $(0,10)$, $(1,9)$, etc. Si *Resultado is Expresion*, *Resultado* sólo puede ser una variable sin instanciar y *Expresion* ha de evaluarse a un número básico. Nótese que las sentencias anteriores no son sentencias de asignación, tal y como se conciben en los lenguajes algorítmicos. La sentencia *is* sólo puede pues asignar valor a una variable *Resultado* una única vez en el cuerpo de un predicado.

Los predicados de la aritmética definidos al comienzo de este apartado presentan sin embargo una ventaja: los múltiples usos que puede hacerse de ellos. Así, plantear una pregunta como

$$?-+(sucesor(0), sucesor(0), sucesor(sucesor(0)))$$

significa comprobar si $1+1=2$, mientras que la consulta

$$?-+(sucesor(0),X,sucesor(sucesor(0)))$$

implica llevar a cabo una substracción, y la pregunta

$$?-+(X,Y,sucesor(sucesor(0)))$$

incluso da al sistema la posibilidad de ofrecer múltiples soluciones.

Ejercicio 3.39 Utilizando los predicados predifinidos de cálculo aritmético de PROLOG, proporcione dos definiciones distintas del predicado *mod* (”módulo”o ”resto de una división entera”). La primera de ellas ha de ser una traducción directa de la definición matemática del resto de una división entera: ”Z es el valor de X *mod* Y si Z es estrictamente menor que Y y existe un número Q tal que $X = Q*Y+Z$. La segunda debe constar de dos cláusulas, de modo que la segunda de ellas sea recursiva. Compruebe la mayor eficiencia de la segunda definición, dada la menor dimensión de los árboles SLD que genera.

Predicado de corte

El corte es el predicado no lógico más controvertido de PROLOG, por la modificación tan importante que supone de la programación lógica. El predicado de corte, denotado con el símbolo !, interfiere directamente en el procedimiento de refutación SLD evitando que se lleve a cabo el retroceso en ciertos puntos. Una vez que el predicado de corte se ha ejecutado en un potencial punto de retroceso, todas las ramas alternativas del árbol SLD que penden del nodo correspondiente se podan automáticamente.

Puesto que el principal objetivo de la programación lógica es permitir que el programador escriba programas declarativos dejando la parte de control para el motor de inferencia genérico implementado en los intérpretes, los cortes deberían evitarse, puesto que sólo pueden entenderse desde una perspectiva procedimental de la programación. Sin embargo, los programadores deben estar familiarizados con los diferentes usos del corte, por lo que a continuación presentamos algunos ejemplos de cómo el predicado de corte permite resolver esencialmente problemas de ineficiencia y, en particular, de computación no terminante.

Ejemplo 3.40 Consideremos la siguiente definición del predicado *mezcla* en PROLOG:

$\%mezcla(Xs,Ys,Zs) :-$

$\%mezcla\ dos\ listas\ ordenadas\ de\ números\ enteros\ Xs\ e\ Ys\ en\ la\ lista\ ordenada\ Zs$

$mezcla([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, mezcla(Xs,[Y|Ys],Zs).$

$mezcla([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, mezcla(Xs,Ys,Zs).$

$mezcla([X|Xs],[Y|Ys],[X,Zs]) :- X > Y, mezcla([X|Xs],Ys,Zs).$

$mezcla(Xs,[],Xs).$

$mezcla([],Ys,Ys).$

Ante cualquier pregunta PROLOG, una y sólo una de las cinco cláusulas es aplicable. Sin embargo en caso de solicitarse todas las respuestas posibles, el intérprete tanteará todas las opciones.

Ejercicio 3.41 Compruebe la ineficiencia del predicado construyendo el árbol SLD correspondiente a la computación del objetivo $mezcla([1,3,5],[2,3],Ks)$.

Ejemplo 3.42 Consideremos ahora la definición de un predicado de ordenación de listas:

$\%ordena(Xs,Ys) :-$

$\%ordena\ la\ lista\ Xs\ en\ la\ lista\ Ys$

$ordena(Xs,Ys)$:- $append(As,[X,Y|Bs],Xs),$
$X > Y,$
$append(As,[Y,X|Bs],Vs),$
$ordena(Vs,Ys).$

$ordena(Xs,Xs) :- ordenada(Xs).$

$\%append(Ps,Qs,PsQs) :-$

$\%PsQs\ es\ la\ concatenación\ de\ las\ listas\ Ps\ y\ Qs$

$append([],Qs,Qs).$

$append([P|Ps],Qs,[P|Zs]) :- append(Ps,Qs,Zs).$

$\%ordenada(Xs) :-$

$\%Xs$ es una lista ordenada

$ordenada([]).$

$ordenada([X]).$

$ordenada([X,Y|Ys]) :- X \leq Y, ordenada([Y|Ys]).$

El programa busca un par de elementos adyacentes desordenados, los intercambia y continúa hasta que la lista está ordenada. Dado que sólo existe una lista ordenada, cualquier alternativa de búsqueda conduce a la misma solución. Sin embargo, ante una solicitud de todas las respuestas posibles el intérprete buscará inútilmente otras alternativas.

Ejercicio 3.43 Comprueba la ineficiencia del predicado construyendo el árbol SLD correspondiente a la computación del objetivo $ordena([3,2,1], Ks)$.

Veamos ahora cómo se puede utilizar el predicado de corte para aumentar la eficiencia de los programas anteriores. Un predicado de corte define un objetivo que siempre se satisface y que compromete todas las elecciones hechas desde que el objetivo que se está resolviendo se unificó con la cabeza de la cláusula en que el corte ha ocurrido, es decir, poda todas las ramas alternativas que penden del punto de retroceso más reciente del correspondiente árbol SLD. En definitiva, si un corte se satisface, no intentarán satisfacerse cláusulas alternativas a la cláusula que lo contiene. Como consecuencia, una conjunción de objetivos seguida de un corte producirá a lo sumo una solución. Obsérvese, sin embargo, que un corte no afecta a los objetivos situados a su derecha, que sí podrán producir más de una solución. Sin embargo, si estos fallan, la búsqueda procederá desde la última alternativa previa a la elección de la cláusula que contiene el corte.Veamos cómo definir formalmente este predicado (ver figura 3.6).

Definición 3.44 Sea una cláusula C en un procedimiento que define el predicado A:
$C = A :- B_1....B_k, !, B_{k+2}, ..., B_n$
Si el objetivo actual G se unifica con la cabeza de C y $B_1....B_k$ se satisfacen, el corte:

1. compromete la elección de C para reducir G; cualesquiera cláusulas alternativas para A unificables con G son ignoradas;

2. si B_i falla para $i > k+1$, el retroceso opera sólo hasta llegar a !. Las computaciones restantes en B_i, $i \leq k$, se podan del árbol de búsqueda;

3. si el retroceso alcanza a ! éste falla y la búsqueda procede desde la última elección hecha antes de elegir a C para la reducción de G.

Ejemplo 3.45 Coloquemos cortes en el predicado mezcla, con el fin de solventar los problemas de eficiencia antes destacados:

$mezcla([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, !, mezcla(Xs,[Y|Ys],Zs).$

$mezcla([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, !, mezcla(Xs,Ys,Zs).$

$mezcla([X|Xs],[Y|Ys],[X,Zs]) :- X > Y, !, mezcla([X|Xs],Ys,Zs).$

$mezcla(Xs,[],Xs) :- !.$

$mezcla([],Ys,Ys) :- !.$

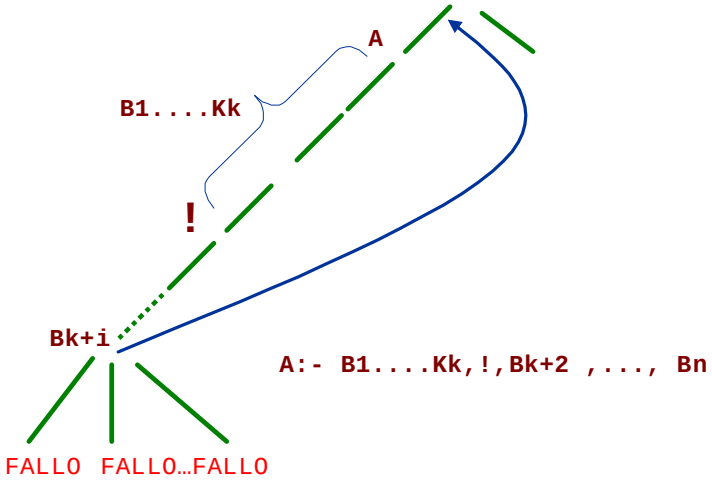


Figura 3.6: Funcionamiento del operador de corte

Obsérvese que el corte se sitúa tras el test en las tres cláusulas recursivas, y como única cláusula del cuerpo de la regla en los casos base.

Ejercicio 3.46 Compruebe la eficiencia del nuevo predicado construyendo el árbol SLD correspondiente a la computación del objetivo $mezcla([1,3,5],[2,3],Ks)$.

Coloquemos también cortes en el predicado ordena:

$$\begin{array}{ll} ordena(Xs,Ys):- & append(As,[X,Y|Bs],Xs), \\ & X > Y,!, \\ & append(As,[Y,X|Bs],Vs), \\ & ordena(Vs,Ys). \end{array}$$

$$ordena(Xs,Xs) :- ordenada(Xs),!.$$

Ejercicio 3.47 Compruebe la eficiencia del nuevo predicado construyendo el árbol SLD correspondiente a la computación del objetivo $ordena([3,2,1],Ks)$.

Nótese que la utilización del predicado de corte ha implicado, en los dos casos anteriores, mejorar la eficicencia tanto espacial como temporal: al podarse ramas del árbol de búsqueda se reduce el tiempo de computación y, al requerirse guardar menos información para usar en caso de retroceso, se gana espacio de almacenamiento. Como contrapartida, al podar dinámicamente los árboles de búsqueda, el corte ha alterado la estrategia de evaluación de los programas. Como ilustran los ejemplos anteriores, el corte permite evitar caminos de computación infructuosos que el programador sabe que no producirán soluciones o producirán soluciones redundantes o indeseadas. En particular, podrá también utilizarse para podar caminos infinitos origen de computaciones no terminantes.

Ejercicio 3.48 El siguiente predicado permite averiguar si el factorial de un número es menor que 100:

$comprueba(N) :- factorial(N,F), minimo(F,100,F)$.

Identifique posibles computaciones no terminantes en la utilización de este predicado e indique cómo evitarlas con la utilización del predicado de corte.

El predicado de corte se introdujo inicialmente para aumentar la eficiencia de los programas PROLOG, sin embargo, su utilización puede tener otros efectos no relacionados con la eficiencia. A este respecto se distingue entre los denominados "cortes verdes" y "cortes rojos".

Los cortes verdes podan ramas de computación que no conducen a nuevos resultados, proporcionando una solución más eficiente sin que su adición o eliminación altere el significado declarativo de los predicados. Estos corte se utilizan, por ejemplo, para hacer explícita la naturaleza mutuamente exclusiva de tests aritméticos (como en el caso del predicado *mezcla*) o para eliminar computaciones redundantes (como en el caso del predicado *ordena*).

Ejercicio 3.49 Reflexiona sobre el uso del corte verde en la definición del predicado:

$\%minimo(X,Y,Min) :-$

%Min es el mínimo de los números X e Y

$minimo(X,Y,X) :- X \leq Y, !.$

$minimo(X,Y,Y) :- X > Y, !.$

Comprueba la eficiencia del predicado construyendo el árbol SLD correspondiente a la computación del objetivo $minimo(5,3,K)$.

Los cortes rojos, por el contrario, podan ramas de computación que podrían conducir a nuevas soluciones, de modo que su adición y eliminación altera el significado declarativo del predicado. Veamos un uso habitual de este tipo de cortes: la omisión de condiciones explícitas.

Previamente definimos una versión del predicado $ordena(Xs,Ys)$ con cortes verdes. Consideremos ahora una nueva definición que incluye cortes rojos:

Ejemplo 3.50

```
%ordena(Xs,Ys) :-
% ordena la lista Xs en la lista Ys
ordena(Xs,Ys):-  append(As,[X,Y|Bs],Xs),
                 X > Y, !,
                 append(As,[Y,X|Bs],Vs),
                 ordena(Vs,Ys).
ordena(Xs,Xs) :- !.
```

Puede observarse que la primera regla se aplica siempre que hay un par de elementos adyacentes en la lista que están desordenados. Debido al corte, cuando se usa la segunda ya no existen tales elementos y la lista está ordenada: la condición $ordenada(Xs)$ puede pues omitirse. Este uso del corte es ciertamente peligroso, ya que puede olvidarse que si el corte se suprime el programa daría soluciones falsas.

Ejercicio 3.51 Reflexione sobre la posible adición de cortes rojos y verdes en la definición del predicado $borra(Lista,X,SinXs)$

```
% borra(Lista,X,SinXs) :- la lista SinXs es el resultado de eliminar todas las ocurrencias
% de X de la lista Lista
borra([X | Xs],X,Ys) :- borra(Xs,X,Ys).
borra([X,Xs],Z,[X | Ys]):-  X =\= Z,
                            borra(Xs,Z,Ys).
borra([],X,[]).
```

Observemos, finalmente, que la utilización de cortes puede conducir a definiciones erróneas de predicados. Consideremos la definición de predicado del siguiente ejemplo:

Ejemplo 3.52 $\%minimimo(X,Y,Z) :-$

$\%Z$ *es el mínimo de los enteros* X *e* Y

$minimo(X,Y,X) :- X =< Y, !.$

$minimo(X,Y,Y).$

El razonamiento implícito en esta definición es: si X es menor o igual que Y, entonces el mínimo es X; de otro modo el mínimo es Y, y cualquier otra comparación entre X e Y es innecesaria. Esta definición de predicado es incorrecta: conduce a la satisfacción de objetivos incorrectos debido a la omisión de la condición $X > Y$ en la segunda regla.

Ejercicio 3.53 Compruebe la satisfacción del objetivo $minimo(2,5,5)$ utilizando la anterior definición del predicado.

El error anterior puede evitarse haciendo explícita la unificación entre los argumentos primero y tercero, que está implícita en la primera regla:

$minimo(X,Y,Z) :- X =< Y, !, Z = X.$

$minimo(X,Y,Y).$

El problema de este modo de usar el corte es que genera un código difícil de interpretar.

El uso de cortes para la eliminación de condiciones explícitas es un buen ejemplo de uso peligroso del operador de corte. Se basa en el conocimiento del comportamiento de PROLOG, específicamente sobre el orden en que se usan las reglas, para omitir condiciones que podrían inferirse como ciertas. Omitir una condición es posible si el fallo de las cláusulas previas la implica; en general, la condición es la negación de las condiciones previas. A veces resulta esencial en la programación práctica ya que las condiciones explícitas, especialmente las negativas, son engorrosas de especificar o ineficientes en ejecución. Sin embargo, la omisión de condiciones es propensa a error, obliga a tener en mente el comportamiento operacional de PROLOG y permite escribir programas que resultan falsos leídos como programas lógicos: proporcionan conclusiones falsas, aunque se comportan correctamente porque PROLOG es incapaz de probarlas. En general, omitir condiciones simples es desaconsejable: la ganancia en eficiencia es mínima comparada con la pérdida de legibilidad y mantenimiento del código. Siempre será preferible escribir el programa lógico correcto y después añadir cortes si es importante para la eficiencia:

$minimo(X,Y,Z) :- X =< Y, !.$

$minimo(X,Y,Y) :- X > Y, !.$

3.4.3 Ventajas de la programación en PROLOG. Principales aplicaciones.

Las características de la programación lógica descritas a lo largo de este capítulo, junto a las propias del lenguaje, proporcionan las siguientes ventajas de la programación en PROLOG frente a la programación convencional:

Programación declarativa, en un alto nivel de abstracción No obstante las objeciones planteadas en la sección anterior, el programador de PROLOG se ve en gran parte liberado de la tarea de especificar como resolver los problemas de forma compleja y paso a paso, y se concentra principalmente en describir el propio problema, mientras que la solución es llevada a cabo por los mecanismos operacionales predefinidos de PROLOG.

Prototipado rápido Tanto la lógica como la programación requieren la expresión explícita de conocimiento y métodos de solución de un problema en un formalismo. Sin embargo, mientras que la formalización del conocimiento en lógica con frecuencia contribuye a profundizar en el problema en consideración, la formalización en términos de un lenguaje de programación convencional raras veces aporta beneficios. PROLOG se ha demostrado muy útil para el diseño de prototipos en las fases de especificación de sistemas, proporcionando al mismo tiempo una especificación estructurada del problema y una fuente compilable.

Facilidad de aprendizaje A pesar de que sus raíces se encuentren en la disciplina matemática de la lógica, el uso de PROLOG no requiere conocimientos de lógica formal. De hecho, su sintáxis simple y compacta lo convierten en un lenguaje muy fácil de aprender, incluso para los que no tienen previa experiencia en programación. No hay que pensar, no obstante que escribir programas en PROLOG es tarea sencilla. Un principiante puede ser rápidamente capaz de escribir sus primeros programas, pero el desarrollo de programas grandes requiere un modo riguroso de pensar y una maestría del lenguaje difíciles de adquirir.

Tiempo de desarrollo corto En PROLOG, el número de líneas de código que se requieren para resolver un problema es típicamente solo una fracción del tiempo requerido con un lenguaje de programación en forma de procedimientos como C o Pascal. Claramente esto puede reducir considerablemente los costes de desarrollo, y puesto que el código es más fácil de modificar, los costes subsiguientes de mantenimiento son con frecuencia también más bajos.

Facilidad de lectura y modificación Muchos de los típicos errores de programación comunes en lenguajes como C o Pascal -por ejemplo, un bucle que itera demasiadas veces o una variable sin instanciar - no se producen cuando se programa en PROLOG. El código puede verse como la especificación de un problema bien estructurada que además es ejecutable. Tal código es además fácilmente legible y modificable cuando se modifican aspectos del dominio en cuestión.

Fácil manipulación de estructuras de datos complejas Trabajar con complejas estructuras de datos como árboles, listas o grafos con frecuencia da lugar a programas grandes y complejos que gestionan la reserva y liberación de memoria. Por el contrario, PROLOG proporciona una notación simple y elegante para acceder a tales estructuras de datos y definirlas recursivamente, ahorrando al programador la implementación de todos los detalles, punteros y gestión explícita del almacenamiento.

Mecanismos de control alternativos El control en los programas PROLOG es en cierto modo similar al de los lenguajes convencionales. La invocación de objetivos se corresponde con la invocación de procedimientos, y la ordenación de los objetivos en el cuerpo de las reglas se corresponde con el secuenciamiento de sentencias. Las diferencias ocurren cuando tiene lugar la reevaluación. En un lenguaje convencional, si la computación no puede proseguir (por ejemplo, todas las ramas de una sentencia "case" son falsas), tiene lugar un error en tiempo de ejecución. En PROLOG, simplemente la computación retrocede hasta el último punto donde se seleccionó una alternativa, y se intenta un camino de computación diferente.

Manipulación flexible de estructuras de datos. Lenguaje no tipado. Las estructuras de datos manipuladas por los programas lógicos, denominadas términos, se corresponden en general con estructuras registro en los lenguajes convencionales. La gestión de las estructuras de datos en PROLOG

es muy flexible. Como el Lisp, PROLOG es un lenguaje no tipado, libre de declaraciones. Las diferencias más importantes entre el PROLOG y los lenguajes convencionales en el uso de estructuras de datos surge de la naturaleza de las variables lógicas. Las variables lógicas se refieren a entes más que a localizaciones de memoria. Consecuentemente, una vez que una variable se ha ligado a un objeto particular, ya no es posible asignársela a otro. El contenido de una variable inicializada no puede cambiar. La manipulación de datos en los programas lógicos es llevada a cabo enteramente por el algoritmo de unificación, que implica asignación, paso de parámetro y alojamiento en memoria. Estas operaciones se llevan a cabo en programación convencional usando listas lincadas y manipulación de punteros. La manipulación de las variables lógicas por medio de la unificación puede verse como una abstracción de la manipulación de bajo nivel de punteros a complejas estructuras de datos.

Diversidad de aplicaciones El uso de los lenguajes de programación lógica no se ha limitado a las aplicaciones de investigación. Entre sus campos de mayor difusión destacan las aplicaciones de Inteligencia Artificial (sistemas expertos - basados en reglas y marcos - , resolución de restricciones, procesamiento del lenguaje natural), la teoría y desarrollo de las bases de datos deductivas, y también aplicaciones más convencionales tales como bases de datos, problemas de planificación dinámica, configuración y logística, implementación de compiladores, etc. Un área de interés actual en la programación lógica es el de los sistemas concurrentes y paralelos. Mientras que PROLOG se ha a daptado particularmente para su ejecución en arquitecturas Von Neumann, lo cierto es que la programación lógica en sus formas más puras, en parte por las características antes señaladas, se adapta particularmente al diseño de lenguajes de programación concurrente (tales como el difundido Parlog.

Bibliografía complementaria

Un excelente libro sobre los fundamentos de la programación lógica, con un estudio del PROLOG como caso particular, es[1]. Otros dos libros interesantes - y en cierto modo complementarios - son [2], que no trata ningún lenguaje particular, y [3], que incluye un capítulo sobre la verificación de programas en PROLOG. Una comparación de distintos lenguajes de programación lógica se encuentra en [4]. Textos recomendados para un estudio más en profundidad del lenguaje Prolog son [5], [6],[7] y el clásico y muy completísimo [8]. El alumno que desee ampliar conocimientos puede estudiar programación lógica concurrente y programación lógica con restricciones respectivamente en las secciones 8.4 y 8.5 de [9].

[1] J. Lloyd. Foundations of Logic Programming. *Addison-Wesley, Reading, MA, 1987. 2ª edición.*

[2] K. Doets, From Logic to Logic Programming. *MIT Press, Cambridge, MA, 1994.*

[3] K.R. Apt. From Logic Programming to Prolog.

[4] E. Shapiro, The family of concurrent logic programming languages *ACM Computing Surveys, 21 (1989) 413-510.*

[5] W.F. Clocksin y C.S. Mellish. Programación en Prolog. *Springer-Verlag (1984). Traducido en Editorial Gustavo Gili, 1987.*

*[6]I. Bratko (1991).*Prolog programming for artificial intelligence. *Addison-Wesley (Segunda edición).*

[7]F. Giannesini, R. H. Kanoui, R. Pasero, M. van Caneghem. Prolog. *Versión traducida (1989). Addison-Wesley Iberoamericana.*

[8]L. Sterling, E. Shapiro (1986). The art of prolog. *MIT Press.*

[9] M. Ben-Ari. Principles of Concurrent and Distributed Programming. *Prentice-Hall International, Londres, 1990.*

Actividades y evaluación

Los ejercicios más interesantes de este tema consisten en construir el árbol de deducción SLD asociado a la ejecución de un pequeño programa lógico; véanse los ejercicios 4, 5 y 6 del capítulo 8 del texto: Ben-Ari M. Mathematical Logic for Computer Science. Springer-Verlag, London, 2001. También es conveniente realizar los ejercicios 4 a 7 tal como están enunciados en el libro. Como apoyo para la realización de estos ejercicios puede ser muy interesante utilizar un programa para la visualización de árbolos SLD, tal como el que ofrece un alumno de la Universidad de Málaga en la dirección Web: http://polaris.lcc.uma.es/ pacog/apuntes/pd/

Finalmente, es muy interesante que el alumno se inicie en la programación en PROLOG mediante la codificación de pequeños programas. Los ejercicios no se centrarán en cuestiones de estilo ni se orientarán al desarrollo de grandes aplicaciones, ya que tan sólo se pretende que el alumno comprenda las idiosincrasias de la programación lógica y los compromisos (programación lógica frente a algorítmica) que supone el diseño de programas eficientes. Adicionalmente, la programación en PROLOG mejorará la comprensión de la lógica proposicional y de predicados, particularmente de los métodos deductivos que utilizan la resolución (junto con los procesos de equiparación y unificación de variables que conlleva), ya que la traducción de las cláusulas Horn a código PROLOG es inmediata.

Para este propósito el alumno instalará en su ordenador un intérprete de PROLOG. En las páginas web de la asignatura "Programación orientada a la inteligencia artificial", cuya dirección es:

http://www.ia.uned.es/asignaturas/prog-ia/util/index.html

se pueden encontrar diferentes intérpretes tanto para los sistemas LINUX como para los sistemas WINDOWS. Por su sencillez, aconsejamos utilizar el entorno SWI. Sugerimos la utilización de las facilidades de traza que los intérpretes proporcionan, con el fin de depurar los programas y comprender en profundidad la semántica operacional que los sistemas PROLOG implementan (algoritmo de búsqueda, mecanismos de unificación y equiparación...). Una vez codificado un predicado, es también altamente recomendable que el alumno ensaye a dibujar el árbol SLD correspondiente a la ejecución de determinadas preguntas PROLOG y compruebe cómo el intérprete lleva a cabo el correspondiente proceso de resolución.

Capítulo 4

VERIFICACIÓN DE PROGRAMAS SECUENCIALES

Resumen

Este tema estudia cómo definir con precisión la semántica de un lenguaje formal mediante la lógica de Hoare y cómo utilizar un método deductivo basado en ella para la verificación de programas secuenciales.

Objetivos

El objetivo principal es que el alumno aprenda a verificar pequeños programas secuenciales mediante la lógica de Hoare.

Metodología

Para poder comprobar que un programa es correcto hace falta establecer con precisión su semántica. Por ello, casi al principio de este tema se define un lenguaje que contiene sólo tres instrucciones: if-then-else, while-do y := (asignación de valores a variables); aunque se trata de un lenguaje aparente muy simple, la mayor parte de las instrucciones de lenguajes de programación más complejos —salvo las relativas a interfaces de entrada/salida— se pueden construir a partir de estas tres instrucciones básicas. Luego se establece la semántica de este lenguaje mediante ciertas expresiones lógicas, las ternas de Hoare, que representan la transformación de estados asociada a cada instrucción. Finalmente, se expone un sistema deductivo para la lógica de Hoare y se explica cómo aplicarlo a la verificación y síntesis de programas.

4.1 Introducción

El hecho de que un programa tenga un error puede resultar sumamente costoso, no sólo en términos económicos, sino que en ciertos casos incluso puede poner en peligro la vida de muchos seres humanos. Por eso es importante disponer de métodos que permitan comprobar que un programa cumple las especificaciones con que fue diseñado. Generalmente las especificaciones suelen venir dadas en lenguaje natural. Por ejemplo: “Quiero un programa que calcule las nóminas de mis empleados, a partir de los siguientes datos…”. Naturalmente, dada la ambigüedad y falta de precisión del lenguaje

natural y la ausencia de métodos que permitan una compresión automática del mismo, se hace necesario contar con descripciones formales que indiquen de forma precisa e inequívoca las especificaciones de cada programa. Una vez que se conocen las especificaciones, sería deseable contar con un método, implementable en un programa de ordenador, que generase automáticamente el programa buscado, libre de errores. Dado que esto es todavía hoy ciencia-ficción, al menos sería deseable contar con métodos que permitan comprobar de forma automática o semiautomática que cierto programa cumple las especificaciones, es decir, que hace lo que se espera de él, sin cometer nunca errores; es lo que se conoce como *verificación de programas*.

Aún estamos lejos de contar con verificadores totalmente automáticos, pero recientemente se han desarrollado ya verificadores semiautomáticos para lenguajes de alto nivel.[1] Los grandes avances que se han producido en las últimas tres décadas y los intereses de los fabricantes de software por garantizar la fiabilidad de sus productos hacen pensar que en los próximos años se seguirán produciendo progresos muy significativos en este campo. Precisamente por la importancia del tema, es de esperar que el conocimiento de los métodos de verificación formal sea una de las cualidades más valoradas de los ingenieros en informática de un futuro no muy lejano.

En este tema vamos a estudiar los métodos de verificación de programas secuenciales. En concreto, vamos a estudiar un sistema deductivo que permite verificar programas de un lenguaje de programación muy sencillo, tan sencillo que sólo tiene tres instrucciones: asignación de variables (`:=`), condicional (`if-then-else`), y bucle (`while-do`). Aunque se trata de un lenguaje aparentemente muy simple, la mayor parte de las instrucciones de lenguajes de programación más complejos —salvo las relativas a interfaces de entrada/salida— se pueden construir a partir de estas tres instrucciones básicas. Luego introduciremos ciertas expresiones lógicas, las ternas de Hoare, que permiten especificar formalmente un programa. Más adelante expondremos un sistema deductivo (el sistema de Hoare) y explicaremos cómo aplicarlo a la verificación de programas escritos en nuestro micro-lenguaje.

4.2 Sintaxis

4.2.1 Un micro-lenguaje de programación

Como hemos dicho en la introducción, la verificación de un programa escrito en un lenguaje de alto nivel es sumamente complicada. Para simplificar nuestro estudio, vamos a definir un pequeño lenguaje que sólo tiene tres instrucciones y dos tipos de datos: booleanos y enteros. En las conclusiones comentaremos las limitaciones de este micro-lenguaje en comparación con los lenguajes habituales.

Utilizando la notación gramatical de Naur Backus (NBF), definimos nuestro micro-lenguaje así:

$$E ::= n \mid x \mid \texttt{(}-E\texttt{)} \mid \texttt{(}E + E\texttt{)} \mid \texttt{(}E - E\texttt{)} \mid \texttt{(}E * E\texttt{)}$$

$$B ::= \texttt{true} \mid \texttt{false} \mid \texttt{!}B \mid \texttt{(}B \texttt{ \& } B\texttt{)} \mid \texttt{(}B \mid B\texttt{)} \mid \texttt{(}E == E\texttt{)} \mid \texttt{(}E \texttt{ != } E\texttt{)} \mid \texttt{(}E < E\texttt{)} \mid \texttt{(}E \texttt{ =< } E\texttt{)} \mid \texttt{(}E > E\texttt{)} \mid \texttt{(}E \texttt{ >= } E\texttt{)}$$

$$S ::= x\texttt{:=}E \mid \texttt{if } B \texttt{ then (}S\texttt{)[else (}S\texttt{)]} \mid \texttt{while } B \texttt{ do (}S\texttt{)} \mid S\texttt{;}S$$

Observe que en este lenguaje tenemos dos tipos de expresiones: boleanas (B) y enteras (E) y tres tipos de instrucciones (S, del inglés, *statement*). Una expresión booleana puede venir dada por un entero (n), tal como `1` ó `3758`, una variable de programa (por ejemplo, `x`), el opuesto de una expresión entera (por ejemplo, `-1` o `-y`) o bien por la suma, resta o multiplicación de dos expresiones enteras.

[1] Vea, por ejemplo, el proyecto KeY, `http://i12www.ira.uka.de/~key/`, que ha conseguido integrar *especificación formal* (en OCL) y *verificación semiautomática* (con una ampliación de las técnicas expuestas en este capítulo) en una herramienta CASE de *modelado mediante objetos* (UML), especialmente diseñada para programar en Java.

Las palabras reservadas `true` y `false` son expresiones boolenas. La negación de una expresión booleana también lo es, así como la conjunción y disyunción de expresiones booleanas y la comparación de expresiones enteras.[2]

El primer tipo de instrucción que tenemos es la asignación de una expresión numérica E a una variable x. Hay también expresiones condicionales (de la forma `if-then` o `if-then-else`) y, en tercer lugar, bucles (instrucciones `while`). La concatenación de dos expresiones (C;C) también es una expresión.

Ejemplo 4.1 El siguiente programa, que denominaremos `fact1`, está escrito en nuestro micro-lenguaje:

```
fact := 1;
i := 0;
while (i != x) do (
    i := i + 1;
    fact := fact * i
)
```

Como el lector habrá adivinado, este programa sirve para calcular el factorial de x. Más adelante vamos a demostrar formalmente que esto es así.

4.2.2 Especificación de estados

Dado el conjunto de variables que aparecen en un programa, un *estado* viene dado por la asignación de un valor a cada una de variables. Por ejemplo, si las variables son `x`, `i` y `fact`, la expresión $\{x = 10 \wedge i = 4 \wedge fact = 24\}$ representa un estado.

Puede haber también expresiones en que no aparezcan todas las variables que se usan en el programa. Tales expresiones no representan un estado, sino un conjunto infinito de ellos. Así, si las variables de un programa son `x`, `i` y `fact`, la expresión $\{i = 4\}$ no representa un único estado, sino el conjunto de todos los estados en que la variable `i` tiene el valor 4; este conjunto tiene tantos estados como valores puedan tomar `x` y `fact`. Otro ejemplo de expresión que representa un conjunto de estados es la siguiente: $\{x \geq 0 \wedge x \leq 10 \wedge i \neq x\}$.

Cualquier expresión lógica, por tanto, representa un estado o un conjunto de estados. En particular, $\{\top\}$ representa todos los estados posibles, y $\{\bot\}$ no representa ningún estado (conjunto vacío).

4.2.3 Ternas de Hoare

Dado un lenguaje que expresa los estados de un sistema y un lenguaje de programación (cada fórmula de este lenguaje representa una instrucción, es decir, un programa), definimos un nuevo lenguaje formado por ternas de la forma

$$\{\text{precondición}\}(\text{instrucción})\{\text{postcondición}\}$$

Estas expresiones se denominan ternas de Hoare. La interpretación intuitiva de esta expresión es que si un sistema que se encuentra en alguno de los estados representados por la precondición ejecuta la instrucción, pasa a alguno de los estados representados por la postcondición.

[2] Observe que en Pascal la asignación de variables se representa mediante "`:=`" y la comparación mediante "`=`", mientras que en C y Java la asignación se representa mediante "`=`" y la comparación mediante "`==`". Para evitar confusiones, en nuestro micro-lenguaje hemos utilizado "`:=`" para la asignación y "`==`" para la comparación. En cambio, en las expresiones lógicas utilizaremos el signo "`=`" para denotar la igualdad porque en ellas no hay confusión sobre su significado.

Ejemplo 4.2 La terna $\{i = 4\}$(`i:=i+1`)$\{i = 5\}$ significa que si el sistema se encuentra en un estado en que `i` vale 4 y ejecuta la instrucción `i:=i+1`, pasa a un estado en que `i` vale 5.

Ejemplo 4.3 $\{i > 0\}$(`i:=i+1`)$\{i > 1\}$.

Ejemplo 4.4 Dado que la condición $\top$ es cierta para todos los estados, la terna $\{\top\}$(`i:=5`)$\{i = 5\}$ significa que, cualquiera que se sea el estado inicial del sistema, la ejecución de la instrucción `i:=5` hace que el sistema pase a un estado en que `i` vale 5.

Nota. Como hemos visto anteriormente al definir nuestro micro-lenguaje, la concatenación de dos instrucciones es una nueva instrucción. En la práctica llamamos *programa* a una instrucción compuesta que realiza cierta tarea. Sin embargo, formalmente no hay diferencia entre programas e instrucciones.

Cuando tenemos una terna de Hoare en que el programa no está definido todavía, se dice que tenemos una *especificación* del programa, pues estamos indicando solamente cuáles son los requisitos que debe cumplir el programa. Por ejemplo, la condición

$$\{x \geq 0\}(\texttt{S})\{fact = x!\} \tag{4.1}$$

es una especificación, porque indica que el programa debe calcular el factorial de un número no negativo (más adelante veremos por qué esta especificación no es del todo correcta).

4.2.4 Variables de programa y variables lógicas

Imagine que queremos expresar mediante una terna de Hoare la propiedad de que, cualquiera que sea el valor inicial de la variable `i`, la instrucción `i:=i+1` hace que el valor de esta variable aumente en una unidad. Por analogía con el ejemplo 4.4, un aprendiz de lógica ingenuo podría intentar representarla así:

$$\{\top\}(\texttt{i:=i+1})\{i = i + 1\} \qquad \text{(incorrecta)}$$

Claramente, esta expresión es incorrecta, porque la condición $i = i + 1$ siempre es falsa. La forma correcta de representar la propiedad anterior es ésta:

$$\forall a,\ \{i = a\}(\texttt{i:=i+1})\{i = a + 1\}$$

Observe que en este caso la variable a no aparece en la instrucción. Estas variables, que no forman parte del programa, sino que se introducen para relacionar la precondición con la postcondición, se denominan *variables lógicas*, para distinguirlas de las *variables de programa*, que son las que aparecen la instrucción S. En las ternas de Hoare, las variables lógicas están siempre sujetas a un cuantificador universal, que en la práctica suele omitirse, de modo que lo habitual será escribir la expresión anterior simplemente así:

$$\{i = a\}(\texttt{i:=i+1})\{i = a + 1\}$$

Pero no debemos olvidar que en realidad hay un cuantificador en esta expresión, aunque no lo hayamos escrito.

Veamos con otro ejemplo la necesidad de introducir variables lógicas. Ya hemos dicho antes que la expresión (4.1) es una especificación para el cálculo del factorial de enteros no negativos. Aunque todavía no hemos desarrollado los métodos formales, no será difícil para el lector comprobar que el

programa `fact1` del ejemplo 4.1 cumple esta condición, lo cual es correcto. Sin embargo, observe que el programa `(x:=1;fact:=1)` también cumple la condición (4.1),

$$\{x \geq 0\}(\texttt{x:=1;fact:=1})\{fact = x!\}$$

a pesar de que no calcula correctamente el factorial cuando $x > 1$. Esto nos muestra que la especificación (4.1) no es satisfactoria.

En cambio, el siguiente programa, que llamaremos `fact2`,

```
fact := 1;
while (x != 0) do (
    fact := fact * x;
    x = x - 1;
)
```

sí calcula correctamente el factorial, pero no cumple la condición (4.1). Por tanto, la especificación (4.1) no es necesaria ni suficiente para garantizar que el programa calcula el factorial.

¿Cuál es la forma correcta de representar esta especificación? Una solución es la siguiente:

$$\forall a,\ \{x = a\}(\texttt{S})\{fact = a!\} \tag{4.2}$$

En esta expresión hemos indicado explícitamente el cuantificador universal para que quede claro que a es una variable lógica, y por tanto no puede aparecer en el cuerpo del programa. (Si el programa utilizase la variable a tendríamos que escoger una variable lógica diferente, como es natural.) Recomedamos al lector que compruebe que el programa `(x:=1;fact:=1)` no cumple esta especificación, pero los programas `fact1` y `fact2` sí la cumplen.

La conclusión que se saca de los ejemplos anteriores es que hay que utilizar variables lógicas cuando queremos que la poscondición haga referencia al valor que toma cierta variable en la precondición y el cuerpo del programa modifica el valor de esa variable. Así, en nuestro ejemplo, queremos que, después de ejectutar el programa S, la variable *fact* contenga el factorial del valor asignado inicialmente a x. Cuando el programa no modifica el valor de x (como era el caso del programa `fact1`), la especificación (4.1) no plantea problemas, porque el valor final de x es el mismo que el inicial. En cambio, cuando el programa modifica el valor de x (como era el caso del programa `fact2`), la postcondición $\{fact = x!\}$, que hace referencia al valor **final** de x, no nos sirve, y por eso debemos utilizar la especificación (4.2).

4.3 Semántica de los programas

4.3.1 Corrección total

Una terna de Hoare, $\{p\}(S)\{q\}$, puede considerarse como una proposición y por tanto es posible asignarle un valor de verdad. Decimos que una terna de Hoare es cierta si todo sistema que parte de un estado (cualquiera) que satisface p pasa a un estado que satisface q, y se representa así:[3]

$$\vDash_{\text{tot}} \{p\}(S)\{q\}$$

[3]En este capítulo estamos definiendo la semántica de modo un tanto informal, basada en el estado inicial y el estado final de una computación. Para un tratamiento riguroso, el lector puede consultar el libro de Francez [1992] o el de Apt y Olderog [1997]. La definición de semántica que ofrece el libro de Ben-Ari [2001] es diferente, y está basada en el concepto de *condición más débil* ("weakest precondition").

Esta propiedad, denominada *corrección total*, es muy importante en la práctica, pues la verificación del programa S consiste precisamente en demostrar que siempre que el sistema satisface ciertas condiciones (p) la ejecución de S hace que se satisfaga la condición que nos interesa (q). Las dos tareas más frecuentes que vamos encontrar en la práctica son:

Verificación: Tenemos las condiciones p y el programa S (codificado por nosotros mismos o por otro programador) y queremos demostrar que el programa es correcto (es decir, la ejecución del programa hace que se satisfaga la condición q).

Programación: Conocemos las condiciones p y q y queremos encontrar el programa S. En este caso, se trata de un problema de programación.

En este capítulo vamos a estudiar métodos formales de verificación. En la sección 4.7.2 mencionaremos brevemente la posibilidad de utilizar estos métodos para realizar a la vez la programación y la verificación.

Para algunas instrucciones sencillas es fácil ver que se satisface la corrección total. Por ejemplo,

$$\vDash_{\text{tot}} \{i = 4\}(\texttt{i:=i+1})\{i = 5\}$$

En la práctica, demostrar la corrección total directamente suele ser bastante complicado. En estos casos podemos resolver el problema dividiéndolo en dos subtareas: (1) demostrar que el programa, **si termina**, llega a q —es lo que se denomina *corrección parcial*— y (2) demostrar que el programa termina.[4] En la práctica, el método que se sigue es demostrar primero la corrección parcial y adaptar luego la demostración para probar la corrección total (véase la sec. 4.6).

4.3.2 Corrección parcial

La *corrección parcial* se expresa así:

$$\vDash_{\text{par}} \{p\}(S)\{q\}$$

y, como hemos dicho, significa que si un sistema que se encuentra en un estado que satisface la condición ejecuta el programa S, **si el programa termina**, llega al estado q.

De esta definición se deduce que un programa que no termina nunca siempre es *parcialmente correcto* (para toda p y toda q), pero nunca es *totalmente correcto*. Por ejemplo,

$$\vDash_{\text{par}} \{p\}(\texttt{while true do (i:=0)})\{q\}$$
$$\nvDash_{\text{tot}} \{p\}(\texttt{while true do (i:=0)})\{q\}$$

Otro ejemplo trivial es que para todo S se cumple que

$$\vDash_{\text{par}} \{\bot\}(S)\{q\} \tag{4.3}$$

Por tanto, todo programa S es *parcialmente correcto* para la precondición "$\bot$" cualquiera que sea la postcondición.[5]

Análogamente tenemos que

$$\vDash_{\text{par}} \{p\}(S)\{\top\} \tag{4.4}$$

[4] Obviamente, las únicas instrucciones que pueden hacer que el programa no termine son los bucles `while`. Por eso, para demostrar que un programa termina basta demostrar que todos sus bucles terminan.

[5] Naturalmente, este ejemplo sólo tiene interés didáctico, porque la precondición "$\bot$" excluye todos los estados. Dado que un ordenador real siempre va a encontrarse en algún estado, la propiedad (4.3) nunca puede aplicarse en la práctica.

porque la condición "$\top$" se satisface siempre, y de ahí se deduce que todo programa es *parcialmente correcto* para la prostcondición "$\top$" cualquiera que sea la precondición.[6]

En la próxima sección vamos a estudiar un sistema deductivo que nos permitirá demostrar la corrección parcial de programas que presentan interés real. En la sección 4.6 discutiremos cómo demostrar la terminación en el caso de programas con bucles.

4.4 El sistema deductivo de Hoare

El sistema deductivo de Hoare se basa en cinco axiomas, que sirven como reglas de deducción:

Asignación:

$$\vdash \{P(E)\}(\texttt{x:=}E)\{P(x)\} \tag{4.5}$$

Condicional:

$$\frac{\vdash \{p \wedge B\}(S_1)\{q\} \qquad \vdash \{p \wedge \neg B\}(S_2)\{q\}}{\vdash \{p\}(\texttt{if } B \texttt{ then (}S_1\texttt{) else (}S_2\texttt{)})\{q\}} \tag{4.6}$$

Bucle:

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do (}S\texttt{)})\{p \wedge \neg B\}} \tag{4.7}$$

Composición:

$$\frac{\vdash \{p\}(S_1)\{q\} \qquad \vdash \{q\}(S_2)\{r\}}{\vdash \{p\}(S_1\texttt{;}S_2)\{r\}} \tag{4.8}$$

Encadenamiento:

$$\frac{\vdash \{p \to p'\} \qquad \vdash \{p'\}(S)\{q'\} \qquad \vdash \{q' \to q\}}{\vdash \{p\}(S)\{q\}} \tag{4.9}$$

A estas reglas habría que añadir los axiomas propios del dominio. En el caso de nuestro microlenguaje el dominio es la aritmética entera, pues el único tipo de datos que admite son enteros (a parte de expresiones booleanas, naturalmente, que formarn parte de la lógica). Por ejemplo, un axioma del dominio puede ser "$\forall x,\ x = x$". Otro axioma puede ser "$\forall x, \forall y, \forall z,\ x = y \to x + z = y + z$". De hecho, más adelante veremos que en la verificación de programas se combinan dos sistemas deductivos, uno para razonar sobre las ternas de Hoare y otro para razonar sobre el dominio (la aritmética).

Vamos a explicar a continuación cada una de estas reglas. Pero antes debemos mencionar que los nombres varían mucho de un texto a otro. Por eso no es importante recordar los nombres de las reglas sino su significado y la forma en que se aplican.

4.4.1 Regla de asignación

La regla de asignación nos dice que si una condición, expresada en forma de predicado, se cumple para la expresión E entonces se cumple también para x después de haber asignado a la variable `x` el valor E (`x:=E`). Dicho de otra forma, si queremos demostrar que la variable x cumple el predicado P tenemos que demostrar que el valor (dado por la expresión E) que hemos asignado a x cumplía el predicado P, porque si no se cumplía $P(E)$, tampoco va a cumplir $P(x)$.

[6] De nuevo encontramos un ejemplo que sólo tiene interés didáctico, a pesar de que la propiedad (4.4) siempre es cierta: la razón es que esta propiedad sólo nos dice que, si el programa S termina, el sistema se va a encontrar *en algún estado*, lo cual es tanto como no decir nada.

Ejemplo 4.5 La regla de asignación nos permite deducir que

$$\vdash \{a = 0\}(\texttt{x:=a})\{x = 0\}$$

En este ejemplo, el predicado P es "igual a cero" y la expresión E (el valor asignado a la variable) es "a"; por eso $P(E)$ es "$a = 0$". El significado de la regla de asignación, en este ejemplo, es que si a valía inicialmente 0, la instrucción `x:=a` hace que x también valga 0.

En la práctica esta regla se aplica hacia atrás, de modo que si queremos demostrar que después de la asignación `x:=E` se cumple $P(x)$, intentaremos demostrar que $P(E)$ era cierto antes de la asignación. La forma de hacer esto consiste en tomar la postcondición y sustituir en ella x por E. Así, en el ejemplo anterior, debemos tomar la postcondición "$x = 0$", y sustituir x por el valor que le asigna la instrucción (`x:=a`), con lo cual obtenemos la precondición "$a = 0$".

Ejemplo 4.6 Si queremos ver qué precondición satisface esta terna

$$\{?\}(\texttt{x:=a+1})\{x = 0\}$$

tenemos que tomar la postcondición, "$x = 0$", y sustituir x por el valor que se le asigna, que es $a + 1$, con lo cual la precondición que obtenemos es "$a + 1 = 0$":

$$\{a + 1 = 0\}(\texttt{x:=a+1})\{x = 0\}$$

Ejemplo 4.7

$$\vdash \{x + y = 7\}(\texttt{z:=x+y})\{z = 7\}$$

Ejemplo 4.8

$$\vdash \{x + 1 = 4\}(\texttt{x:=x+1})\{x = 4\}$$

Como se ve en estos ejemplos, la precondición se obtiene tomando la postcondición y sustituyendo en ella la variable por el valor que se le asigna.

4.4.2 Regla del condicional

La regla del condicional nos dice que si tenemos que demostrar una terna de la forma

$$\{p\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}$$

podemos hacerlo en dos pasos: por un lado demostramos que, cuando partimos de la condición p y B es cierto, la ejecución de S_1 garantiza la condición q,

$$\{p \wedge B\}(S_1)\{q\}$$

y por otro lado, demostramos que, cuando partimos de la misma condición p y B es falso, la ejecución de S_2 también garantiza la condición q,

$$\{p \wedge \neg B\}(S_2)\{q\}$$

Ejemplo 4.9 Para demostrar que

$$\{x \neq 0\}(\texttt{if (x > 0) then (y:=x) else (y:=-x)})\{y > 0\}$$

basta demostrar que

$$\{x \neq 0 \wedge x > 0\}(\texttt{y:=x})\{y > 0\}$$

y que

$$\{x \neq 0 \wedge \neg(x > 0)\}(\texttt{y:=-x})\{y > 0\}$$

Naturalmente, al traducir las expresiones booleanas del programa al lenguaje de la lógica hay que recordar la equivalencia entre los operadores de programa y los operadores lógico-matemáticos. Por ejemplo, las expresiones booleanas `(x!=0)` y `(0=<x & x=<10)` se traducen, respectivamente, como $(x \neq 0)$ y $(0 \leq x \wedge x \leq 10)$.[7]

Como hemos visto, la regla del condicional es muy fácil de entender. Vamos a ver a continuación otra versión de esta regla que, aunque no es tan intuitiva ni tan fácil de recordar, resulta más cómoda de aplicar en la práctica.

4.4.3 Regla del condicional modificada

Queremos que el sistema, después de ejecutar la instrucción "`if` B `then (`S_1`) else (`S_2`)`" satisfaga la condición q.

$$\vdash \{p\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}$$

Supongamos que hemos encontrado dos condiciones, p_1 y p_2, tales que $\{p_1\}(S_1)\{q\}$ y $\{p_2\}(S_2)\{q\}$. Definimos p así

$$p = (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2) \tag{4.10}$$

Como $p \wedge B \rightarrow p_1$, por la regla de encadenamiento tenemos que

$$\{p \wedge B\}(S_1)\{q\}$$

Análogamente, como $p \wedge \neg B \rightarrow p_2$,

$$\{p \wedge \neg B\}(S_2)\{q\}$$

Introduciendo estos dos resultados en la regla del condicional (expresión (4.6)) tenemos que

$$\frac{\vdash \{p_1\}(S_1)\{q\} \qquad \vdash \{p_2\}(S_2)\{q\}}{\vdash \{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}} \tag{4.11}$$

Esta nueva versión de la regla del condicional nos dice que para asegurar que el sistema, después de ejecutar la instrucción "`if` B `then (`S_1`) else (`S_2`)`" satisfaga la postcondición q, es suficiente que satisfaga la precondición $(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$.

Esta nueva versión tiene la ventaja de que es más fácil buscar p_1 y p_2 por separado que tener que buscar directamente una condición p que satisfaga las dos condiciones, $\{p \wedge B\}(S_1)\{q\}$ y $\{p \wedge \neg B\}(S_2)\{q\}$. O visto de otra forma, el buscar p_1 y p_2 por separado es una forma de buscar p, de acuerdo con la ecuación (4.10).

En la sección 4.5.3 veremos cómo se aplica esta regla en la práctica.

[7] Tenga cuidado de no confundir "`x!=0`", que significa x es distinto de 0, con "$x! = 0$", que significa que el factorial de x es 0.

4.4.4 Regla del bucle

La regla del bucle está íntimamente ligada al concepto de invariante.

Definición 4.10 La condición p es un *invariante* para la instrucción "`while` B `do (`S`)`" si y sólo si se cumple que

$$\{p \wedge B\}(S)\{p\} \tag{4.12}$$

De esta definición se deduce que, si p es cierto antes de ejecutar la instrucción `while` también lo será después de haberla ejecutado. ¿Por qué? Supongamos que B es falsa. Entonces el cuerpo S no se ejecuta; por tanto, el estado del sistema no se modifica y la condición p sigue siendo cierta. Supongamos que B es cierta. Eso significa que el cuerpo S se ejecuta en un estado que satisface la condición $p \wedge B$, y entonces la propiedad (4.12) nos garantiza que después de ejecutar S el sistema va a seguir cumpliendo la condición p. Dicho de otro modo, no sabemos a priori cuántas veces se va a ejecutar el cuerpo S dentro de la instrucción `while`, pero sí sabemos que en cada una de las ejecuciones se va a mantener la propiedad p. Por eso p se denomina invariante. Esto es lo que nos permite concluir que

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do } (S))\{p\}}$$

Por otro lado, la instrucción `while` sólo termina cuando B es falsa. Uniendo estos dos resultados tenemos

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do } (S))\{p \wedge \neg B\}}$$

que es precisamente la regla del bucle.

Proposición 4.11 Para toda instrucción `while`, $\top$ es un invariante.

Demostración. La fórmula $\{\top \wedge B\}(S)\{\top\}$ es una tautología, porque la postcondición siempre es cierta. □

Naturalmente, en la práctica nos interesa encontrar invariantes no triviales. Veamos otros ejemplos de invariantes.

Ejemplo 4.12 La regla del bucle nos dice que para demostrar

$$\{x = 0\}(\texttt{while (y!=0) do (z:=3)})\{x = 0 \wedge \neg(y \neq 0)\}$$

basta demostrar que $\{x = 0 \wedge y \neq 0\}(\texttt{z:=3})\{x = 0\}$. (Esta fórmula se demuestra por la regla de asignación, pues en la postcondición "$x = 0$" no aparece z, y por eso al sustituir z por 3 la precondición que se obtiene es la misma, "$x = 0$".) El invariante es $x = 0$. □

En este ejemplo se da una paradoja: inicalmente no sabemos cuál es el valor de y y sin embargo concluimos que, después de ejecutar el bucle, se cumple que $\neg(y \neq 0)$, es decir, $y = 0$, a pesar de que el bucle no ha modificado el valor de y. ¿Cómo se explica esto? Debemos tener en cuenta dos situaciones: si inicialmente $y = 0$, la instrucción `while` no hace nada, y el programa termina, satisfaciendo la condición $\{x = 0 \wedge y = 0\}$. En cambio, si inicialmente $y \neq 0$, el programa entra en un bucle infinito. Por eso no hay contradicción al afirmar que "si el programa termina (algo que no ocurre cuando $y \neq 0$), el valor de y al salir del bucle es 0".

Como ya hemos mencionado, el sistema deductivo de Hoare sólo garantiza la corrección parcial: la afirmación $\vdash \{p\}(S)\{q\}$ es equivalente a $\vDash_{\text{par}} \{p\}(S)\{q\}$, que, como vimos anteriormente, significa que "si el sistema satisface inicialmente la condición p y ejecuta el programa S **y el programa termina**, entonces el sistema satisface la condición q".

En el ejemplo anterior era muy fácil encontrar un invariante: como el cuerpo de la instrucción `while`, que es `z:=3`, no modifica el valor de x ni de y, cualquier condición en que sólo aparezcan estas dos variables será un invariante.

Otro ejemplo similar es el siguiente:

Ejemplo 4.13

$$\vdash \{x < 5\}(\texttt{while (x==0) do (y:=1)})\{x < 5 \wedge \neg(x = 0)\}$$

En estos dos ejemplos, el cuerpo de la instrucción `while`, S, no modifica la condición, B, y por eso hay dos posibilidades: o bien la condición es cierta antes de ejecutar la instrucción `while`, con lo cual el programa entra en un bucle infinito, o bien la condición es falsa antes de ejecutar la instrucción, con lo cual la instrucción `while` no hace nada, y es como si no estuviera en el programa. Por eso en la práctica sólo nos interesan los casos en que el cuerpo, S, puede modificar la condición, B.

Ejemplo 4.14 Queremos demostrar que

$$\vdash \{x \geq 0\}(\texttt{while (x!=0) do (x:=x-1)})\{x = 0\}$$

Para ello, demostramos la expresión

$$\vdash \{x \geq 0 \wedge x \neq 0\}(\texttt{x:=x-1})\{x \geq 0\}$$

que indica que "$x \geq 0$" es un invariante para este bucle. Aplicando la regla del bucle, con las equivalencias $p =$ "$x \geq 0$", $B =$ "$x \neq 0$" y $S =$ "`x:=x-1`" se obtiene que

$$\vdash \{x \geq 0\}(\texttt{while (x!=0) do (x:=x-1)})\{x \geq 0 \wedge \neg(x \neq 0)\}$$

de donde se deduce la expresión que queríamos demostrar.

4.4.5 Regla de composición

La regla de composición es muy sencilla: si queremos demostrar $\{p\}(S_1\,;S_2)\{r\}$ tenemos que buscar una propiedad q tal que $\{p\}(S_1)\{q\}$ y $\{q\}(S_1\,;S_2)\{r\}$.

Ejemplo 4.15 Para demostrar que

$$\{x \geq 0\}(\texttt{y:=x+1;z:=2*y})\{z \geq 0\}$$

basta demostrar estas dos fórmulas:

$$\{x \geq 0\}(\texttt{y:=x+1})\{y \geq 0\}$$
$$\{y \geq 0\}(\texttt{z:=2*y})\{z \geq 0\}$$

o bien estas dos:

$$\{x \geq 0\}(\texttt{y:=x+1})\{y \geq 1\}$$
$$\{y \geq 1\}(\texttt{z:=2*y})\{z \geq 0\}$$

4.4.6 Regla de encadenamiento

La regla de encadenamiento también es muy sencilla y ya la hemos aplicado en algunos de los ejemplos anteriores. Tan sólo vamos a indicar que de la regla de encadenamiento y de la propiedad $p \rightarrow p$ se deducen estas dos reglas, que pueden considerarse como casos particulares de ella:

$$\frac{\vdash \{p \rightarrow p'\} \qquad \vdash \{p'\}(S)\{q\}}{\vdash \{p\}(S)\{q\}}$$

$$\frac{\vdash \{p\}(S)\{q'\} \qquad \vdash \{q' \rightarrow q\}}{\vdash \{p\}(S)\{q\}}$$

4.5 Verificación parcial de programas

4.5.1 Composición y encadenamiento

Supongamos que tenemos un programa S compuesto por una serie de instrucciones, $S = S_1; \ldots; S_n$. Una forma de expresar la verificación de este programa consiste en escribir una cadena de condiciones e instrucciones, de este modo,

$$\begin{array}{ll}
\quad \{p\} & \\
S_1 & \\
\quad \{p_1\} & \text{[Justificación-1]} \\
\vdots & \\
\quad \{p_{n-1}\} & \text{[Justificación-}(n-1)\text{]} \\
S_n & \\
\quad \{q\} & \text{[Justificación-}n\text{]}
\end{array}$$

(Definimos $p_0 = p$ y $p_n = q$.) Para cada eslabón $\vdash \{p_{i-1}\}(S_i)\{p_i\}$ se indica entre corchetes cuál es la regla que lo justifica. Al demostrar todos y cada uno de los eslabones, el programa S queda demostrado, por la regla de composición. Si S_i es una instrucción compuesta, hay que repetir el proceso anterior anidando las cadenas de demostración, y así sucesivamente, hasta llegar a instrucciones simples.

A veces puede resultar que la postcondición de S_i no coincida con la precondición que requiere S_{i+1}. En ese caso es posible insertar varias condiones entre S_i y S_{i+1},

$$\begin{array}{ll}
\quad \{p\} & \\
S_i & \\
\quad \{p_{i,1}\} & \text{[Justificación-}i\text{]} \\
\quad \{p_{i,2}\} & \text{[Justificación-}i\text{,1]} \\
\quad \vdots & \\
\quad \{p_{i,k}\} & \text{[Justificación-}i,k\text{]} \\
S_{i+1} &
\end{array}$$

de modo que la primera condición intermedia, $p_{i,1}$, es la postcondición de S_i, y la última condición intermedia, $p_{i,k}$, sirve de precondición para S_{i+1}. Para cada par de proposiciones consecutivas, la fórmula $\vdash p_{i,j} \rightarrow p_{i,j+1}$ se demuestra por alguno de los axiomas del dominio o por alguna de las propiedades de la lógica de predicados, que se indica entre corchetes como [Justificación-i,j]. La validez de la demostración viene garantizada por la regla de encadenamiento.

Observe que en el proceso de verificación estamos combinando dos sistemas deductivos. Por un lado, tenemos el sistema deductivo de Hoare, que nos permite justificar los eslabones del tipo $\vdash \{p_{i-1}\}(S_i)\{p_i\}$, y por otro lado el sistema deductivo propio del dominio, que nos permite justificar los eslabones del tipo $\vdash p_{i,j} \rightarrow p_{i,j+1}$. Como hemos dicho ya, en el caso de nuestro micro-lenguaje, el dominio es la aritmética de los números enteros.

Ejemplo 4.16 La demostración de $\vdash \{\top\}(\texttt{x:=2;y:=3*x+1})\{y = 7\}$ puede ser ésta:

$\{\top\}$
$\{2 = 2\}$ [$\forall x,\ x = x$]
`x:=2`
$\{x = 2\}$ [Regla de asignación]
$\{3x + 1 = 7\}$ [$3 * 2 + 1 = 7$]
`y:=3*x+1`
$\{y = 7\}$ [Regla de asignación]

Observe que la demostración de $\vdash \{2 = 2\}(\texttt{x:=2})\{x = 2\}$ y de $\vdash \{3x + 1 = 7\}(\texttt{y:=3*x+1})\{y = 7\}$ se basa en la regla de asignación, que pertenece al sistema deductivo de Hoare, mientras que $\vdash \top \rightarrow$ 2=2 y $\vdash (x = 2) \rightarrow (3x + 1 = 7)$ se demuestran mediante el sistema deductivo de la aritmética. □

Aunque resulta más intuitivo entender la demostración leyéndola en el sentido de ejecución del programa, es decir, de arriba a abajo (o hacia adelante, si se prefiere), la construcción de la demostración suele ser más sencilla en sentido inverso, es decir, desde la postcondición hasta la precondición (de abajo a arriba, o hacia atrás). La razón es que generalmente la regla de asignación y la del condicional son más fáciles de aplicar hacia atrás que hacia adelante, como vamos a ver a continuación.

4.5.2 Tratamiento de las asignaciones

La regla de asignación puede expresarse así:

$\{P(E)\}$
`x := E`
$\{P(x)\}$ [Regla de asignación]

Como acabamos de indicar, generalmente esta regla es más fácil de aplicar hacia atrás. Lo vemos volviendo al ejemplo anterior.

Ejemplo 4.17 Queremos demostrar que $\vdash \{\top\}(\texttt{x:=2;y:=3*x+1})\{y = 7\}$. Para ello escribimos la precondición, las instrucciones simples y la postcondición. Entre cada par de instrucciones insertamos la condición correspondiente que, como aún no conocemos, la hemos representado mediante una interrogación. Una interrogación entre corchetes indica que aún no hemos demostrado ese paso. (Si el lector lo prefiere, al construir sus demostraciones puede dejar un espacio en blanco en vez de escribir una interrogación; nosotros utilizamos la interrogación para que quede más claro cuándo hay todavía algún paso pendiente de demostrar.)

$\{\top\}$
`x := 2`
$\{?\}$ [?]
`y := 3*x+1`
$\{y = 7\}$ [?]

Si queremos construir la demostración de arriba a abajo, tenemos que buscar un predicado P que nos permita aplicar la regla de asignación $\{P(E)\}$`(x:=2)`$\{P(x)\}$. El problema es que, en nuestro esbozo de demostración, la precondición de `x:=2` es $\top$; si tomamos este predicado tenemos que $P(E) = P(2) = \top$, con lo que concluimos que $\vdash \{\top\}$`(x:=2)`$\{\top\}$, lo cual es cierto pero no nos sirve para nada. Para esta primera instrucción tampoco podemos buscar el predicado P a partir de su postcondición porque aún no la conocemos (por eso hemos escrito una interrogación).

Vamos a intentar verificar el programa anterior de abajo a arriba. Ahora sí es fácil aplicar la regla de asignación: basta tomar la postcondición de la última instrucción, $y = 7$, y en esta expresión sustituimos y por el valor asignado, con lo cual obtenemos la precondición de `y:=3*x+1`:

$\{\top\}$
`x := 2`
$\{3 * x + 1 = 7\}$ [?]
`y := 3*x+1`
$\{y = 7\}$ [Regla de asignación]

Siguiendo el mismo proceso para la instrucción `x := 2`, tomamos su postcondición y sustituimos x por el valor asignado: con lo cual llegamos a

$\{\top\}$
$\{3 * 2 + 1 = 7\}$ [?]
`x := 2`
$\{3 * x + 1 = 7\}$ [Regla de asignación]
`y := 3*x+1`
$\{y = 7\}$ [Regla de asignación]

Para concluir la demostración, basta probar que $\top \rightarrow 3 * 2 + 1 = 7$, que es equivalente a probar que $3 * 2 + 1 = 7$ (habría que demostrarlo por la aritmética de números enteros). □

4.5.3 Tratamiento de las instrucciones condicionales

Para tratar las instrucciones `if-then-else` aplicamos la regla del condicional modificada (expresión (4.11)), que se traduce en:

$\{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}$
`if` B `then (`
$\{p_1\}$
S_1
$\{q\}$ [Justificación-1]
`) else (`
$\{p_2\}$
S_2
$\{q\}$ [Justificación-2]
`)`
$\{q\}$ [Regla del condicional]

Ejemplo 4.18 Para demostrar que

$$\{x \neq 0\}\texttt{(if (x>0) then (y:=x) else (y:=-x))}\{y > 0\}$$

escribimos este esbozo de demostración:

$\{x \neq 0\}$
$\{?\}$ [?]
`if (x>0) then (`
$\{p_1?\}$
`y:=x`
$\{y > 0\}$ [?]
`) else (`
$\{p_2?\}$
`y:=-x`
$\{y > 0\}$ [?]
`)`
$\{y > 0\}$ [?]

Las condiciones p_1 y p_2 se obtienen por la regla de asignación:

$\{x \neq 0\}$
$\{?\}$ [?]
`if (x>0) then (`
$\{x > 0\}$
`y:=x`
$\{y > 0\}$ [Regla de asignación]
`) else (`
$\{(-x) > 0\}$
`y:=-x`
$\{y > 0\}$ [Regla de asignación]
`)`
$\{y > 0\}$ [?]

La precondición de la instrucción condicional es $(B \to p_1) \wedge (\neg B \to p_2)$, de modo que

$\{x \neq 0\}$
$\{(x > 0 \to x > 0) \wedge (\neg(x > 0) \to (-x) > 0\}$ [?]
`if (x>0) then (`
$\{x > 0\}$
`y:=x`
$\{y > 0\}$ [Regla de asignación]
`) else (`
$\{(-x) > 0\}$
`y:=-x`
$\{y > 0\}$ [Regla de asignación]
`)`
$\{y > 0\}$ [Regla de asignación]

Dejamos como ejercicio para el lector demostrar que $x \neq 0 \to (x > 0 \to x > 0) \wedge (\neg(x > 0) \to (-x) > 0$, con lo cual se completa la verificación del programa. (En este caso, la verificación parcial es una verificación total, porque el programa no contiene bucles.)

4.5.4 Tratamiento de los bucles

Hemos visto anteriormente que el tratamiento de las instrucciones de asignación y condicionales es bastante fácil, pues se limita a la aplicación de un algoritmo. Sin embargo, el tratamiento de los bucles es mucho más complicado, pues requiere encontrar invariantes, una tarea para la cual no existe un algoritmo, sino tan sólo algunas heurísticas.[8]

Cuando en la demostración de un programa nos encontramos con una situación como ésta,

$\{?\}$
`while` B `do (`S`)`
$\{q\}$ [?]

tenemos que encontrar un invariante que nos permita escribir

$\{p\}$
`while` B `do (`
$\{p \wedge B\}$
S
$\{p\}$ [Justificación (de que p es un invariante)]
`)`
$\{p \wedge \neg B\}$ [Regla del bucle]
$\{q\}$ $[p \wedge \neg B \rightarrow q]$

Para ello, el invariante p debe satisfacer tres condiciones:

1. $\vdash \{p \wedge B\}(S)\{p\}$ (es la definición de invariante);
2. $\vdash p \wedge \neg B \rightarrow q$ (para poder obtener la postcondición del bucle);
3. Hace falta que p pueda deducirse a partir de la postcondición de la instrucción anterior al bucle (para poder continuar luego la demostración hacia arriba).

Ejemplo 4.19 Dado el programa `fact1` introducido en el ejemplo 4.1, queremos demostrar que se cumple la condición (4.1).[9] Para ello buscamos un invariante p que cumpla que $\vdash \{p \wedge i \neq x\}$(`i:=i+1;fact:=fact*i;`)$\{p\}$ y $\vdash p \wedge \neg(i \neq x) \rightarrow fact = x!$.

Una heurística que recomiendan algunos autores para encontran invariantes es construir una tabla que refleje el valor que toman las variables del programa en cada ejecución del bucle, y tratar de ver qué propiedad(es) se cumplen para cada una de las filas. Para este ejemplo, tomando $x = 6$ (un valor escogido arbitrariamente, con la única condición de que no sea demasiado grande ni demasiado pequeño), obtenemos la tabla 4.1. En todas las columnas de esta tabla se cumple que $fact = i!$. Vamos a comprobar si esta condición es un invariante:

[8] Como ya sabe el lector, en inteligencia artificial se denomina *heurística* a una regla que ayuda a buscar una solución, aunque generalmente la aplicación de una heurística ni garantiza que se encuentre una solución, ni garantiza que la solución encontrada sea óptima.

Por eso podríamos decir que el tratamiento de las condiciones de asignación y condicionales es una técnica, mientras que el tratamiento de los bucles es un arte, que sólo se aprende con la práctica.

[9] Recordemos que la especificación (4.2) era más correcta, pero la especificación (4.1) también era válida cuando el programa no modifica el valor de la variable `x`, y en nuestro caso es más sencilla de aplicar. Dejamos como ejercicio para el lector comprobar que el programa `fact1` cumple la especificación (4.2).

iteración	x	i	$fact$	B
1^a	6	0	1	$\top$
2^a	6	1	1	$\top$
3^a	6	2	2	$\top$
4^a	6	3	6	$\top$
5^a	6	4	24	$\top$
6^a	6	5	120	$\top$
7^a	6	6	720	$\bot$

Tabla 4.1: Tabla para probar el bucle de la función `fact1`..

$\{fact = i! \wedge i \neq x\}$
```
i := i + 1;
```
$\{?\}$ [?]
```
fact := fact * i;
```
$\{fact = i!\}$ [?]

Aplicando dos veces la regla de asignación llegamos a

$\{fact = i! \wedge i \neq x\}$
$\{fact = i!\}$ $[p \wedge q \rightarrow p]$
$\{fact * (i+1) = (i+1)!\}$ $[\forall a,\ (a+1)! = (a+1) * a!]$
```
i := i + 1;
```
$\{fact * i = i!\}$ [Regla de asignación]
```
fact := fact * i;
```
$\{fact = i!\}$ [Regla de asignación]

lo cual demuestra que $fact = i!$ es un invariante. Podemos integrar este resultado en la verificación del programa completo, que queda así:

$\{x \geq 0\}$
$\{1 = 0!\}$ [Axioma (definición de factorial)]
```
fact := 1;
```
[Regla de asignación]
$\{fact = 0!\}$
```
i := 0;
```
$\{fact = i!\}$ [Regla de asignación]
```
while (i != x) do (
```
$\{fact = i! \wedge i \neq x\}$
$\{fact = i!\}$ $[p \wedge q \rightarrow p]$
$\{fact * (i+1) = (i+1)!\}$ $[\forall a,\ (a+1)! = (a+1) * a!]$
```
    i = i + 1;
```
$\{fact * i = i!\}$ [Regla de asignación]
```
    fact := fact * i;
```
$\{fact = i!\}$ [Regla de asignación]
```
)
```
$\{fact = i! \wedge \neg(i \neq x)\}$ [Regla del bucle]
$\{fact = i! \wedge i = x\}$ $[\forall a, \forall b,\ a \neq b \leftrightarrow \neg(a = b)]$
$\{fact = x!\}$ [Regla de sustitución (propiedad de la lógica)]

Observe que la precondición $\{x \geq 0\}$ no nos ha hecho falta para demostrar $\{0! = 1\}$, porque éste es un axioma del sistema. Por tanto, también habríamos podido demostrar que

$$\vdash \{\top\}(\texttt{fact1})\{fact = x!\}$$

Sin embargo, cuando $x < 0$, el programa `fact1` entra en un bucle infinito. Por eso la precondición $\{x \geq 0\}$ es necesaria para garantizar la correción total, mientras que, de acuerdo con la expresión que acabamos de escribir, no es necesaria ninguna precondición para garantizar la correción parcial.

Ejercicio 4.20 Demostrar que

$$\vdash \forall a,\ \{x = a\}(\texttt{fact1})\{fact = a!\}$$
$$\vdash \forall a,\ \{x = a\}(\texttt{fact2})\{fact = a!\}$$

4.6 Verificación total de programas

El sistema deductivo expuesto en la sección 4.4 sólo garantiza la corrección parcial. Para demostrar la corrección total es necesario sustituir la regla del bucle anterior por esta otra:[10]

Regla del bucle (para corrección total):

$$\frac{\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{p \wedge E \geq 0 \wedge E < a\}}{\vdash \{p \wedge E \geq 0\}(\texttt{while } B \texttt{ do } (S))\{p \wedge \neg B\}}$$

En esta regla p sigue siendo un invariante, porque si se cumple antes de que se ejecute el bucle, se cumple también después. Si hemos demostrado ya la corrección parcial para este bucle, podemos utilizar aquí el mismo invariante.

La expresión E se denomina *variante*,[11] porque si es igual a a (un número entero) antes de que se ejecute el bucle, será estrictamente menor que a cuando el bucle se ha ejecutado, y continuará decreciendo cada vez que se ejecuta el cuerpo del bucle. Como esta expresión no puede decrecer indefinidamente (porque siempre es mayor que 0), el bucle debe terminar.

De este modo, el nuevo sistema deductivo garantiza tanto la corrección parcial como la terminación de los bucles, con lo que se demuestra la corrección total.

Ejemplo 4.21 Dado el programa `fact1` introducido en el ejemplo 4.1, queremos demostrar que se cumple la condición (4.1) en el nuevo sistema deductivo (que garantiza la corrección total). Recordemos que el bucle que aparece en este programa es

```
while (i!=x) do (i=i+1; fact:=fact*i)
```

y que en la verificación parcial habíamos tomado como invariante $fact = i!$. Por tanto, aunque todavía no hemos encontrado E, sabemos que se cumple que

$$\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{p\}$$

[10] Desde un punto de vista conceptual, no era necesario definir primero el sistema deductivo de verificación parcial, sino que podríamos haber definido directamente el de verificación total. Sin embargo, por motivos pedagógicos nos ha parecido mejor abordar el problema en dos pasos, discutiendo primero la verificación parcial y viendo luego cómo se debe modificar la regla del bucle para obtener la verificación total.

[11] Observe que el invariante es *una proposición* (por ejemplo, $fact = i!$), y por eso lo hemos representado por p, mientras que el variante es *una expresión numérica* (por ejemplo, $x - i$), y por eso lo representamos por E. Observe también que, en general, tanto p como E hacen referencia al valor de las variables del programa.

es decir,

$$\vdash \{fact = i! \wedge i \neq x \wedge E \geq 0 \wedge E = a\}(\texttt{i:=i+1;fact:=fact*i})\{fact = i!\}$$

(lo hemos demostrado en el ejemplo 4.19).

Vamos a buscar ahora una expresión E (un variante) que cumpla que

$$\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{E \geq 0 \wedge E < a\}$$

Tomamos la expresión $x - i$, pues cumple que

$$\vdash \{i \neq x \wedge x - i \geq 0 \wedge x - i = a\}(\texttt{i:=i+1;fact:=fact*i})\{x - i \geq 0 \wedge x - i < a\}$$

(Invitamos al lector a que lo demuestre formalmente.) Esto implica que si antes de ejecutar el cuerpo del bucle se cumple $0 \leq x - i = a$, después de su ejecución se cumple $0 \leq x - i < a$. Como la expresión E siempre decrece al menos una unidad en cada ejecución del cuerpo del bucle (en este ejemplo $x - i$ decrece exactamente una unidad) y siempre es mayor o igual que 0, el bucle debe terminar. □

Ejercicio 4.22 Demostrar la corrección total de `fact2` dada la especificación (4.2). □

Como en el caso de los invariantes, encontrar un variante puede ser complicado, ya que no hay reglas algorítmicas, sino sólo algunos consejos heurísticos. Por ejemplo, se recomienda la construcción de una tabla de ejecución, como la del ejemplo 4.19 (tabla 4.1), que puede ayudar a encontrar tanto el invariante como el variante.

4.7 Comentarios adicionales

4.7.1 Consistencia y completitud

El sistema deductivo que hemos definido en la sección 4.4 es **consistente** para la verificación parcial, lo cual significa que toda terna deducida mediante la aplicación de sus reglas es parcialmente correcta:

$$\vdash \{p\}(S)\{q\} \text{ implica que } \vDash_{\text{par}} \{p\}(S)\{q\}$$

Recordemos que $\vDash_{\text{par}} \{p\}(S)\{q\}$ que significa que "si el sistema (el ordenador) se encuentra inicialmente en alguno de los estados representados por la precondición p y ejecuta el programa S y el programa termina, el sistema se encontrará en alguno de los estados representados por la postcondición q" (cf. sec. 4.3.2).

Análogamente, el sistema deductivo que hemos definido en la sección 4.6 es **consistente** para la verificación total:

$$\vdash \{p\}(S)\{q\} \text{ implica que } \vDash_{\text{tot}} \{p\}(S)\{q\}$$

Por tanto, la consistencia significa que un programa verificado mediante este sistema deductivo satisface realmente la especificación $\{p\}(S)\{q\}$.

Para demostrar la consistencia de ambos sistemas deductivos deberíamos establecer primero una semántica formal de nuestro micro-lenguaje de programación (cómo y cuándo se pasa de un estado a otro al ejecutar cada una de las instrucciones del programa), algo que no hemos hecho. Sin embargo, el conocimiento que tiene el lector sobre cómo funciona cada una de esas instrucciones en un lenguaje de programación y las explicaciones que hemos dado para justificar cada regla hacen verosímil la afirmación de que el sistema es consistente. El lector interesado en la demostración formal de la consistencia puede consultar el libro de Francez [1992] o el de Apt y Olderog [1997].

A su vez, cada uno de estos sistemas es **completo en sentido relativo**. La completitud es la propiedad recíproca de la consistencia, y significa que toda terna semánticamente correcta puede ser obtenida mediante el sistema deductivo correspondiente: si se cumple $\vDash_{\text{tot}} \{p\}(S)\{q\}$ entonces esta terna puede ser demostrada (el programa puede ser verificado) por el sistema deductivo de verificación total:

$$\vDash_{\text{tot}} \{p\}(S)\{q\} \text{ implica que } \vdash \{p\}(S)\{q\}$$

La propiedad de completitud para el sistema deductivo de verificación parcial es análoga.

La expresión "completo **en sentido relativo**" significa que completo si consideramos cada fórmula del dominio como un axioma (recordemos que en el caso de nuestro micro-lenguaje el dominio es la aritmética de números enteros). En la práctica es imposible incluir todas las fórmulas del dominio dentro del sistema deductivo, porque son infinitas. Por eso nos interesa que las fórmulas del dominio puedan deducirse a partir de un número finito de axiomas. Sin embargo, no se puede construir un sistema deductivo completo para la aritmética de números enteros. Dado que el sistema deductivo de verificación que hemos tratado en este capítulo (en sus dos versiones, parcial y total), además de tener como axiomas las cinco reglas propias de la verificación, incluye también los axiomas del sistema deductivo del dominio, si éste no es completo tampoco puede serlo aquél. Por eso se dice que el sistema deductivo de verificación de programas **no es completo en sentido absoluto**.

4.7.2 Otras cuestiones

El micro-lenguaje que hemos estudiado en este capítulo está limitado sobre todo porque no utiliza arrays ni procedimientos, dos recursos imprescindibles para desarrollar programas de cierta envergadura. Estos aspectos se estudian en el libro de Francez [1992].

Según los métodos presentados en este capítulo, antes de verificar un programa es necesario haberlo construido. Sin embargo, es posible integrar ambas tareas en una sola, si se escribe el programa mediante bloques que se enlazan y expanden; en cada uno de los pasos se verifica el código generado, hasta llegar a un programa cuya corrección está garantizada. Puede encontrarse más información sobre este método en el libro de Apt y Olderog [1997].

Bibliografía complementaria

La referencia histórica básica para este tema es [Hoare, 1969]. Los libros de de Huth y Ryan y de Ben-Ari que hemos mencionado en la Bibliografía Recomendada *(pág. 3) explican este tema, aunque dejan algunos cabos sueltos (especialmente el libro de Ben-Ari). Un tratamiento mucho más completo y riguroso se encuentra en dos libros excelentes: [Apt y Olderog, 1997] y [Francez, 1992].*

Por otro lado, los recursos bibliográficos e informáticos, disponibles en Internet, que hemos comentado en la sección Motivación para los alumnos de Ingeniería Informática *(pág. 2) pueden ser útiles para hacer más interesante el estudio de este tema.*

Actividades y evaluación

Los ejercicios de evaluación más importantes de este tema consisten, naturalmente, en la verificación de pequeños programas secuenciales mediante la lógica de Hoare; por ejemplo, los ejercicios 7 a 10 del capítulo 9 del libro de texto [Ben-Ari, 2001, pág. 220], o los ejercicios que aparecen al final de cada sección en [Huth y Ryan, 2000, cap. 4].

En el grupo de tutorización telemática se encuentra la resolución detallada de algunos ejemplos sencillos (planteados en exámenes previos).

Parte III

LÓGICA MODAL

Capítulo 5

FUNDAMENTOS DE LÓGICA MODAL

Resumen

Este capítulo se podía haber titulado Lógicas modales proposicionales. *Sintácticamente, todos estos lenguajes se construyen a partir del proposicional, añadiendo uno o más pares de operadores modales. Resultan, en todo caso, sintaxis más sencillas que la de los lenguajes de Primer Orden.*

La semántica habitual de estos sistemas requiere una estructura relacional: un universo y relaciones sobre el mismo. En el caso más simple, la lógica modal básica, basta un universo y una relación binaria. Es decir, intuitivamente, para evaluar la satisfacción de una fórmula se requiere interpretarla sobre un grafo dirigido. Más precisamente, cada fórmula se puede evaluar sobre cada nodo del grafo, satisfaciéndose quizá en unos y no en otros. Esta evaluación local *es lo que caracteriza a este tipo lógicas.*

Las lógicas modales son una insustituible herramienta para analizar estructuras relacionales. Y en computación éstas son las estructuras sobre las que se modela prácticamente toda la actividad del campo (sistemas de transiciones etiquetadas, autómatas, redes, agentes, etc.). La elección de los operadores modales adecuados, siempre con un enfoque semántico común, permite proponer sistemas lógicos específicos para ciertas actividades.

Desde el punto de vista formal, es interesante observar cómo las fórmulas modales pueden traducirse sistemáticamente a fórmulas de Primer Orden. Y cómo definen, con una sintaxis sencilla y una evaluación local, propiedades globales de la estructura.

Objetivos

El primer escalón lo constituye la lógica modal básica: su sintaxis y su semántica. A partir de estos conceptos se pueden ampliar nuestros horizontes considerando sistemas polimodales: su sintaxis, semántica y propiedades.

Desde el punto de vista aplicado, un mismo operador modal abstracto se puede leer, se puede interpretar, de unas formas u otras dependiendo del dominio. O se pueden diseñar sistemas lógicos adecuados al problema.

El paso siguiente puede ser el estudio formal de los diversos sistemas deductivos de estas lógicas: de sus propiedades comunes y de las particuliaridades de ciertas familias.

Más allá, pero no menos interesante, resulta el estudio de la reinterpretación de estas fórmulas como sentencias en Lógica de Predicados: qué puede expresarse de forma modal, como fragmento de la Lógica de Predicados, y qué ventajas computacionales aporta su expresión en estos lenguajes.

Metodología

Le recomendamos que se centre inicialmente en la compresión semántica de la lógica modal básica: se puede conseguir intuitiva y gráficamente. Y todos los resultados serán fácilmente exportables a otros sistemas lógicos.

A partir de este punto, continúe con la perspectiva semántica (no se preocupe de los sistemas deductivos) ampliando este lenguaje básico con otros operadores. Reflexiones sobre las diversas lecturas propuestas.

Sólo después de estos pasos le recomendamos que aborde las diversas opciones de sistemas deductivos y opcionalmente su implementación.

En el estado actual de estas notas, dispone de material introductorio sólo para cubrir la primera parte de este camino.

5.1 Perspectiva

Esta sección anticipa los principales conceptos abordados. Se pretende ofrecer una perspectiva en un tono coloquial. Las definiciones precisas se posponen a las secciones siguientes.

5.1.1 Estructuras

Estructuras relacionales

Observe la figura (fig.5.1.1). Es la representación gráfica de un conjunto y de una relación R entre sus elementos. En este ejemplo,

- el elemento 5 está relacionado con el 1: $R(5,1)$,
- pero el 1 no lo está con el 5;
- sólo el elemento 3 está relacionado consigo mismo: $R(3,3)$.

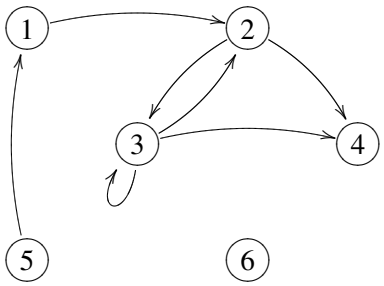


Figura 5.1: Estructura relacional

A este tipo de construcciones matemáticas se les denomina *estructuras relacionales*: un conjunto no vacío y relaciones sobre el mismo. También se les denominará *marcos*.

Modelos

Considere de nuevo la figura (fig.5.1.1), ahora como representación de una imprecisa *'relación entre escenarios de un videojuego'*. Pueden surgir, de inmediato, algunas objeciones razonables: ¿no debería ser reflexiva?, ¿qué sentido tienen los escenarios aislados?, ¿no deberían considerarse transiciones

etiquetadas?. Estas y otras cuestiones dependen del tipo de relación (o relaciones) entre escenarios que se pretende modelizar.

Las estructuras relacionales se utilizan para modelizar sistemas, situaciones, procesos. En general, una modelización adecuada requiere que la estructura relacional verifique ciertas *propiedades formales*.

Los lenguajes modales permitarán, por un lado, expresar lo que ocurre en cada nodo y por otro, formalizar las particularidades de la relación entre nodos

Proposiciones en cada nodo

Volvamos al ejemplo del videojuego, sobre el grafo propuesto. El comportamiento de nuestro 'héroe virtual', *en cada escenario*, puede venir descrito por un conjunto de proposiciones: $\{p,q,r,...\}$. Resulta entonces un grafo como el de la figura (fig.5.2), donde cada nodo se etiqueta con las letras proposicionales localmente verdaderas en el mismo.

Observe que la proposición p puede ser verdadera en unos nodos y falsa en otros, pero en todos ellos describe el mismo comportamiento, enuncia la misma proposición.

A este proceso de etiquetado se le denomina *asignación*. Un marco adecuado con una asignación es un *modelo*.

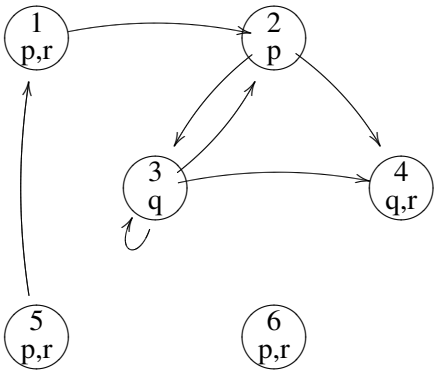


Figura 5.2: Un modelo: marco y asignación

5.1.2 Fórmulas

Razonamiento local proposicional

Sitúese en cualquiera de estos escenarios. En el 1, por ejemplo, puesto que son ciertas p y r, admitiremos que debe ser verdadera la proposición $(p \wedge r)$. No se satisface esta conjunción en el 2, donde sólo es verdadera p. Es decir, en cada nodo 'razonaremos' correctamente por igual, dentro de la lógica de proposiciones; pero los valores de las proposiciones atómica ('los detalles dentro de cada escenario') varían de nodo en nodo.

Es decir, (de momento) las fórmulas en cada nodo sólo se refieren a él mismo: son mundos cerrados e inconexos. Ahora bien, supongamos que una decisión de nuestro ciberprotagonista en un escenario dependa de su conocimiento del estado de otros. Veamos cómo se formalizan este tipo de expresiones.

Razonamiento local modal

La inspección de otros escenarios se simboliza mediante el uso de operadores modales: $\Diamond p$, $\Diamond(q \wedge r)$, $(\Diamond q) \wedge (\Diamond r)$ son ejemplos de fórmulas que incluyen un operador modal $\Diamond$.

Como se parte del lenguaje proposicional, es preciso aumentar su sintaxis con reglas que delimiten dónde puede insertarse este nuevo símbolo.

La interpretación de $\Diamond p$ se produce *localmente, en cada nodo, previa inspección de otros nodos determinados*. Permítanos explicarlo sobre el ejemplo de la figura (fig.5.2):

- en el escenario 3, $\Diamond p$ es verdadera porque '*existe al menos un nodo relacionado* (el 2) donde p es verdadera';
- sin embargo, en el escenario 2, $\Diamond p$ no es verdadera porque no es cierto que *exista* algún nodo relacionado con 2 donde es verdadera p (observe que 2 no está relacionado consigo mismo).

Observe cómo una misma fórmula (p.ej. $\Diamond p$) es verdadera en un nodo y falsa en otros: dependerá de dos factores. Primero, de los escenarios que sean visibles (estén relacionados) desde el nodo donde se evalúa la fórmula. Segundo, del estado de esos escenarios (del valor de las variables proposicionales).

5.1.3 Modelos adecuados

Lecturas de los operadores modales

Un operador como $\Diamond$ permite representar situaciones, sistemas o conceptos diversos. Uno de ellos es el concepto de posibilidad: *una proposición como, por ejemplo,* $(p \wedge q)$ es posible *en el mundo w si* $\Diamond(p \wedge q)$ *es verdadera en w, es decir, si en algún mundo accesible desde w se verifica* $(p \wedge q)$

Es decir, "nuestro 'héroe virtual' considera, en un nodo, que es posible reunir la llave y el cofre si es capaz de ver un escenario accesible desde el mismo donde están ambos".

Sólo una ligera objeción: con la semántica propuesta para $\Diamond$, la proposición p puede ser verdadera en el escenario actual sin que lo sea $\Diamond p$. Es decir, puede ser localmente verdadero algo sin que se acepte que 'es posible ese algo'. Formalmente, basta considerar un escenario que no se relaciona consigo mismo.

El problema planteado se soluciona admitiendo que, efectivamente, $\Diamond$ representa adecuadamente el concepto de posibilidad *pero sólo en marcos reflexivos*. Otras aplicaciones del operador $\Diamond$, como representante formal de otros conceptos, pueden requerir nuevas restricciones sobre los marcos.

Caracterización modal de relaciones

Se acaba de considerar que la lectura de $\Diamond$ como 'posibilidad' sólo es adecuada sobre marcos reflexivos.

Por otro lado, independientemente de que lo que represente (de su lectura), $\Diamond$ tiene una propiedad formal contrastable: la fórmula $p \rightarrow (\Diamond p)$ es verdadera en un mundo si y sólo si está relacionado consigo mismo. Es decir, esta fórmula es verdadera *en todo mundo de un marco* si y sólo si éste es reflexivo.

Así, si se diseña una teoría modal de la posibilidad, ya no es necesario adjuntar la advertencia, externa e informal, 'utilícese sólo sobre marcos reflexivos'. Basta añadir la fórmula $p \rightarrow (\Diamond p)$ como uno de los axiomas de la teoría. Se produce el mismo efecto de restricción sobre los marcos adecuados, de forma local, interna y expresada en el mismo lenguaje modal.

Se pueden caracterizar otras muchas propiedades de las relaciones mediante fórmulas modales. De ello se ocupa la Teoría de la Correspondencia.

Aplicaciones

Existe una gran variedad de lenguajes modales, con más o menos operadores, cuya lectura pretende modelizar un concepto o un sistema: relaciones entre tiempos, interacción de agentes, protocolos de comunicación ...

Todos ellos se basan en las mismas ideas sintácticas y semánticas: las que se pueden encontrar en la lógica modal básica, objeto del capítulo siguiente. Estos formalismos permiten disponer de un lenguaje para expresar qué ocurre en cada nodo (en cada tiempo, en cada estado mental de un agente) y qué relaciones son admisibles en esa lectura. Además, los sistemas deductivos asociados a estos lenguajes permiten obtener conclusiones, localmente, dependientes del contexto facilitado por los estados accesibles.

5.2 Estructuras relacionales

Definición 5.1 (Estructuras relacionales) Una estructura relacional es una tupla $\langle W, R_1, \ldots, R_k \rangle$ donde:

- W es un conjunto no vacío, y
- $R_1, \ldots, R_k$ son relaciones sobre W.

Existe al menos una relación en la estructura.

Las relaciones definidas sobre W pueden ser de distinto tipo: monarias (predicados Pw), binarias (Rw_iw_j), ternarias ($R'w_iw_jw_k$), etc. Así, una estructura puede contener, por ejemplo, dos relaciones monarias, tres binarias y una ternaria, siempre sobre el mismo conjunto W.

5.2.1 Sistemas de transiciones etiquetadas

Entre las posibles estructuras relacionales, se prestará especial interés a las que *sólo* constan de relaciones *binarias* (una o más). Se pueden ver dos ejemplos en la figura (fig.5.3).

La figura (fig.5.3a) consta de una única relación binaria. En (fig.5.3b) se han superpuesto, en la misma representación gráfica, dos relaciones binarias sobre el mismo conjunto. Se puede generalizar la representación, etiquetando como v los arcos de cada nueva relación binaria R_v.

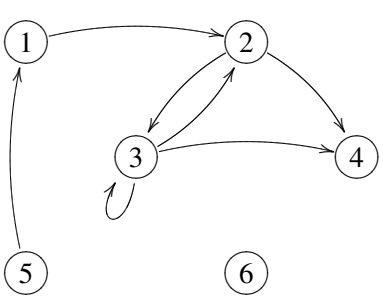
El estudiante de Informática habrá encontrado multitud de representaciones como las del ejemplo, donde las etiquetas se suelen interpretar como entradas externas que producen una transición desde un nodo a otro. Por eso, genéricamente, se denominan *Sistemas de transiciones etiquetadas* a estas estructuras relacionales.

Definición 5.2 (Sistemas de transiciones etiquetadas) Un sistema de transiciones etiquetadas es una estructura relacional donde cada relación R_k es una relación binaria:

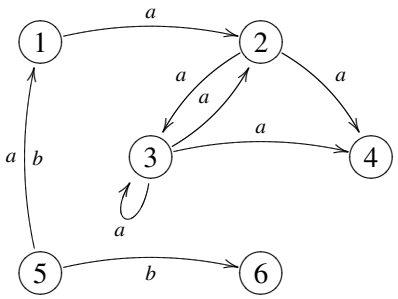
$$R_k \subset W \times W \qquad \text{para toda } k \in L, \text{conjunto de etiquetas}$$

Otro ejemplo, entre muchos, de sistemas de transiciones son las redes semánticas utilizadas para representación del conocimiento. Aquí, una de las etiquetas de los arcos puede ser *'encima-de'*, si se consideran relaciones espaciales entre los objetos representados por los nodos. Observe que, a pesar de su denominación, no siempre una etiqueta se interpretará como una transición entre estados.

Los sistemas de transiciones etiquetadas *con una única relación* también están ubicuamente presentes en el análisis y el desarrollo informático. Un ejemplo son las estructuras de árbol de un directorio, definidas tras imponer ciertas restricciones formales a la relación entre nodos.



(a) Relación R_a



(b) Relaciones R_a y R_b

	1	2	3	4	5	6
1		a				
2			a	a		
3		a	a	a		
4						
5	a/b					b
6						

(c) tabla con ambas relaciones

Figura 5.3: Relaciones binarias

5.2.2 Propiedades de una relación binaria

Obviamente, las propiedades impuestas sobre las relaciones definirán unas u otras familias de estructuras, más o menos adecuadas para modelizar cierta aplicación en estudio. Las propiedades de estas relaciones binarias se suelen describir formalmente en lógica de primer orden. A lo largo de este capítulo se presentará la lógica modal *como un lenguaje alternativo sobre el que describir y analizar estructuras relacionales*.

Revisemos algunas de las propiedades formales que puede verificar (o no) una relación R. La tabla (tabl.5.2.2) reune algunas de estas propiedades, expresadas como sentencias de primer orden.

Reflexiva	$\forall x Rxx$
Simétrica	$\forall x \forall y (Rxy \rightarrow Ryx)$
Transitiva	$\forall x \forall y \forall z (Rxy \wedge Ryz \rightarrow Rxz)$
Euclidea	$\forall x \forall y \forall z (Rxy \wedge Rxz \rightarrow Ryz)$
Lineal	$\forall x \forall y \forall z (Rxy \wedge Rxz \rightarrow (Ryz \vee y = z \vee Rzy))$
Funcional	$\forall x \exists y (Rxy \wedge \forall z (Rxz \rightarrow y = z))$
Determinista	$\forall x \forall y \forall z (Rxy \wedge Rxz \rightarrow y = z)$
Serial	$\forall x \exists y Rxy$
Total	$\forall x \forall y (Rxy \vee Ryx)$
Densa	$\forall x \forall y (Rxy \rightarrow \exists z (Rxz \wedge Rzy))$

Tabla 5.1: Propiedades de una relación binaria, expresadas en lógica de primer orden

Para fijar ideas, considere el grafo de una única relación R como el de la figura (fig.5.3a) o su expresión como tabla, como matriz, en (fig.5.3c). Los párrafos siguiente pretende facilitar la compresión de las sentencias de primer orden utilizadas.

Reflexividad Una relación es reflexiva si todo elemento está relacionado consigo mismo. Gráficamente, cada nodo del grafo tiene un arco hacia sí mismo. En la tabla de la relación, toda las celdas de la diagonal pertenecen a la relación.

Una relación es irreflexiva si ningún elemento está relacionado consigo mismo: $\forall x \neg Rxx$. Ningún elemento de la diagonal pertenece a la relación. Observe que esta sentencia no es la negación de la anterior. Para que una relación no sea reflexiva (para que verifique $\neg \forall x Rxx$) basta con que al menos un elemento no esté relacionado consigo mismo ($\exists x \neg Rxx$).

Simetría, asimetría, antisimetría Una relación simétrica no exige a ningún par de elementos estar relacionado. Ahora bien, si lo están en un sentido (Rxy), también deben estarlo en el otro (Ryx). Gráficamente, los arcos en los grafos dirigidos ocurren por pares, en ambos sentidos. En la tabla, si una celda pertenece a la relación, su simétrica respecto a la diagonal debe pertenecer.

Una relación asimétrica exige justo lo contrario: que si una relación se produce en un sentido, no se produzca en el otro ($\forall x \forall y(Rxy \rightarrow \neg Ryx)$). En la tabla, no puede haber dos celdas de la relación simétricas respecto a la diagonal. En particular, ningún elemento puede estar relacionado consigo mismo.

Una relación antisimétrica mantiene la exigencia de la asimetría, pero permite que existan elementos relacionados consigo mismo (no necesariamente todos): $\forall x \forall y(Rxy \wedge Ryx \rightarrow x = y)$

Observe que ninguna de las tres sentencias precedentes es la negación de alguna de las otras. Considere en qué casos una relación no es simétrica, no es asimétrica o no es antisimétrica.

Transitividad, relaciones euclídeas, lineales Coloquial y gráficamente, la transitividad exige que haya un arco directo desde cada elemento x a cualquier otro z accesible desde x por cualquier camino de arcos.

En las relaciones euclídeas, dos elementos cualesquiera relacionados con un mismo x deben estar relacionados entre sí. Es decir, todos los 'vecinos accesibles' desde x deben estar relacionados entre sí. Observe que, si w_1 y w_2 están ambos relacionados con x, se satisface tanto $((Rxw_1 \wedge Rxw_2) \rightarrow Rw_1w_2)$ como $((Rxw_2 \wedge Rxw_1) \rightarrow Rw_2w_1)$. Asimismo $((Rxw_1 \wedge Rxw_1) \rightarrow Rw_1w_1)$. En las relaciones lineales, los elementos distintos relacionados con x sólo deben estar relacionados entre sí en, al menos, un sentido. Además, $((Rxw_1 \wedge Rxw1) \rightarrow (Rw_1w_1 \vee w_1 = w_1 \vee Rw_1w_1))$ ya no requiere que todo elemento relacionado con x esté relacionado consigo mismo.

Relaciones funcionales, deterministas, seriales Una función de W en W no es más que una relación de la que se exige que cada elemento x esté relacionado con un único elemento $y \in W$. En el grafo dirigido de una función, de cada nodo parte un arco y sólo uno. Observe que no se descarta que un elemento esté relacionado (tenga por imagen) a sí mismo.

En una relación determinista, de un nodo pueden o no partir arcos, pero si parte un arco será único. En una relación serial, de cada nodo parte un arco al menos (si no más).

Relaciones totales, densas En una relación total, cualesquiera dos elementos deben estar relacionados, en un sentido u otro. Los números enteros, los racionales o los reales, con su relación usual de orden, verifican esta propiedad.

Muy coloquialmente, en una relación densa, si dos nodos x e y están relacionados, debe 'aparecer' (existir) un nodo intermedio z, tal que Rxz y Rzy. Si la relación R es reflexiva, ese nodo intermedio z podría ser siempre el mismo x. La relación 'menor que' entre números naturales o enteros no es densa; sí lo es entre números racionales o reales.

Satisfacción de un conjunto de propiedades

A una relación se le puede exigir que verifique varias de estas propiedades. Por ejemplo, que sea reflexiva, simétrica y transitiva (es decir, una relación de equivalencia). O que sea reflexiva, antisimétrica y transitiva (es decir, una relación de orden).

Considere, por separado, todas las estructuras $\langle W, R\rangle$ con relación R reflexiva o con relación R transitiva. Las estructuras con relación R reflexiva y transitiva forman el conjunto intersección de los dos anteriores. Cada nueva propiedad exigida a R reduce el número de estructuras que la verifican. En el caso extremo, puede que en ninguna estructura se satisfaga una relación con un determinado conjunto de propiedades. Es decir, que ese conjunto de propiedades (de sentencias de primer orden) sea insatisfacible. Por ejemplo, una relación no puede ser simultáneamente reflexiva y asimétrica.

Por otra lado,de la exposición coloquial se intuye que si una relación satisface un conjunto determinado de propiedades no podrá dejar de satisfacer ésta o aquélla otra. En términos más formales, alguna de estas sentencias es consecuencia lógica de ciertos conjuntos de otras. La conexión entre insatisfacibilidad y consecuencia se encuentra en cualquier exposición de lógica de primer orden.

El lector puede comprobar que:

• Si R es simétrica y transitiva,	entonces	es euclídea
• Si R es reflexiva	entonces	es serial
• R es simétrica, transitiva y serial	si y solo si	
es reflexiva y euclídea	si y sólo si	
es una relación de equivalencia		
(reflexiva, simétrica y transitiva)		

5.2.3 Cierres

Una relación R cualquiera puede perfectamente no ser reflexiva: quizá algunos elementos estén relacionados consigo mismo, pero no todos. Si se añaden a R todos los pares (x,x) que faltaban se obtiene otra relación R', que se denomina *cierre reflexivo* de la relación R.

Observe que a la relación R de partida se le pueden añadir todos esos pares (x,x) que faltaban y algún par (x,y) más. La relación resultante será también reflexiva e incluirá también a la relación R inicial. Lo que caracteriza al cierre reflexivo es que se añaden sólo los pares (x,x) necesarios y ningún otro. Es decir, que es la *menor relación reflexiva que incluye a R*.

Definición 5.3 (Cierre reflexivo) Sea W un conjunto no vacío y R una relación binaria sobre el mismo. Entonces R', el cierre reflexivo de R se define como:

$$R' = \cap\{R'' \mid R'' \text{ es reflexiva y } R \subset R''\}$$

De igual forma se pueden definir el cierre simétrico o el cierre transitivo de una relación, entre otros.

Notemos como R^+ al cierre transitivo de una relación R. Se puede visualizar gráficamente como sigue: desde cada nodo w, produzca un arco directo a cualquier otro accesible desde w mediante un camino de arcos en R:

$$R^+ = \cap\{R'' \mid R'' \text{ es transitiva y } R \subset R''\}$$

Para ciertas definiciones convendrá considerar cierres más complejos, por ejemplo, el cierre R^*, transitivo y reflexivo, de una relación R:

$$R^* = \cap\{R'' \mid R'' \text{ es reflexiva y transitiva, y } R \subset R''\}$$

De cualquiera de estas definiciones se sigue que si una relación ya satisface una propiedad, su cierre respecto a ella coincide con la relación de partida. Para fijar ideas, se presenta la definición de árbol, que utiliza el cierre transitivo y el transitivo-reflexivo de su relación.

Definición 5.4 (Arbol) Un árbol es una estructura relacional $\langle W, R\rangle$ tal que:

1. existe un único elemento (raíz) $r \in W$ que verifica $\forall w R^* rw$
2. para cada $w \neq r$ existe un único w' tal que Rww'
3. $\forall w \neg R^+ ww$

5.3 Lógica monomodal

5.3.1 Lógica modal básica

Permítanos presentarle el lenguaje modal más simple. Se construye sobre el lenguaje de la lógica de proposiciones, con la incorporación de dos nuevos símbolos. Como es usual, la sintaxis del lenguaje facilita el conjunto de fórmulas, de expresiones correctamente formadas.

Dada una fórmula de la lógica modal básica, mediante su semántica se comprueba si es verdadera o no en una determinada estructura relacional. En concreto, las estructuras sobre las que se evalúa una fórmula de este lenguaje constan de *una única relación binaria*.

Lenguaje

Alfabeto y lenguaje

Definición 5.5 (Alfabeto) El alfabeto de la lógica modal básica consta de los siguientes elementos:

1. letras proposicionales: $p_o, p_1, p_2, \ldots$
2. símbolos lógicos:
   (a) constantes proposicionales: $\bot$,$\top$
   (b) conectivas monarias ($\neg$) y binarias: $\wedge$,$\vee$,$\rightarrow$
   (c) operadores modales: $\Box$,$\Diamond$
3. símbolos auxiliares: '(' y ')'

Cuando el número de proposiciones que se consideran simultáneamente es pequeño, se suelen utilizar como letras proposicionales las últimas del alfabeto latino: $(p, q, r, s, ...)$. El lector quizá esté familiarizado con un lenguaje proposicional sin constantes; simplemente representan proposiciones que son siempre falsas ($\bot$) o verdaderas ($\top$).

Se utilizará el símbolo $\leftrightarrow$ como abreviatura. Así, $(\phi \leftrightarrow \psi)$ abreviará la fórmula $((\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi))$. Análogamente, se podía haber considerado un número menor de conectivas primitivas, por ejemplo $\{\neg, \wedge\}$. El resto se definirían como abreviaturas metalingüisticas en la forma usual. Este planteamiento acorta las definiciones y demostraciones sobre el lenguaje.

Definición 5.6 (Fórmulas del lenguaje modal básico) Las fórmulas del lenguaje se definen por la siguiente expresión BNF:

$$\phi ::= p \mid \bot \mid \top \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \Box\phi \mid \Diamond\phi$$

Es decir, una fórmula es, exclusivamente, cualquier cadena que se genere por aplicación finita de las siguientes reglas.

1. Cada letra proposicional es una fórmula
2. Las constantes proposicionales $\bot$ y $\top$ son fórmulas
3. Si ψ es una fórmula, entonces $\neg\psi$ es una fórmula
4. Si ψ y χ son fórmulas, entonces son fórmulas $(\psi \wedge \chi)$, $(\psi \vee \chi)$ y $(\psi \rightarrow \chi)$
5. Si ψ es una fórmula, entonces son fórmulas $\Box\psi$ y $\Diamond\psi$

Notación Notaremos como *Var* al conjunto de las letras proposicionales y como *Form* al conjunto de las fórmulas.

Ejemplo 5.7 De la definición anterior se sigue:

- Si sólo se utilizan las reglas 1-4, las fórmulas generadas son todas las fórmulas de la lógica de proposiciones: $(p \wedge \neg q)$, $(p \rightarrow \neg\bot)$. Esta es la parte, el subconjunto, no modal del lenguaje.
- Considere la fórmula $(p \wedge \neg q)$, que abreviaremos como ψ. De la regla 5 se sigue que $\Box\psi$ es una fórmula: $\Box(p \wedge \neg q)$. A su vez, esta nueva fórmula puede utilizarse para formar otras más complejas: (regla 4) $(\Box(p \wedge \neg q) \rightarrow (p \vee \neg\bot))$.
- La aplicación reiterada de la regla 3 permite la construcción de fórmulas con negaciones sucesivas: $\neg\neg\neg p$. De igual forma, la regla 5 permite la generación de fórmulas con operadores modales consecutivos: $\Box\Box\Diamond(p \rightarrow q)$.
- Todas las expresiones siguientes son fórmulas del lenguaje de la lógica modal básica: p, $(\top \wedge \Box\neg p)$, $\Diamond(\Box\Box p \rightarrow \neg\Diamond\neg q)$,$((q \rightarrow \neg r) \wedge p)$

Subfórmulas

Al igual que en la lógica de proposiciones o de predicados, las demostraciones sobre el lenguaje utilizan el principio de inducción estructural. Las definiciones se producen recursivamente. La siguiente definición es un primer ejemplo de ello.

Definición 5.8 (Conjunto de subfórmulas) A cada fórmula ϕ le corresponde un (único) conjunto de fórmulas $Subf(\phi)$, el conjunto de sus subfórmulas, definido como:

- $Subf(\Box\phi) = \{\Box\phi\} \cup Subf(\phi)$,
  $Subf(\Diamond\phi) = \{\Diamond\phi\} \cup Subf(\phi)$
- $Sub((\phi * \psi)) = \{(\phi * \psi)\} \cup Sub(\phi) \cup Sub(\psi)$
  para toda conectiva binaria $*$
- $Subf(\neg\phi) = \{\neg\phi\} \cup Subf(\phi)$
- $Subf(\bot) = \{\bot\}$,
  $Subf(\top) = \{\top\}$

- $Subf(p) = \{p\}$
  para cada letra proposicional

Ejemplo 5.9 De acuerdo con la definición anterior,

$$Subf(\Box((p \wedge q) \vee \neg\Diamond r)) = \{\Box((p \wedge q) \vee \neg\Diamond r), ((p \wedge q) \vee \neg\Diamond r), (p \wedge q), \neg\Diamond r, p, q, \Diamond r, r\}$$

Árboles sintácticos

Entre las propiedades que pueden demostrarse inductivamente se encuentra el *Principio de análisis sintáctico único*: toda fórmula ϕ (por muy compleja que sea) está en una y sólo una de las 9 siguientes categorías:

- $\Box\psi$, $\Diamond\psi$, $\neg\psi$ (para una ψ determinada)
- $(\psi * \chi)$ para ψ,χ y conectiva binaria determinadas
- $\bot$,$\top$ o es una letra proposicional

Así, la descomposición sintáctica no presenta ambigüedad en ningún caso. En pocas palabras: a toda fórmula se le podrá hacer corresponder un único árbol sintáctico como el que se presenta (sin definición formal) en la figura (fig.5.4). Y viceversa.

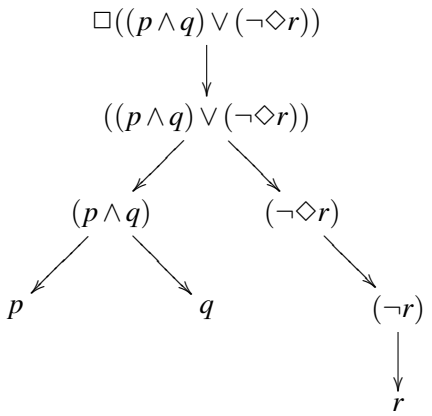


Figura 5.4: Árbol sintáctico de $\Box((p \wedge q) \vee (\neg\Diamond r))$

Antes de continuar, asegúrese de que es capaz de dibujar el árbol sintáctico de cualquiera de las fórmulas que se han visto hasta este punto.

Sustituciones

La sustitución uniforme permite escribir una fórmula a partir de otra. Se puede presentar, intuitivamente, mediante un ejemplo inicial.

Ejemplo 5.10 Considere la fórmula

$$\phi := ((p \rightarrow q) \vee p)$$

y sustituya todas la apariciones de:

- p, por la fórmula $\psi := (r \wedge s)$
- q, por la fórmula $\chi := (t \rightarrow u)$

el resultado es la fórmula

$$(\underbrace{(r \wedge s)}_{\psi} \rightarrow \underbrace{(t \rightarrow u)}_{\chi}) \vee \underbrace{(r \wedge s)}_{\psi})$$

donde las marcas de subrayado se han añadido ocasionalmente para facilitar la correspondencia.

Cuidado Observe que la sustitución se produce uniformemente, por igual, en cada aparición de la letra proposicional sustituida.

Sobre la misma fórmula ϕ de partida, se podían haber sustituido todas las instancias de:

- p, por la fórmula $\psi := q$
- q, por la fórmula $\chi := (q \rightarrow \neg p)$

el resultado sería entonces la fórmula

$$(\underbrace{q}_{\psi} \rightarrow \underbrace{(q \rightarrow \neg p)}_{\chi}) \vee \underbrace{q}_{\psi})$$

Cuidado Observe que la sustitución no vuelve a aplicarse sobre instancias nuevamente aparecidas de la letra proposicional. Se producen todas simultáneamente: *no se 'convierten' primero* las p en q y, luego, todas las q (las primitivas y las recién aparecidas) en $(q \rightarrow \neg p)$.

Algo más formalmente, una sustitución es una función del conjunto de letras proposicionales en el conjunto de fórmulas: $\sigma : \mathit{Var} \mapsto \mathit{Form}$. Gracias a esta función se puede definir otra que asigna a cada fórmula su transformada (por una cierta sustitución): $()^{\sigma} : \mathit{Form} \times \mathit{Sust} \mapsto \mathit{Form}$.

Definición 5.11 (Instancia, por sustitución, de una fórmula) Dada una sustitución $\sigma : \mathit{Var} \mapsto \mathit{Form}$, la transformada ϕ^{σ} de una fórmula ϕ se define como:

1. $(\Box\phi)^{\sigma} = \Box\phi^{\sigma}$,
   $(\Diamond\phi)^{\sigma} = \Diamond\phi^{\sigma}$
2. $(\phi * \psi)^{\sigma} = (\phi^{\sigma} * \psi^{\sigma})$,
   para toda conectiva binaria $*$
3. $(\neg\phi)^{\sigma} = \neg\phi^{\sigma}$
4. $\bot^{\sigma} = \bot$,
   $\top^{\sigma} = \top$
5. $p^{\sigma} = \sigma(p)$,
   para cada letra proposicional

La fórmula ϕ^{σ} es la instancia de ϕ por la sustitución σ.

Ejemplo 5.12 Sea $\sigma : Var \mapsto Form$ tal que

$$\sigma(p) = (q \vee \Diamond p) \qquad \sigma(q) = (p \wedge \Box q)$$

Entonces está definida la transformada $(\phi)^\sigma$ de cualquier fórmula ϕ, por ejemplo $\phi := (p \rightarrow \neg\Box q)$:

$$\begin{array}{rclcrclcrcl} (\phi)^\sigma & = & (p \rightarrow \neg\Box q)^\sigma & = & (p)^\sigma & \rightarrow & (\neg\Box q)^\sigma & = & (p)^\sigma & \rightarrow & \neg(\Box q)^\sigma \\ & = & (p)^\sigma & \rightarrow & \neg\Box(q)^\sigma & = & \sigma(p) & \rightarrow & \neg\Box\sigma(q) & = & (q \vee \Diamond p) \rightarrow \neg\Box(p \wedge \Box q) \end{array}$$

Las sustituciones del ejemplo previo hacen corresponder a p y q con sus fórmulas respectivas. Si p y q no son las únicas letras proposicionales del alfabeto, la sustitución σ debe precisar qué fórmula le corresponde a cada una de las letras restantes. En este caso existirá todo un conjunto de sustituciones distintas que coincidan en la asignación de p y q y difieran en la de alguna de las letras restantes.

Afortunadamente, cualquiera de estas sustituciones producirá la misma tranformación sobre una fórmula ϕ, siempre que en ϕ sólo aparezcan las letras proposicionales p y q.

Instancias, por sustitución, de tautologías Especial interés van a tener las sustituciones sobre tautologías clásicas proposicionales (no modales). Por ejemplo, la que transforma $(p \vee \neg p)$ en:

$$(\Box(q \rightarrow \Diamond r) \vee \neg\Box(q \rightarrow \Diamond r))$$

Semántica relacional

Las fórmulas de la lógica modal se interpretan sobre objetos matemáticos denominados *modelos*. Un modelo (5.18) se define a partir de un *marco* ('el grafo', 5.13) y de una *asignación* (5.16), que precisa qué letras proposicionales son ciertas en cada nodo.

Dada una fórmula ϕ y un modelo, se definirá (5.22) cuándo 'ϕ se satisface (es verdadera) en un nodo determinado del modelo'. Sobre esta semántica se consideran los conceptos usuales de validez, equivalencia y consecuencia lógica.

Marcos

Definición 5.13 (Marco) Un marco F (para la lógica modal básica) es un par $\langle W, R\rangle$, donde W es un conjunto no vacío y R es una relación binaria sobre W.

Notación Dependiendo del contexto o de la aplicación, los elementos de W se denominan *mundos*, *mundos posibles*, *estados*, *instantes*, ... A lo largo de estas notas usaremos indistintamente 'mundos', 'estados' y 'nodos'.

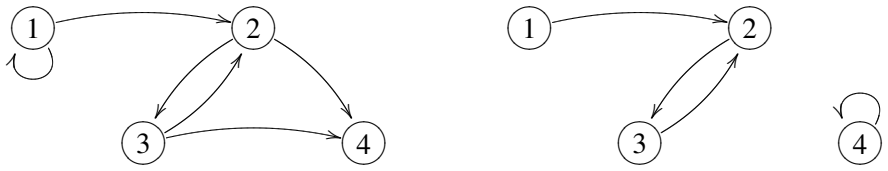
Cuando el mundo w_1 esté relacionado con w_2 diremos que 'desde w_1 se accede a w_2' o que 'w_2 es accesible desde w_1'. Se notará como Rw_1w_2. A la relación R se la denomina *relación de accesibilidad*.

En la notación se han mantenido las iniciales en inglés de los conceptos: $F = \langle W, R\rangle$ corresponde a $Frame = \langle Worlds, Relation\rangle$

Un marco es, en general, *una estructura relacional*: un conjunto y relaciones diversas n-arias sobre el mismo. En este caso, para interpretar la lógica modal básica sólo se necesita una relación binaria. Para fijar ideas seguiremos considerando ejemplos de marcos con pocos elementos, fácilmente representables como grafos dirigidos (fig.5.5).

Ejemplo 5.14 Observe la figura (fig.5.5). En ambos marcos, el mundo w_2 es accesible desde el w_1, pero el w_1 no lo es desde el w_2. Observe que el w_2 tampoco es accesible desde sí mismo. El mundo w_3 es accesible desde el w_1 sólo en el marco de la derecha.

En el marco izquierdo, desde el nodo w_4 no se accede a ningún otro: es un mundo final o terminal; de hecho, el único. En el marco derecho, el estado w_4 se relaciona consigo mismo, luego no es un nodo final.

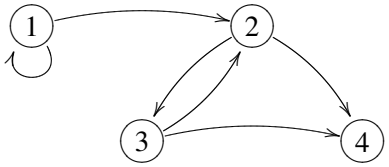


(a) marco $F_1 = \langle W, R_1 \rangle$ (b) marco $F_2 = \langle W, R_2 \rangle$

Figura 5.5: dos marcos sobre un mismo conjunto W

Conjuntos o familias de marcos Algunos resultados teóricos serán válidos no sólo para un cierto marco en estudio, sino para todos los de un determinado conjunto: por ejemplo, 'todos los marcos con relación reflexiva', o 'los que sean reflexivos y transitivos'. Se utilizará la notación $\mathcal{F}$ para referirse a un conjunto (no vacío) de marcos.

Ejercicio 5.15 Con ayuda de la figura (fig.5.6), calcule cuántos marcos distintos pueden definirse sobre un conjunto de 4 elementos. Observe que, independientemente, cada casilla de la tabla puede pertenecer o no a la relación. ¿Cuántas de las relaciones anteriores son reflexivas, es decir, verifican que todo estado es accesible desde sí mismo?



(a) marco $\langle W, R_1 \rangle$

	1	2	3	4
1	x	x		
2			x	x
3		x		x
4				

(b) relación R_1

Figura 5.6: una relación sobre un conjunto de 4 elementos

Modelos

Definición 5.16 (Asignación) Una asignación ν en un marco $\langle W, R \rangle$ es una función $\nu : Var \mapsto \mathcal{P}(W)$ que asocia a cada letra proposicional el subconjunto de mundos donde es verdadera.

Ejemplo 5.17 La figura (fig.5.7) representa gráficamente un marco $\langle W, R \rangle$ con una asignación ν, donde:

- $W = \{1,2,3,4,5,6\}$
- $R = \{(1,2),(2,3),(2,4),(3,2),(3,3),(3,4)\}$
- $\nu(p) = \{1,2,5,6\}$, $\nu(q) = \{3,4\}$, $\nu(r) = \{1,4,5,6\}$

Observe que se define qué letras proposicionales son ciertas en qué estados mediante una función ν del conjunto de las letras proposicionales en el conjunto de subconjuntos de estados.

Otras exposiciones consideran una función ν' que asocia a cada estado el subconjunto de letras proposicionales verdaderas en él; por ejemplo, $\nu'(1) = \{p, r\}$. También puede definirse una función

ν'' que asocia a cada par estado-letra proposicional el valor verdadero o falso, según sea o no verdadera esa letra en ese estado: $\nu''(1,p) = verdadero$.

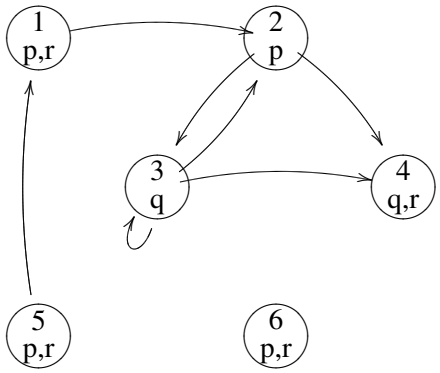


Figura 5.7: Un modelo: marco y asignación

Definición 5.18 (Modelo) Un modelo M (para la lógica modal básica) es una terna $\langle W,R,\nu\rangle$, donde $\langle W,R\rangle$ es un marco y ν es una asignación sobre el mismo. Se notará también como $\langle F,\nu\rangle$.

Cuidado El término *modelo*, en lógica de primer orden, se aplica a toda estructura sobre la que *se satisface* una fórmula. En lógica modal, se denomina modelo *a la estructura matemática sobre la que se interpreta* la fórmula, con independencia de su satisfacción sobre la misma.

Conjuntos o familias de modelos Algunos resultados teóricos son aplicables a todos los modelos de un determinado conjunto. Se utilizará la notación $\mathcal{M}$ para designar a un conjunto de modelos.

Definición 5.19 (Conjunto de modelos sobre un marco) Dado un marco $F = \langle W,R\rangle$, cada asignación distinta sobre el mismo produce un modelo diferente. Si se consideran todas las asignaciones posibles, cada marco determina un conjunto de modelos.

Ejemplo 5.20 Considere de nuevo la figura (fig.5.7). Supongamos que el conjunto de variables es sólo $Var = \{p,q,r\}$. La asignación del modelo de la figura se puede representar con la tabla siguiente:

	1	2	3	4	5	6
p	v	v			v	v
q			v	v		
r	v			v	v	v

Como cada celda (por ejemplo, la (p,1)) se puede marcar o no indistintamente del resto, existen $2^{3\times 6}$ asignaciones distintas sobre este marco. *El conjunto de modelos basados en este marco F* consta de los 2^{18} modelos:

- sobre ese mismo conjunto W,
- con esa misma relación R representada en el grafo,
- para cada una de las 2^{18} asignaciones distintas posibles

Observe que, si se varía la relación R, se obtiene otro marco F' sobre ese conjunto W. El conjunto de modelos basados en el nuevo marco F' también constará de 2^{18} modelos, generados aplicando todas las posibles asignaciones distintas.

Si se considera el conjunto de ambos marcos $\mathcal{F} = \{F, F'\}$, el conjunto de modelos basados en el conjunto de marcos $\mathcal{F}$ será la unión de los modelos basados sobre F y sobre F'.

Ejemplo 5.21 Se pueden definir conjuntos de modelos que no son conjuntos basados en marcos. Un primer ejemplo lo constituye el conjunto de modelos $\mathcal{M}$ que consta sólo del modelo representado en la figura (fig.5.7).

Sobre ese mismo marco, se pueden considerar 'sólo' los 2^{17} modelos en que la variable p es verdadera en el mundo 1. También se pueden considerar los modelos, sobre ese conjunto W, en que la variable p es verdadera en 1 sea cual sea la relación (el marco).

Intuitivamente, un conjunto genérico $\mathcal{M}$ de modelos puede constar: de algunos de los posible modelos sobre cierto marco F, o de algunos modelos escogidos sobre el marco F_1 y de otros sobre $F_2, F_3, \ldots$ O bien, escogiendo 'paquetes completos', de todos los modelos sobre un marco F, o de todos los modelos sobre los marcos de un conjunto $\mathcal{F}$.

Satisfacción

La definición de una semántica, en un sistema lógico, permite responder a la pregunta ¿se satisface (es verdadera) la fórmula ϕ, interpretada sobre el objeto matemático O?. Para el sistema que nos ocupa:

"¿se satisface la fórmula ϕ, interpretada en el mundo w del modelo M?"

Observe el modelo de la figura (fig.5.8). Sitúese localmente en un estado cualquiera. En el mundo w_4, por ejemplo, son verdaderas *sólo* las letras proposicionales q y r. Con la semántica usual de la lógica de proposiciones, *en ese mundo*, son verdaderas las fórmulas $(q \wedge r)$, $(p \rightarrow r)$, $(\neg p \vee \neg q)$ ó $\top$. Y falsas otras como: p, $(r \rightarrow p)$ ó $\bot$.

¿Cómo se interpretan las fórmulas que incluyen operadores modales? Sitúese localmente en un estado cualquiera y *tenga en cuenta sus mundos accesibles*. Por ejemplo, el estado w_2 está relacionado tanto con el estado w_3 como con el w_4 (sin embargo, no está relacionado consigo mismo):

- una fórmula como $\Diamond r$ se satisface en w_2 porque *existe al menos un estado accesible* (w_4) donde es r verdadera; también en w_2 se satisface $\Diamond(r \rightarrow p)$, puesto que $(r \rightarrow p)$ se satisface en su estado accesible w_3
- una fórmula como $\Box q$ se satisface en w_2 porque *en todos sus estados accesibles* (w_3 y w_4) es q verdadera; también en w_2 se satisface $\Box(r \rightarrow q)$

Para evaluar fórmulas más complejas se necesita una definición precisa y formal del concepto de satisfacción.

Definición 5.22 (Satisfacción) La satisfacción de una fórmula ϕ en un mundo w del modelo M, notada como $M, w \Vdash \phi$, se define recursivamente como se adjunta en la tabla 5.2.

Observe cómo la satisfacción de una fórmula compleja se hace depender recursivamente de la satisfacción de sus subfórmulas, hasta llegar al caso base: las fórmulas atómicas 1[a-c].

- Las líneas 1[a-c] y 2[a-d] son una copia de la semántica de la lógica proposicional. Definen cómo interpretar, en un mundo, una fórmula sin operadores modales: simplemente proceda como ya lo hacía en lógica proposicional (considerando qué letras proposicionales son verdaderas en ese mundo).

1*a*	$M, w \Vdash \bot$		en ningún caso
1*b*	$M, w \Vdash \top$		en todo caso
1*c*	$M, w \Vdash p$	si y sólo si	$w \in \nu(p)$
2*a*	$M, w \Vdash \neg\phi$	si y sólo si	no $M, w \Vdash \phi$
2*b*	$M, w \Vdash (\phi \wedge \psi)$	si y sólo si	$M, w \Vdash \phi$ y $M, w \Vdash \psi$
2*c*	$M, w \Vdash (\phi \vee \psi)$	si y sólo si	$M, w \Vdash \phi$ ó $M, w \Vdash \psi$
2*d*	$M, w \Vdash (\phi \rightarrow \psi)$	si y sólo si	no $M, w \Vdash \phi$ ó $M, w \Vdash \psi$
3*a*	$M, w \Vdash \Diamond\phi$	si y sólo si	existe un $w' \in W$ tal que Rww' y $M, w' \Vdash \phi$
3*b*	$M, w \Vdash \Box\phi$	si y sólo si	para todo $w' \in W$, si Rww' entonces $M, w' \Vdash \phi$

Tabla 5.2: definición recursiva de satisfacción $M, w \Vdash \phi$

- En la línea 2a, conviene fijarse en que, coloquialmente, el 'no' metalingüistico externo se convierte en un 'no' interno del lenguaje ($\neg$). Y viceversa. Se abreviará 'no $M, w \Vdash \phi$' como $M, w \nVdash \phi$.

- También informalmente, 3a se puede ver desde una perspectiva procedimental: 'para verificar que $\Box\phi$ se satisface en un mundo, inspeccione todos sus mundos accesibles y compruebe que en cada uno (sin excepción) se satisface la subfórmula ϕ'.

- La misma perspectiva se puede ofrecer de 3b: 'para verificar que $\Diamond\phi$ se satisface en un mundo, inspeccione sus mundos accesibles hasta encontrar al menos uno donde se satisface la subfórmula ϕ'.

La satisfacción es el concepto clave de la lógica modal. Para fijarlo, revisemos algunos ejemplos de complejidad creciente.

Ejemplo 5.23 (Fórmulas sin operadores modales) Considere el modelo M de la figura 5.8:

Una fórmula modal básica sin operadores modales resulta ser una fórmula proposicional. Su satisfacción en un mundo, como fórmula modal, utiliza sólo las entradas 1[a-c] y 2[a-d] de la definición de satisfacción.

La negación $\neg p$ se satisface sólo en los mundos w_3 y w_4. La conjunción $(q \wedge r)$ sólo en el mundo w_4. Un condicional como $((p \rightarrow \neg(q \vee r))$ sólo puede ser falso en los mundos en que p es verdadera (y de éstos, sólo aquéllos en que el consecuente sea falso: w_1, w_5 y w_6).

> *Para todo modelo y mundo, la satisfacción de estas fórmulas, sin operadores modales, no depende de la relación de accesibilidad.*

La fórmula atómica $\top$ se satisface en todo mundo w. Pero también cualquier otra tautología; por ejemplo: $M, w \Vdash (q \rightarrow (p \vee \neg p))$, no importa cuál sea el mundo w. Incluso, independientemente de cuál sea el modelo M.

> *En todo mundo de todo modelo: se satisface cualquiera de las tautologías de la lógica de proposiciones, y no se satisface ninguna de las contradicciones.*

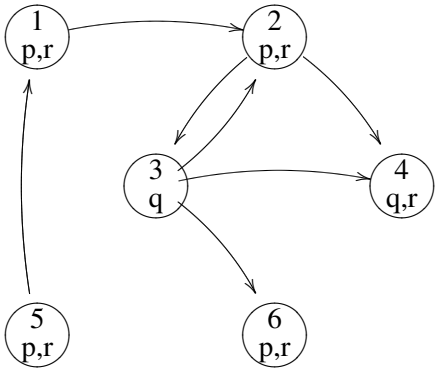


Figura 5.8: satisfacción de una fórmula

Ejemplo 5.24 (Fórmulas simples con operadores modales) Sigamos considerando el modelo M de la figura (fig.5.8). La fórmula $\Diamond(p \wedge r)$ se satisface sobre w_3, puesto que 'en alguno de sus mundos accesibles se satisface la subfórmula $(p \wedge r)$'. Más formalmente:

$$M, w_3 \Vdash \Diamond(p \wedge r) \qquad \text{si y sólo si} \qquad \exists w \in W \text{ tal que } Rw_3w \text{ y } M, w \Vdash (p \wedge r)$$

Los únicos mundos relacionados con w_3 son w_2, w_4 y w_6. Como el conjunto W es finito y pequeño, expandamos las opciones:

$$M, w_3 \Vdash \Diamond(p \wedge r) \quad \text{sii} \quad (\bigvee) \begin{cases} M, w_2 \Vdash (p \wedge r) & \text{sii} \quad M, w_2 \Vdash p \text{ y } M, w_2 \Vdash r & [\surd: \text{ si}] \\ M, w_4 \Vdash (p \wedge r) & \text{sii} \quad M, w_4 \Vdash p \text{ y } M, w_4 \Vdash r & \\ M, w_6 \Vdash (p \wedge r) & \text{sii} \quad M, w_6 \Vdash p \text{ y } M, w_6 \Vdash r & [\surd: \text{ si}] \end{cases}$$

Efectivamente, en los mundos w_2 y w_6 se satisface $(p \wedge r)$, luego en w_3 se satisface $\Diamond(p \wedge r)$. Observe que en w_3 no se satisface $\Box(p \wedge r)$ puesto que la subfórmula $(p \wedge r)$ debería satisfacerse en los tres mundos relacionados:

$$M, w_3 \Vdash \Box(p \wedge r) \quad \text{sii} \quad (\bigwedge) \begin{cases} M, w_2 \Vdash (p \wedge r) & \text{sii} \quad M, w_2 \Vdash p \text{ y } M, w_2 \Vdash r & \\ M, w_4 \Vdash (p \wedge r) & \text{sii} \quad M, w_4 \Vdash p \text{ y } M, w_4 \Vdash r & [\times: \text{ no}] \\ M, w_6 \Vdash (p \wedge r) & \text{sii} \quad M, w_6 \Vdash p \text{ y } M, w_6 \Vdash r & \end{cases}$$

Un resultado negativo, como $M, w_2 \nVdash \Box r$ se demuestra simplemente señalando un mundo accesible desde w_2 (por ejemplo w_3) donde no se satisface r. ¿Qué ocurre en estados como w_4 ó w_6 donde no se puede mostrar un sólo contraejemplo de mundo accesible donde no se satisfaga r? ¿Se verifica o no $M, w_4 \Vdash \Box r$? Denominaremos a estos mundos *finales* o *terminales*, y los analizamos en el ejemplo siguiente.

Ejemplo 5.25 (Satisfacción en mundos finales) Considere, de nuevo, el modelo M de la figura (fig.5.8). Desde el mundo w_4 no es accesible ningún otro (ni siquiera él mismo). En el modelo M sólo encontramos dos de estos mundos terminales: w_4 y w_6. La definición de satisfacción se aplica en ellos de igual forma que en el resto, pero esta aplicación produce resultados poco intuitivos que conviene resaltar:

$M, w_4 \Vdash \Diamond\phi$ se produce si y sólo si existe un mundo accesible desde w_4 donde se satisfaga ϕ. Pero w_4 no tiene mundos accesibles. No importa cuál sea la subfórmula ϕ: ese mundo relacionado donde

debía satisfacerse no existe, y por tanto, $M, w_4 \nVdash \Diamond\phi$ *cualquiera que fuera* ϕ*, incluso aunque* ϕ *fuese una tautología.*

Si w es un mundo final, entonces $M, w \nVdash \Diamond\phi$ *(en todo modelo M, para toda fórmula* ϕ*)*

La satisfacción $M, w_4 \Vdash \Box\phi$ se produce si y sólo si en todo mundo de W se verifica que 'si es accesible desde w_4 entonces se satisface ϕ'. Esta expresión condicional será verdadera salvo que algún mundo verifique el antecedente(ser accesible desde w_4) sin verificar el consecuente (satisfacer ϕ). Y, efectivamente, en ningún caso deja de ser verdadero el condicional, porque ningún mundo verifica el antecedente. Luego $M, w_4 \Vdash \Box\phi$ *para toda fórmula* ϕ*, incluso aunque* ϕ *fuese una contradicción.*

Si w es un mundo final, entonces $M, w \Vdash \Box\phi$ *(en todo modelo M, para toda fórmula* ϕ*)*

Insistiendo en estos resultados anteriores, recuerde que $M, w \nVdash \Diamond\top$ y $M, w \Vdash \Box\bot$ *sólo en mundos finales w*, por la inexistencia de mundos relacionados. En cualquier otro caso, cuando w tenga mundos relacionados, en todos ellos se satisfará $\top$ y en ninguno de ellos se satisfará $\bot$.

$$\text{Si } w \text{ no es un mundo final } \begin{cases} M, w \Vdash \Diamond\top & M, w \Vdash \Box\top \\ M, w \nVdash \Diamond\bot & M, w \nVdash \Diamond\bot \end{cases}$$

Ejemplo 5.26 (Fórmulas $\Diamond^n$ ó $\Box^n$) La sintaxis permite generar fórmulas como $\Diamond\Diamond p$. Para verificar si esta fórmula se satisface en un mundo w hay que considerar la subfórmula $\Diamond p$ en los mundos accesibles de w, que, de nuevo, requiere inspeccionar los mundos accesibles de estos últimos:

$$M, w \Vdash \Diamond\Diamond p \quad \text{si y sólo si} \quad \exists x Rwx M, x \Vdash \Diamond p \quad \text{si y sólo si} \quad \exists y Rxy M, y \Vdash p$$

Así, en el modelo de la figura (fig.5.8), $\Diamond\Diamond p$ se satisface en w_5:

$$M, w_5 \Vdash \Diamond\Diamond p \quad \text{ya que} \quad M, w_2 \Vdash \Diamond p \quad \text{puesto que} \quad M, w_1 \Vdash p \qquad (\text{Y } w_5 \xrightarrow{R} w_2 \xrightarrow{R} w_1)$$

Sin embargo, $\Diamond\Diamond p$ no se satisface en w_1. En general:

$M, w \Vdash \Diamond^n\phi$ si y sólo si existe un mundo z al que se puede acceder desde w tras recorrer n arcos tal que $M, z \Vdash \phi$

$M, w \Vdash \Box^n\phi$ si y sólo si para todo mundo z al que se puede acceder desde w tras recorrer n arcos se verifica $M, z \Vdash \phi$

Observe que no se descarta el uso de 'arcos reflexivos' en este camino. Es decir, si un mundo está relacionado consigo mismo y p es allí verdadero, entonces $\Diamond^n p$ para toda n. Sin embargo, no se puede garantizar que se satisfaga $\Box^n p$, ¿por qué?.

Ejemplo 5.27 (Fórmulas generales) De nuevo sobre el modelo de la figura (fig.5.8), comprobaremos que $M, w_1 \Vdash \Diamond(p \to \Diamond\Box(q \to r))$:

$$\begin{array}{lll} M, w_1 \Vdash \Diamond(p \to \Diamond\Box(q \to r)) & & \text{ya que existe } w_2, \text{ con } Rw_1w_2, \text{ donde} \\ M, w_2 \Vdash (p \to \Diamond\Box(q \to r)) & & \text{condicional que se satisface, puesto que} \end{array}$$

si

$$M, w_2 \Vdash p$$

entonces

$$M, w_2 \Vdash \Diamond\Box(q \to r)$$

Como el antecedente $M, w_2 \Vdash p$ efectivamente se satisface, para completar el desarrollo hay que mostrar que $M, w_2 \Vdash \Diamond\Box(q \to r)$:

$$\begin{array}{rcll} M, w_2 & \Vdash & \Diamond\Box(q \to r) & \text{ya que existe } w_3, \text{ con } Rw_2w_3, \text{ donde} \\ M, w_3 & \Vdash & \Box(q \to r) & \text{ya que, para todo } w \text{ relacionado con } w_3 \\ M, w & \Vdash & (q \to r) & \text{en todo } w \in \{w_2, w_4, w_6\} \end{array}$$

si

$$M, w \Vdash q$$

entonces

$$M, w \Vdash r$$

Ejercicio 5.28 Determine los mundos (si los hay) en el modelo de la figura (fig.5.8) donde se satisfacen las fórmulas:

$$(\Box(p \to q) \wedge \Diamond r), \quad \Box\Box(p \to \Diamond q), \quad (p \to \Box(q \to \Diamond q))$$

Retomando el hilo de la exposición, tras los ejemplos, conviene resaltar la relación que existe entre los dos operadores modales. Tan estrecha, que cada uno no es sino una representación tranformada del otro.

Definición 5.29 (Operadores duales) Para todo operador modal $\triangle$ que se defina, consideraremos el operador $\neg\triangle\neg$, al que denominaremos su operador *dual*.

Los operadores $\Diamond$ y $\Box$ son duales el uno del otro. Observe que, con la semántica definida:

$M, w \Vdash \Box\phi$

si y sólo si para todo w' accesible desde w se satisface ϕ

si y sólo si no existe un w' accesible desde w donde no se satisfaga ϕ

si y sólo si

no $M, w \Vdash \Diamond\neg\phi$

si y sólo si

$M, w \Vdash \neg\Diamond\neg\phi$

El operador $\Diamond$ tiene un sentido existencial (sobre el dominio de mundos localmente relacionados), mientras que el operador $\Box$ tiene un sentido universal. Se podía haber definido el lenguaje con uno sólo de los dos operadores y considerar el otro como una abreviatura. Así, el lenguaje de la lógica modal básica es monomodal: sólo requiere definir un operador modal.

> En general, los operadores modales que se irán definiendo, ocurrirán por pares duales. A veces, ambos símbolos se definen como operadores primitivos del alfabeto; si no, se suele definir un símbolo adicional como abreviatura del dual del primitivo.

Sobre satisfacción y satisfacibilidad Para terminar este apartado, y entre otros problemas relativos, resaltaremos los dos siguientes:

- Dada una fórmula ϕ, así como un modelo M y uno de sus mundos w ¿se satisface ϕ en M, w?
- Dada una fórmula ϕ, ¿es satisfacible: existen M, w tales que $M, w \Vdash \phi$?

Conceptos semánticos básicos

En el apartado anterior se ha definido recursivamente la relación de satisfacción. Basándose en ella se definirán otros conceptos semánticos fundamentales como *verdad, validez, consecuencia o equivalencia*.

Definición 5.30 (Fórmula verdadera) Si $M, w \Vdash \phi$, es decir, si una fórmula se satisface en un estado w de un modelo M diremos que es *localmente verdadera* o *verdadera* en ese estado.

Validez

Recuerde que, en lógica de proposiciones, la satisfacción de una fórmula se producía, o no, en una línea de la tabla de verdad. La fórmula era válida (una tautología) si se satisfacía en todas las líneas. Análogamente, una fórmula modal será válida si se satisface en toda interpretación posible.

Definición 5.31 (Validez) Una fórmula es válida si se satisface en todo mundo de todo modelo.

Notación Para expresar la validez de una fórmula ϕ se utilizará la notación $\Vdash \phi$.

Un fórmula válida se satisface en todo caso: en todo dominio W y con cualquier relación R (es decir, en todo marco), cualquiera que sea la asignación ν y el mundo w donde se evalúa. Así, la validez no puede depender de una elección particular de estas opciones: depende sólo de la propia estructura de la fórmula y de la semántica de las fórmulas modales.

Teorema 5.32 Toda tautología proposicional clásica es una fórmula válida.

Considere un marco y un mundo cualesquiera. No importa cuál sea la asignación, (los valores de las letras proposicionales en ese mundo): la fórmula (por ser tautología) será verdadera en él.

Las tautologías no son las únicas fórmulas válidas. La fórmula del siguiente teorema es un ejemplo de ello, suficientemente importante como para recibir un nombre propio: se denomina K en honor del lógico Saul Kripke.

Teorema 5.33 La fórmula $(\Box(p \to q) \to (\Box p \to \Box q))$ es una fórmula válida.

Para demostrar $\Vdash K$ se precisa demostrar $M, w \Vdash K$ para todo mundo w de todo modelo M. El esquema de demostración que sigue se refiere a un mundo w y a un modelo M cualesquiera. Como no se utilizan características particulares de los mismos, resultará un esquema de demostración correcto para todo mundo y todo modelo.

La fórmula K es un condicional. El único caso en que un condicional no se satisface es cuando sí lo hace el antecedente pero no el consecuente. Pero esto no se produce para K: si se supone la satisfacción del antecedente esto nos conduce lógicamente a la satisfacción del consecuente.

Así, supongamos $M, w \Vdash \Box(p \to q)$. Entonces (1): se satisface $(p \to q)$ para todo mundo de M accesible desde w. Por otro lado, el consecuente de K es asímismo un condicional. Sólo será falso si se satisface su antecedente pero no su consecuente. Supongamos, entonces, $M, w \Vdash \Box p$. Es decir, (2): se satisface p para todo mundo M accesible desde w.

De (1) y (2) se sigue que se satisface q en todo mundo accesible desde w, y por tanto $M, w \Vdash \Box q$.

Algunas fórmulas, sin llegar a ser válidas, pueden aún cumplir propiedades menos exigentes. Por ejemplo: pueden ser verdaderas en todo mundo de cierto modelo M, o de cierta familia de modelos. En estos casos, $W, R, \nu, w \Vdash \phi$ aún se requiere para todo w pero ya sólo para ciertos modelos, ciertas elecciones de marco y asignación. Un caso particular se produce cuando esa familia de modelos coincide con todos los modelos basados en cierto marco ó en un conjunto de ellos.

Definición 5.34 (Validez en un conjunto de modelos)

- Una fórmula ϕ es válida en un modelo M si es verdadera en todo mundo de M.
- Una fórmula ϕ es válida en un conjunto de modelos $\mathcal{M}$ si es válida en todo modelo $M \in \mathcal{M}$.

Como caso particular, resultará especialmente interesante la validez sobre todos los modelos de un marco o de un conjunto de marcos:

- Una fórmula ϕ es válida en un marco F si es válida en todo modelo basado en F.
- Una fórmula ϕ es válida en un conjunto de marcos $\mathcal{F}$ si es válida en todo marco $F \in \mathcal{F}$.

Notación Las fórmulas válidas en un modelo, un conjunto de modelos, un marco y un conjunto de marcos se notan, respectivamente, como $M \Vdash \phi$, $\mathcal{M} \Vdash \phi$, $F \Vdash \phi$, $\mathcal{F} \Vdash \phi$.

Ejemplo 5.35 Elija un marco F cualquiera (universo W y relación R). Considere sólo las asignaciones tales que cada letra proposicional es cierta a lo sumo en un mundo. Obtendrá un conjunto $\mathcal{M}$ de modelos.

Una fórmula ϕ como $\neg(p \wedge q)$ es verdadera en todo mundo de cualquiera de estos modelos $M \in \mathcal{M}$; es decir, $\mathcal{M} \Vdash \phi$, ϕ es válida en $\mathcal{M}$. Si este conjunto de modelos $\mathcal{M}$ coincidiera con todos los modelos basados en F, utilizaríamos la notación (más expresiva) $F \Vdash \phi$. Pero no es el caso.

El lector puede construir conjuntos de modelos como el anterior: escogiendo uno o varios marcos y definiendo asignaciones peculiares. Observará que el diseño de fórmulas válidas en ese conjunto depende de las restricciones sobre las relaciones y asignaciones utilizadas.

Si el conjunto de modelos incluye todos los modelos basados en uno o varios marcos, sólo serán relevantes las restricciones sobre las relaciones. Observe que, en este caso, la fórmula debe ser válida no sólo en todo mundo sino frente a cualquier asignación.

Ejemplo 5.36 Considere un marco con una relación reflexiva. La fórmula $T := (\Box p \rightarrow p)$ será verdadera en cualquier mundo de este marco, no importa cuál sea la asignación utilizada. Es decir, si $\mathcal{F}$ es el conjunto de los marcos reflexivos entonces T es válida en todos los modelos basados en estos marcos: $\mathcal{F} \Vdash T$.

Sobre el conjunto de fórmulas válidas en $\mathcal{M}$ En este punto, concentramos nuestra atención sobre el siguiente problema: *'dado un conjunto $\mathcal{M}$ de modelos, ¿qué fórmulas son válidas en él?'*. Recapitulemos algunos resultados parciales y anticipemos otros:

- Cualquier fórmula válida, por ser cierta en todo modelo, lo es sobre cualquier subconjunto de modelos. De momento, en esta categoría, conocemos las tautologías y la fórmula K
- Aparte de estas fórmulas generales, pueden existir otras particularmente válidas en $\mathcal{M}$. La fórmula T era un ejemplo de ello
- Existen operaciones sintácticas que preservan la validez: dada una fórmula válida en un conjunto $\mathcal{M}$, se garantiza que su transformada resultará asimismo válida en $\mathcal{M}$.

La sustitución uniforme se encuentra entre estas operaciones: preserva la validez, pero ¡cuidado! sólo en conjuntos de modelos 'completos' basados en marcos.

Teorema 5.37 Suponga que ψ es una instancia por sustitución uniforme de ϕ. Para cualquier conjunto de marcos $\mathcal{F}$:

$$\text{Si} \quad \mathcal{F} \Vdash \phi \quad \text{entonces} \quad \mathcal{F} \Vdash \psi$$

Observe que el conjunto de 'todos los modelos posibles' se puede definir como el conjunto de 'los modelos basados en todos los marcos posibles'. Las tautologías y K son válidas sobre este $\mathcal{F}$ global. Y, por tanto, todas sus instancias por sustitución seguirán siendo válidas sobre este conjunto $\mathcal{F}$. Es decir:

> *Las instancias por sustitución de tautologías y de la fórmula K resultan ser válidas, y por tanto, son válidas en cualquier conjunto de modelos $\mathcal{M}$.*

Ejercicio 5.38 Plantéese un conjunto $\mathcal{M}$ de modelos que conste de un único modelo M: un marco F y una asignación adecuada que consiga que cierta fórmula ϕ sea ocasionalmente válida en M.

Puesto que éste no es un conjunto de todos los modelos basados en ciertos marcos, el teorema (teor.5.37) no garantiza que toda instancia de ϕ siga siendo válida en M. Trate de encontrar una sustitución que, efectivamente, no preserve la validez.

Teorema 5.39 Si $\mathcal{M}$ es una colección de modelos, el conjunto de fórmulas válidas en $\mathcal{M}$:

1. incluye todas las instancias de tautologías
2. incluye todas las fórmulas de la forma $(\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi))$
3. si incluye ϕ y $(\phi \rightarrow \psi)$, entonces incluye ψ
4. si incluye ϕ, entonces incluye $\Box\phi$

Además, si $\mathcal{F}$ es una colección de marcos, el conjunto de fórmulas válidas sobre $\mathcal{F}$ es cerrado respecto a la sustitución.

Las dos primeras categorías son fórmulas válidas, en general, y por tanto fórmulas válidas en cualquier conjunto restringido de modelos. El conjunto $\mathcal{M}$ puede contar además con otras fórmulas válidas en él (aunque no válidas en general). Lo que garantizan las dos últimas condiciones es que en $\mathcal{M}$ el conjunto total de sus fórmulas válidas (las generales y las particulares) está cerrado respecto a ellas; es decir:

3. en cada mundo en que ϕ y $(\phi \rightarrow \psi)$ se satisfacen también se satisface ψ. Si se supone que ϕ y $(\phi \rightarrow \psi)$ se satisfacen en todo mundo de todo modelo de $\mathcal{M}$, igualmente ocurrirá con ψ.
4. si ϕ fuese una fórmula cualquiera, puede satisfacerse ϕ sin que se satisfaga $\Box\phi$. Ahora bien, si se parte de que ϕ es válida (verdadera en todo mundo), lo será en todos los vecinos accesibles, luego se satisface $\Box\phi$ en todo mundo (para el mismo conjunto de modelos en que ϕ sea válida).

Además, sólo en el caso de que se considere un conjunto de modelos basados en marcos, la sustitución uniforme preserva la validez sobre este conjunto.

Dado un conjunto de marcos $\mathcal{F}$, denominaremos la lógica basada en $\mathcal{F}$ al conjunto de fórmulas válidas en $\mathcal{F}$.

Consecuencia

Se dirá que un conjunto Γ de fórmulas se satisface en un mundo w del modelo M si $M, w \Vdash \gamma$ para toda fórmula $\gamma \in \Gamma$. Se nota como $M, w \Vdash \Gamma$.

Definición 5.40 (Consecuencia) Una fórmula ϕ es consecuencia de un conjunto de fórmulas Γ cuando:

$$\text{si } M, w \Vdash \Gamma \text{ entonces } M, w \Vdash \phi, \qquad \text{(para todo mundo } w \text{ de todo modelo } M)$$

Notación Esta relación semántica entre el conjunto de fórmulas Γ y la fórmula ϕ se notará $\Gamma \models \phi$. Se suele escribir $\psi, \ldots, \chi \models \phi$ en vez de $\{\psi, \ldots, \chi\} \models \phi$, omitiendo las llaves que delimitan las fórmulas del conjunto Γ. Así, se notará $\psi \models \phi$ cuando el conjunto Γ conste de una única fórmula ψ.

La definición requiere algo más que el mero hecho de que las fórmulas Γ y la fórmula ϕ coincidan, puntual y ocasionalmente, en ser verdaderas. Se requiere que 'donde quiera que se evalúen (mundo y modelo) no pueda ocurrir que se satisfacen las fórmulas Γ y no se satisfaga ϕ'.

Ejemplo 5.41 Como ejemplo trivial, considere $p, q \models (p \wedge q)$. Menos trivialmente, también puede demostrarse $\Box(\phi \to \psi) \models (\Box\phi \to \Box\psi)$

Contrajemplos Para demostrar un resultado negativo, que ϕ no es consecuencia lógica del conjunto de fórmulas Γ ($\Gamma \not\models \phi$) basta mostrar *un sólo mundo w de un modelo M* donde:

$$M, w \Vdash \Gamma \text{ y, sin embargo, } M, w \nVdash \phi,$$

Desgraciadamente, los resultados positivos ($\Gamma \models \phi$) no pueden decidirse extensivamente, verificando todos los (infinitos) modelos posibles.

Consecuencia restringida a un conjunto de modelos La relación de consecuencia propuesta es muy exigente: debe verificarse en todo mundo de todo modelo. Así, $p \not\models \Diamond p$, ya que existen mundos donde p es verdadera sin que lo sea en ninguno de sus mundos accesibles. No obstante, *si sólo consideráramos los modelos basados en marcos reflexivos*, en todo mundo en que sea verdadera p debe serlo $\Diamond p$. Es decir, $\Diamond p$ es consecuencia lógica de p (restringida al conjunto de modelos basados en marcos reflexivos).

Esta relación restringida se notará como $\Gamma \models_{\mathcal{M}} \phi$ y se define como:

$$\text{si } M, w \Vdash \Gamma \text{ entonces } M, w \Vdash \phi, \qquad \text{(para todo mundo } w \text{ de todo modelo } M \in \mathcal{M})$$

Es importante tener en cuenta que, restringidos a un cierto conjunto de modelos $\mathcal{M}$:

- todas las relaciones generales de consecuencia se siguen manteniendo; por ejemplo, $\Box(\phi \to \psi) \models_{\mathcal{M}} (\Box\phi \to \Box\psi)$
- se satisfacen relaciones que no se satisfacían en el conjunto total de modelos; por ejemplo, $p \models_{\mathcal{M}} \Diamond p$ (para $\mathcal{M}$ igual al conjunto de modelos sobre marcos reflexivos).

Equivalencia

Definición 5.42 (Equivalencia) Las fórmulas ϕ y ψ son equivalentes si $\phi \models \psi$ y $\psi \models \phi$. Para representar esta relación semántica entre ambas fórmulas se utilizará la notación $\phi \equiv \psi$.

Observe que la definición de equivalencia se ha formalizado utilizando el concepto de consecuencia (5.40). De esta definición resultará que dos fórmulas son equivalentes si en todo mundo de todo modelo se satisface una si y sólo si se satisface la otra.

Ejemplo 5.43 (Equivalencia de fórmulas sin operadores modales) Las fórmulas $(p \rightarrow q)$ y $(\neg p \vee q)$ son equivalentes en lógica de proposiciones: $(p \rightarrow q) \equiv_p (\neg p \vee q)$. Como esta propiedad se mantiene en todo mundo de todo modelo, resultan ser fórmulas modales equivalentes: $(p \rightarrow q) \equiv (\neg p \vee q)$.

Ejemplo 5.44 (Equivalencias modales básicas) Entre las equivalencias modales más útiles e inmediatas se encuentran:

- $\neg\Box\neg\phi \equiv \Diamond\phi$
- $\Box(\phi \wedge \psi) \equiv (\Box\phi \wedge \Box\psi)$
- $\Diamond(\phi \vee \psi) \equiv (\Diamond\phi \vee \Diamond\psi)$

Ejercicio 5.45 Defina un ejemplo de modelo y mundo donde se compruebe que:

- $\Box(\phi \vee \psi) \not\equiv (\Box\phi \vee \Box\psi)$
- $\Diamond(\phi \wedge \psi) \not\equiv (\Diamond\phi \wedge \Diamond\psi)$

Ejemplo 5.46 La fórmula K es equivalente a :

$$
\begin{array}{l}
(\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)) \\
\qquad\qquad \equiv \quad (\neg\Box(\phi \rightarrow \psi) \vee (\Box\phi \rightarrow \Box\psi)) \quad \equiv \\
\qquad\qquad \equiv \quad (\neg\Box(\phi \rightarrow \psi) \vee (\neg\Box\phi \vee \Box\psi)) \quad \equiv \\
\qquad\qquad \equiv \quad ((\neg\Box(\phi \rightarrow \psi) \vee \neg\Box\phi) \vee \Box\psi) \quad \equiv \\
\qquad\qquad \equiv \quad (\neg(\Box(\phi \rightarrow \psi) \wedge \Box\phi) \vee \Box\psi) \quad \equiv \\
((\Box(\phi \rightarrow \psi) \wedge \Box\phi) \rightarrow \Box\psi)
\end{array}
$$

5.3.2 Teoría de la correspondencia

Ya se ha expuesto que, dado un conjunto de modelos $\mathcal{M}$ existe un conjunto de fórmulas válidas en él. Algunas, porque son válidas en todo modelo, junto a otras particularmente válidas en $\mathcal{M}$. Considere una fórmula ϕ de este segundo grupo: no es válida (en general), pero sí lo es sobre los modelos de $\mathcal{M}$. No se descarta que esta fórmula ϕ sea también válida sobre otros conjuntos de modelos $(\mathcal{M}_1, \ldots, \mathcal{M}_k)$.

> Cuando no ocurre esto, *cuando una fórmula es válida sobre un conjunto de modelos y sólo sobre ese conjunto*, esta fórmula modal caracteriza sintácticamente a todo ese conjunto (normalmente infinito) de modelos.

En esta sección nos restringiremos al estudio de los conjuntos de modelos basados en marcos y de las fórmulas modales que los caracterizan. Ejemplos de estos conjuntos son: todos los modelos con marco reflexivo, o con marco transitivo, o con marco transitivo y euclídeo, o cuya relación sea una relación de equivalencia, o de orden ... En general, todas estas propiedades son definibles por sentencias de la lógica clásica de primer (o de segundo) orden.

Esta línea de investigación suele denominarse *Teoría de la Correspondencia* porque trata de buscar la correspondencia adecuada entre las fórmulas modales y las fórmulas en otros lenguajes, cuando todas ellas se satisfacen exclusivamente sobre las mismas estructuras relacionales.

En general, los teoremas presentan una estructura parecida a la del siguiente, donde una propiedad de la relación se presenta si y sólo si se satisface una fórmula (o, equivalentemente, un esquema):

Teorema 5.47 Sea $F = \langle W, R\rangle$ un marco, entonces:

$$R \text{ es transitiva} \quad \text{si y sólo si} \quad F \Vdash (\Box p \to \Box\Box p) \quad \text{si y sólo si} \quad F \Vdash (\Box\phi \to \Box\Box\phi)$$

• Si R es transitiva entonces $F \Vdash (\Box p \to \Box\Box p)$:

Considere un modelo M basado en F (con relación transitiva, por hipótesis) y un mundo $w \in W$. Se satisface este condicional $M, w \Vdash (\Box p \to \Box\Box p)$ si y sólo si la suposición $M, w \Vdash \Box p$ conduce a $M, w \Vdash \Box\Box p$. Supongamos entonces $M, w \Vdash \Box p$. Es decir, (1) que para todo w' accesible desde w se satisface p.

Consideremos todos los mundos w'' relacionados con estos w'. Como la relación R es transitiva, también están relacionados con w. Por el resultado (1), en todos ellos se satisface p. Luego en cada w' se satisface $\Box p$, y en w se satisface $\Box\Box p$.

• Si $F \Vdash (\Box p \to \Box\Box p)$ entonces la relación R de F es transitiva:

Si la fórmula se satisface en F entonces se satisface para todo mundo y toda asignación sobre F. Luego, se satisface para todo mundo dada una asignación particular (que construiremos).

Intentaremos demostrar que la satisfacción sobre esta asignación particular implica la transitividad de R. O lo que es equivalente, que si R no fuera transitiva, la fórmula no sería satisfacible para esa asignación. Así, se habrá demostrado que cuando es cierto que la fórmula se satisface para todos los modelos de F debe ser transitiva (o fallaría, al menos, en ese modelo).

Elija un mundo cualquiera t y considere una asignación $\nu(p) = \{w \in W \mid Rtw\}$; es decir, p es verdadera sólo en los mundos relacionados con t. Esta asignación garantiza (1) que $F, \nu, t \Vdash \Box p$.

Inicialmente supusimos $F \Vdash (\Box p \to \Box\Box p)$, que junto con (1) implica que (2) $F, \nu, t \Vdash \Box\Box p$.

Así pues, para todo w', w'' tales que Rtw' y $Rw'w''$, de (2) se sigue que en w'' se satisface p. Y por tanto, que Rtw'', puesto que sólo satisfacian p los mundos relacionados con t.

Observe, del párrafo anterior, que se ha garantizado que para todo w', w'' tales que Rtw' y $Rw'w''$, entonces Rtw'': la transitividad de la relación, puesto que no se ha utilizado ninguna propiedad específica del mundo t, y por tanto el desarrollo es válido para todo t.

• Si $F \Vdash (\Box p \to \Box\Box p)$ entonces $F \Vdash (\Box\phi \to \Box\Box\phi)$:

La segunda fórmula es una instancia por sustitución de la primera. Se preserva la validez porque nos hemos ceñido a conjuntos de modelos basados en marcos.

• Si $F \Vdash (\Box\phi \to \Box\Box\phi)$ entonces $F \Vdash (\Box p \to \Box\Box p)$:

La segunda fórmula se ajusta al esquema de la primera, simplemente considerando $\phi = p$.

La tabla (tabl.5.3.2) facilita algunas fórmulas modales junto al tipo de relaciones que caracterizan. Se adjunta también el nombre histórico dado a estas fórmulas (o esquemas de fórmulas).

Reflexiva	$(\Box p \to p)$	(T)
Simétrica	$(p \to \Box\Diamond p)$	(B)
Transitiva	$(\Box p \to \Box\Box p)$	(4)
Euclídea	$(\Box p \to \Box\Diamond p)$	(5)
Determinista	$(\Diamond p \to \Box p)$	(Q)
Serial	$(\Box p \to \Diamond p)$	(D)
Funcional	$(\Box p \leftrightarrow \Diamond p)$	

Tabla 5.3: Fórmulas modales que caracterizan relaciones binarias

La demostración de estas correspondencias se efectúa de forma análoga a la del teorema (teor.5.47). El único paso no trivial consiste en demostrar la propiedad dada la satisfacción de la fórmula. Para ello es preciso encontrar una asignación particular afortunada, de la que se siga este resultado.

5.3.3 Lógicas normales

Cada lógica normal es un conjunto determinado de fórmulas. Como tales conjuntos, si una fórmula pertenece a uno de ellos y no a otro, resultan ser conjuntos distintos: lógicas normales distintas. Existen multitud de lógicas normales distintas. Sin embargo, todas ellas cumplen, para serlo, características comunes.

Definición 5.48 (Lógica normal) Una lógica normal es un conjunto de fórmulas que, incluye:

1. todas las tautologías
2. todas las fórmulas de la forma $(\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi))$

y es cerrado respecto de:

3. *modus ponens*: si incluye ϕ y $(\phi \rightarrow \psi)$, entonces incluye ψ
4. *generalización*: si incluye ϕ, entonces incluye $\Box\phi$
5. *sustitución uniforme*: si incluye ϕ, entonces incluye cualquier instancia suya por sustitución uniforme

Observe que esta definición no hace ninguna referencia a estructuras relacionales. Cualquier conjunto de fórmulas que sintácticamente garantice estas propiedades es una lógica normal. No obstante, esta definición recuerda a los resultados del teorema (teor.5.39). Efectivamente, toda lógica basada en marcos (todas las fórmulas válidas sobre un conjunto de marcos) resultará ser una lógica normal. Sin embargo, no a toda lógica normal le corresponderá una lógica basada en marcos. En estas notas no se considerará ninguna de estas lógicas normales no basadas en marcos.

Lógica K Es la menor de las lógicas normales incluye sólo las tautologías y la fórmula K, así como las fórmulas que resultan del cierre frente a las tres condiciones propuestas. Todas sus fórmulas son válidas.

Lógica KT Resulta tras añadir el axioma $T := \Box p \rightarrow p$, a la lógica K (junto a todas las fórmulas que se obtienen por cierre). Todas sus fórmulas son válidas sobre marcos reflexivos.

Lógica $K4$ Resulta tras añadir el axioma $4 := \Box p \rightarrow \Box\Box p$, a la lógica K (junto a todas las fórmulas que se obtienen por cierre). Todas sus fórmulas son válidas sobre marcos transitivos.

Lógica $KT4$ ó $S4$ Resulta tras añadir T y 4 a la lógica K (considerando todos sus cierres). Todas sus fórmulas son válidas sobre el conjunto de marcos reflexivos y transitivos.

Lógica $KT45$ **ó** $S5$ Resulta tras añadir T, 4 y 5 $:= (\Box p \rightarrow \Box\Diamond p)$ a la lógica K (considerando todos sus cierres). Todas sus fórmulas son válidas sobre el conjunto de marcos reflexivos, euclídeos y transitivos, que resultan también ser los marcos reflexivos, simétricos y transitivos, es decir, sobre todas las relaciones que sean de equivalencia.

Existen otras muchas lógicas normales de interés. En los ejemplos citados (y en otros omitidos) estos conjuntos de fórmulas presentan 3 perspectivas complementarias que las hacen especialmente atrayentes:

- Por un lado, todas sus fórmulas son válidas sobre conjuntos de marcos que tienen una propiedad formal de interés.

- Así, cuando se quiere modelizar un concepto o un sistema (tiempo, agentes, etc.) se escoge la lógica adecuada a las propiedades que presentan las relaciones del sistema (simetría, transitividad, ...)

- Por otro lado, todas estas lógicas tienen diversos sistemas de demostración bien definidos. Por ejemplo, a partir de los axiomas K, T y 4, mediante reglas de inferencia adecuadas, se puede ir generando sintácticamente fórmulas de $KT4$, de las que se garantiza que son válidas en su conjunto de marcos (reflexivos y transitivos).

La exposición de estos sistemas de demostración (de tipo Hilbert, de deducción natural y basados en tablas semánticas) es un material avanzado que no se incluye en los objetivos básicos del curso (en esta edición).

Lecturas del operador modal

Este apartado podía haberse incluido dentro de una sección genérica dedicada a la *Ingeniería Lógica*. Antes de abordar (ligeramente) este concepto, analicemos un ejemplo estándar.

Se pretende formalizar el concepto de necesidad. Es factible hacerlo sobre la lógica modal presentada si se considera que 'algo es necesario en un mundo (una situación) si se satisface en todo mundo accesible desde él'. Informalmente, allá hasta donde se puede mirar se produce p, luego desde esta situación p es necesario. En este contexto, una fórmula como $\Box p$ se puede leer como 'es necesario p'.

Como $\Diamond p$ equivale a $\neg\Box\neg p$, debiera leerse como 'no es necesario que no se satisfaga p'; es decir: 'es posible p'.

La semántica modal definida hasta el momento parece ajustarse perfectamente a la modelización de este concepto. No es extraño, gran parte del trabajo en lógica monomodal se hizo con esta lectura en mente.

Cuando se desarrolla en detalle esta modelización se encuentra una objeción evidente: ¿qué ocurre en un mundo no relacionado consigo mismo? Puede satisfacerse $\Box p$ sin que se satisfaga p en ese mundo. Con la lectura propuesta, supone admitir que p es necesario en un situación sin que se verifique en la misma.

La adecuación del formalismo se produce, sin más sorpresas, si se requiere que la interpretación de este concepto se produzca sobre marcos reflexivos. Es decir, si uno se restringe a la lógica KT. Todas las fórmulas que se pueden demostrar partiendo de estos axiomas serán válidas en marcos formales, y todas ellas expresarán relaciones aceptables entre los conceptos de necesidad ($\Box$), posibilidad ($\Diamond$) y expresiones lógicas proposicionales.

La *Ingeniería Lógica*, en general, parte de una aplicación que pretende formalizar y evalúa qué sistema lógico es adecuado. Si se escoge un sistema modal, se considera qué restricciones formales son exigibles al sistema para representar adecuadamente la aplicación. Formalmente, cuando es posible, esta adición de restricciones, supone añadir axiomas (fórmulas modales) al sistema.

5.4 Lógicas polimodales

El lenguaje utilizado hasta este punto es monomodal: contiene un único operador modal $\Box$, junto a su operador dual $\Diamond$. Gran parte de las aplicaciones de interés requieren lenguajes con más operadores modales.

5.4.1 Sintaxis

Ejemplo 5.49 Considere un lenguaje con dos operadores modales $[a]$ y $[b]$, junto a sus respectivos operadores duales $\langle a\rangle$ y $\langle b\rangle$. Definiremos la sintaxis de este lenguaje como una ampliación de la sintaxis monomodal:

> Si ϕ es una fórmula, entonces $[a]\phi, [b]\phi, \langle a\rangle\phi$ y $\langle b\rangle\phi$ son fórmulas

Luego, todas las siguientes expresiones son fórmulas:

- $(p \rightarrow q)$, $(p \wedge \neg q)$
- $[a](p \rightarrow q)$, $[b](p \wedge \neg q)$, $\langle a\rangle(p \rightarrow q)$
- $[a][a](p \rightarrow q)$, $[a]\langle a\rangle(p \wedge \neg q)$, $\langle b\rangle\langle b\rangle(p \rightarrow q)$
- $[a][b](p \rightarrow q)$, $[b]\langle a\rangle(p \wedge \neg q)$, $\langle b\rangle\langle a\rangle[a](p \rightarrow q)$
- $[a][b](p \rightarrow ([b]\langle a\rangle(p \wedge \neg q)))$

Un lenguaje modal proposicional con n operadores se define de forma análoga.

Definición 5.50 (Fórmulas de un lenguaje polimodal) Se parte de un alfabeto proposicional que incluye el siguiente conjunto de operadores modales $\{[a], \ldots, [n], \langle a\rangle, \ldots, \langle n\rangle\}$. Las fórmulas del lenguaje son, exclusivamente, todas aquellas que se pueden generar por aplicación finita de las siguientes reglas:

1. Cada letra proposicional es una fórmula
2. Las constantes proposicionales $\bot$ y $\top$ son fórmulas
3. Si ψ es una fórmula, entonces $\neg\psi$ es una fórmula
4. Si ψ y χ son fórmulas, entonces son fórmulas $(\psi \wedge \chi)$, $(\psi \vee \chi)$ y $(\psi \rightarrow \chi)$
5. Si ψ es una fórmula, entonces son fórmulas $[k]\psi$ y $\langle k\rangle\psi$, para cada uno de los operadores modales

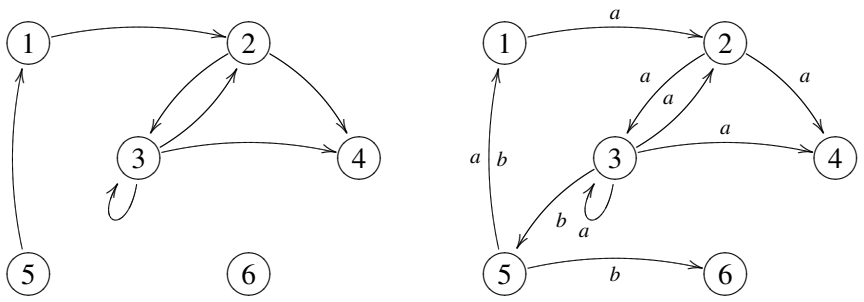
Usualmente, no suelen incluirse los operadores duales como símbolos primitivos del alfabeto. Se les asigna un símbolo posterior, como abreviatura.

5.4.2 Semantica

Como en el apartado anterior, comencemos considerando un lenguaje con dos únicos operadores $[a]$ y $[b]$ (junto a sus duales).

- Interpretar, en un mundo w, una fórmula como $[a]p$ o como $\langle a\rangle p$ requiere inspeccionar el valor de p en sus mundos a-relacionados
- Interpretar, en un mundo w, una fórmula como $[b]p$ o como $\langle b\rangle p$ requiere inspeccionar el valor de p en sus mundos b-relacionados
- Los mundos a-relacionados con w pueden ser distintos de sus mundos b-relacionados

Es decir, un lenguaje con dos operadores modales (junto a sus duales) requiere un marco $\langle W, R_a, R_b\rangle$ con dos relaciones binarias. En la figura (fig5.9b) se muestra un marco de este estilo. La figura (fig5.9a) aisla la relación R_a sobre W, que sería suficiente para las fórmulas de este lenguaje que sólo utilizaran $[a]p$ o $\langle a\rangle p$.



(a) La relación R_a, separada (b) Marco, con relaciones R_a y R_b

Figura 5.9: Marco para un lenguaje bimodal

Ejemplo 5.51 Sobre el marco del la figura (fig.5.9b) construyamos un modelo M sencillo de recordar: p y q son verdaderas sólo en el mundo 3 y falsas en el resto. Es decir, $\nu(p) = \{w_3\}$ y $\nu(q) = \{w_3\}$. Entonces:

- $M, w_3 \Vdash (p \wedge q)$, $M, w_3 \nVdash \neg p$
- $M, w_3 \Vdash \langle a\rangle \neg p$, ya que $\neg p$ es verdadera en alguno de sus mundos a-relacionados (p.ej. w_2)
- $M, w_3 \nVdash [a]\neg p$, ya que $\neg p$ no es verdadera en todos sus mundos a-relacionados (en concreto es falsa en w_3)
- $M, w_3 \Vdash \langle b\rangle \neg p$, ya que $\neg p$ es verdadera en alguno de sus mundos b-relacionados (w_5). Es más, como éste es su único mundo b-relacionado, también $M, w_3 \Vdash [b]\neg p$
- $M, w_1 \Vdash \langle a\rangle\langle a\rangle p$ ya que existe un mundo a-relacionado con w_1 (en concreto,w_2) donde se satisface $\langle a\rangle p$, puesto que p es verdadero en w_3. Sin embargo, $M, w_1 \nVdash [a][a]p$ (observe que p no se satisface en w_4).
- Como w_1 (o w_2, entre otros) son mundos finales en la relación R_b, para toda fórmula ϕ se verifica $M, w_1 \Vdash [a]\phi$ pero $M, w_1 \nVdash \langle b\rangle\phi$

Mayor interés tiene la interpretación de fórmulas donde aparecen ambos operadores:

- $M, w_2 \Vdash \langle a\rangle\langle b\rangle\neg p$ ya que existe un mundo a-relacionado con w_2 (en concreto,w_3) donde se satisface $\langle b\rangle\neg p$, puesto que $\neg p$ es verdadero en w_5.
- $M, w_2 \Vdash \langle a\rangle\langle b\rangle[b]\neg p$. Desde w_2 existe un a-arco seguido de un b-arco que nos sitúa en un mundo (w_5) donde se satisface $[b]\neg p$.

En el caso general, con n operadores modales, basta modificar la definición previa de satisfacción monomodal:

$3'a$	$M, w \Vdash \langle k\rangle\phi$	si y sólo si	existe un $w' \in W$ tal que $R_k w w'$ y $M, w' \Vdash \phi$
$3'b$	$M, w \Vdash [k]\phi$	si y sólo si	para todo $w' \in W$, si $R_k w w'$ entonces $M, w' \Vdash \phi$

Observe que esta definición exige que cada operador modal tenga asignada un relación binaria en el marco sobre el que se interpreta.

Interdependencia entre relaciones El marco de la figura (fig.5.9) facilita una relación R_a para interpretar el par de operadores $[a], \langle a\rangle$ y otra R_b para interpretar $[b], \langle b\rangle$. Perfectamente R_a y R_b se podían haber escogido tales que $R_a = R_b$. Gráficamente, partiendo de R_a, basta añadir la etiqueta b a cada arco etiquetado como a. De esta forma, varios pares de operadores modales pueden interpretarse sobre la misma relación.

Existen otras muchas formas de definir una relación R_b a partir de una R_a. Se puede requerir, por ejemplo, que dos mundos estén relacionados en un sentido por R_a si y sólo si lo están en el otro por R_b. Es decir, que R_a y R_b sean relaciones recíprocas o inversas. Más generalmente: ciertas aplicaciones pueden requerir *alguna dependencia entre las relaciones* de los marcos donde se interpretan.

La formulación de estas dependencias se puede expresar en lógica de predicados. Por ejemplo, la reciprocidad a la que antes se aludía, se presenta en los marcos que verifican

$$\forall x \forall y (R_a xy \leftrightarrow R_b yx)$$

Resultará, sin embargo, mucho más útil si la dependencia entre relaciones se puede caracterizar mediante una o varias fórmulas modales. Por ejemplo, en todos los marcos en que son válidas las fórmulas

$$p \rightarrow [a]\langle b\rangle p \qquad p \rightarrow [b]\langle a\rangle p$$

las relaciones R_a y R_b son recíprocas.

5.4.3 Lógica temporal básica

Sintaxis y semántica

La lógica temporal básica es una lógica modal proposicional con dos operadores: $\langle P\rangle$ y $\langle F\rangle$. De cada uno de ellos resultará útil considerar su respectivo operador dual, e incluso reservarle un símbolo específo:

$$[H] = \neg\langle P\rangle\neg \qquad [G] = \neg\langle F\rangle\neg$$

La lectura pretendida para $\langle P\rangle$ y $\langle F\rangle$ es:

$\langle P\rangle\phi$	existe al menos un mundo, un estado (un instante) en el pasado en que se satisface ϕ
$\langle F\rangle\phi$	existe al menos un mundo, un estado (un instante) en el futuro en que se satisface ϕ

Con esta lectura, sus operadores duales deben interpretarse como:

$[H\phi] = \neg\langle P\rangle\neg\phi$ no existe un instante en el pasado en que no se satisfaga ϕ
en todo instante pasado se satisface ϕ
$[G\phi] = \neg\langle F\rangle\neg\phi$ no existe un instante en el futuro en que no se satisfaga ϕ
en todo instante futuro se satisface ϕ

Notación Se han respetado las iniciales usuales, que en inglés corresponden con:

- $\langle P\rangle$: para algún instante en el pasado (Past)
- $\langle F\rangle$: para algún instante en el futuro (Future)
- $[H]$: para todo instante pasado (it always Has been ...)
- $[G]$: para todo instante futuro (it always is Going to ...)

Usualmente se escriben P, F, H, G (sin corchetes). En esta introducción se han añadido para recalcar el caracter existencial de P y F frente al universal de H y G, sus respectivos operadores duales.

Modelización adecuada del tiempo

El lenguaje temporal básico se puede interpretar en cualquier marco que facilite dos relaciones binarias R_P y R_F. Incluso en el marco de la figura (fig.5.9b). No obstante, la relación entre nodos (entre instantes) en esta figura no capta las restricciones mínimas de una modelización del tiempo.

Transitividad Parece razonable que, si en el futuro de w_1 se encuentra w_2 y en el futuro de w_2 se encuentra w_3, entonces en el futuro de w_1 debiera estar w_3. Es decir, que la relación R_F sea *transitiva*:

$$\forall w_i w_j w_k (R_F w_i w_j \wedge R_F w_j w_k \rightarrow R_F w_i w_k)$$

La misma transitividad se puede pedir de R_P, que relaciona a un instante w_i con otro w_j en su pasado: $R_P w_i w_j$. Así, como mínimo, no se interpretará la lógica temporal básica sobre cualquier marco $\langle W, R_P, R_F\rangle$, sino sobre aquéllos donde tanto R_P como R_F sean transitivas. Es decir, en los marcos en que sean válidas las fórmulas:

$$[H]\phi \rightarrow [H][H]\phi \qquad [G]\phi \rightarrow [G][G]\phi$$

o sus equivalentes

$$\langle P\rangle\langle P\rangle\psi \rightarrow \langle P\rangle\psi \qquad \langle F\rangle\langle F\rangle\psi \rightarrow \langle F\rangle\psi$$

Reciprocidad Pasado-Futuro Otra restricción razonable introduce una dependencia entre ambas relaciones: querríamos que en el futuro de w_1 se encuentre w_2 si y sólo si en el pasado de w_2 se encuentra w_1. Es decir, que una relación sea la recíproca de la otra. Ya se ha mencionado que los marcos donde esto se verifica son aquellos en que son válidas las dos fórmulas siguientes:

$$p \rightarrow [H]\langle F\rangle p \qquad p \rightarrow [G]\langle P\rangle p$$

Cuando se verifica esta reciprocidad, se puede considerar que existe una única relación R (por ejemplo, $R = R_F$). Entonces, la relación 'hacia el pasado' utiliza R pero hacia atrás. Formalmente, basta redefinir la semántica de $\langle P\rangle$ y $[H]$:

$M, w \Vdash \langle F\rangle\phi$ si y sólo si existe un $w' \in W$ tal que Rww' y $M, w' \Vdash \phi$
$M, w \Vdash [G]\phi$ si y sólo si para todo $w' \in W$, si Rww' entonces $M, w' \Vdash \phi$
$M, w \Vdash \langle P\rangle\phi$ si y sólo si existe un $w' \in W$ tal que $R\mathbf{w}'\mathbf{w}$ y $M, w' \Vdash \phi$
$M, w \Vdash [H]\phi$ si y sólo si para todo $w' \in W$, si $R\mathbf{w}'\mathbf{w}$ entonces $M, w' \Vdash \phi$

Observe cómo, en la semántica de $\langle P\rangle$ y $[H]$, los instantes se relacionan por R de forma inversa a cómo lo hacen para $\langle F\rangle$ y $[G]$.

Otras restricciones opcionales Añadiendo restricciones adicionales se pueden ir consiguiendo unas u otras modelizaciones más específicas. Por ejemplo, si se exige que todo instante tenga uno y sólo un R-sucesor, nos limitamos a los marcos temporales lineales; si se permiten varios R-sucesores, consideramos líneas de tiempo que se bifurcan. Asimismo se pueden considerar modelos discretos o modelos densos.

5.4.4 Lógica epistémica

En la economía, los juegos, los protocolos de comunicación o la cooperación entre agentes, no sólo es importante saber si se verifica cierta relación lógica entre proposiciones; además interesa precisar 'si Antonio es consciente de ello' ó si 'Luis conoce que Antonio sabe que se verifica eso'. Más formalmente, notaremos como:

$[K_j]\phi$ a la expresión 'la persona o el agente j conoce ϕ.

La formalización modal de este concepto se produce admitiendo tantos operadores modales $[K_j]$ distintos como agentes distintos estén interactuando. Sus respectivos operadores duales se leerían como:

$\neg[K_j]\neg\phi$, 'j no conoce que no se satisfaga ϕ'; es decir, hasta donde sabe, es posible ϕ.

Este operador dual se representa como $\langle M_j \rangle$. El uso de los corchetes anticipa cómo se interpretan estos operadores sobre las estructuras relacionales.

Considere de momento un único agente. El operador [K] se comporta formalmente como $\Box$; es decir,

$M, w \Vdash [K](p \to q)$ si y sólo si en todo mundo relacionado con w se satisface $(p \to q)$.

En este caso, se satisface si en todos los mundos relacionados con w nunca ocurre que p es verdadera y q falsa. Informalmente, 'el agente sabe algo si eso se satisface en todo mundo que alcanza a ver'.

Igual que se ha hecho anteriormente, desearíamos forzar, restringir el modelo hasta captar formalmente nuestro concepto de conocimiento. Por ejemplo, puede parecer razonable que sea válida la siguiente expresión:

1. $([K]p \to p)$: si en una situación w el agente conoce p entonces efectivamente se satisface p en w; es decir, lo que se conoce es verdad (en ese mismo mundo)

Esta fórmula, con la semántica antes definida no es verdadera en todo caso. Falla en los mundos no relacionados consigo mismo. El lector reconocerá fácilmente que se trata de una reescritura del esquema T. Así pues, si se asume como axioma en un sistema de demostración, todas las fórmulas que se deduzcan serán válidas sobre marcos reflexivos.

Puede considerarse la adición de otras expresiones como axiomas del sistema. Por ejemplo:

2. $[K]p \to [K][K]p$: si el agente conoce p entonces conoce que conoce p; es decir, tiene capacidad de introspección positiva

3. $\neg[K]p \to [K]\neg[K]p$: si el agente no conoce p entonces conoce que no conoce p; es decir, tiene capacidad de introspección negativa

Puede que un modelo realista del conocimiento no incluya todas estas capacidades. En todo caso, si se miran detenidamente, las fórmulas (2) y (3) son adaptaciones de los conocidos esquemas 4 y 5 (éste, en una versión equivalente). Es decir, si estas fórmulas se incluyen como axiomas del sistema (junto a la fórmula K monomodal), se estaría modelizando el conocimiento sobre la lógica $KT45$ (también conocida como $S5$).

En este caso, los marcos sobre los que se interpretan las fórmulas se restringena a aquéllos con relación de equivalencia; es decir, donde los mundos se dividen en clases de equivalencia disjuntas.

Bibliografía complementaria

El texto [Huth y Ryan 2000] tiene sus contenidos ordenados por su aplicación más que por la relación entre sus sistemas lógicos. El lector puede encontrar en su capítulo 5 una buena introducción a la lógica modal y a su aplicación para modelizar la relación entre agentes. Anteriormente, en el capítulo 3, se encuentra también una excelente introducción a la lógica modal temporal y a su uso, que quizá debiera haberse situado tras el capítulo 5.

Para una introducción más formal y extensa conviene consultar [Popkorn 94], [Gabbay et al. 93 y 95] y [Blackburn et al., 01], quizá en ese orden creciente de dificultad.

La lógica epistémica sólo se ha esbozado en estas notas introductorias. En [Meyer y van der Hoek, 1995] y [Fagin et al., 1995] se pueden encontrar varios de estos sistemas junto a las aplicaciones que se han venido desarrollando sobre ellos: razonamiento en sistemas multiagentes, verificación de protocolos de comunicación, sistemas con operadores tanto epistémicos como temporales, ... Posiblemente, un lector novel encuentre [Fagin et al., 1995] algo más fácil de seguir que [Meyer y van der Hoek, 1995].

Actividades y evaluación

El alumno dispone de ejemplos y actividades en el grupo de tutorización telemática del curso, así como exámenes resueltos de años pasados.

Capítulo 6

Lógica Modal Temporal

6.1 Introducción

6.1.1 Diseño de sistemas

Las fórmulas de lógica modal se interpretaban sobre estructuras de Kripke, como la de la figura 6.1. Una estructura así puede utilizarse para modelar muy diversas relaciones, asignando adicionalmente propiedades a cada una de las entidades relacionadas.

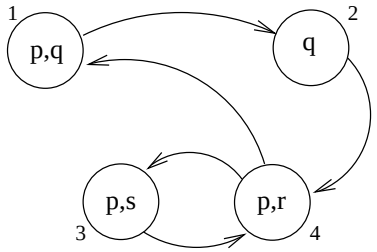


Figura 6.1: Diagrama de transiciones

En el diseño de sistemas físicos o de programas, una figura como la 6.1 se utiliza para describir transiciones entre estados de un sistema. Un microondas o nuestro último móvil no responde siempre igual a la pulsación de la misma tecla: conforme esté en un estado u otro ejecutará o no alguna acción y pasará o no a otro estado. La descripción funcional de un sistema, o su diseño físico íntegro, pueden expresarse mediante estas relaciones entre estados etiquetados.

Gran parte de estos sistemas se diseñan para mantener una permanente actividad reactiva: de una máquina de café o de un sistema operativo se espera un funcionamiento continuo. En estos casos no hay nodos, estados, finales: de todo estado surge al menos una transición hacia otro. Formalmente, en cualquier diagrama con n estados finales se pueden redirigir éstos hacia un nuevo nodo adicional (con un bucle hacia sí mismo). A partir de este punto consideraremos que estamos trabajando con este tipo de sistemas, sin estados finales.

En ese caso, cualquier traza de ejecución de transiciones es ilimitada. Si se empieza en el nodo 1 de la figura 6.1 se puede pasar al 2 y de éste al 4,... Un posible camino que parte del estado 1 es:

$$1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 3 \longrightarrow 4 \longrightarrow 3 \longrightarrow 4 \longrightarrow 1 \ldots$$

La figura 6.2 presenta todas las opciones de recorrido del diagrama de transiciones desde el nodo 1. Siempre se puede construir, desde un nodo cualquiera del diagrama de transiciones, este tipo de

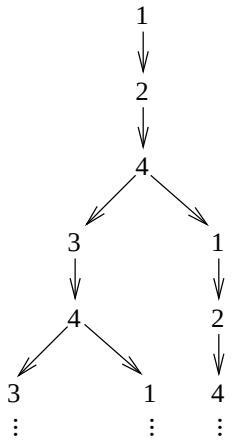


Figura 6.2: Transiciones desde el nodo 1 de la figura 6.1

árbol de alternativas. Observe que debajo del nodo 1 final vuelve a repetirse toda la estructura de alternativas que se muestra desde la raíz. Algo análogo ocurre desde cualquier nodo.

6.1.2 Propiedades de un diseño

De una máquina de café no se espera que atienda peticiones mientras está sirviendo un café, pero sí que (tras pasar por otros estados intermedios) llegue de nuevo al estado de recepción de peticiones.

Esta es una propiedad 'positiva' del sistema, que se espera que ocurra infinitamente a menudo: en cualquier tiempo futuro siempre se confia que en otro instante posterior se presentará esa propiedad. Sobre el árbol de la figura 6.2: "en toda rama se debe asegurar que, situados en cualquier nodo, otro nodo posterior de esa rama tendrá esa propiedad". De otras propiedades 'negativas', como el bloqueo del sistema, se espera que no ocurran en ningún instante de ninguna de las alternativas de ejecución: "que para toda rama, en todo estado de esa rama, no ocurra".

Hay multitud de propiedades de un sistema que pueden expresarse, escogiendo los términos temporales adecuados. De forma abstracta, y apoyándonos en el árbol 6.2 se pueden citar entre otros:

- que algo ocurra en el instante siguiente, para alguna de las alternativas de ejecución (para alguna de las ramas que se abren desde el nodo considerado)
- o que ocurra en el instante posterior, para toda rama
- o que ocurra en algún tiempo futuro, en alguna de las ramas que se abren desde el nodo en cuestión
- o que que se garantice que en toda rama en algún tiempo futuro ocurrirá (aunque en cada rama ese futuro vendrá más tarde o más temprano)
- o que en toda rama se garantice que algo ocurre hasta que se produce otra cosa (de nuevo, más tarde o más temprano en cada rama).

6.1.3 Verificación de las propiedades de un diseño

Cualquier lenguaje formal que pretenda facilitar este tipo de sentencias debe disponer de operadores temporales adecuados. Si además ese lenguaje es el de un sistema lógico, debe precisar una semántica (un significado) para cada operador.

Este documento presenta algunos sistemas lógicos temporales diseñados para facilitar el análisis de sistemas. Las fórmulas se interpretan sobre estrucuturas de Kripke, que resultan ser una descripción total o parcial de la funcionalidad del sistema. El sistema cumple la propiedad expresada en la fórmula si esa estructura de Kripke satisface la fórmula.

Observe que esta verificación de propiedades utiliza *la satisfacción* de una fórmula: dadas una interpretación (la estructura de Kripke, el diseño del sistema) y una fórmula se trata sólo de comprobar si aquélla satisface ésta. No se requiere un análisis de *satisfacibilidad*, que supondría recorrer todas las (quizá infinitas) interpretaciones posibles para una fórmula. U otros conceptos relacionados con la satisfacibilidad, como el de consecuencia.

En concreto, a esta técnica de verificación se la conoce como 'comprobación de modelo' (*model checking*) y está siendo utilizada con éxito para verificar diseños con billones de estados posibles. Una buena página de inicio (en inglés) la facilita Carnegie Mellon

`http://www.cs.cmu.edu/ modelcheck/`

donde también se presenta algún programa verificador, como SMV.

Lógicas temporales para la verificación Este documento se limita a presentar la lógica denominada CTL (*Computation Tree Logic*), que permite expresar propiedades sobre todas las posibles ejecuciones del sistema a partir de un cierto estado (figura 6.2). Otros sistemas lógicos, como LTL (*Linear Temporal Logic*), se restringen al análisis de una posible ejecución, de una rama. Siendo más precisos, LTL no es exactamente un fragmento de CTL: hay propiedades que pueden expresarse indistintamente en ambas lógica, y otras que pueden expresarse en una y no en la otra. Ambas lógica sí son un fragmento de una propuesta posterior, denominada CTL*. Más información sobre estos sistemas puede encontrarse en el texto de Huth y Ryan:

> *"Logic in Computer Science: modelling and reasoning about systems"*; M. Huth y M. Ryan, (segunda edición) 2004 Cambridge University Press

Y un estudio más detallado sobre verificación en:

> *"Model Checking"*; E. Clarke, O. Grumberg y D. A. Peled, 2000 MIT Press

6.2 CTL: Sintaxis

Definición 6.1 La sintaxis de las fórmulas CTL se puede expresar inductivamente como:

$$\begin{array}{rcll} \varphi & ::= & \bot \mid \top \mid p \mid & (1) \\ & & (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \mid & (2) \\ & & AG\varphi \mid AF\varphi \mid AX\varphi \mid & (3a) \\ & & EG\varphi \mid EF\varphi \mid EX\varphi \mid & (3b) \\ & & A[\varphi U \varphi] \mid & (4a) \\ & & E[\varphi U \varphi] & (4b) \end{array}$$

Aunque la definición es un poco extensa, basta apreciar que, sobre la sintaxis proposicional, se proponen algunos nuevos operadores monarios (3) y otros binarios (4).

Simplemente hay que destacar que los nuevos operadores monarios (como ya lo era la negación $\neg$), se escriben con dos símbolos, inseparables. Y que los nuevos operadores binarios (4), como ya lo eran $\wedge$ o $\rightarrow$, se escriben en una 'extraña forma mixta': podían haberse insertado ambos entre las fórmulas $[\varphi AU \varphi]$ o antepuesto a ellas $AU[\varphi, \varphi]$.

Ejemplo 6.2 El proceso de construcción, desde las fórmulas del caso base, es el usual:

1. p y $\top$ son fórmulas CTL (véase (1)), así como $(p \rightarrow \top)$ (por (2)) ó $(\neg(p \rightarrow \top))$. Igualmente, $(q \wedge r)$.

2. $AG(q \wedge r)$ lo es (3), así como $EF(\neg(p \rightarrow \top))$. Y su conjunción o su disyunción (2). También lo es $AGEF(\neg(p \rightarrow \top))$, que tiene la forma $AG\varphi$, donde φ es una fórmula CTL previa.

3. $A[pUq]$ es una fórmula CTL correcta, así como cualquier sustitución de las proposiciones p o q por cualquier fórmula CTL más compleja (cualquiera de las anteriores). Incluso por fórmulas en cuya composición hayan ya intervenido estos nuevos operadores binarios:

$$A[\ \underbrace{E[\,\overline{AGp}\,U\,\overline{EXp}\,]}_{\varphi_1}\ \ U\ \ \underbrace{\overline{AX\overline{EF\overline{(p \rightarrow q)}}}}_{\varphi_2}\ ]$$

Ejercicio 6.3 Construya el árbol sintáctico de la fórmula previa. Observe que cada nodo puede tener un hijo o dos a lo sumo: uno, si la conectiva principal de la subfórmula es la negación o las conectivas temporales monarias (3a,3b); y dos si la conectiva principal de la subfórmula es alguna de las usadas en (2, 4a, 4b). Etiquete el nodo con AU o EU, en el caso de las conectivas 4a o 4b.

Nota 1 (Convenio de precedencia) La omisión de paréntesis produce ambigüedad de interpretación. Se pueden proponer convenios de precedencia entre las conectivas, que fijan la interpretación sintáctica adecuada cuando se duda entre varias.

En este documento, todas las conectivas monarias tendrán la mayor precedencia ($\neg$ y las temporales AG, AF, AX, EG, EF, EX). Seguidas de la conjunción y disyunción; después, del condicional, y por último de las binarias temporales AU y EU.

Ejemplo 6.4 Con el convenio anterior, se tiene que:

1. $\neg AFp \rightarrow q$ se interpreta como $(\neg(AFp)) \rightarrow q$. Cualquier otra interpretación sintáctica, fuera de convenio, requiere ser forzada explícitamente por paréntesis: $\neg AF(p \rightarrow q)$.

2. Como interpretación de una fórmula más compleja:

$$\begin{array}{c} AFq \wedge r \rightarrow A[\neg p \wedge q\ U\ \neg p \vee AGq] \\ \underbrace{((AFq) \wedge r)} \rightarrow \underbrace{(A[\underline{((\neg p) \wedge q)}\ U\ \underline{((\neg p) \vee (AGq))}])} \end{array}$$

6.3 CTL: semántica

6.3.1 Introducción informal

Antes de facilitar una semántica precisa, conviene tener un primer acercamiento informal. En primer lugar, la lógica CTL permite construir (como un fragmento de la misma) fórmulas de la lógica proposicional: sin operadores temporales. En estos casos, en cada nodo, la fórmula será o no verdadera conforme lo sean los átomos (p,q,...) en ese nodo, siguiendo la semántica proposicional. Ya ocurría ésto en las lógicas modales estudiadas.

La aparición de operadores temporales requiere 'consultar otros estados, relacionados con el actual' para evaluar el valor de verdad de la fórmula en el estado que se considera. Remitimos de nuevo a la semántica de las lógicas modales previas. En estas lógicas (modales) temporales, los estados

que deben consultarse son un subconjunto de los estados accesibles en un futuro. Recuerde que cada operador temporal consta de dos signos inseparables: primero A ó E, seguido de X, G, F ó U. Cada uno de los dos signos contribuye a perfilar la semántica del operador.

Así, el primer signo actúa como 'cuantificador sobre las ramas':

- (A) para toda rama (desde el nodo evaluado), para toda posible ejecución del sistema
- (E) existe una rama, una posible ejecución, desde el nodo evaluado

Mientras que el siguiente signo se centra en los estados de una rama, de una posible ejecución del sistema:

- X (*neXt*) el *siguiente*
- F (*some Future state*) *existe* algún estado futuro
- G (*Globally*) *en todo* estado futuro
- U (*Until*) *existe* algún estado futuro *donde ocurre ...* y *hasta el cual* ocurre ...

Observe que todos ellos se aplican sobre el *conjunto de futuros estados* en esa ejecución en particular. Ante esto cabe optar por incluir o no el estado actual en ese conjunto de estados. Así se hace en este documento: cuando se pide que *algún* estado futuro cumpla algo, basta con que lo cumpla el actual. Y si se pide que *todos* los estados futuros cumplan algo, no debe incumplirlo el actual: el futuro incluye el presente. La referencia al siguiente estado (X) requiere evaluar el primer estado futuro distinto al actual.

Ejemplo 6.5 Anticipamos algunos ejemplos de asignación semántica, todos ellos sobre el sistema de la figura 6.1, al que llamaremos *modelo M*. Todas sus ejecuciones se se pueden visualizar alternativamente sobre el árbol de la figura 6.2, que se podía haber enriquecido con la mención de qué letras proposicionales son verdad en cada estado.

- $M,4 \models EXs$ La fórmula EXs se satisface en el estado 4: existe al menos una ejecución (la que comienza pasando al estado 3) donde en el siguiente estado es verdad s. No se satisface en el estado 1.
- $M,4 \models AX(q \vee \neg r)$ La fórmula $AX(q \vee \neg r)$ también es verdad en 4: para toda ejecución desde ese estado, en su estado siguiente se satisface $(q \vee \neg r)$. En la ejecución que comienza en 3, porque allí es verdad $\neg r$. Y en la que comienza en 1, porque allí es verdad tanto q como $\neg r$.
- $M,1 \models EFs$ La fórmula EFs, evaluada en 1, requiere que exista al menos una posible ejecución tal que en algún estado futuro de la misma se cumpla s. (Observe que es una composición de cuantificador existencial sobre ejecuciones y existencial sobre estados de la ejecución). Efectivamente, no toda ejecución desde 1 tiene esa propiedad, pero sí la que pasa por el estado 3.
- $M,1 \not\models AFs$ No se satisface esta fórmula en 1: no es verdad que *toda* ejecución desde 1 acceda a algún estado futuro donde se satisfaga s. En particular, no en la ejecución que repite cíclicamente $1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 1 \ldots$
- $M,1 \models EG(q \vee r)$ Existe efectivamente al menos una ejecución desde 1 tal que permanentemente, en *todo* estado de esa ejecución, es verdad $(q \vee r)$. En particular, la ejecución cíclica antes citada: $1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 1 \ldots$

- $M,1 \not\models AG(q \vee r)$ No es verdad que *en toda* ejecución, en todos sus estados, sea verdad $(q \vee r)$. No en las ejecuciones desde 1 que acceden al estado 3.

- $M,1 \models A[qUr]$ En toda ejecución, desde 1, ocurre *(a)* que *existe* (se acaba accediendo) a un estado donde se satisface r y *(b)* en todo estado desde 1 inclusive hasta ése (sin incluirlo) ocurre q. No importa si q sigue ocurriendo, o no, en y a partir de ese estado.

- $M,1 \models E[(p \vee q)Us]$ Existe al menos una ejecución desde 1 tal que $(p \vee q)$ se satisface en todo estado hasta que se llega a un estado que satisface s. No toda ejecución desde 1 tiene esta propiedad: requiere que efectivamente exista en ella ese estado que satisface s.

6.3.2 Definición de la semántica de CTL

Las fórmulas de CTL se interpretarán sobre un modelo $M = (S, \longrightarrow, L)$ (una estructura de Kripke), que se recuerda que consta de: un conjunto S de estados, una relación $\longrightarrow$ entre ellos y un etiquetado L tal que a cada estado se le asigna el conjunto de letras proposicionales que se satisfacen en el mismo.

Siguiendo, como en todo este documento, la notación del texto de Huth y Ryan, llegamos a la siguiente definición formal de semántica para CTL.

Definición 6.6 Dados un modelo M y uno de sus estados s, la satisfacción de una fórmula φ de CTL en el estado s del modelo M se define recursivamente como:

- $\mathbf{M},\mathbf{s} \models \top$ en todo estado.

- $\mathbf{M},\mathbf{s} \models \bot$ en ningún estado.

- $\mathbf{M},\mathbf{s} \models \mathbf{p}$ si el estado s se ha etiquetado con p, $p \in L(s)$

- $\mathbf{M},\mathbf{s} \models \neg\varphi$ si no ocurre que "$M,s \models \varphi$"

- $\mathbf{M},\mathbf{s} \models (\varphi \wedge \psi)$ si "$M,s \models \varphi$ y $M,s \models \psi$"

- $\mathbf{M},\mathbf{s} \models (\varphi \vee \psi)$ si "$M,s \models \varphi$ o $M,s \models \psi$"

- $\mathbf{M},\mathbf{s} \models (\varphi \rightarrow \psi)$ si "no $M,s \models \varphi$ o $M,s \models \psi$"

- $\mathbf{M},\mathbf{s} \models (\varphi \leftrightarrow \psi)$ si "$M,s \models \varphi$ si y sólo si $M,s \models \psi$"

- $\mathbf{M},\mathbf{s} \models \mathbf{AX}\varphi$ si para todo $s' \in S$ tal que $s \longrightarrow s'$ se cumple $M,s' \models \varphi$

- $\mathbf{M},\mathbf{s} \models \mathbf{EX}\varphi$ si existe un $s' \in S$ tal que $s \longrightarrow s'$ tal que $M,s \models \varphi$

- $\mathbf{M},\mathbf{s} \models \mathbf{AG}\varphi$ si para todo posible camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) se satisface φ en todo estado s_k del camino.

- $\mathbf{M},\mathbf{s} \models \mathbf{EG}\varphi$ si existe al menos un camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) donde se satisface φ en todo estado s_k del camino.

- $\mathbf{M},\mathbf{s} \models \mathbf{AF}\varphi$ si en todo posible camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) existe al menos un estado donde se satisface φ.

- $\mathbf{M},\mathbf{s} \models \mathbf{EF}\varphi$ si existe al menos un camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) donde existe al menos un estado que satisface φ.

- $\mathbf{M}, \mathbf{s} \models \mathbf{A}[\varphi \mathbf{U} \psi]$ si en todo camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) existe un estado s_i tal que $M, s_i \models \psi$ y, para todo estado estrictamente anterior s_j, $j < i$, se cumple $M, s_j \models \varphi$.
- $\mathbf{M}, \mathbf{s} \models \mathbf{E}[\varphi \mathbf{U} \psi]$ si existe al menos un camino $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$ que comience en s ($s_0 = s$) donde exista un estado s_i tal que $M, s_i \models \psi$ y, para todo estado estrictamente anterior s_j, $j < i$, se cumple $M, s_j \models \varphi$.

6.3.3 Expresión de propiedades en CTL

Sobre CTL (así como sobre otras lógicas temporales) pueden expresarse propiedades del diseño de un sistema. Una recopilación de estos patrones se encuentra en la página:

`http://patters.projects.cis.ksu.edu/`

Revisamos el posible uso algunas fórmulas CTL sencillas. Para ello, suponemos que se están evaluando sobre un determinado modelo M en cierto estado s:

[$AG\varphi$] Ésta es una fórmula 'muy exigente'. Para que se satisfaga en el estado s debe ocurrir:

- *(Sobre árbol desplegado del modelo, figura 6.2):* en toda rama y en todo estado en cada rama debe satisfacerse φ; es decir, en todo estado del subárbol situado debajo de s.
- *(Sobre modelo, figura 6.1):* en todo estado de toda posible ejecución desde s se cumple φ.

Se utiliza para expresar propiedades permanentes del sistema que se desean (AGp) o que se quieren evitar ($AG\neg q$). Complicando un poco la fórmula se pueden obtener patrones como $AG(q \rightarrow AG\neg p)$: de todo el subárbol del estado s, donde se evalúe, se espera que 'si algún estado s' tiene la propiedad q, entonces, a partir de él no ocurra p (en todo estado de este subárbol de s' no ocurra p).

[$AF\varphi$] Para que se satisfaga en el estado s debe ocurrir:

- *(Sobre árbol desplegado del modelo, figura 6.2):* que de cualquier rama que se escoja (a partir de s), antes o después (según la rama) aparecerá un estado que satisface φ.
- *(Sobre modelo, figura 6.1):* en toda ejecución del sistema a partir de s se acaba llegando a un estado donde se cumple φ. Esto ocurre trivialmente si el mismo estado s ya satisface φ (recuerde que en nuestra formulación, el futuro incluye el presente).

Es usual que un sistema, a partir de cierto estado, puede evolucionar de muy diversas formas. Un patrón como $p \rightarrow AFq$ reclama que, si p se ha requerido en el estado actual, no deje de verificarse q en un futuro estado (antes o después, según la ejecución que tome el sistema).

Ejercicio 6.7 Sitúese en el contexto de un sistema reactivo, como un sistema operativo o una máquina de café. Hay una propiedad de interés p_+ que se desea que aparezca "ilimitadamente a menudo": ocasionalmente sabiendo que siempre habrá una próxima ocasión. Y otra propiedad negativa p_- que se desea garantizar que, evolucione como evolucione el sistema, acaba definitivamente dejando de ocurrir.

Sustituya adecuadamente φ por p_+ o p_- en las siguientes fórmulas, para expresar los enunciados anteriores sobre estas propiedades.

- $AGAF\varphi$
- $AFAG\varphi$

[$EG\varphi, EF\varphi$] La primera fórmula afirma que, "si acertamos con la ejecución adecuada, todos sus estados tendrán cierta propiedad". Y la segunda que, "si acertamos con la ejecución adecuada, algún estado acabará teniendo cierta propiedad". Estos patrones tal cual, sin que formen parte de otros más complejos, tienen poco interés como garantía de propiedades positivas del sistema: quizá no se produzcan en toda ejecución.

Más interés tienen para señalar la existencia de estados "peligrosos": $EF\,bloqueo$. O para afirmar la ausencia de tales estados, $M,s \models \neg EF\,bloqueo$: a partir de s no existe ejecución tal que se llegue a estado con *bloqueo*. Observe que esta misma propiedad podría haberse expresado como $M,s \models AG\neg\,bloqueo$.

Patrones que fijan precedencia entre eventos El uso adecuado de condicionales (antes citado) junto a los operadores que incluyen X o U permiten modelar el orden relativo de aparición de propiedades esperadas.

Conviene aquí revisar la semántica del operador $(A$ ó $E)[pUq]$: se comporta como un existencial sobre estados, requiere que acabe existiendo un estado s' donde se satisfaga q (y que ocurra p hasta entonces). Así:

1. si q ya ocurre en el estado s actual, se satisfacen AU y EU: ocurre p en todos los estados donde estaba obligada a ocurrir (en cero estados, en un conjunto vacío de estados).

2. si q no ocurre en la rama analizada a partir de s, [pUq] no se satisface, por mucho que ilimitadamente todos los estados desde s satisfagan p

3. p está obligada a ocurrir hasta el estado estrictamente anterior al de ocurrencia de q; que ocurra o no p en s' o después es irrelevante, no influye en la satisfacción de $[pUq]$; en todo caso, no se descarta: $[pUq]$ no afirma (ni niega) que p deje de ocurrir cuando q aparezca.

Es usual definir un operador 'menos exigente', denominado W (*Weak Until*), que se comporta como U salvo por la restricción fijada en el anterior punto 2: pWq se satisface incluso aunque q no llegue a ocurrir (siempre que siga verificándose p). Los patrones recopilados en la página Web recomendada utilizan W en lugar de U.

6.3.4 Equivalencias básicas

Ya se ha mencionado que $\neg EF\varphi \equiv AG\neg\varphi$. Si se contempla el primer signo de un operador temporal como cuantificador sobre ramas (ejecuciones) y el segundo como cuantificador sobre estados de esa rama, se llega a:

$$\begin{aligned} \neg AF\varphi &\equiv EG\neg\varphi \\ \neg EF\varphi &\equiv AG\neg\varphi \\ \neg AX\varphi &\equiv EX\neg\varphi \end{aligned}$$

Es decir, muy coloquialmente: "donde quiera que aparezca en una fórmula CTL el operador AG es posible sustituir esta cadena por $\neg EF\neg$ y obtener así una fórmula equivalente. O sustituir AX por $\neg EX\neg$, es decir, cada operador por su forma dual". Adicionalmente:

$$\begin{aligned} AF\varphi &\equiv A[\top U\varphi] \\ EF\varphi &\equiv E[\top U\varphi] \end{aligned}$$

Como es posible reescribir unos operadores temporales en términos de otros, podría utilizarse un menor número de ellos. En lógica proposicional ya se vió cómo algunos subconjuntos de conectivas eran suficientes para expresar cualquier fórmula boolena: p. ej. $\{\neg,\vee\}$, $\{\neg,\wedge\}$. En la elección de un conjunto adecuado de conectivas CTL hay que tener presente el siguiente teorema:

Teorema 6.8 (A. Martin) Un conjunto de conectivas temporales de CTL es adecuado si y sólo si contiene:

- al menos una entre $\{AX, EX\}$ y
- al menos una bien entre $\{EG, AF, AU\}$, o bien EU.

6.4 Verificación CTL: un primer algoritmo

El objetivo de este tema es proponer un algoritmo para comprobar la satisfacción de una fórmula CTL sobre un modelo M. En su aplicación práctica, el modelo será un diseño de un sistema real y la fórmula expresará una propiedad de interés sobre el mismo.

En el plano formal previo hay simplemente que garantizar que toda fórmula CTL podrá evaluarse sobre cualquier modelo M finito. Como ejemplo, basta considerar la fórmula de la figura 6.3 evaluada sobre un modelo como el de la figura 6.1.

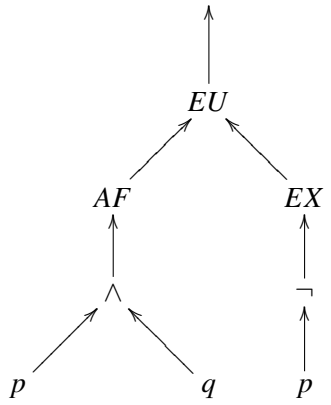


Figura 6.3: $E[\, AF(p \wedge q)\ U\ EX(\neg p)\,]$

6.4.1 Fórmulas sin operadores temporales

Inicialmente, del modelo la única información de la que se dispone es el etiquetado $L(s)$ de las proposiciones de cada estado s. De la fórmula se puede requerir su descomposición sintáctica, su árbol sintáctico.

A partir de ahí, se pueden recorrer uno a uno los estados y marcar explícitamente con la fórmula $(\neg p)$ aquellos estados que no estuvieran inicialmente etiquetados con p. Y marcar explícitamente con la fórmula $(p \wedge q)$ aquéllos estados previamente marcados tanto con p como con q. Análogamente para el resto de las conectivas proposicionales.

Si todas las subfórmulas fueran proposicionales (sin operadores temporales) es fácil definir un proceso iterativo de marcado explícito de fórmulas cada vez más complejas en cada estado , dependiendo de las marcas que existan previamente. Este proceso considera las fórmulas según aparecen, de abajo arriba, en el árbol sintáctico de la fórmula global evaluada.

Al final del proceso, algunos estados presentarán la marca con la fórmula global evaluada y otros no (quizá sólo satisfagan alguna de las subfórmulas de aquélla). Recuerde que la satisfacción de una fórmula es local: en unos estados del modelo puede ocurrir, mientras que en otros no.

Cuidado 1 Este proceso se lleva gráficamente a cabo sobre el modelo del sistema (como el de la figura 6.1), y computacionalmente sobre su codificación. No se trabaja en ningún momento sobre una representación como la del árbol 6.2, que se presentó para fijar ideas.

6.4.2 Fórmulas con operadores temporales

Cuando aparecen operadores temporales el proceso se complica: las marcas que confirman que una fórmula se satisface en cada estado no se calculan exclusivamente a partir de las marcas previas de ese estado. También han de considerarse las de otros estados relacionados.

Fórmulas del tipo $AF\varphi$

[Caso base] Si un estado presenta ya la marca φ, puede asegurarse sin más que también satisfará $AF\varphi$: puesto que se satisface ya en el presente de cualquier ejecución que parta de ese estado y nuestra definición de estados futuros incluía el estado inicial.

[Propagación] Ahora bien, (es suficiente pero) no es necesario que un estado satisfaga φ para satisfacer $AF\varphi$: basta que en toda ejecución desde ese estado se llegue a otro que sí satisface φ. Este hecho se calcula por propagación, hacia atrás, de esta información.

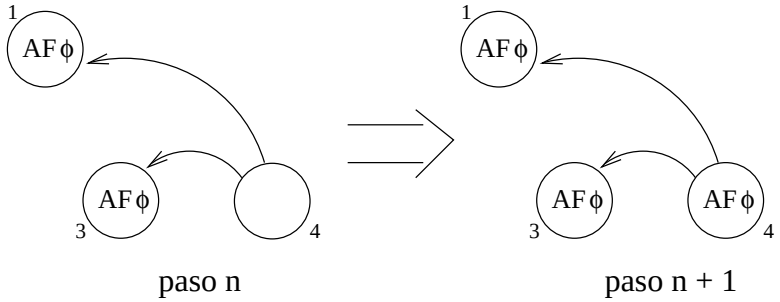


Figura 6.4: Propagación de la marca AF

- *hipótesis 1:* los estados 1 y 3 satisfacen $AF\varphi$
- *hipótesis 2:* los estados 1 y 3 son *todos* los sucesores inmediatos del estado 4
- *conclusión:* el estado 4 también satisface $AF\varphi$

El caso base facilita las primeras marcas $AF\varphi$ y se entra en un proceso iterativo que propaga estas marcas hacia otros nodos si sus sucesores cumplen las hipótesis anteriores. Se finaliza el bucle tras la primera iteración que no produce cambios.

Nota 2 Un sencillo ajuste de la hipótesis 2 permite diseñar un algoritmo análogo para $EF\varphi$: un estado se marca como $EF\varphi$ si *alguno* de sus sucesores inmediatos satisface $EF\varphi$.

Ejercicio 6.9 Sobre el modelo de la figura 6.1, ejecute manualmente este algoritmo para calcular qué estados satisfacen las siguientes fórmulas: AFr, EFs, AFs.

Fórmulas del tipo $AX\varphi$

Bastar recorrer una única vez todos los estados y etiquetar con $AX\varphi$ cada estado tal que *todos* sus sucesores inmediatos tengan marca φ. O requerirlo sólo de uno de sus sucesores si se quiere etiquetar $EX\varphi$.

Ejercicio 6.10 Ejecute este algoritmo sobre la figura 6.1 para calcular qué estados satisfacen AXp, AXq, EXp, EXq, $AX(p \vee q)$.

Fórmulas del tipo $AG\varphi$

[Caso base] Se marcan primero *todos* los estados del modelo con $AG\varphi$. Y se elimina esta marca en aquéllos donde no se cumpla φ.

[Propagación] Se entra en un proceso iterativo donde, en cada iteración, se elimina esta marca de todo estado tal que *alguno* de sus sucesores inmediatos carezca de la marca $AG\varphi$. La iteración finaliza cuando ya no hay cambios de un paso a otro.

Se puede adaptar este algoritmo al cálculo de la marca $EG\varphi$: en el proceso de propagación basta eliminar la marca de los estados tales que *ninguno* de sus sucesores inmediatos la tenga previamente.

Ejercicio 6.11 Sobre el modelo de la figura 6.1 aplique este algoritmo para calcular qué estados satisfacen EGp, $AG(p \vee q)$.

Fórmulas del tipo $A[\varphi U \psi]$

[Caso base] Se marcan inicialmente como $A[\varphi U \psi]$ todos los estados que tenían previamente marca ψ.

[Propagación] Se marca con $A[\varphi U \psi]$ todo estado tal que:

1. tuviera ya una marca φ
2. y *todos* sus sucesores inmediatos tengan marca $A[\varphi U \psi]$

La iteración finaliza cuando ya no hay cambios de un paso al otro.

Ejercicio 6.12 Sobre el modelo de la figura 6.1, ejecute manualmente este algoritmo para calcular qué estados satisfacen $A[qUp]$.

Si lo que se quiere calcular es el conjunto de estados que satisfacen $E\varphi U \psi]$, basta hacer unas ligeras modificaciones sobre el procedimiento previo:

[Caso base] Se marcan inicialmente como $E[\varphi U \psi]$ todos los estados que tenían previamente marca ψ.

[Propagación] Se marca con $E[\varphi U \psi]$ todo estado tal que:

1. tuviera ya una marca φ
2. y *alguno* de sus sucesores inmediatos tengan marca $E[\varphi U \psi]$

La iteración finaliza cuando ya no hay cambios de un paso al otro.

6.4.3 Pseudocódigo

El pseudocódigo del algoritmo 1 utiliza las consideraciones anteriores para construir un proceso ESTADOS(φ, M), que acepta una fórmula CTL y un modelo M; y devuelve el conjunto de estados del modelo que satisfacen esa fórmula.

Simplificable Puede ofrecerse una versión reducida, considerando menos casos, si se reescriben algunas de las conectivas proposicionales y temporales en función de otras. Recuerde que, en lógica proposicional, cualquier fórmula era reescribible equivalentemente a otra con, p. ej., sólo negaciones y conjunciones. En este caso, la fórmula φ de entrada tiene que haberse reescrito sobre este conjunto adecuado de conectivas.

Proceso recursivo Es un algoritmo que procesa recursivamente la fórmula, desde arriba abajo de su árbol sintáctico, haciendo las llamadas oportunas a las subrutinas que van procesando las subfórmulas. Estas subrutinas, para las conectivas proposicionales, se resumen en una nueva llamada al programa ESTADOS. Y para procesar las subfórmulas temporales se llama a subrutinas específicas para cada una (basadas en los algoritmos de etiquetado y propagación antes citados).

Subrutinas específicas Para facilitar el pseudocódigo de alguna de las subrutinas temporales específicas conviene utilizar una notación que se encuentra en el texto de Huth y Ryan citado. Si Y es un conjunto cualquiera de nodos de un modelo:

- $pre_{\exists}(Y)$ designará el conjunto de estados con *algún* sucesor inmediato entre los estados de Y
- $pre_{\forall}(Y)$ designará el conjunto de estados con *todos* sus sucesores inmediatos entre los estados de Y

Observe que Y se utilizará como conjunto a partir del cual se propaga hacia atrás alguna marca hacia nodos s: todos los elementos de Y tendrán esa marca y se exigirá que *todos* los sucesores inmediatos de s o *alguno* se encuentren en Y.

Los algoritmos 2, 3 y 4 son subrutinas suficientes (junto a las proposicionales) para desarrollar el algoritmo ESTADOS(φ, M). Toda fórmula CTL tiene una expresión equivalente que sólo utiliza operadores AF, EU y EX (aparte de un conjunto adecuado de conectivas proposicionales)

6.5 Consideraciones finales

El caracter introductorio de este documento no aconseja presentar más contenido. Atrás se ha quedado la exposición de otros sistemas temporales como LTL o CTL*, o la exposición de verificadores que los utilizan (SPIN,NuSMV,...) No obstante, queda un punto importante por resaltar: el fenómeno de la explosión de estados, que hace del problema de la verificación un problema difícil.

El algoritmo que se ha esbozado, con diversas mejoras, tiene complejidad lineal respecto al número de estados. Sin embargo, el número de estados del modelo crece exponencialmente respecto a las propiedades que se pretenden evaluar del sistema.

Si en la descripción funcional de un sistema hay 3 propiedades de interés (p, q, r), típicamente se proponen estados suficientes para modelar todas sus ausencias / presencias: pqr, $p\bar{q}r$, $p\bar{q}\bar{r}$, ... Es decir 2^3 estados. Aunque el modelo del sistema puede obviar alguno de estos estados, esta tendencia exponencial persiste.

El texto recomendado de Clarke (y colaboradores) es una excelente recopilación de todas las estrategias de simplificación del problema. Algunas de ellas suficientemente eficaces como para manejar

Algorithm 1 ESTADOS(φ, M)

Require: Una fórmula φ de CTL y un modelo $M = \{S, \longrightarrow, L\}$
Ensure: El conjunto de estados de M que satisfacen φ

if φ es $\bot$ **then**
  return: $\emptyset$
else if φ es $\top$ **then**
  return: S
else if φ es atómica **then**
  return: $\{s \in S \mid \varphi \in L(s)\}$
else if φ es $\neg\varphi_1$ **then**
  return: $S -$ ESTADOS(φ_1, M)
else if φ es $(\varphi_1 \wedge \varphi_2)$ **then**
  return: ESTADOS$(\varphi_1, M) \cap$ ESTADOS(φ_2, M)
else if φ es $(\varphi_1 \vee \varphi_2)$ **then**
  return: ESTADOS$(\varphi_1, M) \cup$ ESTADOS(φ_2, M)
else if φ es $(\varphi_1 \rightarrow \varphi_2)$ **then**
  return: ESTADOS$((\neg\varphi_1 \vee \varphi_2), M)$
else if φ es $(\varphi_1 \leftrightarrow \varphi_2)$ **then**
  return: ESTADOS$(((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)), M)$
else if φ es $AG\varphi$ **then**
  return: ESTADOS$_{AG}(\varphi, M)$
else if φ es $EG\varphi$ **then**
  return: ESTADOS$_{EG}(\varphi, M)$
else if φ es $AF\varphi$ **then**
  return: ESTADOS$_{AF}(\varphi, M)$
else if φ es $EF\varphi$ **then**
  return: ESTADOS$_{EF}(\varphi, M)$
else if φ es $AX\varphi$ **then**
  return: ESTADOS$_{AX}(\varphi, M)$
else if φ es $EX\varphi$ **then**
  return: ESTADOS$_{EX}(\varphi, M)$
else if φ es $A[\varphi U \psi]$ **then**
  return: ESTADOS$_{AU}(\varphi, \psi, M)$
else if φ es $E[\varphi U \psi]$ **then**
  return: ESTADOS$_{EU}(\varphi, \psi, M)$
else
  return: "No es una fórmula CTL"
end if

un elevadísimo número de estados, es decir, suficientes para abordar la verificación de complejos sistemas industriales.

En el texto de Huth y Ryan, que ha sido referencia de este documento, hay una buena exposición de una de estas estrategias: el uso de estructuras de datos eficientes, en particular OBDDs (*Ordered Binary Decision Diagrams*), diagramas binarios de decisión ordenados.

Algorithm 2 ESTADOS$_{EX}(\varphi, M)$

Require: Una fórmula φ de CTL y un modelo $M = \{S, \longrightarrow, L\}$
Ensure: El conjunto de estados de M que satisfacen $EX\varphi$

```
    local var X,Y
2:  X := ESTADOS(φ,M)
    Y := pre∃(X)
4:  return: Y
```

Algorithm 3 ESTADOS$_{AF}(\varphi, M)$

Require: Una fórmula φ de CTL y un modelo $M = \{S, \longrightarrow, L\}$
Ensure: El conjunto de estados de M que satisfacen $AF\varphi$

```
    local var X,Y
2:  X := S
    Y := ESTADOS(φ,M)
4:  repeat
        X := Y
6:      Y = Y ∪ pre∀(Y)
    until X = Y
8:  return: Y
```

Algorithm 4 ESTADOS$_{EU}(\varphi, \psi, M)$

Require: Dos fórmulas, φ y ψ, de CTL y un modelo $M = \{S, \longrightarrow, L\}$
Ensure: El conjunto de estados de M que satisfacen $E[\varphi U \psi]$

```
    local var W,X,Y
2:  W := ESTADOS(φ,M)
    X := S
4:  Y := ESTADOS(ψ,M)
    repeat
6:      X := Y
        Y = Y ∪ (W ∩ pre∃(Y))
8:  until X = Y
    return: Y
```

Bibliografía

[Abramsky et al., 1992-2001] S. Abramsky, D.M. Gabbay y T.S.E. Maibaum. *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992 (primer volumen) - 2001 (quinto volumen).

[Apt, 1996] K.R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, Londres, 1996.

[Apt y Olderog, 1997] K.R. Apt y E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlín. 2ª edición.

[Badesa et al., 1998] . Badesa, I. Jané y R. Jansana. *Elementos de lógica formal.* Ariel, 1998.

[Ben-Ari, 1990] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall International, Londres, 1990.

[Ben-Ari, 2001] M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer-Verlag, Londres, 2001.

[Blackburn et al., 2001] P. Blackburn, M. de Rijke e Y. Venema. *Modal Logic.* Cambridge University Press, Cambridge, RU, 2001.

[Broda et al., 1994] K. Broda, S. Eisenbach, H. Khoshnevisan y S. Vickers. *Reasoned Programming.* Prentice Hall International, 1994.

[Burris, 1998] S. N. Burris. *Logic for Mathematics and Computer Science.* Prentice Hall, 1998.

[Clarke et al., 1994] E. Clarke, O. Grumberg y D. Long. "Verification Tools for Finite-State Concurrent Systems". En: J.W. de Bakker, W.P. de Roever. *A Decade of Concurrency: Reflections and Perspectives.* Springer-Verlag, Berlín, 1994.

[Clarke et al., 2000] E.M. Clarke, O. Grumberg y D. Peled. *Model Checking.* MIT Press, Cambridge, MA, 2000.

[Clocksin et al., 1994] W. Clocksin y C.S. Mellish. *Programming in Prolog.* Springer, 1994. 4^a edición.

[Chagrov y Zakharyaschev, 1997] A. Chagrov y M. Zakharyaschev. *Modal Logic.* Oxford University Press, Oxford, 1997.

[Cuena, 1985] J. Cuena. *Lógica Informática.* Alianza, 1985.

[Dalen, 1997] D. van Dalen. *Logic and Structure* Springer, 1997. 3^a edición.

[Deaño, 1993] A. Deaño. *Introducción a la lógica formal.* Alianza Universidad, 1993. Décima reimpresión.

[Doets, 1994] K. Doets, *From Logic to Logic Programming.* MIT Press, Cambridge, MA, 1994.

[Ebbinghaus et al., 1996] H.-D. Ebbinghaus, J. Flum y W. Thomas. *Mathematical Logic.* Springer, 1996. 2^a edición.

[Fagin et al., 1995] R. Fagin, J.Y. Halpern, Y. Moses y M.Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, MA, 1995.

[Fitting, 1996] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, Nueva York, 1996. 2^a edición.

[Francez, 1992] N. Francez. *Program Verification.* Addison-Wesley, Reading, MA, 1992.

[Gabbay et al., 1993] D.M. Gabbay, C.J. Hogger y J.A. Robinson (eds). *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 1: Logic Foundations.* Oxford University Press, Oxford, 1993.

[Gabbay et al., 1994a] D.M. Gabbay, I. Hodkinson y M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Oxford University Press, Oxford, 1994.

[Gabbay et al., 1994b] D.M. Gabbay, C.J. Hogger y J.A. Robinson (eds). *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 2: Deduction Methodologies.* Oxford University Press, Oxford, 1994.

[Gabbay et al., 1994c] D.M. Gabbay, C.J. Hogger y J.A. Robinson (eds). *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 3: Nonmonotonic Reasoning.* Oxford University Press, Oxford, 1994.

[Gabbay et al., 1995] D.M. Gabbay, C.J. Hogger y J.A. Robinson (eds). *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 4: Epistemic and Temporal Logics.* Oxford University Press, Oxford, 1995.

[Gabbay et al., 1998] D.M. Gabbay, C.J. Hogger y J.A. Robinson (eds). *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 5: Logic Programming.* Oxford University Press, Oxford, 1998.

[Gabbay et al., 2000] D.M. Gabbay, M.A. Reynolds y I. Hodkinson. *Temporal Logic: Mathematical Foundations and Computational Aspects. Vol. 2.* Oxford University Press, Oxford, 2000.

[Garrido, 1995] M. Garrido. *Lógica Simbólica.* Tecnos, 1995. 3^a edición.

[Hoare, 1969] C.A.R. Hoare. "An axiomatic basis for computer programming". *Communications of the ACM*, 12 (1969) 576-580.

[Hughes y Cresswell, 1968] G.E. Hughes y M.J. Cresswell. *An Introduction to Modal Logic.* Methuen, Londres, 1968.

[Hughes y Cresswell, 1996] G.E. Hughes y M.J. Cresswell. *A New Introduction to Modal Logic.* Routledge, Londres, 1996.

[Huth y Ryan, 2000] M.R.A. Huth y M.D. Ryan. *Logic in Computer Science. Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, RU, 2000.

[Klir y Yuan, 1995] G.J. Klir y B. Yuan. *Fuzzy Sets and Fuzzy Logic.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Kowalski, 1986] R. Kowalski. *Lógica, programación e inteligencia artificial.* Díaz de Santos, 1986. Traducida de *Logic for Problem Solving.* Elsevier, 1979.

[Lloyd, 1987] J. Lloyd. *Foundations of Logic Programming.* Addison-Wesley, Reading, MA, 1987. 2ª edición.

[Manna y Pnueli, 1992] Z. Manna y A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. I: Specification.* Springer-Verlar, Nueva York, 1992.

[Manna y Pnueli, 1995] Z. Manna y A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. II: Safety.* Springer-Verlar, Nueva York, 1995.

[McMillan, 1993] K. McMillan. *Symbolic Model Checking.* Kluwer, Dordrecht, Holanda, 1993.

[Mendelson, 1997] E. Mendelson. *Introduction to Mathematical Logic.* Chapman and Hall, Londres, 1997. 4ª edición.

[Meyer y van der Hoek, 1995] J.J. Meyer y W. van der Hoek. *Epistemic Logic for Computer Science and Artificial Intelligence.* Cambridge University Press, Cambridge, RU, 1995.

[Pnueli, 1981] A. Pnueli. "The temporal logic of concurrent programs". *Theoretical Computer Science*, 13 (1981) 45-60.

[Pnueli y Shahar, 1996] A. Pnueli y E. Shahar. "A platform for combining deductive with algorithmic verification". *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV'96)*. Springer-Verlag, Berlín, 1996.

[Popkorn, 1994] S. Popkorn. *First Steps in Modal Logic.* Cambridge University Press, Cambridge, 1994.

[Smullyan, 1995] R.M. Smullyan. *First-Order Logic*. Dover, Nueva York, 1995. La primera edición de esta obra fue publicada por Springer Verlag, Nueva York, 1968.

[Vila, 1994] L. Vila. "A survey on temporal reasoning in artificial intelligence". *AI Communications*, 7 (1994) 4-28.