TEXTOS UNIVERSITARIOS

FACULTAD DE INGENIERIA

LOGICA PARA CIENCIA DE LA COMPUTACION

LEOPOLDO BERTOSSI D.

EDICIONES UNIVERSIDAD CATÓLICA DE CHILE

LOGICA
PARA
CIENCIA DE LA COMPUTACION

Dr. Leopoldo Bertossi D.

Pontificia Universidad Católica de Chile

Escuela de Ingeniería

Departamento de Ciencia de la Computación

ERRATA

- Página 34. La demostración de **2.**: Está dada sólo la de una de las direcciones de la doble implicación.
- Página 48, línea 6, sec. 2.9: "... nos interesa especialmente el caso ...".
- Página 51, línea 1: "... lógica, resolvemos un problema computacional, en el caso ideal, de la siguiente manera:"

  Línea 8: Kowalski
- Página 53, línea -7: "Más generalmente, demostramos que ..."
- Página 57: Falta especificar que f es una función, es decir, que, para cada i, existe exactamente un j, tal que p_{ij}. Esto se puede expresar por medio de una fórmula proposicional.
- Página 60: Un compuerta `XOR` corresponde a un "o exclusivo".
- Página 61: Falta ahí la axiomatización de las operaciones booleanas.

  Debería haber una definición explícita de diagnóstico como una valuación que satisface simultáneamente al modelo y a la observación.
- Página 90, Problema 5: Falta indicar un número k como parte de la instancia. La pregunta es si existe una clique de tamaño a lo más k
- Página 111, línea -9: "... cuantificar en LP ..."
- Página 159, item 2(a): Hay que poner la restricción de que las variables de t no están cuantificadas en φ. Tal como está, hay un contraejemplo: $\forall x \exists y(\neg x = y) \rightarrow \exists y(\neg y = y)$.
- Página 224: Falta la fórmula: $Legal(S_0) \wedge \forall a, s(Legal(s) \wedge Pos(a, s) \rightarrow Legal(ejecutar(a, s))$.

CAPITULO

PROLOGO

La computación es una disciplina que cuenta con sus propios temas, conceptos, y problemas que son susceptibles de ser tratados en forma científica, con los métodos de las ciencias exactas. Como disciplina científica, la computación hace uso fundamental de la matemática, que provee los lenguajes, conceptos y técnicas fundamentales para su construcción y desarrollo. Parte importante de la matemática que provee los fundamentos científicos de la computación es la lógica, más precisamente, la lógica matemática. Este es un libro sobre lógica matemática con una elección de temas y una forma de presentación del material que están fuertemente motivadas por su relación de la lógica con la computación. Sin embargo, el libro también será de utilidad para aquellos que están especialmente interesados en la lógica matemática y que no saben o no se interesen en la computación (a pesar de que el desarrollo histórico de la lógica matemática simpre ha estado estrechamente ligado al desarrollo de la computación).

La lógica matemática es un área joven de la matemática. Podemos decir que se inicia en la forma que hoy la conocemos hace unos 100 años, tratando problemas relacionados con los fundamentos y la filosofía de la matemática. Hoy día podemos observar que conceptos, términos y técnicas de la lógica matemática aparecen con frecuencia en otras disciplinas del conocimiento: lingüística, biología, psicología, sociología, filosofía, y ciencia de la computación, por citar algunas.

Actualmente un mínimo conocimiento de lógica matemática es prácticamente ineludible para cualquiera que, a través de sus estudios, trabajo o investigación, se relacione con ciencia de la computación. Muchas áreas de ella tienen que ver con lógica matemática en forma más o menos directa: bases de datos, semántica de lenguajes de programación, especificación de tipos de datos y procedimientos, inteligencia artificial, complejidad computacional, teoría de algoritmos, modelos de computación, programación en lógica y funcional, demostración mecánica de teoremas, ingeniería de software, etc.

De hecho, un curso de lógica matemática es exigido a los alumnos de pregrado por mucho departamentos de ciencia de computación de las universidades más prestigiosas del mundo. Más aún, en esos departamentos, además de dictar cursos de lógica formal, se hace intensa investigación en el área. Podemos decir que la lógica matemática y algunos temas de matemática discreta forman gran parte de "la matemática de la ciencia de la computación", tal como, por mucho

tiempo, ciertas áreas de la matemática continua, como ecuaciones diferenciales, fueron gran parte de la matemática de la física.

Si bien es cierto que la lógica matemática ha sido un aporte fundamental a la ciencia de la computación, también es cierto que el desarrollo de la lógica matemática se ha visto fuertemente estimulado por los problemas que la ciencia de la computación le ha puesto. Sin exagerar, podemos decir que la ciencia de la computación ha permitido a la lógica matemática reencontrarse con sus orígenes y raíces filosóficas y fundacionales. En este sentido, vemos una mutua retroalimentación y codesarrollo de ambas disciplinas, en la misma forma en que física y matemática se desarrollaron juntas hasta las primeras décadas de este siglo, siendo, de hecho, casi indistinguibles como disciplinas del conocimiento.

Este libro está dirigido a aquellos que por distintos motivos se interesen por iniciarse en la lógica matemática, no sólo a los interesados en ciencia de la computación. Sin embargo, la elección de algunos temas, algunas discusiones y ejemplos está motivada en gran parte por las necesidades que he percibido en el mundo de la ciencia de la computación. Es por esto que se presenta muchos de los temas clásicos e inevitables de la lógica matemática, pero también temas que son propios de lógica matemática en ciencia de computación, y, más particularmente, en bases de datos e inteligencia artificial.

Los conocimientos de matématica por parte del lector que exige este libro son mínimos. Conocimientos adquiridos en un primer curso de álgebra bastan y sobran. No se presupone conocimientos de lógica matemática, aunque cierta familiaridad con lo más básico de lógica proposicional, como tablas de verdad y álgebra proposicional, puede ser útil.

Este libro se originó de las notas del curso "Lógica para Ciencia de Computación" que he estado dictando regularmente durante los últimos tres años a los alumnos de ingeniería de computación de la Pontificia Universidad Católica de Chile, y en una ocasión en la Universidad de Chile. En él ha habido influencia de varios otros. Es difícil decir cuáles y en qué extensión y sentido. Sin embargo, no puedo dejar de destacar la influencia del libro de H. D. Ebbinghaus, J. Flum y W. Thomas [19], y de estos autores en foma directa durante mis estudios con ellos en la Universidad de Freiburg (Alemania). En algunas partes, en particular, en la presentación del sistema formal deductivo, he seguido el libro de S. Reeves y M. Clarke [33]. En el capítulo sobre complejidad de algoritmos hay clara influencia del libro de M.R. Garey y D.S. Johnson [24].

En este libro se incluye un capítulo sobre Programación en Lógica escrito por Raymond Reiter, profesor del Departamento de Ciencia de la Computación

de la Universidad de Toronto, miembro de Canadian Institute for Advanced Research. Este capítulo fue escrito por él hace un tiempo para un libro sobre lógica en computación que nunca se concretó. El material sólo tuvo circulación interna en la Universidad de Toronto. Agradezco a Ray por aceptar generosamente que lo incluyera en este libro. Yo soy responsable por la traducción al castellano.

Agradezco el importante apoyo que he recibido para la escritura de este libro por parte de la Dirección de Escuela de Ingeniería de la Universidad Católica de Chile. También el apoyo de la Dirección de Docencia de la Universidad Católica de Chile, a través de un grant, ha sido fundamental para la realización de este libro. Agradezco el entusiasta y excelente trabajo de Pamela Castro, (ex–)alumna de computación, quien hizo gran parte de la transcripción a LaTex de mis notas de clases. Agradezco también a mi alumna María Alejandra Delaporte, quien revisó exhaustiva y entusiastamente la versión preliminar, sin embargo, los errores que persistan son de mi entera responsabilidad. Mis reconocimientos también a mi colega Ignacio Lira por proporcionarme macros de LaTex; y a los (ex–)alumnos Felipe Castro, Marcelo Matus, Daniel Hirschberg, y Enrique Pinaud por su ayuda computacional y con LaTex, en particular.

Dr. Leopoldo E. Bertossi D.
Algarrobo, Verano 1994; Cáhuil y Rocas de Santo Domingo, Verano 1995.

CONTENIDO

Capítulo 1

Introducción

En lógica matemática se estudia diversas formas de razonamiento desde un punto de vista abstracto. Se diseña modelos matemáticos capaces de dar cuenta de ciertos fenómenos que observamos, en este caso, los relacionados con los procesos de razonamiento. En este sentido, no hay una diferencia en filosofía con un modelo basado en ecuaciones diferenciales o algebraicas para un fenómeno físico, sin embargo, en nuestro caso, los fenómenos son de naturaleza deductiva y los modelos son de naturaleza simbólica y discreta.

Hay bastante consenso entre los lógicos y científicos de la computación en que el principal objeto de estudio de la lógica en el concepto de "conclusión", más ampliamente, se pretende modelar el proceso a través del cual, a partir de un cúmulo de conocimiento expresado a través de proposiciones de algún tipo, se concluye una nueva proposicón. Se trata de sancionar cuáles son las consecuencias admisibles de una base de conocimiento.

La vida diaria nos muestra que hay diversas formas de sacar conclusiones o de efectuar razo-namientos, y la lógica matemática pretende dar cuenta de ellas. Por ejemplo, existe el razonamiento matemático, pero también existe razonamiento asociado al diagnóstico, y ambos son incompatibles (a veces): si en una prueba de matemática el alumno aplica un razonamiento del tipo "Tengo fiebre. Además, siempre que tengo gripe tengo fiebre. En consecuencia, tengo gripe.", va a sacar una mala nota, ya que no es admisible en matemática el deducir la premisa de una implicación si se dispone del concecuente de ella. Esta forma de razonamiento abductivo no es totalmente compatible con el razonamiento matemático. Otro ejemplo es el siguiente: "1 es número primo, 2 es número primo, 3 es número primo, bueno, entonces todos los números son primos" es un razonamiento con sabor inductivo (que no es lo mismo que el razonamiento por inducción matemática de los primeros curso de álgebra) que no es compatible con el razonamiento matemático. Sin embargo, los razonamientos inductivos son importantes y están en la base del aprendizaje y

formulación de teorías científicas en las cuales se postula leyes generales a partir de la observación (no exhaustiva) del fenomeno en estudio. En consecuencia, tambíen hay que buscar modelaciones del razonamiento inductivo.

Todo modelo matemático puede, en determinadas circunstancias, quedar obsoleto o francamente incorrecto, al no dar cuenta de los fenómenos que pretende describir, ya sea porque no hay cómo formular el fenómeno en los términos esenciales del modelo (sin producir contradicciones), o porque el modelo necesita forzosamente extensiones, o porque falla como herramienta predictiva para los fenómenos en estudio. Esto también ha pasado en lógica matemática. Los ejemplos del párrafo anterior sugieren claramente este hecho. Es así como no hay una sola lógica matemática, de hecho, hay muchas, dependiendo de la naturaleza del razonamiento que se pretende modelar. Hay muchas lógicas y cada una de ella se puede ver como la modelación –o una clase de modelos– de cierto tipo de razonamiento.

Algunas lógicas resultarán ser una extensión de otra, y habrá pares de ellas incompatibles. Hay que notar, sin embargo, que una de las grandes tareas de la inteligencia artificial hoy día tiene que ver con formas de integración de distintas lógicas para su uso en sistemas inteligentes que sean capaces de gatillar la intervención de la lógica adecuada –entre otras posibles– según lo demande la tarea presente y posteriormenete, si es necesario, cambiarse a otra lógica. Los seres humanos poseemos esa capacidad –como esperamos ilustrar pronto, así es que ésta es una realidad que parece atractiva y necesario modelar y simular en inteligencia artificial. No obstante esto, desde el punto de vista matemático y didáctico es preferible mostrar las diversas lógicas por separado, eventualmente construyendo una como extensión de la otra cuando corresponda.

En este libro partiremos con la lógica proposicional, que es capaz de modelar cierto tipo de razonamiento, digamos el razonamiento deductivo matemático con cierto tipo simple de aseveraciones. La razón para esta elección tiene que ver con la importancia del razonamiento matemático y su naturaleza paradigmática. La lógica resultante está bien estudiada, tiene características interesantes. Su formulación, desarrollo y resultados establecen todo un programa que sirve de modelo y motivación para la creación y desarrollo de otras lógicas asociadas a otras formas de razonamiento.

Consistentemente con lo dicho anteriormente, la lógica proposicional tiene sus limitaciones, y será necesario extenderla, pasando a la lógica de predicados de primer orden, en la cual, además, tenemos cuantificadores ("para todo ...", "existe ...") sobre individuos de un dominio y la posibilidad de referirnos

a atributos de ellos. Posteriormente, mencionaremos la lógica de segundo orden, donde hay cuantificadores sobre propiedades de individuos. Finalmente, discutiremos la lógica de la programación en lógica y lógicas para modelar razonamiento con (cierto tipo de) sentido común.

1.1 Algunas Lógicas

A continuación queremos mencionar, de manera introductoria e informal, ciertas formas de razonamiento que efectuamos los seres humanos. En algunos casos, no daremos más que el nombre de una lógica y el contexto en que ella emerge. Esto es porque nuestros propósitos, en este punto, son generar la convicción de la naturaleza y necesidad de distintas lógicas, y generar algunas referencias para el lector. Antes de continuar, conviene destacar que no hay por qué ver a la(s) lógica(s) asociada solamente a procesos dedutivos de seres humanos. Hoy día también hay lógicas asociadas a otros tipos de procesos, en particular, a procesos computacionales, como ejecuciones de programas. Por ejemplo, si cada estado de la ejecución de un programa posee ciertas propiedades expresables en un lenguaje de una lógica, es natural preguntarse cómo se modifican esas propiedades a medida que el programa evoluciona en su ejecución. Podríamos entonces razonar sobre la ejecución del programa, en particular, sobre la terminación de éste, o pensar que hay una lógica subyacente al programa (o a una clase de ellos, o al lenguaje de programación).

Algunas Lógicas y Formas de Razonamiento:

1. **Razonamiento Matemático**. Dan cuenta de él la lógica proposicional, la lógica de predicados de primer orden, y extensiones de ellas. Una regla de deducción típica de ellas es: "De $p \to q$ (léase "$\to$" como "implica") y p, conclúyase q".

2. **Razonamiento con Sentido Común**. Un ejemplo canónico es el siguiente: si (sólo) se nos dice que Piolín es un pájaro, entonces concluimos, o conjeturamos, tal vez provisoriamente, que éste vuela. Estamos concientes de la debatibilidad de esta conclusión, de la cual debamos eventualmente retractarnos a la luz de nueva información, como, por ejemplo, que Piolín, además de ser pájaro, es una avestruz. Hay diversas lógicas que pretenden modelar este tipo de razonamiento.

3. **Razonamiento Abductivo y de Diagnóstico**. También se acostumbra a ver como una forma de razonamiento con sentido común. Podemos

tener una regla del tipo: "De $p \to q$ y q, conclúyase (tomando ciertas precauciones) p".

4. **Razonamiento Inductivo**. Tiene que ver con la conclusión de una ley general a partir de ciertas observaciones. En este contexto existe la lógica inductiva. Este es un tema importante en el área de aprendizaje automático en Inteligencia artificial.

5. **Lógica Difusa**. Su proósito es el de modelar ciertos patrones de razonamiento con incerteza. Tiene que ver con razonamiento con conceptos vagos, como "alto", "viejo", etc., y la asignación de grados de vaguedad a las conclusiones obtenidas. Ha tenido exitosas aplicaciones en control automático.

6. **Lógicas Multivaluadas**. La lógica clásica, por ejemplo, proposicional, considera sólo los valores de verdad "verdadero" y "falso". Sin embargo, se puede pensar en lógicas con más valores de verdad, que tienen, además de los dos anteriores, el valor de verdad "indeterminado". Una tal lógica trivalente ha sido usada para dar semánticas, es decir, significados, a programas en lógica.

7. **Lógica Probabilística**. En este caso, se asigna valores de verdad probabilísticos a las proposiciones.

8. **Lógicas Intuicionista y Constructivista**. Ella sirven para modelar el razonamiento matemático constructivo, según el cual, por ejemplo, una conclusión del tipo "existe un objeto ..." es admisible en la medida que se ha dado una prueba constructiva del objeto, o un método para construirlo. Según esta forma de razonar, una conclusión del tipo "*p o no p*" no es automáticamente, tautológicamente, aceptable –como en la lógica proposicional. Es aceptable en la medida que haya una demostración de p o una demoatración de *no p*.

9. **Lógicas Modales**. Modelan el razonamiento con modalidades. El caso clásico es el de las modalidades "necesario" y "posible". Si las denotamos con los operadores L y M, respectivamente, y p es una proposición, tenemos también aseveraciones de la forma Lp y Mp, que significan "necesariamente p" y "posiblemente p". Una tal lógica usualmente va tener entre sus axiomas lógicos, entre lo que se acepta como verdadero por simple lógica, proposiciones de la forma $Lp \to p$, $L(p \to q) \to (Lp \to Lq)$, $Mp \leftrightarrow \neg L \neg p$ (léanse "$\leftrightarrow$" como una doble implicación y "$\neg$" como "no",

respectivamente). Por ejemplo, el último axioma relaciona ambos operadores y dice que es lo mismo decir "es posible p" que decir que "no es necesario que no p".

En ciencia de computación han surgido diversas lógicas modales, por ejemplo, lógica dinámica y lógica temporal (que tiene modalidades como $\Box$ ("para siempre"), $\diamond$ ("finalmente"), $\circ$ ("siguiente"), $\mathcal{U}$ ("hasta que")), que permiten razonar sobre la evolución de un programa computacional. También en el área de representación de conocimiento en inteligencia se han introducido lógicas con las modalidades de creencia B (por "belief") y conocimiento K (por "knowledge"). Una forma de formular nuestra manera de razonar por defecto con Piolín, como vimos arriba, es a través de la proposición "$\mathit{Es\text{–}pájaro\text{–}Piolín} \wedge \neg B \neg\ \mathit{vuela\text{–}Piolín} \rightarrow \mathit{vuela\text{–}Piolín}$" (léase "$\wedge$" como "y"). Es decir, si Piolín es pájaro y no creemos que no vuela, entonces concluimos que vuela. *

10. **Lógica Paraconsistente**. Esta lógica permite modelar formas de razonamiento en pre-sencia de cierto tipo de inconsistencias, sin que todo colapse o trivialice, como en el caso del razonamiento matemático y la lógica proposicional: en la presencia de una contradicción, cualquier conclusión es válida.

11. **Lógica Deóntica**. Es una lógica asociada al razonamiento legal con los conceptos de "deber" y "obligación". También ha cobrado importancia en inteligencia artificial, donde se quiere automatizar algunos aspectos del razonamiento legal.

La lista anterior no pretende ser exhaustiva, pero debería permitirnos concluir siguieente moraleja: **Hay diversas formas de razonamiento y podemos esperar que surjan lógicas ad hoc que las modelen.**

1.2 El Programa para una Lógica

Cabe preguntarse qué es una lógica, cuáles son sus elementos constituyentes y cómo se desarrolla.

Podemos decir, de acuerdo con lo anterior que sus objetos de estudio son las proposiciones y las formas de relacionarlas a través de alguna noción de "consecuencia" o proceso deductivo. Surgirán diversas lógicas, dependiendo del tipo de proposiciones que maneja y el tipo de conclusiones que valida.

*Véase el artículo de Raymond Reiter, "Nonmonotonic Reasoning", en el segundo volumen de los Annuals Reviews in Computer Science [2].

Simplificando, se puede identificar el siguiente "programa" ideal para la formulación y desarrollo matemático de una lógica:

1. Sintaxis del Lenguaje:

   En este punto determinamos el tipo de proposiciones formales que maneja la lógica. Esto se hace a nivel simbólico, dando las reglas sintácticas que permiten reconocer y construir proposiciones admisibles.

2. Semántica:

   En este punto definimos de manera precisa el significado de los constituyentes del lenguaje formal, la noción de verdad, y la noción de consecuencia lógica. Esto se logra recurriendo a contextos, mundos, realidades, ..., donde se interpreta los símbolos del lenguaje y otras construcciones sintácticas.

3. Sistema Deductivo:

   En el punto anterior se ha definido de manera semántica la noción central de "consecuencia lógica", de "qué se concluye de qué". Sin embargo, dada la naturaleza formal, simbólica, de las proposiciones, aparece como una meta natural y deseada, el capturar esa noción de manera puramente sintáctica, formal. Esto permitiría hacer deducciones y determinar conclusiones admisibles a través de la manipulación sintáctica de las proposiciones aceptadas en una determinada base de conocimiento.

4. Correspondencia entre Sintáxis y Semántica:

   Aquí se trata de establecer matemáticamente la correspondencia exacta entre las formulaciones semántica y sintáctica de la noción de consecuencia. La esperanza es que ellas coincidan, de modo que el sistema deductivo formal no permita deducir conclusiones indeseadas (corrección), y que éste sea lo suficientemente poderoso como para derivar todas las conclusiones semánticamente admisibles (completidad).

5. Implementación Computacional:

   Como personas interesadas en ciencia de computación, nos ponemos naturalmente la meta de implementar computacionalmente el sistema deductivo formal. De este modo tenemos la esperanza de poder razonar automatizadamente con las proposiciones de nuestra base de conocimiento. Esta aspiración se ve reforzada por el hecho que la base de conocimiento está formada por proposiciones de naturaleza simbólica, lo que las hace fácilmente representables en el computador.

Como dijimos a la partida, éste es un programa ideal. Puede ser que se invierta el orden, en el sentido que se parte con un sistema formal deductivo en lugar de una modelación semántica del concepto de consecuencia. También es posible que no se haya encontrado la versión alternativa, ya sea semántica o sintáctica. O, más impactante, que se pueda demostrar matemáticamente que no puede existir una contraparte sintáctica a la noción semántica de consecuencia (lo que ocurre para ciertas lógicas). Sin embargo, el programa ideal pone en relieve, de todos modos, ciertos puntos a considerar para el desarrolllo de cualquier lógica.

1.3 Lenguaje Objeto vs. Metalenguaje

En computación solemos usar lenguajes formalizados de la lógica matemática. Desde un punto de vista general, ésta es una situación no muy distinta de aquellas en que usamos lenguajes de programación .

Por ejemplo, en PASCAL conocemos algunas reglas sintácticas que nos permiten construir y reconocer expresiones del lenguaje simbólico. De acuerdo con éstas, podemos decir que 'for ¿ do 0 +' es una expresión simbólica que no pertenece al lenguaje PASCAL. Por otro lado, los programas tienen una interpretación, sentido o significado, es decir, hay una **semántica**. En el caso de PASCAL, la semántica es **procedural**, es decir, está relacionada con el efecto que produce el programa sobre sus estructuras de datos. Rara vez nos basamos en una sintáxis precisa de PASCAL, más bien reconocemos que una expresión pertenece al lenguaje por su efecto.

En el caso de la lógica matemática (y otros contextos donde se trata lenguajes formales), las reglas sintácticas para sus diversos lenguajes formales son muy claras. También la semántica de los lenguajes de la lógica matemática es muy precisa, y es usualmente **denotacional**, es decir, las interpretaciones se dan en términos de objetos externos que son denotados (nombrados) en el lenguaje formal. Esta especial característica de la sintaxis y la semántica de los lenguajes de la lógica formal posibilitan su tratamiento matemático. Para estos efectos, nos referimos a un lenguaje de la lógica, o a un conjunto de ellos, en los términos de otro lenguaje, por ejemplo, del lenguaje matemático usual o de un lenguaje natural como el castellano. De este modo, el lenguaje particular de la lógica pasa a ser el **lenguaje objeto** y el lenguaje en que hablamos sobre este lenguaje objeto pasa a ser el **metalenguaje**.

Esta situación también se da naturalmente cuando estudiamos un lenguaje extranjero como el alemán, en nuestro idioma usual, el castellano. Cuando en clase de alemán se explica "En alemán, si dicen 'Ich moechte ein Bier,

bitte', están pidiendo una cerveza", se está describiendo el alemán (el lenguaje objeto) en catellano (el metalenguaje). Lo mismo ocurre cuando nos referimos al lenguaje PASCAL en otro lenguaje.

Es fácil imaginar que podemos tener un metalenguaje formalizado para referirnos a un lenguaje objeto también formalizado. Esta situación se da en lógica matemática, en el área de compiladores, y también en metaprogramación. En metaprogramación en PROLOG, de hecho, lenguaje objeto y metalenguaje básicamente coinciden. Un ejemplo más claro donde lenguaje objeto y metalenguaje coinciden se da en una clase de castellano dictada en castellano.

Como dijimos, en metaprogramación en computación, la aparición de lenguajes objeto y metalenguaje no es extraña. La metaprogramación presenta un contexto uniforme para dar cuenta de los llamados metaprogramas, que tratan a otros programas computacionales como datos de entrada. Estos metaprogramas incluyen, por ejemplo, compiladores, "debuggers", editores, intérpretes, transformadores de programas y simuladores.

Recientemente, la metaprogramación se ha convertido en un tema de interés práctico y teórico. El lenguaje en que está escrito el metaprograma es llamado usualmente metalenguaje, mientras que el lenguaje del programa que es dato de entrada para el metaprograma se llama lenguaje objeto. Esta es una definición de amplio espectro, segúna la cual lenguajes como el FORTRAN, COBOL o LISP pueden jugar el papel de lenguajes objeto. Por otro lado, el lenguaje de descripción del compilador es un ejemplo particular de un metalenguaje para compiladores.

En la mayoría de los lenguajes objetos familiares, los metalenguajes correspondientes son objetos diferentes y complicados, difíciles de escribir, mantener y comprender. Por esta razón, es importante el caso de metaprogramación, cuando ésta es fácil de escribir, verificar y de considerar las implicaciones de los metaprogramas, en particular, cuando metalenguaje y lenguaje objeto son idénticos, o el priemero es una extensión no esencial del segundo. En este caso, se incluye el caso de los llamados intérpretes metacirculares, es decir, intépretes de un lenguaje que son escritos en el lenguaje que se está interpretando. En programación en lógica, esto se hace posible a través de un lenguaje de "amalgamación", que incluye predicados de demostrabilidad, como "solve" (para resolver una cláusula). Este lenguaje de amalgamación es, a la vez, una extensión del lenguaje objeto del PROLOG puro y un metalenguaje de programación en lógica. En programación en lógica, especialmente, es posible escribir intérpretes metacirculares muy concisos.

Como vemos, en general, puede haber varios niveles de lenguaje, y en este

contexto, un mismo lenguaje puede ser a la vez lenguaje objeto y metalenguaje.

Es especialmente gratificante la situación en que tenemos un lenguaje objeto con una sintáxis precisa, ojalá dada a través de reglas que definen y permiten construir exactamente todas las expresiones del lenguaje. Esta formalización del lenguaje es deseable por varios motivos:

- para la distinción fácil entre lenguaje objeto y metalenguaje;
- para clarificar y evitar ambigüedades;
- para manipular el lenguaje de manera puramente simbólica, sin pensar permanentemente en los significados de las expresiones. Por ejemplo, cuando manipulamos el lenguaje del álgebra, al procesar formalmente ecuaciones, no estamos permanentemente pensando en el significado de los símbolos (de hecho, en esta posibilidad radica la potencia del álgebra);
- para abrir posibilidades al manejo computacional del lenguaje.

En lo que sigue, veremos una familia de lenguajes formales concretos, los de la lógica matemática, incluyendo, entre otros, a los de la lógica proposicional y a los de la lógica de predicados. Estos serán lenguajes objeto que serán estudiados en un metalenguaje que es el castellano con inserciones del lenguaje de la matemática usual, en particular, de la teoría intuitiva de conjuntos.

Capítulo 2

Lógica Proposicional

2.1 Sintáxis de un Lenguaje Proposicional

Partimos con un conjunto de símbolos (alfabeto) formado por:

1. **conectivos lógicos:** $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
2. **símbolos de puntuación**:), (
3. un conjunto fijo P de **letras (variables) proposicionales**.

Usualmente P es finito, por ejemplo, $P = \{p, q, r, s\}$. Sin embargo, puede ser infinito enumerable, por ejemplo, $P = \{p_1, p_2, p_3, \ldots\}$.

Con estos símbolos podemos formar expresiones de un lenguaje de la lógica proposicional. Los símbolos en 1. y 2. siempre están en cada lenguaje particular, pero los de P, en 3., varían según la aplicación. En consecuencia, el lenguaje depende de P y lo denotamos con $L(P)$.

Tenemos que definir el lenguaje formal $L(P)$. Supongamos entonces que P está fijo.

Definición: El lenguaje $L(P)$ está formado exactamente por aquellas palabras, o fórmulas, obtenidas mediante una cantidad finita de aplicaciones de las siguientes reglas:

1. Cada letra proposicional de P es una fórmula. A estas fórmulas las llamamos "atómicas".
2. Si φ es una fórmula, entonces $\neg\varphi$ también es una fórmula.

3. Si φ, ψ son fórmulas, entonces $(\varphi * \psi)$ es una fórmula (donde $*$ representa uno cualquiera de los conectivos lógicos: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$).

Esta definición amerita algunas observaciones:

- La definición de $L(P)$ es **inductiva**. En esa definición, 1. es el caso básico de la inducción; 2. y 3. son los pasos inductivos. Veremos en lo que sigue muchas definiciones inductivas de este tipo.
- Esta definición inductiva permite implementar fácilmente programas inductivos para **generar** fórmulas y programas **recursivos** para **reconocer** fórmulas.
- En esta definición, φ, ψ son variables del metalenguaje (metavariables) para denotar o referirnos a fórmulas del lenguaje objeto. En estricto rigor, no son fórmulas del lenguaje objeto, sin embargo, en muchas ocasiones diremos simplemente "la fórmula φ ... ".
- En 2. y 3., las expresiones $\neg\varphi$, $(\varphi \wedge \psi)$ pueden ser vistas como esquemas para generar fórmulas. En un programa computacional para este fin, φ y ψ corresponderían a variables del programa.
- Hasta este punto, los símbolos $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ [*] no tienen significado alguno. Cuando introduzcamos la semántica de los lenguajes proposicionales, les daremos su significado subentendido esperado, a saber, los de negación lógica, conjunción, disyunción, implicación y bicondicional, respectivamente.

Ejemplo: Si $P = \{p, q, r\}$, entonces son fórmulas de $L(P)$, entre otras: $(p \wedge q)$, $(p \vee \neg q)$, $(r \rightarrow r)$.

■

Convención: A veces, para simplificar la escritura de ciertas fórmulas, omitiremos algunos paréntesis, por ejemplo, en lugar de $((p \wedge q) \rightarrow r)$, escribimos $p \wedge q \rightarrow r$.

Los lenguajes de la lógica proposicional pueden ser usados para la tarea de **representación de conocimiento**, que define una de las áreas importantes

[*] En algunas publicaciones se usa los símbolos $\supset$ y $\equiv$ en lugar de $\rightarrow$ y $\leftrightarrow$, respectivamente.

de inteligencia artificial y otras áreas de la computación. Más precisamente, el problema es el de representar conocimiento en el computador y usarlo en sistemas computacionales. En nuestro caso, la representación sería hecha a través de los lenguajes de la lógica simbólica.

Ejemplo: Tomemos un simple ejemplo ya canónico del área de diagnóstico en inteligencia artificial. Sea $P = \{llovioAnoche, pastoMojado, calleMojada, regadorEstuvoPrendido, pastoMojadoYbrillante, zapatosMojados\}$.

Por supuesto, hay toda una semántica subyacente a las letras proposicionales elegidas, que nos inspira en la creación de nuestra base de conocimiento sobre el mundo, pero que no tomaremos en cuenta en un manejo puramente simbólico del conocimiento. Una posible representación de conocimiento sobre un dominio particular de aplicación está dada a través de las siguientes fórmulas del lenguaje proposicional:

$llovioAnoche \rightarrow pastoMojado$
$llovioAnoche \rightarrow calleMojada$
$regadorEstuvoPrendido \rightarrow pastoMojado$
$pastoMojado \rightarrow pastoMojadoYbrillante$
$pastoMojado \rightarrow zapatosMojados$

Todo este conocimiento causal puede ser representado fácilmente en el computador y de hecho, se puede escribir programas que sean capaces de **razonar** sobre la base de este conocimiento general sobre el dominio y posibles observaciones adicionales que pueden ser representadas en el mismo lenguaje. Por ejemplo, podríamos tener la observación *zapatosMojados* y buscar explicaciones para ella, o tener *regadorEstuvoPrendido*, y buscar consecuencias de ella.

■

Más adelante, abordaremos el problema de razonamiento deductivo (y otras formas de razonamiento) a partir de un conjunto de fórmulas de la lógica proposicional. Por el momento, destaquemos que la lógica proposicional provee lenguajes que pueden ser usados para representación de conocimiento, es decir, para crear bases de conocimiento, en las cuales el conocimiento almacenado está escrito en un lenguaje de la lógica proposicional. Nótese que no es parte de esta lógica la elección del lenguaje apropiado ni de la representación propiamente tal *. Esto está fuera de la lógica y depende del usuario, el que, en

*Aunque podríamos pensar en crear una "lógica de la representación de conocimiento", pero nos pasamos a otro problema.

definitiva, va a determinar el nivel de granularidad de la representación (¿cuáles son los hechos que consideramos básicos?, ¿cuáles son las fórmulas atómicas de partida?, ¿cuáles son y cómo se representan leyes y relaciones causales?, etc.) y va a tener necesarimente que adquirir algunos compromisos ontológicos y epistemológicos, es decir, que tienen que ver con la naturaleza del dominio en descripción y con el conomocimiento que tenemos de él, respectivamente.

2.1.1 Inducción sobre Fórmulas

La definición inductiva del lenguaje $L(P)$ tiene algunas consecuencias interesantes:

- Podemos usar inducción para definir conceptos asociados a fórmulas del lenguaje
- Podemos usar inducción para demostrar propiedades generales de las fórmulas del lenguaje

Esta situación no es nueva. La hemos enfrentado en el contexto de los números naturales, donde tenemos **el axioma de inducción matemática**. Este dice:

Para toda propiedad $P(\cdot)$:

$$[\ P(0) \quad \text{y} \quad \text{para todo } k\text{: } (\ P(k) \Rightarrow P(k+1))] \quad \Rightarrow \quad \text{para todo } n\text{: } \quad P(n)\ ^*$$

Este axioma nos permite probar que todos los números naturales tienen una cierta propiedad P. Para ello, hay que verificar el caso básico, $P(0)$, y el paso inductivo, $P(k) \Rightarrow P(k+1)$). De este modo se suele demostrar, por ejemplo, la fórmula $1 + \cdots + n = \frac{n(n+1)}{2}$. Para esto, basta definir la propiedad (o atributo) de números naturales $P = \{n \in \mathbb{N} \mid n \neq 0 \text{ y } 1 + \cdots + n = \frac{n(n+1)}{2}\}$ y demostrar, usando el axioma de inducción, que todos los números naturales la tienen (o pertenecen a ella).

También es posible dar definiciones basadas en inducción. Un ejemplo típico es el de la definición recursiva de la función factorial:

1. $0! = 1$
2. $(n+1)! = n! \cdot (n+1)$

*Usamos $\Rightarrow$ para la implicación del metalenguaje, digamos, del lenguaje matemático usual. En contraste, $\rightarrow$ lo usamos sólo como símbolo del lenguaje objeto.

El axioma de inducción permite demostrar que esta definición es correcta, es decir, que hay una única función de números naturales que satisface las ecuaciones 1. y 2.

Las ecuaciones 1. y 2. anteriores constituyen una **especificación** de la función factorial. En ella, valga la redundancia, especificamos nuestros requerimientos sobre la función. Ya sabemos que esta especificación funciona bien, en el sentido que las restricciones que plantea fuerzan la existencia de una única función que las satisface, en la medida que tengamos, además, el principio de inducción. La especificación difiere de un **programa** para la función factorial, en el sentido que la primera no dice cómo calcularla. Sin embargo, no es difícil derivar o sintetizar un programa para calcular la función factorial a partir de la especificación.

El principio de inducción de los naturales nos permite ver a $\mathbb{N}$ como un **dominio constructible** a partir de los constructores "libres" 0 y la función sucesor (sumar 1): los elementos de este dominio se construyen partiendo del 0 y sumando 1 a elementos ya construidos. Otra forma de ver esto es la siguiente: en lugar de usar la suma de 1, podemos formular el axioma de inducción a través de la función sucesor $S : \mathbb{N} \to \mathbb{N}$, que a cada número natural asocia su número sucesor en el orden creciente. El axioma, con una cuantificación universal implícita sobre propiedades P, toma la forma:

$$[\ P(0) \quad \text{y} \quad \text{para todo } k\text{: } (\ P(k) \Rightarrow P(S(k)))] \quad \Rightarrow \quad \text{para todo } n\text{: } \quad P(n).$$

Este axioma de inducción obliga a que los números naturales sean exactamente: $0,\ S(0),\ S(S(0)),\ S(S(S(0))), \ldots$*. Se puede ver, entonces, al conjunto de los naturales como un dominio constructible a partir de los constructores 0 y S. Del mismo modo, el conjunto de fórmulas de un lenguaje proposicional se puede ver como un dominio constructible.

A continuación, damos dos ejemplos de inducción en el contexto de un lenguaje $L(P)$.

Ejemplo: Queremos definir el largo de una fórmula de $L(P)$ por inducción en la construcción (o estructura) de la fórmula. Es decir, la función que nos da el largo, $largo(\varphi)$, de fórmulas arbitrarias φ:

*Otros requerimientos que se imponen usualmente a la función sucesor, aparte del especificado por el axioma de inducción, dicen que el 0 no es sucesor de ningún número, y que la función es uno a uno o inyectiva. Esta axiomatización será discutida en detalle en capítulos posteriores.

1. Caso Básico: Si φ es fórmula atómica (letra proposicional), entonces definimos $largo(\varphi) = 1$.

2. Pasos Inductivos:

   - Si φ es $\neg\psi$, definimos: $largo(\varphi) = largo(\psi) + 1$.
   - Si φ es $(\psi * \chi)$, definimos: $largo(\varphi) = largo(\psi) + largo(\chi) + 3$, donde $*$ representa uno cualquiera de los conectivos lógicos: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.

■

Ejemplo: Demostrar que todas las fórmulas de $L(P)$ tienen el mismo número de paréntesis izquierdos y derechos.

Demostración: Sea φ una fórmula arbitraria de $L(P)$. Demostramos la propiedad para φ por inducción en la estructura de la fórmula:

Caso Básico: Si φ es atómica, entonces tiene 0 paréntesis izquierdos y 0 paréntesis derechos: $\#pi(\varphi) = 0 = \#pd(\varphi)$. *

Pasos Inductivos:

1) Si φ es $\neg\psi$: $\#pi(\varphi) = \#pi(\psi) = \#pd(\psi) = \#pd(\varphi)$.

En esta cadena de igualdades, la primera y la última se obtienen de la construcción de φ, y la del medio, de la hipótesis de inducción, que suponemos verdadera para la fórmula más simple ψ.

2) Si φ es $(\psi * \chi)$, tenemos las hipótesis de inducción: $\#pi(\psi) = \#pd(\psi)$ y $\#pi(\chi) = \#pd(\chi)$.

Entonces:

$$\#pi(\varphi) = 1 + \#pi(\psi) + \#pi(\chi) = 1 + \#pd(\psi) + \#pd(\chi) = \#pd(\varphi).$$

■

2.1.2 Ejercicios

1. Demuestre por inducción en la construcción (estructura) de la fórmula que ningún segmento inicial propio de una fórmula es una fórmula.

*Como en el ejemplo anterior, se puede definir funciones $\#pi(\cdot)$ y $\#pd(\cdot)$ que dan el número de paréntesis izquierdos y el número de paréntesis derechos de una fórmula, respectivamente.

2. Demuestre por inducción (en la estructura de la fórmula o en el largo de la fórmula) que si φ es una fórmula y si $\varphi \equiv \neg\psi \equiv \neg\psi'$, entonces ψ es fórmula, y $\psi \equiv \psi'$. Aquí "$\equiv$" significa identidad como palabras.

3. (a) Defina por inducción en la construcción de una fórmula proposicional el número de apariciones de variables proposicionales en la oración.

(b) Demuestre que el largo de una fórmula proposicional (el número de símbolos) está acotado superiormente por un polinomio en el número de apariciones de variables proposicionales en la oración. Discuta bajo qué condiciones esta aseveración es verdadera.

2.2 Semántica de Lenguajes Proposicionales

Hasta ahora hemos tratado los lenguajes de la lógica proposicional de manera solamente sintáctica, formal, sin atribuir significado a sus símbolos, y sin preocuparnos de la verdad o falsedad de las aseveraciones expresadas en estos lenguajes.

Podemos decir que la semántica tiene tres roles importantes: 1. Proporcionar la noción de significado o interpretación, 2. Precisar el concepto de verdad, y 3. Definir el concepto de consecuencia lógica, es decir, de definir cuándo una proposición se puede concluir a partir de otras proposiciones del lenguaje.

La semántica de estos lenguajes es muy simple. Esta tiene que pasar necesariamente por dar una interpretación a los conectivos lógicos, es decir, a $\neg$, $\wedge$, $\vee$, $\rightarrow$ y $\leftrightarrow$. Estos tendrán la interpretación esperada. Además, en estos lenguajes tenemos las letras proposicionales, que son codificaciones en el lenguaje objeto de proposiciones completas, cerradas, y que, en consecuencia, pueden ser verdaderas o falsas. Por ejemplo, la letra proposicional *pastoMojadoYbrillante* codifica naturalmente la proposición "El pasto está mojado y brillante" (también podría codificar otra proposición muy distinta), la cual puede ser verdadera o falsa según el dominio que estemos describiendo.

Entre otras preguntas que nos ayudaría a enfrentar una definición precisa de la semántica de los lenguajes proposicionales, tenemos la de reconocer cuándo un argumento, expresable en el lenguaje objeto, es un argumento válido.

Por ejemplo, ¿es válido el argumento: "Si Sócrates es un hombre, entonces Sócrates es mortal. Y Sócrates es un hombre. En consecuencia, Sócrates es mortal"? Si lo es, ¿por qué? Más precisamente, este argumento puede ser formalizado partiendo con dos letras proposicionales p y q que **denotan** a

las proposiciones "Sócrates es un hombre" y "Sócrates es mortal", respectivamente. En el lenguaje proposicional $L(\{p, q\})$, el argumento se escribe a través de la fórmula: $(((p \to q) \wedge p) \to q)$. Entonces nos preguntamos por qué este argumento simbólico es válido.

Dado un lenguaje proposicional $L(P)$, definimos su semántica declarando a las letras proposicionales en P como verdaderas o falsas. Cada una de ellas toma exactamente uno de esos dos valores de verdad, que denotamos con 1 o 0, respectivamente (a veces, con V y F). Esto tiene sentido, ya que, como dijimos, las letras proposicionales denotan proposiciones, hechos, aseveraciones que están fuera del lenguaje objeto. Esta asignación de valores de verdad se da a través de una función:

Definición: Una **asignación de verdad** es una función σ de P en el conjunto $\{0, 1\}$. En el lenguaje usual de teoría de conjuntos, escribimos: $\sigma : P \to \{0, 1\}$.

Ejemplo: Consideremos $P = \{p, q, r\}$. Son asignaciones (de verdad), entre otras,

$\sigma_1 : p \mapsto 1,\ q \mapsto 0,\ r \mapsto 1.$ *

$\sigma_2 : p \mapsto 0,\ q \mapsto 1,\ r \mapsto 1.$

■

Observaciones:

1. Las asignaciones también se llaman **"valuaciones"**.
2. Si P contiene n letras proposicionales, entonces hay 2^n valuaciones distintas.
3. Es característico de la lógica proposicional clásica el que toda letra proposicional en P tiene exactamente un valor de verdad entre dos posibles. Esto queda reflejado en la definición a través del hecho que σ es una función total, es decir, su dominio es todo P, y su recorrido es $\{0, 1\}$. Es posible concebir otras "lógicas" en las cuales ciertas letras proposicionales no tengan un valor asignado, o tengan un tercer valor, por ejemplo, I, por "indeterminado". De este modo, surgen "lógicas trivalentes", o, más generalmente, "multivaluadas". Si no decimos lo contrario, nos restringiremos a lógica proposicional clásica bivalente.

*Siguiendo con la notación conjuntista, $p \mapsto 1$ significa que a la letra p, σ_1 le asigna el valor 1. En otros términos, la imagen de p con respecto a la función σ_1 es el 1.

El problema que tenemos que enfrentar ahora es el siguiente: dado un lenguaje proposicional $L(P)$ y una asignación σ, ¿cómo asignamos valores de verdad a las proposiciones más complejas, construidas a partir de las letras de P. Es decir, a todas las fórmulas de $L(P)$. Esto requiere de una extensión del dominio de σ desde P hasta $L(P)$ (recordar que P es un subconjunto de $L(P)$). En el ejemplo anterior, ¿cuál es el valor de verdad de la fórmula $(p \vee q) \rightarrow p$ con respecto a σ?

La definición general se hace, como es de esperar, por inducción sobre las fórmulas de $L(P)$. Es en este punto donde daremos, por primera vez, el significado esperado a los conectivos lógicos.

Definición: Consideremos un lenguaje proposicional $L(P)$ y una asignación $\sigma : P \rightarrow \{0,1\}$. Definimos $\sigma : L(P) \rightarrow \{0,1\}$ (notar que usamos la misma letra, σ, para la función extendida) de la siguiente manera: dada una fórmula arbitraria φ de $L(P)$, el valor de verdad $\sigma(\varphi)$ es:

1. Caso Básico: Si $\varphi \in P$ (fórmula atómica), entonces $\sigma(\varphi)$ es el valor $\sigma(\varphi)$ que φ tenía originalmente como elemento del subconjunto P.
2. Pasos Inductivos:

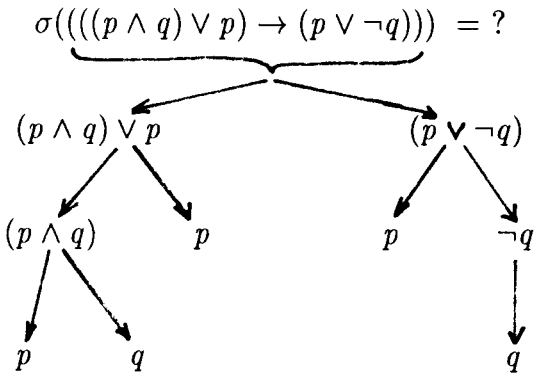
- Si φ es $\neg\psi$, entonces $\sigma(\varphi) = 1 - \sigma(\psi)$. Es decir, el valor de verdad de φ es 1 o 0 según el valor de verdad de ψ sea 0 o 1, respectivamente.
- Si φ es $(\psi \vee \chi)$, entonces $\sigma(\varphi) = max\{\sigma(\psi), \sigma(\chi)\}$.
- Si φ es $(\psi \wedge \chi)$, entonces $\sigma(\varphi) = min\{\sigma(\psi), \sigma(\chi)\}$.
- Si φ es $(\psi \rightarrow \chi)$, entonces $\sigma(\varphi) = 0$ si $\sigma(\psi) = 1$ y $\sigma(\chi) = 0$. En caso contrario, toma el valor 1.

  Es decir, la implicación es falsa sólo cuando el antecedente φ es verdadero y el consecuente χ, falso.
- Si φ es $(\psi \leftrightarrow \chi)$, entonces $\sigma(\varphi) = 1$ si $\sigma(\psi) = \sigma(\chi)$. En caso contrario, toma el valor 0.

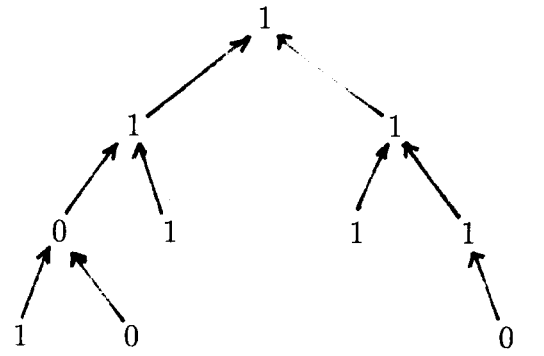
Esta definición puede ser representada a través de las conocidas **tablas de verdad**:

φ	$\neg\varphi$	ψ	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
1	0	1	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	1	0	0	0	1	1

Ejemplo: Sea $P = \{p, q\}$. Consideremos la asignación σ tal, que: $\sigma(p) = 1$ y $\sigma(q) = 0$. Entonces la extensión de σ a todo $L(P)$ asigna valores de verdad a todas las fórmulas de $L(P)$. Por ejemplo, el valor de verdad $\sigma((((p \wedge q) \vee p) \rightarrow (p \vee \neg q)))$ puede ser obtenido en forma descendente (top–down, en inglés), descomponiendo recursivamente la fórmula original en sus subcomponentes:



A continuación podemos evaluar la fórmula de manera ascendente (bottom–up, en inglés), propagando los valores desde las hojas del árbol hacia arriba según la definición de la extensión de σ. Fácilmente se obtiene para la fórmula original el valor de verdad 1:



■

Ahora que sabemos asignar valores de verdad a todas las fórmulas de un lenguaje proposicional, podemos ver que las mismas observaciones que hicimos con respecto a la asignación de valores de verdad a las letras proposicionales son aplicables en este caso. En particular, dada una valuación, toda fórmula del lenguaje tiene como valor de verdad exactamente uno de los valores 0 o 1. No hay otros valores de verdad posibles y cada fórmula tiene exactamente un valor de verdad. Esto se puede demostrar por inducción. En particular, para cualquier asignación σ, una fórmula de la forma $(\varphi \vee \neg\varphi)$ * siempre toma el valor de verdad 1.†

El hecho que, para toda valuación σ, se tenga $\sigma(\varphi \vee \neg\varphi) = 1$, es equivalente a que, para toda valuación σ se tenga: $\sigma(\varphi) = 1$ o $\sigma(\neg\varphi) = 1$. Esto nos indica que siempre φ es verdadera o $\neg\varphi$ es verdadera (pero no ambas). En consecuencia, toda fórmula φ queda determinada en cuanto a su valor de verdad con respecto a una valuación. Esto se da a pesar de que eventualmente no tengamos ninguna pista (ni demostración) sobre la validez de ninguna de las dos en particular. En este sentido, nuestra lógica es "clásica", en contraste con algunas "lógicas no clásicas" que dan a una fórmula como $(p \vee \neg p)$ el valor 1 en la medida que haya evidencia en favor de la verdad de p o en favor de la verdad de $\neg p$ (por ejemplo, una demostración para alguna de las dos).

Como ya hemos advertido, nos concentraremos principalmente en el tratamiento de lógicas clásicas. Sin embargo, es necesario destacar que lógicas no clásicas, por ejemplo, trivalentes, aparecen en distintos contextos en ciencia de computación. Aparecen explícitamente en inteligencia artificial en modelación de razonamiento con incerteza, y en programación en lógica a través de semánticas para programas en lógica.

Destaquemos finalmente que los conectivos lógicos $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ tienen una interpretación fija, determinada por la lógica, no así las letras proposicionales que tienen significado (dado a través de un valor de verdad) variable.

2.2.1 Conectivos Lógicos como Funciones

Los conectivos lógicos pueden ser vistos como funciones que envían valores (o pares de valores) de verdad en valores de verdad. Por ejemplo, la negación lógica puede ser vista como la función:

---

*$(\varphi \vee \neg\varphi)$ no es exactamente una fórmula, pues mezcla metavariables con símbolos del lenguaje objeto. Por esto decimos "una fórmula de la forma $(\varphi \vee \neg\varphi)$". Informalmente podríamos decir simplemente "la fórmula $(\varphi \vee \neg\varphi)$"

†Para ver esto basta considerar los dos casos posibles; aquel en que σ asigna a φ el valor 1, y el caso en que σ asigna a φ el valor 0. En ambos casos, se obtiene valor 1 para la fórmula.

$\neg : \{0,1\} \to \{0,1\}$

$0 \mapsto 1,\ 1 \mapsto 0.$

El conectivo $\wedge$ puede ser visto como la función:

$\wedge : \{0,1\} \times \{0,1\} \to \{0,1\}$

$(0,1) \mapsto 0,\ (0,0) \mapsto 0,\ (1,0) \mapsto 0,\ (1,1) \mapsto 1.$

Notemos que hay $2^4 = 16$ funciones distintas de $\{0,1\} \times \{0,1\}$ en $\{0,1\}$, es decir, podríamos definir 16 funciones distintas de dos argumentos binarios con resultado binario, o, lo que es lo mismo, 16 tablas de verdad distintas para conectivos lógicos $*$ con dos argumentos. Para verificar esto, obsérvese que se puede dar dos valores posibles, 0 o 1, a cada uno de los cuatro pares en $\{0,1\} \times \{0,1\}$.

Aquí hemos introducido sólo cuatro de los 16 conectivos lógicos posibles. A pesar de eso, algunos de éstos son redundantes en el sentido que pueden ser definidos a partir de los otros y la negación, por medio de composiciones usuales de funciones.

Por ejemplo, los conectivos $\wedge$, $\to$ y $\leftrightarrow$ pueden ser definidos a partir de $\neg$ y $\vee$:

1. $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$.
2. $(\varphi \to \psi) = (\neg\varphi \vee \psi)$.
3. $(\varphi \leftrightarrow \psi) = ((\varphi \to \psi) \wedge (\psi \to \varphi))$.

La conjunción $\wedge$ en 3. puede ser eliminada usando 1. El símbolo de igualdad en 1. – 3. tiene el sentido usual de igualdad de funciones, es decir, las funciones son iguales siempre y cuando tengan las mismas imágenes (o, en otros términos, den los mismos valores) para los mismos argumentos.

Cuando los posibles argumentos son una cantidad finita y pequeña, la igualdad se puede establecer verificando la igualdad de valores para todos los posibles argumentos. Una manera de ordenar y representar este proceso de verificación consiste en recurrir a la comparación de las tablas de verdad, con el propósito de observar si ambas tablas producen las mismas columnas de valores.

La igualdad 1. se muestra en la tabla siguiente:

φ	ψ	$\varphi \wedge \psi$	$\neg(\neg\varphi \vee \neg\psi)$
1	1	1	1
1	0	0	0
0	1	0	0
0	0	0	0

Por ser todos los conectivos lógicos introducidos inicialmente definibles en términos de, por ejemplo, $\neg$ y $\vee$ (ambos simultáneamente), decimos que el conjunto de conectivos $\{\neg, \vee\}$ es **funcionalmente completo**.

Por último, digamos que dos fórmulas son **lógicamente equivalentes** si exactamente las mismas valuaciones las hacen verdaderas. Por ejemplo, $(p \rightarrow q)$ y $(\neg p \vee q)$ son lógicamente equivalentes.

2.2.2 Ejercicios

1. ¿Cuántos functores–de–verdad con n argumentos hay?

2. Se define los conectivos lógicos:

$|\ (x, y) := \neg(x \wedge y), \quad x, y \in \{0, 1\}$ (no simultáneamente x e y)

$NOR(x, y) := \neg x \wedge \neg y$ (ni x ni y)

Demuestre que cada uno de los conjuntos $\{|\}$ y $\{NOR\}$ es funcionalmente completo.

3. Verifique si las siguientes equivalencias se cumplen:

(a) $(\varphi * \varphi) \leftrightarrow \varphi$ (idempotencia)

(b) $(\varphi * \psi) \leftrightarrow (\psi * \varphi)$ (conmutatividad)

(c) $(\varphi * (\psi * \chi) \leftrightarrow ((\varphi * \psi) * \chi)$ (asociatividad)

Aquí, $*$ puede ser $\wedge, \vee, \leftrightarrow, \rightarrow$.

4. Demuestre las siguientes equivalencias:

(a) $(\varphi \wedge \neg\varphi) \leftrightarrow \bot$, donde $\bot$ es una variable proposicional especial que siempre toma el valor falso f.

(b) $(\varphi \vee \neg\varphi) \leftrightarrow \top$, donde $\top$ es una variable proposicional especial que siempre toma el valor verdadero t. Esta es la llamada "ley del tercero excluido" o

"tertium non datur".

5. (a) Una fórmula proposicional está en **forma normal conjuntiva (FNC)** si es de la forma: $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} l_{ij}$, donde l_{ij} es un "literal", es decir, una variable proposicional o una negación de variable proposicional. Por ejemplo, $(\neg p \vee q) \wedge (p \vee q \vee \neg r)$ está en FNC.

Demuestre que toda fórmula proposicional $\varphi \in L(P)$ es lógicamente equivalente a una fórmula proposicional ψ que está en FNC.

Ind.: Demuéstrelo por inducción en φ.

(b) Defina **forma normal disyuntiva (FND)** y formule y demuestre las proposiciones correspondientes a (a).

2.2.3 Fórmulas Válidas

Una de las preguntas que queríamos responder a través de la semántica para los lenguajes proposicionales era: ¿cuándo es una fórmula válida? o ¿ cuándo es un argumento válido?

Definición: Decimos que una fórmula φ de $L(P)$ es **válida** siempre y cuando es siempre verdadera, es decir, si y sólo si *, **para toda** valuación $\sigma : P \to \{0, 1\}$: $\sigma(\varphi) = 1$.

Vemos, entonces, que una fórmula es válida cuando toda asignación de verdad la hace verdadera. A las fórmulas válidas también las llamamos **tautologías**.

Ejemplo: La fórmula $((p \to q) \wedge p) \to q$ es válida. Toda asignación de valores de verdad a p y q hacen que la fórmula completa tome el valor 1. Esto se puede verificar exhaustivamente a través de una tabla de verdad:

p	q	$p \to q$	$(p \to q) \wedge p$	$(p \to q) \wedge p) \to q$
1	1	1	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	1

El ejemplo de Sócrates que vimos antes es de este tipo. Ese argumento puede ser representado como una fórmula válida. ■

*Este "si y sólo si" del metalenguaje a veces lo escribiremos como $\Leftrightarrow$. No hay que confundir este bicondicional del metalenguaje con el bicondicional $\leftrightarrow$ del lenguaje objeto.

Ejemplo: Nuestra semántica hace que la fórmula

$$(HayMuchosAlumnosEnIIC2212 \wedge HaceCalorEnInvierno) \rightarrow HaceCalorEnInvierno$$

también sea válida. Sin embargo, la fórmula

$$(HayMuchosAlumnosEnIIC2212 \vee HaceCalorEnInvierno) \rightarrow HaceCalorEnInvierno,$$

no lo es.

■

Una noción más débil, pero también natural e importante, es la de satisfacibilidad:

Definición: Decimos que una fórmula φ de $L(P)$ es **satisfacible** siempre y cuando **existe** una valuación $\sigma : P \rightarrow \{0, 1\}$ tal, que $\sigma(\varphi) = 1$.

En otros términos, una fórmula es satisfacible cuando existe alguna interpretacion, algún mundo, donde ella es verdadera. En cambio, para que una fórmula sea válida, pedimos que sea verdadera en todos los mundos posibles, bajo todas sus posibles interpretaciones. Toda fórmula válida es satisfacible, pero la afirmación recíproca no es verdadera.

Ejemplo: La fórmula $((p \rightarrow q) \wedge q) \rightarrow p$ es satisfacible, pero no es válida. Esto lo muestra claramente su tabla de verdad, en la cual hay un renglón que termina en 1, pero no todos los renglones tienen esta propiedad.

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge q$	$(p \rightarrow q) \wedge q) \rightarrow p$
1	1	1	1	1
1	0	0	0	1
0	1	1	1	0
0	0	1	0	1

■

Cuando una fórmula no es satisfacible, decimos que ella es **insatisfacible**. Por ejemplo, una fórmula de la forma $(\varphi \wedge \neg\varphi)$ es insatisfacible. Ninguna asignación la hace verdadera. Ella es siempre falsa. También decimos que es una **contradicción**.

Con respecto a estos conceptos tenemos los siguientes hechos:

1. φ es satisfacible si y sólo si $\neg\varphi$ no es válida.
2. φ es insatisfacible si y sólo si $\neg\varphi$ es válida.
3. φ es válida si y sólo si $\neg\varphi$ es insatisfacible.

Vemos que los conceptos de validez y satisfacibilidad están relacionados a través de la negación lógica. Los hechos anteriores pueden ser demostrados fácilmente recurriendo a las definiciones ahí involucradas.

Por último, notemos que dos fórmulas son lógicamente equivalentes si y sólo si la doble implicación de ambas es una tautología. Por ejemplo, $(p \to q) \leftrightarrow (\neg p \vee q)$ es una tautología.

2.2.4 Ejercicios

1. Determine si las oraciones proposicionales siguientes son tautología, contradicción o satisfacible:

(a) $(p \to (q \to p))$

(b) $((p \leftrightarrow \neg q) \vee q)$

(c) $((p \to (q \leftrightarrow r)) \to ((p \to q) \to (p \to r)))$

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

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

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

2. Demuestre que $(\varphi_1 \vee \varphi_2) \to \varphi$ es lógicamente equivalente a $(\varphi_1 \to \varphi) \wedge (\varphi_2 \to \varphi)$.

2.3 Conjuntos de Fórmulas

En la sección anterior vimos conceptos relativos a fórmulas aisladas. Estos conceptos pueden ser extendidos naturalmente a casos en que tenemos conjuntos de fórmulas. Para esto, basta definir qué significa que una asignación σ hace verdadero, o satisface, a un conjunto de fórmulas Σ. Esto ocurre siempre

y cuando σ hace verdadera a cada una de las fórmulas del conjunto. Esto se denota así: $\sigma \models \Sigma$, y decimos que σ es un modelo de Σ. Entonces:

$$\sigma \models \Sigma : \Leftrightarrow \quad \text{para toda } \varphi \in \Sigma : \quad \sigma(\varphi) = 1.^*$$

Una forma alternativa de definir la noción de satisfacción de un conjunto de fórmulas pasa por la extensión del concepto de valor de verdad de una fórmula al de valor de verdad de un conjunto de fórmulas: $\sigma(\Sigma) := min\{\sigma(\varphi) : \varphi \in \Sigma\}$. Entonces podemos definir: $\sigma \models \Sigma \quad :\Leftrightarrow \quad \sigma(\Sigma) = 1$.

Como una fórmula se puede ver como el conjunto formado sólo por ella, en lo que sigue, a veces, escribiremos $\sigma \models \varphi$, queriendo decir, $\sigma \models \{\varphi\}$. Esto último es lo mismo que decir $\sigma(\varphi) = 1$.

Notemos que si Σ es un conjunto finito de fórmulas, entonces $\sigma \models \Sigma$ siempre y cuando σ hace verdadera a la conjunción de las fórmulas en Σ.

Tal como en el caso de fórmulas aisladas, un conjunto de fórmulas Σ puede ser válido, satisfacible o insatisfacible, dependiendo de si toda, alguna o ninguna asignación satisface a Σ, respectivamente*. Los conjuntos insatisfacibles de fórmulas también se llaman "inconsistentes".

Ejemplos:

1. El conjunto de fórmulas $\Sigma = \{p \wedge q, \neg q \vee p\}$ es satisfacible, ya que para la asignación $\sigma_0 : p \mapsto 1, q \mapsto 1$ se tiene: $\sigma_0 \models \Sigma$. Sin embargo, Σ no es válido, pues la signación $\sigma_1 : p \mapsto 1, q \mapsto 0$ no satisface Σ. Esto último lo escribimos así: $\sigma_1 \not\models \Sigma$.

2. El conjunto de fórmulas $\{p \wedge (q \vee \neg r), \neg q, r\}$ es insatisfacible, ya que, para toda valuación σ, se tiene $\sigma \not\models \{p \wedge (q \vee \neg r), \neg q, r\}$. Esto, a pesar de que todas las fórmulas del conjunto son satisfacibles por separado.

3. El conjunto $\{p, \neg p\}$ es insatisfacible.

4. El conjunto $\{p, p \to q, \neg q\}$ es insatisfacible. ■

*Las notaciones $=$ y $:\Leftrightarrow$ significan que lo que está a la izquierda de $=$ o $\Leftrightarrow$ se está definiendo en términos de lo que está a la derecha del símbolo correspondiente. Toda esta es notación en el metalenguaje.

*Se entiende, cuando hablamos de asignaciones, estando fijo el lenguaje objeto, digamos $L(P)$, que estas asignaciones son compatibles con el lenguaje, es decir, que son funciones de P en $\{0, 1\}$.

Las afirmaciones en los ejemplos anteriores pueden ser verificadas fácilmente a través de tablas de verdad.

2.4 Teorías, Bases de Conocimiento, Especificaciones

Conjuntos de fórmulas surgen naturalmente al querer representar conocimiento en un lenguaje de la lógica formal, y reciben distintos nombres según el área. En matemática y ciencia básica, en general, usualmente se habla de **teorías**, en inteligencia artificial, de **bases de conocimiento**, y en otras áreas de ciencia de computación, por ejemplo, ingeniería de software, se habla de **especificaciones**. En cualquiera de estas situaciones, lo que tenemos es cierto conocimiento declarativo sobre cierto(s) dominio(s).

En matemática, tendremos usualmente un conjunto de axiomas iniciales, por ejemplo, los de la geometría, definiciones de nuevos conceptos, y teoremas que se desprenden de los axiomas. Todo esto escrito a través de fórmulas. En ciencia de computación, el conjunto de fórmulas puede especificar un procedimiento, es decir, puede describir el comportamiento subentendido de un algoritmo. Esa especificación puede posteriormente ser realizada o implementada computacionalmente a través de un programa en algún lenguaje de programación particular [*].

En este libro usaremos indistintamente los tres términos. Haremos, además, la convención de que teorías, bases de conocimientos y especificaciones, son satisfacibles. Es decir, son verdaderas en algún mundo o domino, al menos. Como veremos más adelante, la insatisfacibilidad de una teoría tiene que ver, de acuerdo con la lógica que estamos presentando, con la existencia de contradicciones internas en la teoría. En general, no parece deseable la representación de conocimiento a través de bases de conocimientos contradictorias [†].

Es importante destacar que, a través de la lógica proposicional, estamos mod-

[*]Existe una tendencia creciente en computación a separar nítidamente las especificaciones de las implementaciones. Las primeras tienen un sentido declarativo y las segundas, un sentido procedural. Un problema importante en computación es el de generar automáticamente un implementación a partir de una especificación, o en otros términos, el de hacer a una especificación, ejecutable.

[†]Sin embargo, hay ciertas publicaciones recientes en inteligencia artificial que presentan aplicaciones de lógicas, como la lógica paraconsistente, que modelan algunas formas de razonamiento con inconsistencias y todavía obteniendo resultados interesantes. En lógica clásica como la que estamos mostrando hasta ahora, cualquier situación de inconsistencia reduce todo el proceso a algo trivial y carente de interés: a la luz de conocimiento inconsistente cualquier aseveración puede ser concluida. Esto se verá en detalle en la sección 2.7.

elando ciertas formas de razonamiento que efectuamos los seres humanos. En este sentido, la lógica proposicional es una abstracción, un modelo, de algo que percibimos como una realidad, en este caso, la de ciertos procesos deductivos[‡]. Estos procesos deductivos son los que aparecen en algunas formas de razonamiento cuotideano y en matemática usual. Sin embargo, está claro, desde los puntos de vista de la lógica filosófica y de la inteligencia artificial, que esta lógica es insuficiente para modelar o capturar muchas formas de razonamiento que los seres humanos también efectuamos. Por ejemplo, razonamiento con sentido común, razonamiento con incerteza, razonamiento con inconsistencias, razonamiento para aprendizaje inductivo, razonamiento abductivo o de diagnóstico, y otros procesos mentales (y de computación) estarían regidos por sus propias lógicas, y éstas requieren de una modelación matemática correspondiente.

De hecho, a través de la historia de la matémática, también han surgido proposiciones en favor de otras lógicas, por ejemplo, lógica intuicionista, distintas de la que estamos presentando, que reflejarían mejor cierta forma de práctica matemática "constructivista". En matémática constructivista, una disyunción de la forma $\varphi \vee \neg\varphi$ no sería aceptada como verdadera por la sola forma lógica de la proposición. Se aceptaría como verdadera en la medida en que se cuente con una demostración de φ o una demostración de $\neg\varphi$.

A pesar de la discusión anterior, el modelo deductivo que presentaremos en este capítulo será fundamentalmente el de la lógica proposicional clásica. Este constituye un núcleo de muchos procesos deductivos y un buen paradigma de modelación de este tipo de procesos.

Cualquier lógica, para modelación de cualquier forma de razonamiento, debe dar cuenta de la relación entre el conocimiento almacenado en una base de conocimiento y las consecuencias que se puede desprender de ella. La lógica debe sancionar qué es lo que se puede concluir de una teoría, validando así nuevo conocimiento obtenido a partir del original. Esta relación es la que queremos modelar a través de la noción de **consecuencia lógica**.

Por todos los antecedentes vertidos en esta sección, debería estar claro que puede haber más de una versión del concepto de consecuencia lógica, dependiendo de la lógica que pretendamos modelar. En la sección siguiente precisaremos esta noción para el caso de la lógica proposicional clásica.

[‡]En este sentido, no habría mayor diferencia de fondo entre una modelación de una realidad física a través de un sistema de ecuaciones diferenciales y las leyes del análisis, y la modelación matemática, a través de la lógica formal, de las leyes del razonamiento.

2.5 La Noción de Consecuencia Lógica

En esta sección caracterizaremos aquellas proposiciones formales que son consecuencia de (son "implicadas" por) un conjunto de proposiciones que representan cierto conocimiento. El definir este concepto es la tarea principal de cualquier lógica. Como vimos en el primer capítulo, para distintas formas de razonar y de concluir, hay que definir esta noción de manera ad hoc. En esta sección nos concentraremos en el caso de la lógica proposicional.

2.5.1 El Caso de la Lógica Proposicional

La noción de consecuencia lógica es una relación entre un conjunto de fórmulas y una fórmula particular. Intuitivamente, una fórmula es una consecuencia lógica de un conjunto de fórmulas siempre y cuando, cada vez que el conjunto de fórmulas es verdadero, necesariamente también la fórmula particular es verdadera. Como esta noción apela al concepto de verdad de fórmulas, ella es una noción semántica.

Definición: Sea Σ un conjunto de fórmulas de $L(P)$ y φ una fórmula de $L(P)$. Decimos que φ **es consecuencia lógica de** Σ si, para cada valuación $\sigma : P \to \{0,1\}$: $\quad \sigma \models \Sigma \quad \Rightarrow \quad \sigma \models \varphi$.

Notación: $\Sigma \models \varphi$

Observaciones:

- A través de este concepto estamos modelando la idea de que una fórmula es "implicada" por la base de conocimiento Σ.
- De acuerdo con esta definición, para verificar si una fórmula es consecuencia lógica de un conjunto de fórmulas, hay que verificar la verdad de esa fórmula sólo con respecto a las valuaciones que hacen verdadero al conjunto de fórmulas (que no necesariamente son todas las valuaciones que admite el lenguaje).
- La notación recién introducida, $\Sigma \models \varphi$, puede prestarse para confusiones con la notación $\sigma \models \varphi$. La primera denota una relación entre un conjunto de fórmulas y una fórmula particular, es decir, entre dos objetos sintácticos; en cambio, la segunda denota una relación entre una valuación y una fórmula. A pesar de este peligro inicial, mantendremos estas notaciones, pues están ya fuertemente establecidas en la literatura.

- Otra forma de ver la relación $\Sigma \models \varphi$ es la siguiente: no es posible que Σ sea verdadera y φ, falsa.

Ejemplos:

1. $\{p\} \models p$.

2. $\{p \to q, p\} \models q$.

En efecto: Si σ es una valuación, entonces $\sigma \models \{p \to q, p\} \Rightarrow \sigma \models (p \to q)$ y $\sigma \models p$.

Para que el consecuente de esta (meta)implicación sea verdadero, es necesario que $\sigma(p) = 1$ y $\sigma(q) = 1$. Luego, $\sigma \models q$.

3. $\{p \to q, q\} \not\models p$.

En efecto: basta exhibir una valuación que haga verdadera a $\{p \to q, q\}$, pero que no haga verdadera a p. La valuación $\sigma : p \mapsto 0, q \mapsto 1$ tiene esta propiedad. Esto se observa el la fila destacada, la segunda, de la siguiente tabla:

p	q	$p \to q$	$(p \to q) \wedge q$
0	0	1	0
0	**1**	**1**	**1**
1	0	0	0
1	1	1	1

4. $\{p, \neg p\} \models q$.

En efecto: lo que hay que establecer es que, para toda valuación $\sigma : \{p, q\} \to \{0, 1\}$, la siguiente implicación es válida: $(\sigma \models p$ y $\sigma \models \neg p) \Rightarrow \sigma \models q$.

Como el antecedente de esta implicación es falso (una valuación no puede satisfacer simultáneamente a p y a $\neg p$), toda la implicación es verdadera. Esta forma de razonamiento, a metanivel, fue modelada precisamente a través de la tabla de verdad de la implicación $\to$.

Otra forma de establecer 4. consiste en verificar que en la siguiente tabla de verdad no hay ninguna asignación que hace verdadera al conjunto de fórmulas $\{p, \neg p\}$, y no, a la fórmula q.

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

El hecho anterior, que puede ser generalizado al caso de fórmulas complejas en la forma $\{\varphi, \neg\varphi\} \models \psi$, es destacable: cualquier fórmula ψ es consecuencia lógica del conjunto inconsistente de fórmulas $\{\varphi, \neg\varphi\}$.

5. $\{(p \vee \neg r) \rightarrow (q \vee r), \neg q\} \models (\neg r \rightarrow \neg p)$

Esto se puede verificar en la siguiente tabla de verdad, observando las filas destacadas, en las cuales el conjunto de fórmulas de la izquierda se hace verdadero:

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

2.5.2 Ejercicios

1. Verifique cuáles de las relaciones de consecuencia lógica es verdadera:

(a) $\{p \rightarrow q\} \models (\neg p \rightarrow \neg q)$

(b) $\{p \rightarrow q\} \models q$

(c) $\{(p \vee q) \rightarrow r, \neg r\} \models \neg p$

(d) $\varnothing \models ((p \rightarrow q) \rightarrow p) \rightarrow p$ ($\varnothing$ es el conjunto vacío de fórmulas)

(e) $\models (p \rightarrow (\neg q \rightarrow F)) \rightarrow (p \rightarrow q)$

(f) $p \models (\neg q \rightarrow F) \rightarrow (p \rightarrow q)$

(g) $p \models (\neg q \rightarrow F) \rightarrow (p \rightarrow q)$

(h) $\{p, \neg q\} \models (F \to (p \to q)$

(i) $\models ((p \wedge \neg q) \to F) \to (p \to q)$

En los cinco últimos ejercicios, se entiende que F es una fórmula contradictoria cualquiera (por ejemplo, $(p \wedge \neg p)$).

2.6 Lógica y Práctica Matemática

Hasta ahora hemos estado modelando ciertos aspectos de la lógica, especialmente de la que usamos en razonamiento matemático. Es natural entonces tratar de relacionar este modelo con la práctica matemática usual. Para ello, tomemos una teoría matemática relativamente bien conocida, la de los espacios vectoriales.

Sea Σ el conjunto de axiomas para la teoría de espacios vectoriales. No nos preocuparemos aquí de la forma en que son escritos esos axiomas en un lenguaje de la lógica formal. De hecho, los lenguajes de la lógica proposicional no son lo suficientemente expresivos como para escribir estos axiomas en su forma usual. Sin embargo, igual podemos ilustrar algunas ideas en este contexto informal.

Un teorema conocido, denotémoslo con φ, de la teoría de espacios vectoriales dice que: "Una transformación lineal es uno a uno (inyectiva) si y sólo si su kernel es $\{0\}$". Este es un teorema de la teoría porque cada vez que los axiomas de espacio vectorial en Σ son verdaderos, es decir, cada vez que tenemos un espacio vectorial particular, también el teorema φ es verdadero, es decir, es verdadero en ese espacio vectorial. En otros términos, tenemos que φ es consecuencia lógica de Σ. Esto lo denotamos así: $\Sigma \models \varphi$.

Notar que no estamos caracterizando esta noción de teorema de una teoría en términos explícitamente deductivos. Los aspectos deductivos aparecen en la práctica matemática cuando queremos demostrar la aseveración $\Sigma \models \varphi$. En ese proceso, aparecen ciertas reglas de deducción, ciertas leyes de la lógica, que nos gustaría capturar y modelar explícitamente a nivel objeto. De este modo, en lugar de efectuar el proceso matemático y semántico usual que establece que $\Sigma \models \varphi$, podríamos pensar en proceder sólo a nivel objeto, deduciendo formalmente, sintácticamente, simbólicamente, la proposición φ a partir del conjunto de axiomas Σ. Esto proporcionaría una contraparte sintáctica a la noción semántica de consecuencia lógica. Eso es algo que presentaremos en secciones posteriores.

Consideremos ahora la teoría de grupos. Esta parte usualmente de tres axiomas que dicen:

- La operación es asociativa.
- Existe un elemento neutro para la operación.
- Existen inversos con respecto a la operación para todos los elementos.

Sea Σ este conjunto de axiomas. Consideremos ahora otra proposición φ que dice que "la operación es conmutativa".

La proposición φ no es un teorema de la teoría de grupos, es decir, $\Sigma \not\models \varphi$. ¿Cómo se establece esto usualmente en matemática? Exactamente como hemos mostrado en secciones anteriores: exhibiendo un grupo, es decir, un mundo donde son verdaderos los axiomas de Σ, que no es conmutativo, es decir, que no satisface φ.

2.7 Algunos Resultados sobre Consecuencia Lógica

A continuación presentaremos algunos teoremas sobre la lógica proposicional. Estos son teoremas del tipo matemático usual, expresados en el metalenguaje, sobre los lenguajes objeto de la lógica y las nociones asociadas a ellos que hemos ido definiendo. Todos ellos pueden ser demostrados fácilmente a partir de las definiciones involucradas.

1. Monotonía: Sean $\Sigma \subseteq \Sigma_1 \subseteq L(P)$, y $\varphi \in L(P)$. Se tiene: $\Sigma \models \varphi \Rightarrow \Sigma_1 \models \varphi$.

Este teorema dice que, al agregar nuevas fórmulas a un conjunto de fórmulas, las consecuencias lógicas del conjunto inicial siguen siendo consecuencias lógicas del conjunto expandido. En otros términos, conclusiones previas no son invalidadas por nueva información agregada a la base de conocimiento. Esta es una propiedad característica de esta lógica, y, de hecho, la que impide usarla tal como está en el modelamiento de razonamiento con sentido común, en el cual es frecuente que tengamos que retractarnos de ciertas conclusiones a la luz de nueva información. Sin embargo, en razonamiento matemático usual, nunca nos retractamos de teoremas ya establecidos aunque agreguemos nuevos axiomas a la teoría. Por ejemplo, todos los teoremas de la teoría de grupos siguen siendo parte de la teoría de grupos conmutativos, que se obtiene al agregar, a los axiomas iniciales, el axioma que habla de la conmutatividad de la operación. Se obtendrá nuevos teoremas, además de todos los antiguos.

Como dijimos, esto último contrasta con el razonamiento basado en sentido común, según el cual se concluye provisionalmente ciertas proposiciones que

posiblemente van a ser rechazadas posteriormente, a la luz de nueva información. Un ejemplo típico es el siguiente: si se nos dice solamente que Piolín* es un pájaro, entonces concluimos que (o actuamos como si) Piolín vuela. Posiblemente se nos diga posteriormente que Piolín es una avestruz, con lo cual la conclusión anterior quedará invalidada.

Algo similar ocurre con las bases de datos. Si preguntamos si tal objeto tiene tal propiedad, el sistema administrador de la base de datos responderá NO si no encuentra almacenado en la base el hecho que el objeto tiene el atributo. Notar que esta conclusión se basa en una forma de razonamiento por sentido común: al no encontar la información positiva, supone que el hecho es falso. Esta suposición se llama la "suposición del mundo cerrado"*. Conclusiones obtenidas sobre la base de esta suposición son frecuentemente no monótonas, pues pueden ser rechazadas al ingresar nueva información a la base de datos.

Demostración de 1.: Supongamos que $\Sigma \models \psi$. Sea ahora σ una asignación tal, que $\sigma \models \Sigma_1$. En particular, $\sigma \models \Sigma$. Entonces, por la hipótesis, $\sigma \models \psi$.

2. $\Sigma \subseteq L(P)$ es insatisfacible $\Leftrightarrow$ para toda $\varphi \in L(P)$: $\Sigma \models \varphi$.

Es decir, de una base de conocimiento proposicional inconsistente se puede concluir cualquier afirmación. En presencia de inconsistencia, la noción de consecuencia lógica de trivializa.

Demostración: La demostración se basa en el argumento dado en el ejemplo 4. de la sección 2.5.1. Hay que verificar que: para toda asignación σ, si $\sigma \models \Sigma$, entonces $\sigma \models \varphi$. Como el antecedente de esta implicación, a saber, "$\sigma \models \Sigma$", es falso (por ser Σ inconsistente), la implicación se hace verdadera (para todo σ).

3. Para toda contradicción $F \in L(P)$ (una fórmula insatisfacible) y conjunto de fórmulas Σ, se tiene: Σ es insatisfacible $\Leftrightarrow$ $\Sigma \models F$.

Este teorema dice que un conjunto de fórmulas es inconsistente si y sólo si se puede obtener como consecuencia lógica de ella, una contradicción fija, por ejemplo, una fórmula de la forma $(\varphi \wedge \neg\varphi)$.

Demostración: ($\Rightarrow$) Sea Σ inconsistente. Hay que probar que, para toda σ,

$$(\sigma \models \Sigma \quad \Rightarrow \quad \sigma \models F).$$

*Introducimos a Piolín para familiarizar al lector con la mascota del área de representación lógica de conocimiento en inteligencia artificial.

*Esta suposición fue enunciada por primera vez por Raymond Reiter, y en inglés se llama "Closed World Assumption".

Esta aseveración es trivialmente verdadera porque "$\sigma \models \Sigma$" es falsa (para toda σ).

($\Leftarrow$) Supongamos que $\Sigma \models F$. Supongamos, razonando por contradicción, que Σ es consistente. Entonces existe σ tal, que $\sigma \models \Sigma$. Por la hipótesis, se tiene $\sigma \models F$. Esto no es posible, y hemos llegado a una contradicción en nuestro razonamiento a metanivel. En consecuencia, la suposición de que Σ es consistente es falsa.

4. Sea $\varphi \in L(P)$ y $\varnothing$, el conjunto vacío de fórmulas. Se tiene: φ es válida $\Leftrightarrow$ $\varnothing \models \varphi$.

Es decir, φ es una tautología si y sólo si es consecuencia lógica del conjunto vacío de fórmulas. En otros términos, si y sólo si se puede obtener sin premisas, axiomas o hipótesis. En este sentido, es una aseveración verdadera por "lógica pura".

Notación: En lugar de $\varnothing \models \varphi$, escribimos $\models \varphi$. Ya sabemos que esto significa que φ es válida.

Ejemplos: a) $\models ((p \wedge (p \wedge q)) \rightarrow q)$. b) $\models (p \vee \neg p)$.

5. Teorema de Deducción: Si $\Sigma \subseteq L(P)$ y $\varphi, \psi \in L(P)$, se tiene:

$$\Sigma \cup \{\varphi\} \models \psi \;\Leftrightarrow\; \Sigma \models (\varphi \rightarrow \psi)$$

Este teorema permite poner las premisas de una implicación como hipótesis en la base de conocimiento o, alternativamente, como antecedentes de lo que se quiere demostrar, sacándolas de la base de conocimiento. Esto permite ir deshaciéndose de hipótesis en la base de conocimiento para ir pasándolas a la derecha, es decir, al teorema (como antecedentes).

Demostración: ($\Rightarrow$) Supongamos que $\Sigma \cup \{\varphi\} \models \psi$. Sea ahora σ tal, que $\sigma \models \Sigma$. Hay que probar que $\sigma \models (\varphi \rightarrow \psi)$. Supongamos que $\sigma \models \varphi$. Entonces $\sigma \models \Sigma \cup \{\varphi\}$. Por la hipótesis inicial, se tiene $\sigma \models \psi$.

($\Leftarrow$) Supongamos ahora que $\Sigma \models (\varphi \rightarrow \psi)$. Sea σ tal, que $\sigma \models \Sigma \cup \{\varphi\}$. Hay que probar que $\sigma \models \psi$. Como $\sigma \models \Sigma$, se tiene $\sigma \models (\varphi \rightarrow \psi)$. De los hechos $\sigma \models \varphi$ y $\sigma \models (\varphi \rightarrow \psi)$ se concluye que $\sigma \models \psi$.

6. Reducción entre Consistencia y Consecuencia Lógica: Si $\Sigma \subseteq L(P)$ y $\varphi \in L(P)$, se tiene:

$$\Sigma \models \varphi \quad \Leftrightarrow \quad \Sigma \cup \{\neg\varphi\} \text{ es inconsistente.}$$

Este teorema es de enorme importancia, pues permite reducir la noción de consecuencia lógica a la noción de inconsistencia (o insatisfacibilidad), y recíprocamente. En una dirección nos dice que si queremos demostrar que una proposición φ es consecuencia lógica de un conjunto de fórmulas, basta con agregar al conjunto la negación de la proposición que queremos probar, e intentar obtener una contradicción a partir del nuevo conjunto de fórmulas. Este teorema está en la base de las demostraciones por contradicción que hacemos usualmente en matemática. Un ejemplo de esto es nuestra demostración de 3., más arriba.

Veremos que este resultado juega un papel clave en demostraciones mecánicas de teoremas. Tanto por razones fundamentales como prácticas. No es difícil imaginar por qué: un demostrador mecánico de teoremas que sea capaz de demostrar directamente (en contraste con el método indirecto de "reducción al absurdo") cualquier proposición φ a partir de cualquier conjunto de fórmulas Σ (siempre que sea la primera consecuencia lógica del segundo) deberá tener programadas heurísticas muy generales, es decir, diferentes métodos o estrategias para, a partir del conjunto Σ, intentar alcanzar el blanco φ, que es distinto en distintas aplicaciones. Sin embargo, al tratar de demostrar mecánicamente una contradicción a partir de $\Sigma \cup \{\neg\varphi\}$ (por el resultado 6., esto bastaría para establecer que $\Sigma \models \varphi$), el demostrador podría concentrarse siempre en un mismo blanco o meta, a saber, una contradicción fija (ver resultado 3., más arriba). Las heurísticas que sería necesario almacenar o programar en el sistema computacional estarían diseñadas para alcanzar siempre la misma fórmula. Indudablemente esto parece más simple de realizar que la primera opción. Entonces la estrategia sería la siguiente: para demostrar una fórmula a partir de una base de conocimiento, se agrega la negación de la fórmula a la base de conocimiento, y se intenta demostrar una contradicción fija y elegida a priori.

Una observación, interesante en sí misma, y que también tiene un papel importante en demostración mecánica de teoremas, es la siguiente:

Si Φ es un conjunto finito de fórmulas, digamos, $\{\phi_1, \ldots, \phi_n\}$, entonces:

- $\sigma \models \Phi \quad \Leftrightarrow \quad \sigma \models (\phi_1 \wedge \cdots \wedge \phi_n)$.
- Φ es (in)consistente si y sólo si $(\phi_1 \wedge \cdots \wedge \phi_n)$ lo es.
- Φ es válido si y sólo si $(\phi_1 \wedge \cdots \wedge \phi_n)$ lo es.

En consecuencia, para establecer que $\Sigma \models \varphi$, cuando Σ es $\{\varphi_1, \ldots, \varphi_n\}$, bastaría con establecer que la fórmula $(\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \neg\varphi)$ es inconsistente. Veremos más adelante que, aunque Σ sea infinito, siempre se puede reducir, al menos en teoría, la verificación de la inconsistencia de $\Sigma \cup \{\neg\varphi\}$, a una verificación finita de este tipo.

Ejemplo: Para verificar que $\{(r \vee p) \rightarrow q \vee \neg s, p, s\} \models q$, basta demostrar que $\{(r \vee p) \rightarrow q \vee \neg s, p, s\} \cup \{\neg q\}$ es inconsistente.

Demostración de 6.: Hay que demostrar las dos direcciones de la implicación "$\Leftrightarrow$".

"$\Rightarrow$": Supongamos que:

(Hipótesis): $\Sigma \models \varphi$.

Hay que probar que: $\Sigma \cup \{\neg\varphi\}$ es inconsistente. Esto lo podemos hacer por contradicción. Supongamos que existe una valuación σ tal, que:

(Suposición): $\sigma \models \Sigma \cup \{\neg\varphi\}$.

Entonces: (1) $\sigma \models \Sigma$ y (2) $\sigma \models \neg\varphi$.

Combinando la (Hipótesis) con (1), obtenemos: $\sigma \models \varphi$. Esto contradice a (2). Luego, la (Suposición) es falsa. *

"$\Leftarrow$": Ahora nuestra hipótesis es:

(Hipótesis): $\Sigma \cup \{\neg\varphi\}$ es inconsistente.

Hay que probar que $\Sigma \models \varphi$, es decir, que, para toda valuación σ: $(\sigma \models \Sigma \Rightarrow \sigma \models \varphi)$.

Sea σ una valuación arbitraria tal, que: (1) $\sigma \models \Sigma$. Hay que probar que $\sigma \models \varphi$. Esto lo hacemos por contradicción:

(Suposición): $\sigma \not\models \varphi$.

De la (Suposición) se obtiene: (2) $\sigma \models \neg\varphi$.

De (1) y (2), obtenemos: $\sigma \models \Sigma \cup \{\neg\varphi\}$. Esto contradice la (Hipótesis). Luego, la (Suposición) es falsa. ■

*Esta demostración que acabamos de hacer es una demostración matemática usual, a metanivel. La pregunta que surge es cómo podemos capturar esta forma de razonar por contradicción a nivel objeto para hacer demostraciones por contradicción sólo a través de manipulación simbólica de fórmulas.

2.7.1 Ejercicios

1. Demuestre los siguientes metateoremas sobre la relación de consecuencia lógica: (como siempre, letras griegas minúsculas denotan oraciones proposicionales, y letras griegas mayúsculas, conjuntos de oraciones)

(a) $\Sigma, \varphi \models \varphi$. (Escribimos Σ, φ en lugar de $\Sigma \cup \{\varphi\}$.)

(b) Si $\Sigma \models \varphi$, entonces $\Sigma, \psi \models \varphi$. Esta es otra forma de la propiedad de monotonía de la lógica proposicional.

(c) Si $\Sigma \models \varphi$ y $\Sigma, \varphi \models \psi$, entonces $\Sigma \models \psi$. Este es el teorema del "corte".

(d) $\Sigma \models \varphi$ y $\Sigma \models \psi$ si y sólo si $\Sigma \models (\varphi \wedge \psi)$. Esta es la propiedad de conclusión de la conjunción.

(e) $\Sigma, \varphi \models \psi$ y $\Sigma, \chi \models \psi$ si y sólo si $\Sigma, (\varphi \vee \chi) \models \psi$.

(f) $\Sigma, \varphi \models \psi$ y $\Sigma, \psi \models \varphi$ si y sólo si $\Sigma \models \varphi \leftrightarrow \psi$.

(g) $\Sigma \not\models \varphi$ si y sólo si $\Sigma \cup \{\neg\varphi\}$ es consistente.

Interprete intuitivamente los hechos anteriores. Hay algunas lógicas, para otras formas de razonamiento, que no tienen algunas de estas propiedades. ¿Se puede imaginar qué tipo de razonamiento podrían éstas modelar?

2. En el resultado 3. de la sección 2.7 se dio una demostración por contradicción. Explique por qué ese tipo de razonamiento a metanivel puede ser modelado en lógica proposicional por los resultados del ejercicio 1.(e) (o (f) o (g) o (h)) de la sección 2.5.2.

2.8 Demostraciones Formales: Resolución

Nos preguntamos si existe algún sistema deductivo formal, simbólico, que permita establecer si $\Sigma \models \varphi$. Hasta ahora sabemos hacer esto de manera semántica, es decir, recurriendo a valuaciones. Más adelante veremos que esto sí es posible, y mostraremos un sistema general de reglas de deducción formales que permite obtener, con pura manipulación simbólica, a φ a partir de Σ (cuando y sólo cuando φ es consecuencia lógica de Σ).

En esta sección enfrentaremos el problema restringido de establecer, de manera puramente simbólica, si un conjunto de fórmulas es inconsistente. Ya sabemos, por la sección anterior, que el problema de establecer consecuencia lógica se puede reducir al de establecer inconsistencia. A pesar de esta equivalencia, el

método es especialmente apropiado para determinar directamente inconsistencia, más que para obtener directamente una fórmula arbitraria a partir de un conjunto arbitrario de fórmulas del cual es consecuencia lógica*.

En esta sección, el método será presentado, principalmente, en sus aspectos procedurales, sin entrar en demasiados detalles sobre sus fundamentos.

El sistema deductivo formal es apropiado para manejar fórmulas de forma sintáctica restringida, las llamadas cláusulas.

Cláusula: es una fórmula de la forma:

$$p_1 \vee \ldots \vee p_r \vee \neg p_{r+1} \vee \ldots \vee \neg p_s,$$

es decir, una disyunción de literales, es decir, de letras proposicionales o negaciones de letras proposicionales.

Ejemplo: $p \vee \neg q \vee r$ es una cláusula.

No es difícil demostrar la siguiente:

Proposición: Todo conjunto de fórmulas proposicionales Σ puede ser transformado en un conjunto de cláusulas Σ' tal, que:

Σ es consistente (inconsistente) si y sólo si Σ' es consistente (inconsistente).

■

El teorema dice, en particular, que Σ y Σ' son **equiconsistentes**, es decir, ambos son simultáneamente satisfacibles o simultáneamente insatisfacibles. De hecho, al menos en el caso de la lógica proposicional, Σ y Σ' en la proposición anterior son lógicamente equivalentes, es decir, son satisfechos exactamente por las mismas asignaciones de verdad. Esta forma más fuerte del teorema no será necesaria para nuestros propósitos.

Ejemplo: El conjunto de fórmulas $\{(r \vee p) \rightarrow (q \vee \neg s), p, s\} \cup \{\neg q\}$ puede ser transformado en un conjunto de cláusulas.

La única fórmula que no es cláusula es la primera fórmula. Pasémosla a forma normal conjuntiva (FNC), es decir a una conjunción de disyunciones de literales (ver sección 2.2.2). Ella es equivalente a la fórmula

*Téngase en cuenta la discusión que sigue al resultado 6. de la sección anterior.

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

Esta fórmula genera dos cláusulas, las dos de la conjunción.

Así, el conjunto original es equiconsistente con el conjunto de cláusulas

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

que es inconsistente si y sólo si el original lo es.

■

Recordemos que una cláusula es una disyunción de literales $l_1 \vee \ldots \vee l_n$. A veces, ésta se escribe como conjunto: $\{l_1, \ldots, l_n\}$. Por ejemplo, $\neg r \vee q \vee \neg s$ se escribe $\{\neg r, q, \neg s\}$.

Con respecto a esta notación y concepción de cláusulas como conjuntos, debemos hacer algunas observaciones:

- Hay que tener cuidado: una asignación de verdad σ satisface una **cláusula** $\{l_1, \ldots, l_n\}$ si satisface **al menos** un literal l_i.
- Una cláusula particular es la **cláusula vacía** $\square$, que no contiene literales. Por el item anterior, $\square$ nunca es satisfacible, es decir, es una contradicción. Esta será nuestra contradicción particular mencionada en la sección anterior.
- Para establecer que un conjunto de cláusulas es inconsistente, vamos a "deducir" la cláusula $\square$.

Definición: Una **demostración (deducción, derivación) formal por resolución** de una cláusula C a partir de un conjunto de cláusulas $\mathcal{C}$ es una sucesión finita de cláusulas $C_1, \ldots, C_n$ tal, que C_n es C y cada C_i:

- es elemento de $\mathcal{C}$, o
- se obtiene de dos cláusulas precedentes en la sucesión a través de una aplicación de la **regla de deducción de resolución**.

La Regla de Resolución:

Motivación: Si una asignación de verdad σ satisface las cláusulas $\neg r \vee s \vee \neg p$ y $\neg q \vee s \vee p$, entonces satisface la cláusula $\neg r \vee s \vee \neg q \vee s$. Es decir, la última es consecuencia lógica de las anteriores.

La Regla: Si $\{l_1, \ldots, l_s\}$ y $\{l'_1, \ldots, l'_t\}$ son cláusulas, donde, digamos, l_s y l'_t son literales complementarios (uno es la negación del otro), entonces pásese a la cláusula $(\{l_1, \ldots, l_s\} - \{l_s\}) \cup (\{l'_1, \ldots, l'_t\} - \{l'_t\})$. Esquemáticamente:

$$\frac{\begin{array}{c} l_1 \vee \ldots \vee l_{s-1} \vee l_s \\ l'_1 \vee \ldots \vee l'_{t-1} \vee l'_t \end{array}}{\bigvee(\{l_1, \ldots, l_s\} - \{l_s\}) \vee \bigvee(\{l'_1, \ldots, l'_t\} - \{l'_t\})}$$

- Esta es una regla sintáctica, formal. Se pasa de dos expresiones simbólicas a una nueva expresión simbólica.
- No es necesario hacer chequeos semánticos (aunque haya una motivación semántica).
- La regla se puede implementar computacionalmente.

Ejemplos:

(a) $\neg p \vee q \vee s$

$\underline{\neg p \vee \neg q \vee \neg s}$

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

(b) $\neg p \vee q \vee s$

$\underline{\neg p \vee \neg q \vee \neg s}$

$\neg p \vee s \vee \neg p \vee \neg s$

(c) p

$\underline{\neg p}$

$\square$

(d) $p \vee q$

$\underline{\neg p \vee \neg q}$

$q \vee \neg q$

(e) $p \vee q$

$\underline{\neg p \vee \neg q}$

$p \vee \neg p$

■

Una estrategia muy general para determinar la inconsistencia, consiste en aplicar sucesivamente la regla de resolución para llegar a una situación como la de (c), donde, al aplicar por última vez la regla, se obtiene la cláusula vacía.

En (d) y (e) no podemos cancelar simultáneamente p y q, pues obtendríamos la clásula vacía $\square$ a partir de $p \rightarrow q$ y $q \rightarrow p$, es decir, a partir de $p \leftrightarrow q$, que no es una fórmula contradictoria.

- La última cláusula en una demostración formal por resolución se llama "conclusión".
- Si la conclusión es la cláusula vacía, decimos que la demostración formal es una **refutación**.

Como **método general**, si queremos establecer que una fórmula φ es una consecuencia lógica de un conjunto de fórmulas Σ, es decir, $\Sigma \models \varphi$:

1. Agregamos $\neg\varphi$ a Σ.
2. Se genera un conjunto de cláusulas $\mathcal{C}$ que sea equiconsistente (o lógicamente equivalente) con $\Sigma \cup \{\neg\varphi\}$.
3. Se intenta obtener la cláusula $\square$ a partir de $\mathcal{C}$ por medio de una refutación formal basada en resolución.

Así, hemos propuesto un método sintáctico que pretende capturar el concepto semántico de inconsistencia. Este puede ser implementado computacionalmente a través de diversas estrategias. Sin embargo, no hemos justificado

el método en sentido que queda pendiente la pregunta: Dado un conjunto de cláusulas $\mathcal{C}$, ¿se tiene siempre que $\mathcal{C}$ es inconsistente si y sólo si hay una refutación formal por resolución a partir de $\mathcal{C}$?

Ejemplo (continuación): Demostrar que si Σ es $\{r \vee p \rightarrow q \vee \neg s, p, s\}$ y φ es q, entonces $\Sigma \models \varphi$.

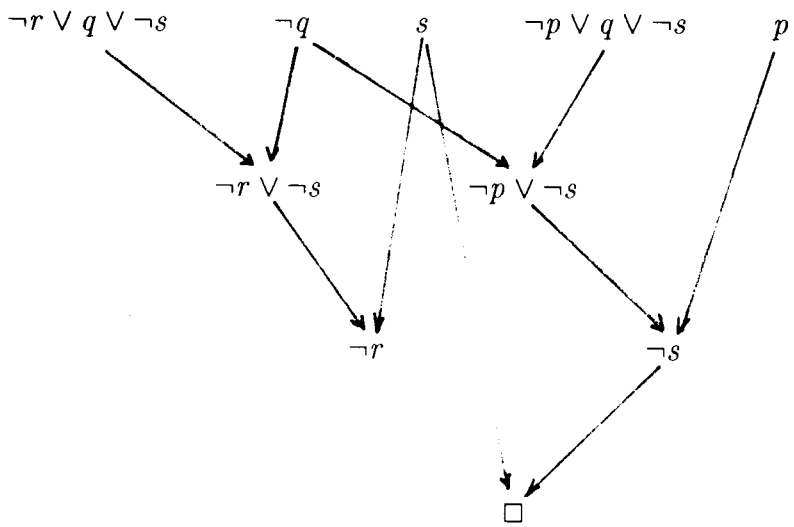
Una posibilidad consiste en proceder semánticamente de manera directa o indirecta, chequeando inconsistencia a través de valuaciones.

El método alternativo de este capítulo consiste en establecer inconsistencia sintácticamente, mediante resolución. Ilustraremos el método:

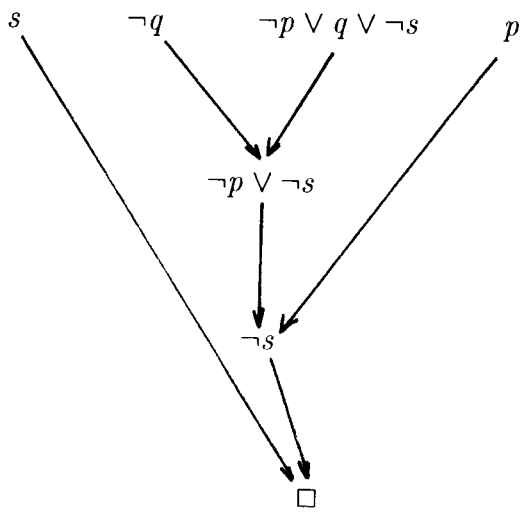
Primero, nos pasamos al conjunto $\mathcal{C}$ de cláusulas:

$$\{\neg r \vee q \vee \neg s, \neg p \vee q \vee \neg s, p, s, \neg q\}.$$

Y ahora demostramos su inconsistencia a través de una refutación por resolución a partir de $\mathcal{C}$:



Este diagrama muestra demostraciones fallidas (la que lleva a $\neg r$), y exitosas (la que lleva a $\square$). De él podemos extraer el siguiente **árbol de refutación**:



En este árbol de refutación, las cláusulas del primer nivel, las de partida, son las hojas o nodos terminales del árbol, $\square$ es la raíz. Por ejemplo, el nodo $\neg s$ tiene como hijos a los nodos superiores $\neg p \vee \neg s$ y p.

También podemos dar esta demostración como una sucesión de fórmulas (así la definimos):

1. $\neg q$	(hipótesis)
2. $\neg p \vee q \vee \neg s$	(hipótesis)
3. p	(hipótesis)
4. $\neg p \vee \neg s$	(resol. 1. y 2.)
5. $\neg s$	(resol. 3. y 4.)
6. s	(hipótesis)
7. $\square$	(resol. 5. y 6.)

Luego, tenemos que $\mathcal{C}$ es inconsistente. Con esto hemos probado que:

$$\{(r \vee p) \rightarrow (q \vee \neg s), p, s\} \models q.$$

■

Ya vimos, en la motivación de la regla de resolución, que la cláusula resultante de la aplicación de la regla de resolución, la llamada "resolvente", es consecuencia lógica de las cláusulas a las cuales se aplicó la regla (las cláusulas "progenitoras"). Esta es una **propiedad de corrección** de la regla, que se

traspasa del siguiente modo al "método de refutaciones por resolución" que se usa para establecer la inconsistencia de un conjunto de cláusulas:

Teorema: (de corrección de refutaciones por resolución) Si Σ es un conjunto de cláusulas y hay una refutación por resolución a partir de Σ, entonces Σ es inconsistente.

Este teorema es fácil de demostrar por inducción en el largo n de la refutación $\varphi_1, \ldots, \varphi_n$ (donde φ_n es $\Box$) usando la corrección de la regla de resolución.

Intuitivamente, el teorema dice que este método deductivo formal no sanciona como inconsistentes a conjuntos de cláusulas que no lo son.

Nos preguntamos si el recíproco del teorema anterior también es verdadero. Es decir, si, en presencia de un conjunto inconsistente de cláusulas, existe una refutación por resolución a partir de él. Efectivamente; este es el teorema de completidad de las refutaciones por resolución:

Teorema: (de completidad de refutaciones por resolución) Si $\mathcal{C}$ es un conjunto de cláusulas inconsistente, entonces existe una refutación por resolución a partir de $\mathcal{C}$.

Demostración:* Daremos la demostración sólo en el caso en que $\mathcal{C}$ es finito†.

La siguiente notación es útil: si $\mathcal{C}$ es un conjunto de cláusulas y u es un literal, denotamos con $\mathcal{C}(u)$ al conjunto de cláusulas $\{C - \{\bar{u}\} \mid C \in \mathcal{C} \ y \ u \notin C\}$ ($\bar{u}$ es el literal complementario de u). Se tiene el siguiente resultado técnico preliminar, cuya demostración se deja como ejercicio:

Lemma: Si $\mathcal{C}$ es inconsistente, entonces también lo es $\mathcal{C}(u)$.

Ahora demostraremos el teorema por recursión (o inducción) en el número n de variables proposicionales que aparecen en $\mathcal{C}$. Si n es cero, $\mathcal{C}$ se reduce a la cláusula vacía, y el resultado es inmediato. En el caso general, fijemos un literal u cuya variable proposicional correspondiente aparece en $\mathcal{C}$. Consideramos los conjuntos de cláusulas $\mathcal{C}(u)$ y $\mathcal{C}(\bar{u})$. Como ambos conjuntos son contradictorios (por el lemma anterior) y contienen sólo $n-1$ variables proposicionales,

*En esta demostración seguimos el libro de Stern [37].

†En realidad, de este caso es posible obtener el caso general, pues el llamado "teorema de compacidad de la lógica proposicional", que aparecerá con frecuencia más adelante, nos dice que un conjunto de fórmulas proposicionales (no necesariamente cláusulas) es inconsistente si y sólo si existe un subconjunto finito de él que lo es. Equivalentemente, un conjunto de fómulas es satisfacible si y sólo si cada subconjunto finito de él lo es.

se puede dar dos árboles de resolución T_0 y T_1 que refutan a $\mathcal{C}(u)$ y $\mathcal{C}(\bar{u})$, respectivamente. Necesitamos un árbol de refutación para $\mathcal{C}$.

Modifiquemos el árbol T_0 por medio de las siguientes reglas:

i) Agregar $\bar{u}$ a las hojas de T_0 que no pertenecen a $\mathcal{C}$.

ii) Agregar $\bar{u}$ a un nodo no terminal si se le ha agregado a alguno de sus dos hijos.

Con la regla (i) se garantiza que las hojas de árbol T_0' (T_0 modificado) son todas cláusulas de $\mathcal{C}$. La regla (ii) asegura que el nuevo árbol es un árbol de resolución: en efecto, los literales que permiten las resoluciones en el árbol original T_0 no son nunca ni u ni $\bar{u}$. La raíz de T_0' es la cláusula $\{\bar{u}\}$ o la cláusula $\square$. En el segundo caso, la prueba estaría terminada (tendríamos un árbol de refutación para $\mathcal{C}$). En caso contrario, se procede a generar T_1' modificando T_1 (usando u en lugar de $\bar{u}$), que tiene como raíz a $\{u\}$ o a $\square$. En el segundo caso, estaríamos listos. En el primer caso, tendríamos árboles de resolución T_0' y T_1' con raíces $\{\bar{u}\}$ y $\{u\}$, respectivamente. Se completa la demostración combinando estos dos árboles en un árbol que tiene a T_0' como subárbol izquierdo, a T_1' como subárbol derecho, y a $\square$ como raíz. Este es un árbol de refutación para $\mathcal{C}$.

■

Este teorema nos dice que el método de refutaciones por resolución es lo suficientemente poderoso, completo, como para capturar la noción semántica de conjunto de cláusulas inconsistente. Sin embargo, el teorema no nos habla sobre la forma o estrategia para conseguir refutaciones concretas que permitan detectar inconsistencia. Este es un tema de enorme importancia en computación y es estudiada en el área de "demostración mecánica de teoremas".

2.8.1 Ejercicios

1. Suponiendo que se puede realizar las siguientes reacciones químicas:

$MgO + H_2 \rightarrow Mg + H_2O$
$C + O_2 \rightarrow CO_2$
$CO_2 + H_2O \rightarrow H_2CO_3$

y que se dispone de algunas cantidades de MgO, H_2, O_2, y C, se afirma que se puede producir H_2CO_3.

(a) Represente toda esta situación, incluyendo la afirmación, en lógica proposicional.

(b) Explique en forma precisa la relación entre el problema original y la representación proposicional.

(c) Demuestre formalmente usando **resolución** que la afirmación es correcta.

2. Demuestre que la regla de resolución es correcta en el sentido que la resolvente es consecuencia lógica de las cláusulas progenitoras.

3. Demuestre según se indicó en el texto que el método de refutaciones por resolución es correcto en el sentido que si él conduce a partir de un conjunto de cláusulas Σ a la cláusula vacía, entonces Σ es inconsistente.

4. Construya demostraciones por resolución para establecer que:

(a) $(p \vee \neg q \vee r) \wedge \neg p \wedge (q \wedge r \wedge p) \wedge (p \vee \neg r)$ no es satisfacible.

(b) $\neg\neg p \wedge (\neg p \vee ((\neg q \vee r) \wedge q)) \wedge \neg r$ no es satisfacible.

(c) $p \vee (q \wedge r) \vee (\neg p \wedge \neg q) \vee (\neg p \wedge q \wedge \neg r)$ es tautología.

d) $\neg q, \neg p \vee r, \neg(r \wedge \neg q) \models \neg p$.

e) $r \vee p, \neg r \models p \vee q$.

(f) $p \vee q \vee (\neg p \wedge \neg q)$ es tautología.

(g) $\neg(\neg p \wedge \neg q) \vee \neg(p \vee q)$ es tautología.

(h) $\neg p \vee q, r \vee \neg(p \wedge q) \models \neg p \vee (q \wedge r)$.

5. Esta es otra demostración de la corrección de la regla de resolución. Sea Σ un conjunto de cláusulas.

(a) Si R es la cláusula que se obtiene de $C_1, C_2 \in \Sigma$ mediante una aplicación de la regla de resolución, demuestre que Σ y $\Sigma \cup \{R\}$ son satisfechas por las mismas valuaciones, es decir, son lógicamente equivalentes.

(b) Obtenga que si Σ es satisfacible, entonces es imposible obtener la cláusula vacía $\square$ por sucesivas aplicaciones de resolución.

(c) Obtenga que si de Σ se puede obtener por resolución la cláusula $\square$, entonces Σ es insatisfacible.

6. Use resolución para justificar como válidas las siguientes formas estándar de razonamiento (explicando claramente qué significa "justificar"):

(a) q es una consecuencia lógica de $p \to q$ y p. (Modus Ponens)

(b) $\neg p$ es una consecuencia lógica de $p \to q$ y $\neg q$. (Modus Tollens)

(c) r es consecuencia lógica de $p \to r$ y $\neg p \to r$. (Demostración por Casos)

7. (a) Demuestre el lemma técnico de la demostración del teorema de completidad de las refutaciones por resolución.

Ind.: Razone por contradicción, construyendo una asignación que satisfaga a $\mathcal{C}$.

(b) ¿En que parte de la demostración del teorema de completidad de refutaciones por resolución se usa la finitud de $\mathcal{C}$?

(c) Ilustre la demostración del teorema de completidad con un ejemplo concreto.

2.9 Programación en Lógica Proposicional

Consideraremos aquí el caso más simple de programación en lógica, es decir, el caso proposicional, y, más aún, dentro de ésta, programación por medio de cláusulas de Horn positivas.

Notemos que una cláusula $p_1 \vee \cdots \vee p_n \vee \neg q_1 \vee \cdots \vee \neg q_m$ (los p_i, q_j son literales positivos) puede ser escrita en la forma $(q_1 \wedge \cdots \wedge q_m) \to (p_1 \vee \cdots \vee \cdots p_n)$. En esta sección nos interesa el caso en que $n = 1$, es decir, el de clausulas de la forma: $q_1 \wedge \cdots \wedge q_m \to p$. Estas cláusulas, llamadas de Horn positivas, representan conocimiento definitivo, en el sentido que no hay disyunciones en el consecuente de la cláusula. Si se tiene $q_1, \ldots, q_m$, entonces, definitivamente, se tiene p. Un caso particular interesante es aquel en que $m = 0$, es decir, el antecedente es vacío. Una cláusula de este tipo se llama "un hecho" (fact, en inglés) y es equivalente a afirmar p, incondicionalmente. En lugar de escribir $\to p$, escribimos p.

En programación en lógica, una cláusula de la forma $q_1 \wedge \cdots \wedge q_m \to p$ se acostumbra escribir en la forma $p \leftarrow q_1, \ldots, q_m$. La "cabeza" de la cláusula es p, y $q_1, \ldots, q_m$, es el "cuerpo" de la cláusula.

Un programa en lógica consiste en un conjunto de cláusulas de Horn positivas que describen, en forma declarativa, cómo es un dominio de aplicación. Lo

interesante es que este programa declarativo, puramente descriptivo, puede ser usado para realizar computaciones, por ejemplo, para responder preguntas relativas al dominio descrito por el programa. En el capítulo 9 se discutirá programación en lógica en detalle. Aquí sólo haremos una pequeña introducción a sus aspectos lógicos en el caso proposicional, con el principal objetivo de relacionar programación en lógica con la sección anterior sobre refutaciones por resolución. El siguiente es un ejemplo de programa en lógica con cláusulas de Horn positivas:

$p \leftarrow q, s$

$s \leftarrow t$

t

q

A partir de este programa, denotémoslo con $PROG_1$ (hasta ahora no es más que un conjunto de fórmulas proposicionales), intentaremos concluir nuevas aseveraciones. En este caso simple, nos interesará concluir proposiciones atómicas, por ejemplo, p. Siguiendo con la idea de demostrar p por contradicción, agregaremos al programa la proposición $\neg p$. Como esta fórmula se puede escribir en la forma $\leftarrow p$, pasamos al conjunto extendido de cláusulas:

$\leftarrow p$

$p \leftarrow q, s$

$s \leftarrow t$

t

q

Podemos aplicar resolución para intentar obtener la cláusula vacía, en este caso, la cláusula "$\leftarrow$". Por ejemplo, una aplicación de esta regla se verá en la siguiente forma:

$$\begin{array}{c} p \leftarrow q, s \\ \underline{\quad s \leftarrow t \quad} \\ p \leftarrow q, t \end{array}$$

En programación en lógica, para demostrar la "meta" (goal, en inglés) p, usualmente se parte con la cláusula $\leftarrow p$ (aunque esto no es imprescindible)

y se busca una cláusula que tenga a p en la cabeza (la cláusula se toma de la cabeza) para poder resolver. Por ejemplo, si resolvemos $\leftarrow p$ con $p \leftarrow q, s$, obtenemos la cláusula resolvente $\leftarrow q, s$. En consecuencia, para establecer p es necesario establecer (como submetas) a q y s. Llegar a la cláusula vacía es equivalente a haber demostrado (eliminado) todas las submetas originadas.

Hay un subentendido en programación en lógica: la cláusula $p \leftarrow q, s$ (junto con las otras que tengan a p como cabeza) está definiendo a p. De hecho, puede entenderse que la cláusula define un "procedimiento" p, que, al ser ejecutado, llama a las subrutinas q y s.

Esta podría ser una demostración de p a partir del programa $PROG_1$:

Agregamos al programa la cláusula $\leftarrow p$, e intentamos obtener la cláusula vacía:

1. $\leftarrow p$ (la negación de p)
2. $p \leftarrow q, s$ (la definición de p)
3. $\leftarrow q, s$ (resolución con las dos cláusulas anteriores)
4. $s \leftarrow t$ (la definición de s)
5. $\leftarrow q, t$ (resolución con las dos cláusulas anteriores)
6. t (definición de t, incondicional, es un hecho)
7. $\leftarrow q$ (resolución con las dos cláusulas anteriores)
8. q (definición de q)
9. $\leftarrow$ (con resolución obtenemos la cláusula vacía)

Vemos que hay varios puntos que aclarar si queremos que este proceso sea efectuado de manera automática. Primero, podría ser que en el programa original haya varias cláusulas que tengan la cabeza p, por ejemplo, $p \leftarrow q, s$ y $p \leftarrow u$. Esto es equivalente a tener $((q \wedge s) \vee u) \rightarrow p$. Entonces, ¿con cuál cláusula comenzamos? Otra pregunta es la siguiente: si se ha decidido comenzar con la primera de las dos cláusulas, ¿cuál de las dos submetas (q o s) establecemos primero?

Distintas implementaciones de programación en lógica se casarán con distintas estrategias. Antes de ver una particular, destaquemos algunas ideas generales sobre programación el lógica.

Con programación en lógica, resolvemos, en el caso ideal, de manera computacional, de la siguiente manera:

- **Describiendo** el dominio en lógica. Para esto, usamos usualmente cláusulas. Esa descripción constituye un **programa declarativo***.
- Usando, por debajo, de forma transparente para el usuario, un **mecanismo de control** de fórmulas e inferencias, estas últimas usualmente basadas en resolución.

De este modo se materializa la llamada "Ecuación de Kowalsky":

$$Algoritmo \quad = \quad Lógica \quad + \quad Control,$$

según la cual, cualquier algoritmo se puede ver descompuesto en una parte declarativa, escrita en lógica, más una parte de control, o manejo, de las fórmulas que constituyen la primera parte.

Como dijimos, estando fija la estructura de control, nuestra tarea de programación es puramente declarativa: describimos el dominio y qué queremos obtener computado (,pero no cómo computarlo).

Una implementación particular de programación en lógica, es decir, correspondiente a una forma de control específica, es el lenguaje de programación PROLOG, por PROgramming in LOGic. La estrategia de éste es "búsqueda primero en profundidad† con backtracking", que es lo siguiente: se parte con la meta $\leftarrow p$ (será una demostración descendente‡) y, en seguida, se elige la primera de las cláusulas con cabeza p. Intentará obtener la cláusula vacía por medio de resolución. Esto irá generando nuevas submetas que se intentará demostrar, con el propósito de llegar a la cláusula vacía. Si no se logra, se hará "backtracking", considerando la siguiente cláusula que permita aplicar resolución. Con respecto a las submetas, irá considerándolas de izquierda a derecha (en el cuerpo), postergando las submetas que están más a la derecha hasta que las de la izquierda estén demostradas. Este proceso da lugar a sub–submetas, lo que hará que las submetas que estaban más a la derecha sean más y más postergadas.

Consideremos ahora el programa $PROG_2$:

*En contraste con programas de lenguajes imperativos, donde hay que especificar en el programa los pasos a ejecutar.

†Depth-First Search, en inglés

‡top–down, en inglés

a. $p \leftarrow q, s$

b. $p \leftarrow u$

c. $s \leftarrow t$

d. $q \leftarrow v$

e. v

f. u

La siguiente es una demostración de p según la estrategia de PROLOG:

1. $\leftarrow p$ (la negación de p)
2. $p \leftarrow q, s$ (cláusula a. en la definición de p)
3. $\leftarrow q, s$ (resolución con 1. y 2.)
4. $q \leftarrow v$ (definición de q, primera submeta)
5. $\leftarrow v, s$ (resolución con 3. y 4., se genera la sub–submeta v)
6. v (v es un hecho del programa)
7. $\leftarrow s$ (resolución con 5. y 6.)
8. $s \leftarrow t$ (la definición de s)
9. $\leftarrow t$ (resolución con 7. y 8., no se puede resolver con ésta)
10. $p \leftarrow u$ (cláusula b., al hacer backtracking)
11. $\leftarrow u$ (resolución con 1. y 10.)
12. u (u es un hecho del programa)
13. $\leftarrow$ (resolución con 11. y 12. conduce a la cláusula vacía)

La sintaxis de PROLOG es algo distinta de la que hemos usado aquí. Para ser precisos, el programa $PROG_2$ se escribe así:

p :– $q, s.$

p :– $u.$

s :– $t.$

q :– $v.$

$v.$

$u.$

Al correr el programa anterior con PROLOG, obtendremos el "prompt" `| ?-` (o algo parecido, según el PROLOG particular que usemos), que nos invita, por ejemplo, a preguntar por p: `| ?- p.` . PROLOG comenzará a demostrar p y finalmente nos responderá `yes` . Esto lo podemos intepretar como $PROG_2 \models p$ (¿por qué?).

2.9.1 Ejercicios

1. Una **cláusula de Horn positiva** es una fórmula que tiene una de las dos formas:

- p (p es una variable proposicional)
- $p_1 \wedge \ldots \wedge p_n \rightarrow p$ (las p_i, p son variables proposicionales

Estas son las fórmulas que aparecen en los programas en PROLOG, o, más generalmente, en programación en lógica. Su nombre proviene de Alfred Horn, el primero que las estudió.

Demuestre que un conjunto de cláusulas de Horn positivas es siempre consistente.

2. Demuestre que a partir de un conjunto de cláusulas de Horn positivas Σ no se puede demostrar por resolución un literal negativo. Más generalmente, que $\Sigma \not\models \neg p$.

3. Haga demostraciones por resolución usando los ejemplos de la sección anterior por medio de diversas estrategias: demostraciones descendentes (top–down) (se parte con (la negación de) lo que se quiere demostrar como meta), ascendentes (bottom–up) (se parte de hechos positivos y se llega a lo que se quiere demostrar), diversos órdenes de uso de cláusulas, diversos órdenes de

evaluación de submetas, ...

4. Demuestre que si $PROG$ es un programa en PROLOG que tiene claúsulas de Horn positivas y p es un literal positivo, entones que PROLOG responda `yes` a la consulta `| :? p` implica que $PROG \models p$. ¿Qué opina de la implicación inversa?

5. Demuestre p según la estrategia de PROLOG a partir del programa:

a. $p \leftarrow q, s$

b. $p \leftarrow u$

c. $s \leftarrow t$

d. $s \leftarrow d$

e. $q \leftarrow v$

f. $d \leftarrow w$

g. v

h. u

2.9.2 Negación en Programación en LP

Ya vimos en los ejercicios de la sección anterior que no es posible obtener conocimiento negativo a partir de un programa en lógica $PROG$ que contiene sólo cláusulas de Horn positivas. Más precisamente, para un literal positivo p , puede tenerse $PROG \models p$ o no (y, en el primer caso, esto podrá ser determinado por resolución), sin embargo, nunca se tiene $PROG \models \neg p$. Podríamos estar interesados en "concluir" $\neg p$ cuando $PROG \not\models p$. Es decir, concluyendo $\neg p$ por una especie de "suposición del mundo cerrado" como la que aparece en bases de datos relacionales, por medio de la cual se puede "concluir" información negativa si es que la afirmación positiva correspondiente no aparece explícitamente representada*. Podemos extender esta suposición del mundo cerrado al caso de programas en lógica, vistos como conjuntos de cláusulas, es decir, sin ningún aspecto procedural asociado: un literal negativo es "consecuencia" de un programa de Horn positivo si el literal positivo correspondiente

*Por ejemplo, si en una base de datos de vuelos no aparece explícitamente un vuelo directo de Santiago a Hong Kong, concluimos que no existe tal vuelo.

no es consecuencia lógica del programa. Nótese que ya no basta con verificar si el literal positivo está explícitamente en el programa, pues éste podría ser consecuencia del prgrama.

Esta suposición del mundo cerrado generalizada es algo inoperante, en el sentido que volvemos al concepto de consecuencia lógica, en circunstancias que nos habíamos pasado al terreno deductivo. Para mantenernos en él, consideramos una suposición modificada: un literal negativo es "consecuencia" de un programa si es que el literal positivo correspondiente no puede ser demostrado por resolución a partir de él (posiblemente con una estrategia fija elegida a priori). En este caso decimos que la negación es verdadera (con respecto al programa). Sin embargo, hay que notar que esta forma de negación es distinta de la negación lógica (de la lógica proposicional); ella se llama *negación como falla*. Distintas implementaciones de programación en lógica, en particular, PROLOG, incorporan esta regla.

Negación como falla es, en general, más débil que la suposición del mundo cerrado generalizada, es decir, esta última, en caso de indeterminación, acepta más conocimiento negativo que negación como falla, ya que "la falla" puede no ser detectada dependiendo de la estrategia de resolución implementada. Por ejemplo, a partir del programa: $p :- p$, la suposición del mundo cerrado acepta a $\neg p$ como verdadera (p no es consecuencia lógica del programa), sin embargo, la negación como falla no lo acepta, pues al tratar de demostrar p cae en una iteración infinita (¿por qué?), sin detectar la falla. Enfaticemos, entonces, que la falla debe ser finita, es decir, no se logra demostrar algo habiendo agotado (finitamente) el proceso de demostración (búsqueda).

Usaremos la notación "*not*" para negación como falla, en lugar de la negacion clásica "$\neg$".

Ejemplos:

1) Consideremos las siguientes cláusulas:

$$p \ :- \ q.$$
$$q \ :- \ r.$$

Al hacer la consulta $:- p?$, e intentar probar p, se falla, es decir, no se logra demostrar p y la búsqueda termina. La respuesta sería NO.

Ahora, si preguntamos por $not\ p$, donde not es negación como falla, entonces, respondemos SI, pues primero intentamos demostrar p. Como a esta pregunta se reponde NO, obtenemos para $not\ p$ la respuesta SI. Es decir,

del programa concluimos $not\ p$, a pesar de que $\neg p$ no es consecuencia lógica (clásica) del programa.

2) Consideremos ahora el programa:

$$p\ :-\ p.$$

Si ahora preguntamos por $not\ p$, primero intentamos demostrar p. En este caso, se cae en una itaración infinita y no se logra demostrar p. Sin embargo, no se falla finitamente en la demostración de p, y en consecuencia, no respondemos SI a la pregunta original.

3) Podemos incluir la negación como falla en cuerpos de cláusulas de un programa, por ejemplo, en

$$\begin{array}{l} p\ :-\ q. \\ q\ :-\ not\ r. \end{array}$$

al intentar demostrar p, se genera la submeta q, la cual, a su vez, genera la submeta $not\ r$. Lo que se hace, en este punto, es intentar demostrar r, y si se falla (finitamente), se responde SI a $not\ r$, y en consecuencia, SI para q, y en seguida, también SI para p.

Sin embargo, con la misma pregunta, pero con el programa

$$\begin{array}{l} p\ :-\ q. \\ q\ :-\ not\ r. \\ r. \end{array}$$

se obtiene la respuesta NO, pues no se falla al intentar demostrar r.

2.10 Dos Aplicaciones de Lógica Proposicional

Hemos mencionado antes que la lógica proposicional proporciona lenguajes que son lo suficientemente expresivos como para permitirnos describir y codificar ciertas situaciones interesantes. En esta sección mostraremos dos aplicaciones de este tipo.

2.10.1 El Principio de los Cajones

El principio de los cajones* es un principio combinatorio que dice:

"No se puede colocar $n+1$ objetos en n cajones sin que quede algún cajón con al menos dos objetos".

Este principio puede ser formulado en términos funcionales:

"No existe una función $f : \{1, ..., n+1\} \longrightarrow \{1, ..., n\}$ que sea uno a uno (inyectiva)".

El principio de los cajones puede ser demostrados por inducción usual en n. Sin embargo, lo que nos interesa aquí es la representación, o codificación, de este problema en lógica proposicional, es decir, el principio de los cajones puede tomar una forma puramente proposicional. Esto es frecuente en el área de combinatoria. De hecho, en la posibilidad de representar este problema en lógica proposicional, radica la posibilidad de obtener resultados matemáticos sobre lógica proposicional, por ejemplo, sobre la complejidad computacional de las demostraciones por resolución. La dificultad del problema combinatorio se traspasa al contexto lógico. Veremos en otros capítulos frecuentes apariciones de esta situación.

Podemos partir con letras proposicionales elegidas convenientemente. Pensando en representar la proposición:

"No hay una función $f : \{1, ..., n+1\} \longrightarrow \{1, ..., n\}$ que es $1-1$", $\qquad (*)$

denotemos la proposición "$f(i) = j$" con la letra proposicional p_{ij}. Tenemos entonces las siguientes variables proposicionales:

$$p_{ij} \begin{cases} 1 \leq i \leq n+1 \\ 1 \leq j \leq n \end{cases}$$

En el lenguaje proposicional determinado por estas variables, podemos representar la parte "$f : \{1, ..., n+1\} \longrightarrow \{1, ..., n\}$ es $1-1$" de $(*)$ a través de la fórmula:

*En inglés es el "Pigeon–Hole Principle".

$$\varphi: \quad \bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} p_{ij} \quad \bigwedge$$

(a cada i le corresponde algún j)

$$\bigwedge_{\substack{1 \le i\,,\, j \le n+1 \\ i \neq j \\ 1 \le k \le n}} (\neg p_{ik} \vee \neg p_{jk})$$

(no simultáneamente $f(i) = k$ y $f(j) = k$)

Notar que esta fórmula está en forma normal conjuntiva (FNC) (ver sección 2.2.2). Se tiene la siguiente

Proposición : La proposición $(*)$ es verdadera si y sólo si φ es insatisfacible.

Con esto tenemos una representación proposicional del problema combinatorio original, que ahora ha tomado una forma proposicional.

2.10.2 Ejercicios

1. Dé un conjunto inconsistente de 5 fórmulas proposicionales tal, que todo subconjunto propio es consistente.

2. Sea $P = \{p_1, \ldots, p_5\}$.
(a) Construya una fórmula $\varphi(p_1, \ldots, p_5) \in L(P)$ tal, que φ es verdadera si y sólo si exactamente una de las p_i es verdadera.

(b) Dé una fórmula $\psi \in L(P)$ que esté en FNC y sea lógicamente equivalente a φ (puede hacerlo usando el problema anterior o directamente).

3. En el problema anterior Ud. vio un ejemplo de la **representación** de una función de verdad $F : \{0,1\}^5 \to \{0,1\}$ mediante una fórmula proposicional. Indique en forma precisa cuál es la representación y qué propiedades tiene.

Los ejercicios siguientes están en el mismo espíritu.

4. Represente mediante fórmulas proposicionales en FNC y FND las siguientes funciones:

(a) $\Phi(p_1, \ldots, p_5) = \mathbf{t}$ si y sólo si al menos tres de las p_i son $\mathbf{t}$ (ésta es la función mayoría)

(b) $\Phi(p_{12}, p_{13}, p_{14}, p_{23}, p_{24}, p_{34}) = \mathbf{t}$ si y sólo si el grafo $G = (V, E)$ tiene un nodo de grado al menos 2.

Aquí, $V = \{1, 2, 3, 4\}$ y E está definido por: $(i, j) \in E$ si y sólo si p_{ij} es $\mathbf{t}$.

(c) $\Phi(p_1, \ldots, p_4) = \mathbf{t}$ ssi $(p_1 p_2 p_3 p_4)_2$ es un número primo impar.

Aquí, $(p_1 \ldots p_4)_2$ puede ser visto como la codificación binaria de un número.

2.10.3 Diagnóstico Basado en Modelos

El diagnóstico de un sistema que se comporta de manera anormal consiste en localizar aquellos componentes cuya falla da cuenta del comportamiento erróneo observado. Existen al menos dos aproximaciones al problema de diagnóstico.

En el primero, se intenta codificar heurísticas y experiencia previa de expertos humanos en el área particular del problema tratado. Los sistemas expertos de primera generación, como MYCIN, son un ejemplo.

En la segunda, se parte con una descripción o modelo* del sistema, que describe explícitamente su estructura, es decir, sus componentes y las conexiones entre ellos, y se aplica métodos deductivos para determinar las causas del mal funcionamiento. A esta forma de diagnóstico se le llama "diagnóstico basado en modelos". El problema de diagnóstico surge cuando una observación del comportamiento actual del sistema entra en conflicto con el comportamiento esperado de él.

Ejemplo:†

En la figura A2.1 tenemos un sumador binario, que recibe tres entradas, A B y C, y entrega dos salidas, D y E. El comportamiento esperado del sumador es que $A + B + C = ED$, en aritmética binaria.

---

*Nótese este uso habitual de la palabra "modelo". Usualmente en ciencia e ingeniería, se entiende este concepto como una descripción o representación simplificada de una realidad. En este sentido, es una abstracción que se expresa usualmente a través de ecuaciones algebraicas, ecuaciones diferenciales, o axiomas de un lenguaje formal. Sin embargo, en lógica matemática es frecuente usar el término "modelo" para una estructura que satisface una descripción.

†Se agradece la colaboración de Cristián Ferretti, quien desarrolló la mayor parte esta sección sobre diagnóstico.

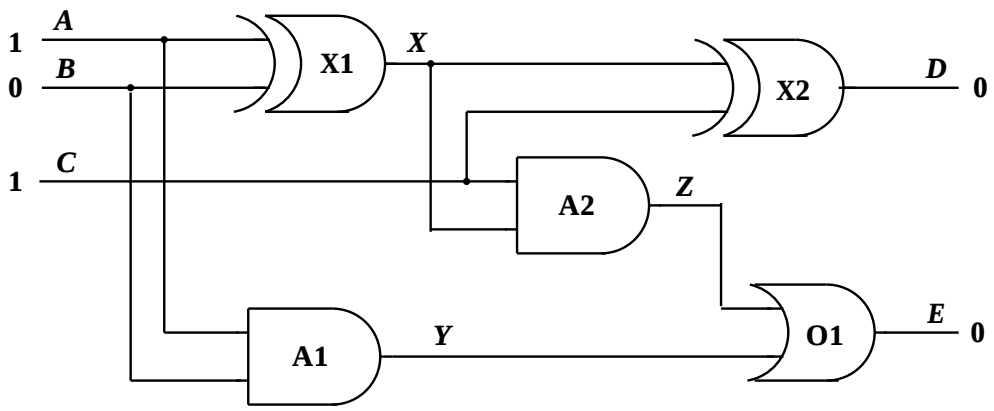


Figura A2.1: Sumador binario formado por dos compuertas `XOR` (`X1` y `X2`), dos compuertas `AND` (`A1` y `A2`) y una compuerta `OR` (`O1`). La salida $E = 0$ es errónea.

El funcionamiento de un sumador como el de la figura es verificado a través de un conjunto de valores de prueba, y se encuentra que, al darle la entrada $(A = 1, B = 0, C = 1)$, entrega $(D = 0, E = 0)$, lo que es erróneo. La salida esperada es $(D = 0, E = 1)$.

Nos interesa identificar aquella(s) compuerta(s) que fallan para reemplazarlas, y así reparar el sumador. Una posibilidad sería verificar cada compuerta por separado. En este ejemplo, en que la cantidad de compuertas es pequeña, esto podría parecer plausible, pero en circuitos más complejos es demasiado costoso.

Tenemos más información disponible que el simple hecho de que el sumador falla. A partir de la estructura del sumador (sus componentes, la forma cómo trabajan y cómo estan conectadas) y la observación que se efectuó (que permitió detectar la falla), es posible reducir la cantidad de compuertas a verificar (un ejemplo de ello es que la falla de la compuerta X2 no bastaría para explicar la salida observada).

Una descripción del funcionamiento normal del sumador en lógica proposicional podría ser la siguiente:*

- compuerta X1: $(A \otimes B) \leftrightarrow X$

*El símbolo $\otimes$ se referirá a la funcion booleana equivalente a `XOR`

- compuerta A1: $(A \wedge B) \leftrightarrow Y$
- compuerta A2: $(X \wedge C) \leftrightarrow Z$
- compuerta X2: $(X \otimes C) \leftrightarrow D$
- compuerta O1: $(Y \vee Z) \leftrightarrow E$

El comportamiento (normal) de cada puerta queda expresado explícitamente por la fórmula a su derecha. Las conexiones entre las puertas están implícitamente representadas a través de las variables que comparten (X, Y y Z).

Ahora bien, lo anterior no es válido para nuestro sumador defectuso (¡verificarlo!). Necesitamos agregar al modelo la posibilidad de falla de las compuertas. Para ello, incorporaremos nuevas variables proposicionales, que representarán el estado normal ($= 1$) o defectuoso ($= 0$) de cada puerta. Si llamamos $X1$, $X2$, $A1$, $A2$, y $O1$ a estas variables, nuestro modelo queda:

$$X1 \rightarrow ((A \otimes B) \leftrightarrow X)$$
$$A1 \rightarrow ((A \wedge B) \leftrightarrow Y)$$
$$A2 \rightarrow ((X \wedge C) \leftrightarrow Z)$$
$$X2 \rightarrow ((X \otimes C) \leftrightarrow D)$$
$$O1 \rightarrow ((Y \vee Z) \leftrightarrow E)$$

Este nuevo modelo no obliga a X a tomar el valor de $(A \otimes B)$ cuando $X1$ esta defectuosa. En ese caso, el valor de X queda libre. A este tipo de modelos, que especifican sólo las condiciones normales de funcionamiento, dejando libres las posibilidades cuando hay fallas, se les llama "modelos débiles de falla".

Si al modelo del comportamiento del sumador en condiciones normales, agregamos nuestras observaciones: $(A \wedge \neg B \wedge C \wedge \neg D \wedge \neg E)$, podremos determinar –ojala deductivamente– cuáles compuertas pueden estar funcionando mal, es decir, cuáles de –o subconjuntos de– $X1$, $A1$, $A2, X2, O1$ deben tomar el valor 0 (es decir, no están correctas), para que no haya una contradicción en la representación del modelo más la observación.

2.10.4 Ejercicios

1. Determine, a partir del modelo y las observaciones, qué compuertas, o conjuntos de ellas, podrían estar funcionando mal. Explique su metodología.

Capítulo 3

Problemas y Algoritmos de Decisión

La sintáxis de los lenguajes de lógica proposicional nos enfrenta naturalmente a algunos problemas de naturaleza computacional.

Ejemplo: (Reconocimiento de Fórmulas Proposicionales) Ya mencionamos que es posible escribir programas computacionales que son capaces de reconocer fórmulas de un lenguaje proposicional dado. Más precisamente tenemos una solución para el siguiente **problema de decisión:**

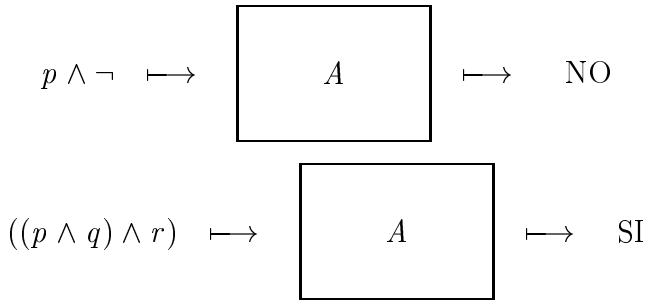
- Fijemos el conjunto P de letras proposicionales.
- Sea L el conjunto de símbolos formado por los paréntesis y conectivos lógicos.
- Podemos formar palabras finitas sobre el alfabeto $L \cup P$, i.e. elementos de $(L \cup P)^*$.
- Algunas de estas palabras son fórmulas del lenguaje $L(P)$.

El problema es: ¿Existe algún algoritmo A que, aplicado a una palabra cualquiera $w \in (L \cup P)^*$, responda

- SI si $w \in L(P)$
- NO si $w \in (L \cup P)^* - L(P)$?

Por ejemplo, si $P = \{p, q, r\}$, el problema de decidir si una palabra arbitraria pertenece a $L(P)$ tiene solución, es decir, hay un algoritmo A que resuelve este problema de decisión computacional. Decimos que el problema "ser fórmula proposicional" es **decidible**.

El algoritmo debería dar las siguientes respuestas en el caso de $L(P)$:



■

Ejemplo: (El Problema de Satisfacibilidad) En secciones previas vimos que la noción de consecuencia lógica puede ser reducida a la noción de satisfacibilidad de un conjunto de fórmulas. En muchas aplicaciones ese conjunto de fórmulas será finito*. En consecuencia, usualmente estaremos interesados en determinar si una fórmula es satisfacible o no.

La pregunta que nos surge naturalmente es la siguiente: Dado un lenguaje proposicional $L(P)$, existe un algoritmo que, aplicado a cualquier fórmula φ de $L(P)$, responde (después de un número finito de pasos) SI, si φ es satisfacible, y NO, en caso contrario.

Obviamente un tal algoritmo existe. Basta con verificar la tabla de verdad de la fórmula φ para determinar si hay alguna valuación que la hace verdadera. El algoritmo es general, es decir, es un algoritmo para todas las fórmulas de $L(P)$; tiene un comportamiento determinista (la sucesión de pasos y la respuesta final no varían de una ejecución a otra del algoritmo con la misma entrada); simpre entrega una respuesta (SI o NO); y el número de pasos es finito y depende de la entrada.

El algoritmo indicado para resolver el problema de satisfacibilidad es obviamente poco eficiente, ya que, si una fórmula contiene n variables distintas, la

*El "Teorema de Compacidad" de la lógica proposicional, que veremos más adelante, nos permitirá reducir la satisfacibilidad de un conjunto infinito de fórmulas a la satisfacibilidad de conjuntos finitos.

tabla de verdad puede llegar a tener que considerar, en el peor caso, 2^n valuaciones para determinar si la fórmula es satisfacible o no. En este caso hablamos de una explosión combinatoria. Nos preguntamos si hay un algoritmo mejor.

3.1 Algoritmos

Problemas de decisión computacional (que entregan respuestas SI o NO a preguntas sobre una clase de objetos) aparecen con frecuencia en lógica, ciencia de la computación, y en muchos otros contextos. Para saber qué tan difícil de resolver computacionalmente es un problema de decisión, o para demostrar que simplemente no es soluble, necesitamos una definición precisa de la noción intuitiva de algoritmo.

Mencionemos otro ejemplo. En el año 1900, el famoso matemático David Hilbert, en su conferencia plenaria del Congreso Mundial de Matemáticos, dio una lista de problemas matemáticos abiertos, sin solución a esa fecha, que, a su juicio, iban a marcar el rumbo del desarrollo de la matemática en el siglo XX. El décimo de ellos planteaba la pregunta por un algoritmo, más precisamente, por un método o procedimiento mecánico, que, al ser aplicado a cualquier ecuación diofantina, es decir, multipolinomial con coeficientes enteros (por ejemplo, $2x^2y^3 - 8xz^4 = 0$), respondiera SI o NO, dependiendo de si la ecuación tiene solución en los enteros o no. Este problema fue formulado mucho antes de que existiera una definición matemática, precisa, del concepto de algoritmo (que se dio a mediados de los años 30). No es extraño, entonces, que la respuesta negativa (no existe tal algoritmo, o, lo que es lo mismo, el problema es indecidible) se obtuvo recién en 1970, por el matemático ruso Y. Matiyasevich.

Una definición clásica de algoritmo se basa en en el modelo computacional de las máquinas de Turing (por Alan Turing, $\sim$ 1936). La figura A3.1 nos permite imaginarnos una máquina de Turing.

El aspecto dinámico de la máquina está representado por una matriz (o función) de transición. Una fila de la matriz podría ser, por ejemplo,

$$q_1 \quad a \quad \not b \quad +1 \quad q_3$$

que la leemos de la siguiente manera: "si la máquina está en el estado q_1 leyendo el símbolo a, escríbase un blanco, $\not b$, muévase a la derecha, y quede en el estado q_3".

Se establecen convenciones para escribir entradas $w \in \sum^*$ en la cinta y se distingue los estados inicial y finales (o de parada).

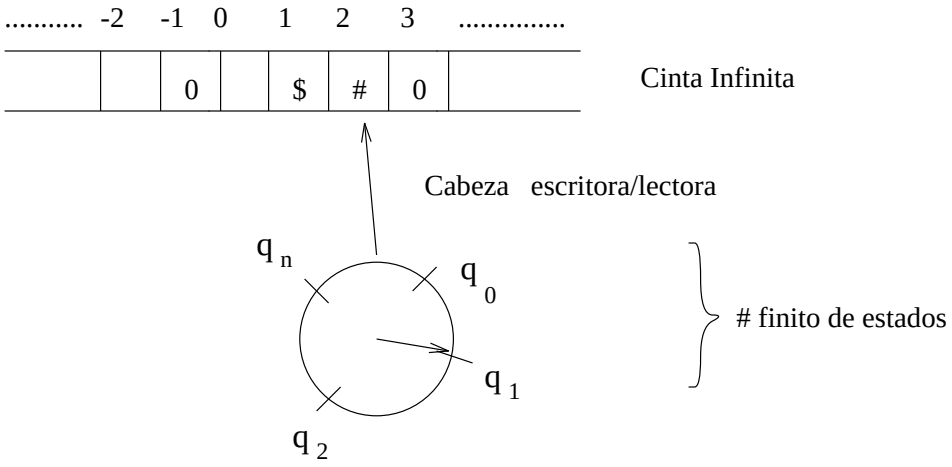


Figura A3.1: $\sum$: conjunto de símbolos (alfabeto de cinta). Q: conjunto de estados.

Ejemplo: Sean $\sum = \{0, 1\}$. Entonces $\sum^*$ contiene palabras binarias. Sea $L \subseteq \sum^*$ el conjunto:

$$L = \{w \in \sum^* \mid \text{el número respresentado por } w \text{ es divisible por } 4\}$$

L es decidible por medio de un algoritmo programable como MT.

$$w \in \sum^* : \ A(w) \mapsto \begin{cases} \text{SI} & \text{si} \quad w \in L \\ \text{NO} & \text{si} \quad w \in \sum^* - L \end{cases}$$

En efecto, si $w = w_1 \ w_2 \ \cdots\cdots \ w_n$, con $w_i \in \sum$, entonces

$$w \in L \iff w_{n-1} = w_n = 0.$$

Una MT posible es la siguiente:

$$\sum = \{0, 1\} \ (\cup \{\not b\})$$
$$Q = \{q_0 \ , \ q_{SI} \ , \ q_{NO} \ , \ \cdots\cdots\}$$

- q_0 es estado inicial.
- q_{SI} y q_{NO} son estados de parada.

La respuesta se leerá de los estados de parada. La entrada se escribirá en la cinta según la siguiente convención:

	-1	0	1					
$\cdots$	$\not b$	w_1	w_2	w_3	$\cdots\cdots$	w_n	$\not b$	$\cdots$

La cabeza estará sobre la celda número 1; y la matriz de transición es:

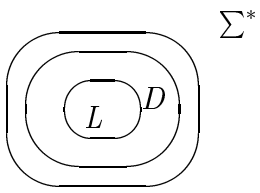
q_0	0	0	$+1$	q_0
q_0	1	1	$+1$	q_0
q_0	$\not b$	$\not b$	-1	q_1
q_1	0	0	-1	q_2
q_2	0	0	?	q_{SI}
q_1	1	?	?	q_{NO}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Necesitamos sólo 5 estados.

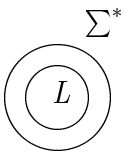
■

Como vimos en el ejemplo de "divisibilidad por 4", las máquinas de Turing pueden ser usadas para programar algoritmos de decisión.

En general, la situación es la siguiente: un problema de decisión aparece como un subconjunto distinguido L de un dominio de instancias D, las cuales son palabras sobre un alfabeto Σ. El problema es determinar si existe un algoritmo, programable como una máquina de Turing M, para decidir la pertenencia a L de las instancias arbitrarias de D. Usualmente el alfabeto Σ_1 de la máquina M contendrá al alfabeto Σ del problema de decisión.



Para simplificar la presentación de los aspectos más esenciales de los problemas de decisión, pensaremos que D, $\Sigma^\star$ y $\Sigma_1^\star$ coinciden. Así es que básicamente el problema de decisión se verá en la forma siguiente:



Usualmente se identifica el problema de decisión con el subconjunto L de instancias positivas (respuestas SI). Si existe un algoritmo para decidir la pertenencia a L, decimos que "el problema L es decidible".

Las máquinas de Turing pueden ser usadas para calcular funciones si se establecen convenciones sobre escritura/lectura de entradas/salidas en/de la cinta. De esta manera, aparecen las llamadas funciones computables (o Turing–computables). Obviamente, los problemas de decisión pueden ser vistos como problemas de cómputo de funciones que entregan el valor 1 para argumentos en L y valor 0, para argumentos en $\Sigma^\star - L$. Más precisamente, cada problema de decisión (D, L) tiene asociada su "función característica" $f_L : D \to \{0, 1\}$ que asigna valor 1 a los argmentos en L, y 0 a los argumentos en $D - L$. L es decidible siempre y cuando f_L es computable.

Hemos dado un modelo matemático de computador, el de las máquinas de Turing, que, de paso, nos da definiciones precisas de los conceptos informales de algoritmo (de decisión, en particular) y de función computable. La pregunta natural es: ¿qué tan bueno es este modelo de computación (ideal)?

Una aseveración muy conocida que apoya la bondad del modelo es la:

Tesis de Church:

- Lo algorítmico es lo programable (representable) como MT.
- Las *funciones computables* son exactamente las calculables mediante máquinas de Turing.

Observaciones:

- Esta tesis no puede ser demostrada, pues mezcla un concepto informal (algorítmico , computable) con uno formal (calculable mediante MT).

- Se cree en ella o no se cree.
- Hay evidencia en favor de esta tesis:
  - Con "todos" los algoritmos concretos que se ha intentado programar en MT se ha tenido éxito.
  - Muchos otros modelos de algoritmo y computación (ideal) han resultado equivalentes al de las MT.
    - "equivalentes" significa que calculan exactamente las mismas funciones o se pueden simular mutuamente.
    - Esta equivalencia se puede demostrar matemáticamente.
    - Los diversos modelos surgieron independientemente (casi al mismo tiempo), pero, al final, todas las intuiciones convergieron a lo mismo (a pesar de que los modelos son formalmente distintos).
- Algunos otros modelos (equivalentes) son: funciones recursivas de Gödel y Kleene (ver próxima sección), λ–cálculo (Church), máquinas de registro (Sturgis y Sheperdson), sistemas de Post, sistemas de Markov, autómatas celulares, etc.

En lo que sigue, aceptaremos como verdadera la tesis de Church. Esto se verá materializado, en particular, en la demostración de la indecidibilidad del "problema de la parada" de una máquina de Turing (MT). Además, cuando hablemos de algoritmos, problemas decidibles, funciones computables, etc., se debe entender que nos referimos a estos conceptos modelados a través de máquinas de Turing.

3.2 Un Problema Indecidible

En esta sección consideraremos el problema de la parada de una MT ("halting problem", en inglés). Para plantear el problema partamos de los siguientes hechos:

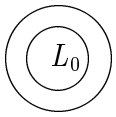
- Una MT, digamos con alfabeto $\{0, 1\}$, es un objeto finito.
- Una tal MT puede ser, a su vez, codificada como una palabra binaria.
- Hay entonces una función de codificación (de la cual no veremos detalles técnicos)

$$M \mapsto \#(M)$$

Planteamos entonces el problema de decidir si una MT arbitraria, con alfabeto binario, alimentada con su propio código como entrada, para o no. Por supuesto, el problema de determinar si una máquina arbitraria, alimentada con entrada también arbitraria, para o no, es más interesante. Sin embargo, este último problema resultará indecidible, si el primero, menos general, resulta indecidible. Nos concentramos, entonces, en el siguiente:

Problema:

$$\sum^* = \{0,1\}^*$$



$$L_0 \;:=\; \{w \in \{0,1\}^* \mid w \text{ es } \#(M) \text{ de alguna MT } M \text{ y } M \text{ con entrada } \#(M) \text{ para } \} \subseteq \sum^*$$

¿Es L_0 decidible?

En este problema alimentamos las máquinas con sus propios códigos. Estas pueden parar o no, y es esto lo que queremos determinar algorítmicamente. Más precisamente, queremos un algoritmo M_0 que, aplicado a cualquier palabra $w \in \sum^*$, decide si w es código de una MT que alimentada con su propio código para o no (es decir, responde SI o NO).

$$\sum^* \ni w \longrightarrow M_0 \begin{cases} \nearrow \text{ SI si } w \in L_0 \\ \searrow \text{ NO si } w \notin L_0 \end{cases}$$

Cuando preguntamos por un algoritmo para decidir L_0, se entiende que es uno programable como MT.

Notar que éste es un problema de naturaleza simbólica y que no hay una razón a priori para que no sea decidible computacionalmente. Sin embargo, la posibilidad de autoreferencia (alimentar máquinas con sus propios códigos) será la base de la demostración de la insolubilidad de este problema de decisión.

Teorema: L_0 es indecidible.

En efecto: Supongamos que existe una MT M_0 que decide L_0 , es decir

$$\sum{}^{*} \ni w \longrightarrow M_0 \begin{cases} \nearrow \text{SI si } w \in L_0 \\ \searrow \text{NO si } w \notin L_0 \end{cases}$$

Usaremos M_0 como subrutina de una MT M_0' tal, que :

$$M_0'(w) \begin{cases} \nearrow \text{SI} & \text{si } M_0(w) \longrightarrow \text{NO} \\ \searrow \text{no para} & \text{si } M_0 \longrightarrow \text{SI} \end{cases}$$

Para construir esta máquina, basta con programar la respuesta SI o una iteración infinita, según la respuesta de M_0.

Entonces:

$$\begin{aligned} M_0'(\#(M_0')) \text{ para} &\Leftrightarrow M_0'(\#(M_0')) \text{ dice SI} \\ &\Leftrightarrow M_0(\#(M_0')) \text{ dice NO} \\ &\Leftrightarrow \#(M_0') \notin L_0 \\ &\Leftrightarrow M_0'(\#(M_0')) \text{ no para} \end{aligned}$$

Esta contradicción nos muestra que no hay ningún algoritmo para decidir si una MT cualquiera, alimentada con su propio código, para o no.

■

Tenemos así nuestro primer problema computacional indecidible. Además de su interés por sí mismo, podremos usar este problema para demostrar que otros problemas también son indecidibles. Esto se hace "reduciendo" el problema L_0 a otro problema que se quiere demostrar indecidible. Por ejemplo, usando este método se puede demostrar que el problema de las ecuaciones diofantinas es indecidible.

La noción de reducibilidad entre problemas la conocemos de la vida diaria: para resolver un problema que no sabemos resolver fácilmente, trasladamos el problema, posiblemente con otra representación, a un escenario que nos es más favorable.

Un ejemplo de reducción que tenemos a la mano es el de la reducción del problema de validez lógica de fórmulas proposicionales al de insatisfacibilidad:

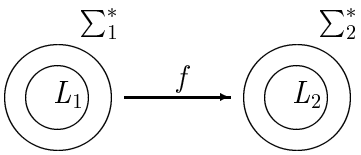
si queremos determinar si una fórmula φ es logicamente válida, nos pasamos a $\neg\varphi$, y determinamos si esta última es satisfacible. Notar que:

- φ es lógicamente válida si y sólo si $\neg\varphi$ es insatisfacible
- el paso de φ a $\neg\varphi$ es computable por un algoritmo general

Estas observaciones nos motivan a dar la siguiente definición general de:

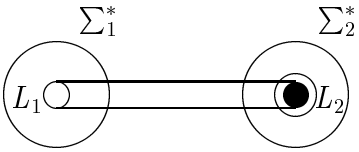
El Concepto de Reducción:

Sean $L_1 \subseteq \Sigma_1^\star$ y $L_2 \subseteq \Sigma_2^\star$. Decimos que L_1 es **reducible** a L_2, y escribimos $L_1 \leq L_2$, si existe una función f de $\Sigma_1^\star$ en $\Sigma_2^\star$



tal, que:

- para todo $w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$
- $f : \sum_1^* \to \sum_2^*$ es computable



$L_1 \leq L_2$

Observación: Si $L_1 \leq L_2$, obtenemos de manera inmediata de la definición los siguientes hechos:

1. L_2 decidible $\Rightarrow$ L_1 decidible.
2. L_1 indecidible $\Rightarrow$ L_2 indecidible.

En efecto, si, como en 1., L_2 es decidible, entonces, para obtener un algoritmo de decisión para L_1, basta con computar, para cada $w \in \Sigma_1^\star$, su imagen $f(w)$ y decidir si $f(w) \in L_2$. Por otro lado, si, como en 2., L_1 es indecidible, entonces el problema más general L_2, en el cual estamos sumergiendo L_1 de manera efectiva (computable), tampoco puede ser decidible.

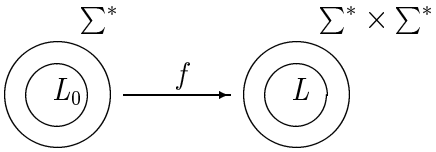
Con el resultado 2. podemos obtener la indecidibilidad de otros problemas computacionales.

Otro Problema Indecidible: El Problema General de la Parada de una MT.

En este problema, las entradas para las máquinas serán arbitrarias. Es de esperar que el problema de parada más general también sea indecidible. Más precisamente, el problema es el siguiente:

$$\begin{aligned} L \;=\; \{(w_1, w_2) \in \textstyle\sum^* \times \sum^* \mid w_1 \text{ es } \#(M) \\ \text{de alguna MT } M, \text{ y } M, \text{ alimentada con } w_2, \text{ para } \} \end{aligned}$$

Podemos suponer que $\Sigma = \{0, 1\}$. Se tiene : $L_0 \leq L$. Es decir, el problema de la parada restringido que ya vimos es reducible a este problema de parada más general:



$w \in L_0 \Leftrightarrow f(w) \in L$

En este caso, la función computable f está definida así: para $w \in \sum^*$, $f(w) := (w, w) \in \sum^* \times \sum^*$. Para ella, se tiene: $w \in L_0 \Leftrightarrow f(w) \in L$.

Por el hecho 2. de la observación anterior, obtenemos que L es indecidible. Es decir, no hay ningún algoritmo que permita decidir si una máquina cualquiera con una entrada cualquiera para o no. Este resultado se refiere a algoritmos programables como MT's , sin embargo, también puede ser demostrado para otros modelos y lenguajes de computación. Por ejemplo, se puede establecer que no hay ningún programa en PASCAL que, aplicado a cualquier programa en PASCAL (junto con su entrada), decida si el programa para o no*.

*Un enfoque de la teoría de computabilidad en términos de lenguajes de programación cercanos a los tradicionales, por ejemplo, PASCAL, puede ser encontrado en "A Programming

3.3 Problemas Recursivamente Enumerables

En la sección anterior vimos que hay problemas de decisión, de naturaleza simbólica, que no son solubles mediante métodos computacionales. Sin embargo, esto no significa que esos problemas no sean susceptibles de ser tratados computacionalmente. Hay una noción más débil que la de decidibilidad, pero que todavía es natural e interesante. Se trata de la noción de "problema recursivamente enumerable".

Ejemplo: Consideremos el conjunto $L(P)$ de fórmulas proposicionales basado en en un conjunto fijo P de letras proposicionales, digamos, finito. Ya sabemos que éste es un conjunto decidible con respecto al conjunto de palabras arbitrarias formadas con letras de P y los símbolos lógicos. Sin embargo, veremos otra propiedad computacional interesante del conjunto $L(P)$.

Hay un algoritmo que echado a funcionar (sin entrada) va entregando una a una todas y sólo las fórmulas del lenguage proposicional $L(P)$. Es fácil construir un tal algoritmo; basta con ir aplicando "sistematicamente'' las reglas inductivas para construir fórmulas de $L(P)$. Esto sólo requiere de que haya una subrutina que vaya generando los elementos de P, lo cual no parece demasiado exigente. Obviamente esto se da, en particular, cuando el conjunto P es finito.

■

Definición: Sea $L \subseteq \Sigma^\star$. Decimos que el L es **recursivamente enumerable** (r.e.) si existe un algoritmo (una MT) que, puesto a funcionar, va entregando una a una todas y sólo las palabras que pertenecen a L.

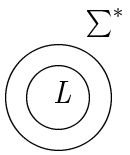
El ejemplo y la definición anteriores nos dicen que $L(P)$ es decidible y recursivamente enumerable.

Observación: Si un problema es r.e., la lista de sus elementos que va generando el algoritmo puede incluir repeticiones y no tiene por qué seguir un orden prescrito. Si exigiéramos que las palabras sean entregadas en un orden prescrito, estaríamos forzando a un problema r.e. a ser, a la vez, decidible, ya que para decidir una palabra cualquiera, bastaría con observar si apareció en la posición predeterminada. Obviamente queremos que el concepto de enumerabilidad recursiva sea un concepto distinto del de decidibilidad.

Proposición: Si L es decidible, entonces es recursivamente enumerable.

En efecto: supongamos que L es decidible. Sea A un algoritmo para decir L.

Approach to Computability", Kfoury, Moll & Arbib, Spriger 1980.



Necesitamos un algoritmo para generar L. Un algoritmo es el siguiente:

1. Vaya generando internamente en forma sistemática (por ejemplo, en orden lexicográfico) todas las palabras $w \in \sum^*$.
2. Para cada una, a medida que se van generando, antes de entregarla como salida, use como subrutina el algoritmo de decisión A para L para decidir si $w \in L$ o no. Si la respuesta de A es SI, entréguese w como salida.

Con esto tenemos un algorimo para enumeración recursiva de L.

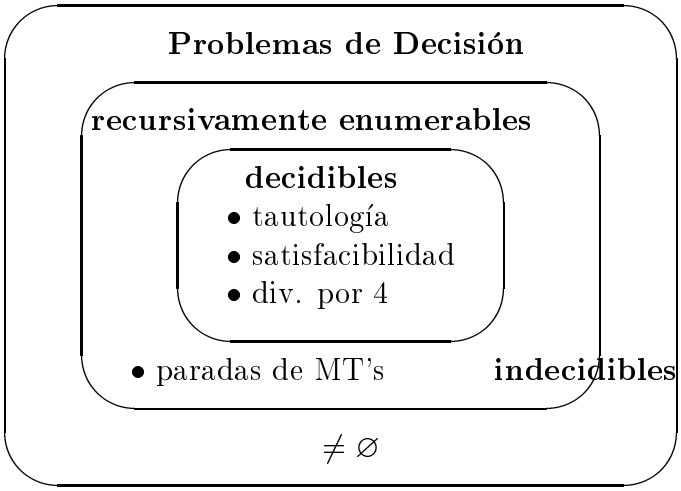
■

Observación:

- El concepto de decidibilidad es más fuerte que el de enumerabilidad recursiva. Es decir, hay problemas que son recursivamente enumerables, pero no decidibles. Por ejemplo, se puede demostrar que el problema de la parada de MT's es recursivamente enumerable, es decir, hay una MT que va entregando uno a uno todos y sólo los códigos de MT's que paran con su propio código como entrada. Veremos otros ejemplos más adelante.
- En primera instancia, se podría pensar en aprovechar un algoritmo de enumeración recursiva de un conjunto L, para decidir L de la siguiente manera: dada una palabra cualquiera $w \in \Sigma^\star$, para determinar si $w \in L$, échese a funcionar el algoritmo de enumeración recursiva y decídase según aparezca w en la lista o no. Este argumento no funciona, pues no tenemos la seguridad de que el algoritmo propuesto va a parar (con la respuesta correcta) después de un número finito de pasos. Si la palabra está en L, obtendremos la respuesta SI en algún momento. Pero en caso contrario, estaremos esperando la respuesta para siempre, y, lo que es peor, sin saberlo.

- Una caracterización útil del concepto de enumerabilidad recursiva es la siguiente: $L \subseteq \sum^*$ es r.e. si y sólo si existe una relación decidible $R \subseteq \sum^* \times \sum^*$ tal, que, para todo $x \in \sum^*$: $x \in L \Leftrightarrow$ existe $y \in \sum^*$ tal, que $(x, y) \in R$.

Tenemos entonces la siguiente situación:



El problema de la parada de máquinas de Turing, digamos, el problema general, aparece como recursivamente enumerable. Esto significa que que hay una máquina de Turing que es capaz de ir listando uno a uno, todos y sólo los pares (código de máquina M, entrada w) tales, que $M(w)$ para.

La idea de la demostración es la siguiente: se va generando sistemática e internamente en una máquina $\mathcal{M}$, los distintos códigos de máquina M y entradas posibles w. Una idea es ir probando (internamente) si $M(w)$ para. Si para, producimos como salida el par (M, w). El problema es que si no para, no lo sabremos y la máquina $\mathcal{M}$ quedará "colgada". Lo que se hace, entonces, es ir simulando internamente, en forma creciente, una cantidad finita de pasos de la computación $M(w)$, pero con pruebas de la misma naturaleza para otras máquinas (M', w') en forma intercalada. Es un proceso sistemático que va generando diversas máquinas, entradas para ellas **y** cantidades finitas (crecientes) de pasos de las máquinas con esas entradas. Cada vez que se llegue a una parada, $\mathcal{M}$ entrega el correspondiente par (M, w). Este proceso permitirá revisar todas las máquinas, sus entradas posibles y sus computaciones finitas.

3.4 Ejercicios

1. Costruya máquinas de Turing que decidan los siguientes lenguajes (como subconjuntos de $\{a, b\}^*$):

(a) $\{ w \mid w$ contiene al menos una $a\}$

(b) $\varnothing$ (el **conjunto** vacío)

(c) $\{\epsilon\}$ (ϵ es la **palabra** vacía)

(d) $\{a\}$

(e) $\{ w \mid |w|$ es par $\}$

2. Construya máquinas de Turing que calculen las siguientes funciones numéricas:

(a) $f(n, m) = n + m$

(b) $f(n, m) =$ el mayor entero menor o igual a n/m si $m > 0$, o 0 si $m = 0$

(c) $f(m, n) = n \dot{-} m$ (definida por $n - m$ si $n - m \geq 0$; y 0, si no)

(Primero debe especificar cuáles son sus convenciones sobre escritura/lectura de inputs/outputs en la cinta.)

3. Costruya máquinas de Turing que calculen las siguientes funciones cuyos argumentos y resultados son palabras de $\{a, b\}^*$:

(a) $f(u, w) = wu$

(b) $f(w) = ww$

(El mismo comentario que en el problema anterior)

4. Demuestre usando reducción que los siguientes problemas son indecidibles:*

(a) Dada una MT arbitraria, ¿para con la cinta vacía? Más precisamente, el siguiente problema es indecidible:

$M_e = \{u \in \{0, 1\}^* \mid u = \#(M)$, donde M es una MT que, echada a funcionar con entrada ϵ, para $\}$.

*Esto se hace usualmente basándose en la indecidibilidad de otros problemas, haciendo la demostración por contradicción, es decir, suponiendo que el problema en cuestión es decidible, y estableciendo que otro problema indecidible sería decidible. Detrás de esto hay una reducción implícita. Se trata, además, de hacer explícita y precisa la reducción subyacente. Por supuesto, también se puede partir directamente de la reducción.

(b) Dada una MT arbitraria, ¿hay alguna palabra de entrada con la que pare?

Solución: Se puede hacer por reducción del problema general de la parada a éste, es decir, si $L_g = \{(u, w) \mid u \text{ es código de máquina } M \text{ y } M(w) \text{ para}\}$, y $L_a = \{v \mid v \text{ es código de una máquina } M \text{ y existe } x \text{ tal, que } M(x) \text{ para}\}$, entonces hay que probar que $L_g \leq L_a$.

La reducción: dado el par (u, w), si u es código de máquina M, asóciese al par (u, v) el código v de una máquina M' que hace lo siguiente: con una entrada cualquiera, digamos x, M' borra de su cinta a x, escribe w y funciona como la máquina original M. Obviamente,

$v \in L_e \Leftrightarrow$ existe x t.q. $M'(x)$ para $\Leftrightarrow M(w)$ para $\Leftrightarrow (u, w) \in L_g$.

(c) Dada una MT arbitraria, ¿para con todas las entradas?

(d) Dadas dos MT's arbitrarias, ¿paran en las mismas entradas?

5. Sea $L \subseteq \Sigma^*$ un problema decidible. ¿Es posible que exista $L' \subseteq L$ que sea indecidible (con respecto a Σ^*)? Demuestre, refute, discuta, ...

6. Demuestre que la función predecesor $p : \mathbb{N} \to \mathbb{N}$ definida por $p(n) = max(n - 1, 0)$ es (Turing–) computable.

7. Demuestre que las siguientes relaciones entre números naturales son (Turing–) decidibles:

(a) $\{ (m, n) \mid m \mid n\}$ (divisibilidad)

(b) $\{ n \mid n$ es primo $\}$ (primalidad)

8. (a) Demuestre que si R, S son relaciones decidibles para números naturales, entonces también son decidibles: $R \cap S$, $R \cup S$, $R - S$.

(b) Sea $M \subseteq \mathbb{N}$ decidible. Demuestre que también lo es $M \cup \{n\}$ para $n \in \mathbb{N}$ fijo. En particular, obtenga que todo subconjunto finito de $\mathbb{N}$ es decidible.

9. Sea $f : \mathbb{N} \to \mathbb{N}$ computable. Demuestre (informalmente, intuitivamente):

(a) Si f es estrictamente creciente, entonces el recorrido de f, $Rec(f)$, es un conjunto decidible.

(b) Si f es inyectiva (1 a 1), entonces f^{-1}, la inversa de f con dominio $Rec(f)$, también es computable.

10. Demuestre y discuta la siguiente caracterización de los problemas recursi-

vamente enumerables:

$L \subseteq D$ es r.e. $\Leftrightarrow$ existe una relación $R \subseteq D \times D$ decidible tal que: para todo $x \in D$: $x \in L \Leftrightarrow \exists\, y \in D$ tal, que $(x, y) \in R$.

3.5 Funciones Recursivas

En esta sección presentaremos las llamadas funciones recursivas, un modelo de computación alternativo al de las máquinas de Turing, que se originó en trabajos de Gödel, Church y Kleene. Es posible presentar el modelo general, que permite trabajar con alfabetos tan generales como los que manejan las máquinas de Turing. Sin embargo, por razones históricas y de simplicidad en la presentación, nos concentraremos solamente en el cómputo de funciones de números naturales.

La clase de las funciones recursivas es la menor clase de funciones de números naturales que satisface los siguientes requerimientos:

(R1) La **función cero** $Z : \mathbb{N} \to \mathbb{N}$, definida por $Z(n) = 0$, para todo $n \in \mathbb{N}$, es una función recursiva.

(R2) La **función sucesor** $S : \mathbb{N} \to \mathbb{N}$ es función recursiva.

(R3) Para cada $n \geq 1$ y cada $1 \leq i \leq n$, la **función proyección** $P_{ni} : \mathbb{N}^n \to \mathbb{N}$, definida por $P(x_1, \ldots, x_n) = x_i$, es función recursiva.

(R4) Si $g : \mathbb{N}^s \to \mathbb{N}$ y $g_i : \mathbb{N}^t \to \mathbb{N}$ son funciones recursivas, entonces también lo es la **función composición** $f : \mathbb{N}^t \to \mathbb{N}$ definida por $f(\bar{x}) = g(g_1(\bar{x}), \ldots, g_s(\bar{x}))$.

(R5) Si $g : \mathbb{N}^{n+2} \to \mathbb{N}$ y $h : \mathbb{N}^n \to \mathbb{N}$ son funciones recursivas, entonces también lo es la función $f : \mathbb{N}^{n+1} \to \mathbb{N}$ definida por las ecuaciones de **recursión primitiva**:
$f(\bar{x}, 0) = h(\bar{x})$
$f(\bar{x}, S(y)) = g(\bar{x}, y, f(\bar{x}, y))$.

(R6) Si $g : \mathbb{N}^{n+1} \to \mathbb{N}$ es función recursiva, entonces también lo es la función $f : \mathbb{N}^n \to \mathbb{N}$ definida por el **operador de minimización**:
$f(\bar{x}) = \mu\, y\, \{g(\bar{x}, y) = 0\}$ (el menor y tal, que $g(\bar{x}, y)$ se hace cero).

Las cláusulas (R1) – (R6), de Gödel y Kleene, se pueden ver como reglas para construir funciones recursivas a partir de otras funciones recursivas más simples, tal como se construyen fórmulas de un lenguaje proposicional, siguiendo un esquema de definición inductivo.

Ejemplo: La siguiente es una definición por recursión primitiva de la función suma de naturales:

$suma(n, 0) = n$
$suma(n, S(m)) = S(suma(n, m))$.

Dado que, por la regla (R1), la función sucesor es recursiva, también la suma es recursiva. Esto se obtiene por la regla (R5).

Ejemplo: Es fácil ver que la función $F(n, m)$, que da 0 cuando n no divide a m y es mayor que 0, y 1 si no, es recursiva (ver ejercicios de esta sección).

Entonces, la siguiente función también es recursiva:

$f(m) = \mu\ n\ (F(n, m) = 0)$ (el menor n tal, que $F(m, n) = 0$)

Por ejemplo, $f(6) = 4$.

Vemos que el operador μ es un operador de búsqueda. Eventualmente, la búsqueda puede ser infinita, por lo cual el operador puede conducir a funciones definidas parcialmente (no totales). Esto no es extraño si pensamos que las máquinas de Turing, por ejemplo, también pueden definir funciones computables no totales, al haber computaciones infinitas.

■

Este modelo pretende capturar la noción intuitiva de función computable (de números naturales). De hecho, se puede probar que esta clase de funciones coincide exactamente con las funciones (numéricas) que son calculables mediante máquinas de Turing. Esto se puede demostrar de manera precisa, matemática. Para demostrar que toda función recursiva es Turing–computable, basta con construir máquinas de Turing que calculen las funciones recursivas básicas (reglas (R1) – (R3)), y máquinas de Turing que calculen las funciones recursivas definidas por composición, recursión primitiva y minimización, cuando ya se cuenta con máquinas (subrutinas) que calculan las funciones en las que se basan esas definiciones. Por el otro lado, para demostrar que toda función Turing–computable es recursiva, lo que se hace –y esto es más complejo– es aritmetizar (codificar de manera numérica) las máquinas de Turing y describir su funcionamiento a través de funciones numéricas que resultan ser recursivas.

Notar que la regla (R6) puede introducir funciones recursivas parciales (parcialmente definidas) a través de la minimización irrestricta: puede no haber

un y para el cual $g(\bar{x}, y) = 0$. Esto no es extraño, una máquina de Turing también puede definir una función parcial si es que no para con ciertas entradas.

Las funciones construidas con las reglas (R1) – (R5) se llaman "recursivas primitivas". A las funciones construidas con las reglas (R1) – (R6) tambíen se les llama *μ–recursivas* (por el operador μ).

3.5.1 Ejercicios

1. Demostrar que las funciones recursivas primitivas son totales, es decir, están siempre definidas (en su dominio natural).

2. Demostrar que hay funciones computables y totales (en el sentido intuitivo) que no son recursivas primitivas. Para esto utilice un argumento clásico de "diagonalización": liste todas las funciones recursivas primitivas en una matriz infinita enumerable (¿por qué es posible?), en seguida, construya una nueva función yéndose por la diagonal y haciendo que la nueva función difiera de cada una de las funciones listadas. Llene los detalles mostrando que la función así construida no puede ser recursiva primitiva y que es computable.

Este método de diagonalización se usa también para demostrar que el conjunto de números reales del intervalo $[0, 1]$ no es enumerable. Para esto se supone que se puede enumerar los números en una lista
$0, d_{11}\, d_{12}\, d_{13} \ldots$
$0, d_{21}\, d_{22}\, d_{23} \ldots$
$0, d_{31}\, d_{32}\, d_{33} \ldots$
$\ldots$
y se construye un nuevo número real que no está en la lista, yéndose por la diagonal, haciendo que difiera del primero en d_{11}, del segundo, en d_{22}, del tercero, en d_{33}, etc.

3. (a) Trate de aplicar el mismo argumento para mostrar que hay una función computable que no es μ–recursiva (es decir, ahora se acepta minimización). ¿Por qué no funciona?

(b) Modifique la regla (R6) de modo que el operador μ dé siempre funciones totales, aplicando $\mu\, y\ \ g(\bar{x}, y)$ sólo si existe y tal, que $g(\bar{x}, y) = 0$. Trate de aplicar el argumento anterior para demostrar que existe una función total computable no obtenible con las reglas (R1) – (R5), (R6)'. ¿Qué ocurre?

4. Demuestre que las siguientes funciones son computables a partir de las reglas (R1) – (R4):

1. $f(x_2, x_3, x_1)$ (en este caso, más el hecho que $f(x_1, x_2, x_3)$ es computable)
2. $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$ (en este caso, más el hecho que la suma con dos argumentos es computable)
3. La función constante m con $m \in \mathbb{N}$
4. la función $h(x) = f(x, m)$, con $m \in \mathbb{N}$ (en este caso, más el hecho que $f(x, y)$ es computable)

5. Demuestre que la suma, producto, factorial y exponenciación son recursivas primitivas (obtenibles con las reglas (R1) – (R5)).

6. Demuestre que las siguientes funciones son recursivas primitivas: la resta de 1, la diferencia truncada (si es negativa, se deja en 0), la función signo (1 si es positivo, y 0, si no), la otra función signo (1 si es 0, y 0, si no), el módulo de la diferencia, el mínimo de dos números, el máximo de dos números, el resto de la división de dos números (el resto de la división por 0 de un número da el número), el cuociente de la división de dos números (el cuociente de la división de 0 es 0), la función que da 1 si el primer argumento divide al segundo y 0, si no (0 divide a 0, pero 0 no divide a ningún otro)*.

7. Demuestre que las siguientes funciones son μ–recursivas: la función que indica el número de divisores de un número (el número de divisores de 0 es 1); la función que da 1 si el número es primo y 0, si no; la función definida por $f(x) = x$–ésimo número primo ($f(0) := 0$, $f(1) := 2$, etc.); la función definida por $g(x, y) =$ el exponente del y–ésimo número primo en la factorización prima de x ($g(0, 0) := 0$).

(En realidad, no se necesita minimización de manera esencial para demostrar la computabilidad de estas funciones, pero es más fácil con ella.)

8. El siguiente es un ejemplo de una función total concreta que es μ–recursiva, pero no es recursiva primitiva†. Ella es la función de Ackerman, y el ejemplo clásico de una función cuya μ–computabilidad requiere de minimización.

$ack(0, y) = y + 1$
$ack(y + 1, 0) = ack(y, 1)$
$ack(y + 1, x + 1) = ack(y, ack(y + 1, x))$.

*De acuerdo con esto, el predicado de divisibilidad $D(x, y)$ sería decidible, pues su función característica lo es.

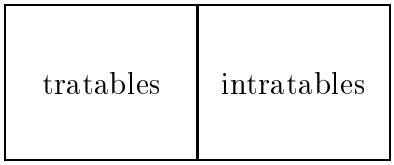
†Se puede demostrar que ella crece más rápido que cualquier función recursiva primitiva.

¿Qué diferencia hay entre esta recursión y la primitiva? ¿Cómo se podría demostrar que estas ecuaciones de recursión definen una función total de números naturales? Demuestre que $ack(2, y) = 2y + 3$. ¿Cuánto es $ack(3, 3)$, $ack(4, 4)$? ¿Por qué esta función sería computable (intuitivamente)?

Capítulo 4

Complejidad de Algoritmos de Decisión

En el capítulo anterior hicimos una separación entre los problemas de decisión que son indecidibles y los que son decidibles. Sin embargo, esta distinción, aunque importante y fundamental, no aporta mucha información al ser aplicada a muchos problemas de decisión que nos aparecen cotideanamente. Es especialmente importante poder discriminar, dentro de la clase de problemas decidibles, entre los que pueden ser resueltos fácilmente y los que no tienen solución fácil. O, para usar la terminología estándar, entre los problemas decidibles que son tratables y los que son intratables.



Surgen varias observaciones obvias:

- Tanto problemas tratables como intratables son decidibles, es decir, dejamos a los problemas indecidibles fuera de estas categorías.
- Tenemos que definir razonablemente el concepto de tratabilidad.
- Con respecto a los problemas intratables, se trata de que para ellos no hay algoritmos de decisión eficientes (otro término por definir) por motivos fundamentales. Es decir, no se trata sólo de que hasta ahora no se ha descubierto algoritmos de decisión eficientes, sino de que nunca los habrá.

- En este sentido, la separación es tan fundamental como la separación entre problemas decidibles e indecidibles.
- Al hacer esta separación, cuando nos referimos a algoritmos, nos referimos a los algoritmos programables como MT's como las ya vistas.

Veamos algunos ejemplos:

Ejemplo: Consideremos nuevamente el problema de divisibilidad por 4 de números naturales en representación binaria. Tenemos un algoritmo de decisión concreto. Veamos cuánto tiempo requiere ese algoritmo para decidir sobre la divisibilidad por 4 de una palabra w de largo n. Como el algoritmo recorre la palabra hasta detectar su extremo derecho, y luego verifica los últimos dos dígitos, el número de pasos de la MT (el número de aplicaciones de la función de transición de la MT) está acotado superiormente por, digamos, $2 \mid w \mid (= 2n)$, es decir, por una expresión polinomial en el largo n de la entrada (en este caso, el polinomio $p(x) = 2x$). El problema de divisibilidad por 4 lo consideraríamos tratable.

■

Ejemplo: El problema de ordenar un archivo por sus llaves es considerado tratable (no es un problema de decisión, pero igual ilustra el punto), pues existe (más de) un algoritmo que lo resuelve en forma que consideraríamos eficiente. Por ejemplo, si usamos el algoritmo de "ordenación por mezcla (de dos archivos ordenados en un tercer archivo ordenado)", el tiempo de ejecución (en este caso el número de comparaciones de dos elementos), en términos del largo n (número de llaves) del archivo original, está acotado superiormente por $cn \log n$, para alguna constante c. Nuevamente tenemos una cota superior polinomial, por ejemplo, cn^2, para el tiempo de ejecución.

■

Ejemplo: Retomemos el problema de satisfacibilidad de fórmulas proposicionales. Este problema es decidible. Sea φ una fórmula proposicional. Podemos medir el tamaño de φ como entrada para un algoritmo a través del número de apariciones de letras proposicionales (no necesariamente distintas)*.

*Una medida tal vez más natural sería el largo de φ como palabra, es decir, el número de símbolos en ella. Sin embargo, en general, exceptuando casos poco naturales como una larga sucesión de símbolos de negación antepuestos uno al otro, el largo de la fórmula como palabra estará acotado superiormente por un polinomio en el número de apariciones de letras proposicionales.

Entonces, si φ es de tamaño n, en el peor de los casos, el algoritmo basado en tablas de verdad tendría que verificar 2^n valuaciones para decidir si la fórmula es satisfacible o no. En consecuencia, el algoritmo ejecutaría al menos 2^n pasos, lo que es una cantidad exponencial en términos del tamaño de la entrada . Los peores casos de tamaño n para este algoritmo surgen cuando φ tiene n letras proposicionales distintas y la última valuación considerada es la que satisface la fórmula.

Dado que no podemos consider a este algoritmo como eficiente, nos preguntamos si existe algún algoritmo para decidir satisfacibilidad que sea eficiente.

■

Definición: Un algoritmo Al que resuelve un problema de decisión es considerado eficiente cuando, para cada instancia w (elemento de D) del problema de decisión, el número de pasos $T_{Al}(w)$ que da el algoritmo hasta dar la respuesta SI o NO para w, está acotado superiormente por $p(\mid w \mid)$, donde $p(x)$ es un polinomio (dependiente del algoritmo y fijo para todas las entradas) y $\mid w \mid$ es el tamaño de w.* Esta cota superior se entiende de manera asintótica, es decir, es válida para $\mid w \mid$ suficientemente grande.

Otra manera de ver la definición es la siguiente: existe un polinomio $p(x)$ tal, que, para cada número natural n suficientemente grande, el número de pasos del algoritmo, en el peor de los casos con entradas de tamaño n, está acotado superiormente por $p(n)$:

$$\text{Para todo } n \geq n_0 : \quad T_{Al}(n) \ := \ max_{|w|=n} \, T_{Al}(w) \quad \leq p(n).$$

Observaciones:

- Aquí estamos definiendo una medida de complejidad de algoritmos, en este caso, la medida "tiempo" o "número de pasos". Podríamos considerar otras medidas, como "espacio" (en el caso de MT's, la cantidad de celdas de la cinta visitadas durante la ejecución).
- Obviamente debe haber relaciones cuantitativas entre medidas de complejidad, como tiempo vs. espacio. Sin embargo, en esta introducción nos limitaremos a complejidad temporal.

*Consideraremos que el tamaño de w es un número natural, que usualmente va a ser el largo de w como palabra, aunque podría ser algún otro parámetro numérico asociado naturalmente al problema representado por w, y que dé cuenta de su tamaño.

- Otro aspecto importante de la definición de eficiencia que dimos, es que está basada en los peores casos (aquellos en que el algoritmo se demora más en responder). Se podría considerar una definición basada, por ejemplo, en un promedio (ponderado o simple) de tiempos de ejecución, para entradas de un mismo tamaño.
- Podría objetarse, con fundamentos, que considerar a un algoritmo como eficiente por el solo hecho de que su tiempo de ejecución está acotado por un polinomio en el tamaño de la entrada, es poco realista. El grado del polinomio podría ser muy alto. Sería natural, entonces, hacer posteriormente un análisis más fino, en el contexto de lo polinomial.

Definición: Un problema de decisión es considerado tratable si es soluble mediante un algoritmo eficiente, es decir, de tiempo polinomial.

Notación: La clase de problemas de decisión tratables se denota con $\mathcal{P}$. Esta es, entonces, la clase de problemas solubles mediante algoritmos (MT's) de tiempo polinomial.

Sigue pendiente la pregunta por la tratabilidad del problema de satisfacibilidad de fórmulas proposicionales. ¿Por qué estamos particularmente interesados en este problema? Algunas razones son las siguientes

- Nos surgió naturalmente en el contexto de la lógica proposicional.
- Su análisis de complejidad puede ser paradigmático para el análisis de otros problemas de decisión.
- Está relacionado estrechamente, en términos de complejidad y codificación mutua, con otros problemas de decisión importantes y naturales.

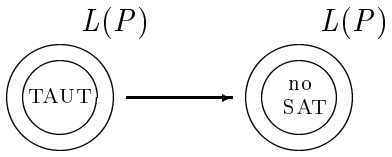
La siguiente proposición ilusta en parte el último punto. Veremos otros ejemplos más adelante.

Proposición: El problema de decidir satisfacibilidad es tan difícil como el problema de decidir tautología.

Demostración: Para demostrar esta proposición es necesario precisar qué significa que un problema sea tan difícil como otro, o al menos (o a lo más) tan difícil como otro. En este caso, viene en nuestra ayuda el concepto de reducibilidad entre problemas de decisión. La única diferencia con lo visto antes, es que ahora consideraremos no sólo aspectos cualitativos, sino también un análisis cuantitativo del costo de las reducciones.

Sabemos que para toda fórmula proposicional φ, se tiene: φ es tautología $\Leftrightarrow$ $\neg\varphi$ no es satifacible.

Tenemos entonces la siguiente reducción de "tautología" a "no satisfacibilidad":



¿TAUT $\ni$? $\quad \varphi \longmapsto \neg\varphi \quad$ ¿$\in$ no SAT?

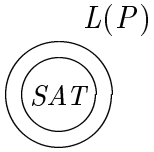
Para decidir si φ es tautología, reducimos el problema a "no satisfacibilidad". Esto cuesta muy poco, un tiempo constante, independiente de φ, el que toma anteponer a φ la negación $\neg$. En seguida, decidimos si $\neg\varphi$ es insatisfacible. Pero es claro que no sólo "satisfacibilidad" es decidible si y sólo si "no satisfacibilidad" lo es, sino que los problemas son igualmente difíciles (o fáciles), ya que, al ser problemas complementarios, el mismo algoritmo sirve para decidir ambos (basta con cruzar las respuestas SI y NO). Luego, decidir "tautología" no es más difícil que decidir "satisfacibilidad" (excepto por el tiempo constante de la reducción).

El hecho "φ satisfacible $\Leftrightarrow$ $\neg\varphi$ no es tautología" y un análisis similar al anterior nos muestran que decidir "satisfacibilidad" no es más difícil que decidir "tautología". En consecuencia, ambos problemas son igualmente difíciles.

■

4.1 Complejidad del Problema de Satisfacibilidad

Retomemos el problema de satisfacibilidad de fórmulas proposicionales:



$\varphi \in SAT \;\Leftrightarrow\; \varphi$ es satisfacible

¿Existe un algoritmo de tiempo polinomial que decide la pertenecia a SAT? Es decir, que, para toda fórmula φ, responde SI si $\varphi \in SAT$ y NO si $\varphi \notin SAT$, y, además, el número de pasos hasta responder es $\leq p(\mid \varphi \mid)$.

Hasta hoy (1995), no se conoce ningún algoritmo eficiente para decidir satisfacibilidad de fórmulas proposicionales. Tampoco se ha probado que no existe un tal algoritmo. Es decir, no sabemos si el problema de satisfacibilidad de fórmulas proposicionales está en la clase $\mathcal{P}$ de los problemas de decisión solubles en tiempo polinomial. ¿Sabemos algo más sobre este problema?

Se tiene lo siguiente:

- El problema de satisfacibilidad es difícil.
- Más precisamente, hay mucha evidencia contundente en contra de la decidibilidad eficiente de este problema.
- Es muy posible que no exista un algoritmo de decisión de tiempo polinomial que lo resuelva.
- Esto no está demostrado (ni refutado) aún.

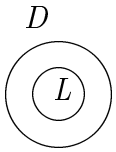
A continuación queremos mostrar la evidencia de que el problema es difícil. En torno a este problema se generó toda una teoría matemática en ciencia de la computación de enorme alcance y repercusiones.

Hay por lo menos dos tipos de evidencia en favor de la dificultad (o intratabilidad) del problema de satisfacibilidad:

- Como dijimos, no disponemos hasta ahora de un algoritmo de tiempo polinomial.
- Pero, más contudentemente, si se pudiera resolver este problema eficientemente, entonces (y esto sí está demostrado) se podría resolver eficientemente muchos otros problemas de decisión y optimización que son muy importantes desde los puntos de vista teórico y práctico, para los cuales hasta hoy tampoco hay algoritmos eficientes. Obviamente, este traspaso de algoritmos de un problema a otro se establece a través del concepto de reducción eficiente.

4.2 Algunos Problemas de Decisión

Presentaremos ahora algunos otros problemas de decisión, y más adelante, veremos cómo están están relacionados con el problema de satisfacibilidad. Recordemos que un problema de decisión se ve de la siguiente manera:



D : dominio de instancias admisibles ($D \subseteq \Sigma^*$)

$L \subseteq D$, conjunto de instancias positivas

Buscamos un algoritmo que aplicado a cualquier $w \in D$, responde :

- SI si $w \in L$
- NO si $w \in D - L$

Problema 1: Problema del Vendedor Viajero

instancia (entrada): un número $n \in \mathbb{N}$ y una matriz de orden $n \times n$ de números no negativos (se puede ver como la matriz de costos de transitar de manera directa entre cada par de ciudades)

salida: una trayectoria cerrada que pasa una vez y sólo una vez por cada ciudad y tiene costo total mínimo (también podría ser sólo el costo de una trayectoria mínima, sin preocuparnos de la trayectoria propiamente tal).

- Este no es un problema de decisión, sino de optimización (lo denotamos con POVV).
- Lo transformaremos en problema de decisión (PDVV):

  instancia: como antes más un numero b.

  pregunta: ¿existe una trayectoria simple de costo $\leq b$?

  b : se llama el "presupuesto" o cota.

  Este sí es un problema de decisión.

- Nos preguntamos por la relación de PDVV con los dos problemas de optimización anteriores. ¿Cuál es más difícil? ¿Si podemos resolver uno, podremos resolver los otros usando el algoritmo para el primero como subrutina? (ver ejercicio 2)
- Un algoritmo ingenuo para PDVV:

  1. Calcular los costos totales de las $(n-1)!$ posibles tratectorias simples y cerradas (supongamos que pueden partir todas de la ciudad 1).

2. Ir comparando uno a uno cada costo total con el presupuesto b.

¡¡ Esto da tiempo exponencial en el largo de la entrada!!

Problema 2: Circuito Hamiltoneano (CH)

instancia: un grafo.

pregunta: ¿hay un ciclo (cerrado) que pasa exactamente una vez por cada uno de los vértices?

Problema 3: Programación Entera (PE)

instancia: un sistema de inecuaciones $A\vec{x} \leq \vec{b}$ con coeficientes en $\mathbb{Z}$

pregunta: ¿existe una solución $\vec{x}$ de ceros y unos?

Problema 4: Coloración con 3 Colores (3–CG)

instancia: un grafo (mapa)

pregunta: ¿se puede colorear con 3 colores los nodos (países) de modo que no haya dos nodos adyacentes (conectados por un arco) con el mismo color?

Problema 5: Subgrafo Completo (CLIQUE)

instancia: un grafo

pregunta: ¿existe un subgrafo tal, que todos los nodos estén conectados entre sí?

Problema 6: Problema de la MOCHILA (Knapsack, en inglés)

Intuitivamente, se trata de seleccionar una cierta cantidad se ítemes para llenar completamente la capacidad de una mochila. Más formalmente:

instancia: números naturales : $m_1, m_2, ..., m_n, c$

pregunta: ¿Se puede obtener c con la suma de algunos de ellos (con distintos subíndices)?

Problema 7: Recubrimiento de Vértices (RV)

instancia: un grafo y un número natural k

pregunta: ¿existe un subconjunto de vértices del grafo de tamaño no superior a k tal, que cada arista del grafo original tiene al menos un extremo en el subconjunto?

Problema 8: SAT

instancia: una fórmula proposicional en forma normal conjuntiva (FNC)*

pregunta: ¿es satisfacible la fórmula?

Esta será nuestra definición oficial del problema SAT. Aunque las fórmulas estén restingidas a FNC, el problema no es más fácil que el de satisfacibilidad de fórmulas proposicionales arbitrarias.

Nótese que este problema puede ser visto como el de determinar consistencia de conjuntos finitos de cláusulas.

Problema 9: SAT(3)

instancia: una formula proposicional en FNC con, a lo más, 3 literales por cláusula (,es decir, por disyunción)

pregunta: ¿es la fórmula satisfacible?

Problema 10: Número Compuesto (NUMCOMP)

instancia: un número natural N

pregunta: ¿existen factores enteros no triviales P y Q de N ($N = P \cdot Q$)?

*En adelante, para uniformar con la literatura existente, nos restringiremos a fórmulas en forma normal conjuntiva (ver sección 2.2.2), es decir, a conjunciones de cláusulas. Veremos que en este caso importante obtenemos una alta complejidad del problema decisión, la que obviamente se traspasa a la clase de todas las fórmulas. Ver también el ejercicio 9.

Obviamente existe un algoritmo para determinar si un número natural arbitrario N es compuesto o no: basta con ir verificando si $2, 3, ..., \sqrt{N}$ dividen a N o no. Este es el algoritmo más inmediato. ¿Cuál es su complejidad en número de pasos?

Uno podría pensar que hay que hacer menos de N chequeos que no son muy costosos (a lo más divisiones). En consecuencia, el algoritmo es de tiempo polinomial en N (¡bastante menos que lineal, de hecho!). Hay que tener cuidado. El número N debe ser entregado a un algoritmo en alguna representación, digamos, binaria, decimal. El tiempo del algoritmo debe ser medido en términos del largo de la representación del número de entrada. Sabemos que el largo de un número, en binario o decimal, es esencialmente $log\ N$ (en base 2 o 10, aunque, por el teorema del cambio de base, esto no es muy relevante). En consecuencia, el número de pasos del algoritmo, sin contar el tiempo de chequeo de la divisibilidad, es, en términos del largo de la entrada, y en el peor caso:

$T(log\ N) \geq \sqrt{N} = N^{\frac{1}{2}} = 2^{(log\ N) \times \frac{1}{2}}$

Es decir, tenemos una cantidad exponencial en el largo de N, es decir, en $log\ N$.

PREGUNTA: ¿Qué tan difíciles son todos estos problemas de decisión?

- Los algoritmos ingenuos para cada uno de ellos (¿cuáles?) son de tiempo exponencial en el tamaño de la entrada.
- **Hasta ahora no hay algoritmos de tiempo polinomial para resolverlos**.
- No se ha probado que son tratables ni que no lo son.
- Tal como SAT, la mayoría de ellos parece ser intrínsecamente difíciles, es decir, intratables.

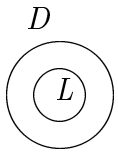
4.3 La Clase $\mathcal{NP}$

Para dar cuenta de la aparente dificultad intrínseca de todos los problemas de la sección anterior, trataremos de enfrentar estos problemas de manera global, insertándolos en una clase natural de problemas.

Para hacer esto, debemos preguntarnos qué tienen todos estos problemas en común. En todos ellos se da un patrón que describiremos a través del problema SAT: dada una fórmula proposicional φ, si ella es satisfacible, entonces hay una

valuación (una solución) σ que se puede verificar (como solución) en tiempo polinomial (en el tamaño del problema de verificación, es decir, en términos de $\mid \varphi \mid + \mid \sigma \mid$). Además, la solución σ es pequeña en comparación con el problema original, más precisamente, $\mid \sigma \mid$ está acotado superiormente por un polinomio (fijo) evaluado en $\mid \varphi \mid$. Por otro lado, las instancias negativas (fórmulas insatisfacibles) no cuentan con tales soluciones σ. En estos casos de instancias negativas, el problema de verificación de presuntas soluciones nunca da la respuesta SI.

En general, para todos los problemas anteriores tenemos:



1. Las instancias positivas (elementos de L), x , tienen certificados de pertenencia, y , cortos (pensar en la solución) que pueden ser verificados en tiempo corto (es decir, en tiempo polinomial).

2. Las instancias negativas no tienen tales certificados de pertenencia.

Intuitivamente, todos éstos son problemas de decisión cuyos correspondientes problemas de verificación de presuntas soluciones son fáciles.

Coleccionaremos, en la clase denotada con $\mathcal{NP}$, todos los problemas con estas dos características. Así, todos los problemas $1 - 10$ anteriores pertenecen a esta clase. La $\mathcal{N}$ antes de la $\mathcal{P}$ (que usamos para la clase de problemas solubles en tiempo polinomial) no es por "no polinomial", sino por la caracterización de esta clase en términos de algoritmos "no deterministas de tiempo polinomial" que veremos a continuación.

Una pregunta que queda pendiente es sobre la relación entre las clases de problemas $\mathcal{P}$ y $\mathcal{NP}$.

4.3.1 Algoritmos No Deterministas

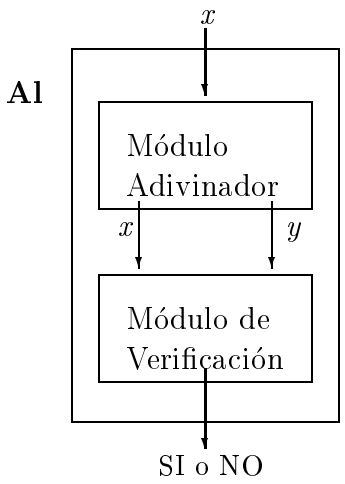
Vamos a dar una definición alternativa para esta clase de problemas de decisión. Tal como ocurre con la clase $\mathcal{P}$, de problemas de decisión solubles por algoritmos de tiempo polinomial, queremos ver a la clase $\mathcal{NP}$ caracterizada como la clase de problemas de decisión que pueden ser resueltos en tiempo

polinomial, pero ya no sólo por algoritmos comunes y corrientes, como los que hemos estado usando (si así fuera, estaríamos declarando por decreto a los problemas dentro de esa clase como pertenecientes a la clase $\mathcal{P}$, lo que no ha sido probado), sino por algorimos de un tipo más general que los ya vistos. Los algoritmos en cuestión son llamados "no deterministas".

¿Qué es un algoritmo no determinista? Intuitivamente, uno que, en contraste con los deterministas, cuando se le alimenta, en distintas ocasiones, con la misma entrada, puede dar lugar a computaciones (en particular, respuestas) distintas. Una definición precisa de algoritmo no determinista se puede dar en términos de máquinas de Turing no deterministas. Lo único que diferencia a éstas de las definidas antes (las deterministas) es que, en lugar de tener una función de transición (que a cada configuración en una computación –determinada por la palabra escrita en la cinta, el símbolo que se lee y el estado actual– asocia a lo más una configuración sucesora), tienen una relación de transición. Esta última puede dar lugar a más de una configuración sucesora, y ahí está presente el no determinismo.

Aquí daremos otra caracterización de algoritmo no determinista, una que está más cercana a las características comunes que detectamos en los problemas anteriores.

Un Algoritmo No Determinista



Aquí:

- **Módulo de Adivinación:** dada una entrada x, produce un y en forma no determinista.

- **Módulo de Verificación:** módulo que produce una respuesta SI o NO en forma determinista.

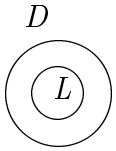
Vemos que:

- Si corremos el algoritmo Al con la misma entrada x distintas veces, podemos obtener distintas y's y, luego, distintas respuestas SI o NO.
- La parte no determinista del algoritmo está concentrada en el módulo adivinador.

Hay dos preguntas que surgen naturalmente:

1. ¿Cuándo podemos decir que un algoritmo no determinista resuelve un problema de decisión? Nótese que para una misma instancia x, este algoritmo podría entregar tanto respuestas SI, como respuestas NO para ella (de hecho, hasta podría dar lugar a computaciones que no paran).
2. ¿Qué significa tiempo polinomial cuando puede, para una misma entrada, producir computaciones de distintos largos, incluso algunas de largo infinito?

Definición: El algoritmo de decisión no determinista resuelve el problema de decisión (D,L) si:



a) $x \in L \Rightarrow$ existe algún y tal que (x, y) produce la respuesta SI (y se llama "certificado de pertenencia" para x);

b) $x \in D - L \Rightarrow$ no existe ningún y tal, que (x, y) produce la respuesta SI.

Con respecto a la segunda pregunta, damos la siguiente

Definición : El algoritmo no determinístico AL decide (D,L) en tiempo polinomial si :

a) $x \in L \Rightarrow$ existe algún y tal que (x, y) produce la respuesta SI;

b) $x \in D - L \Rightarrow$ no existe ningún y tal, que (x, y) produce la respuesta SI;

c) el tiempo de ejecución del algoritmo con el y de (a) es polinomial en el largo de x (este tiempo incluye la producción del posible certificado y).

Como en el caso de los algoritmos deterministas de tiempo polinomial, aquí también el polinomio está asociado al algoritmo y no a la entrada particular; es uniforme para todas las entradas.

Definimos entonces la clase $\mathcal{NP}$, como la clase de problemas de decisión que pueden ser resueltos por un algoritmo no determinista en tiempo polinomial. Esta es una caracterización más precisa que la de la sección anterior y que captura la misma intuición.

Ejemplo: Consideremos el siguiente algoritmo no determinista para decidir NUMCOMP:

Dado un número N, sobre el cual se quiere decidir, se le entrega este número al primer módulo, el adivinador. Este genera un par de números P y Q, con $1 < P, Q < N$. Esto puede ser hecho en tiempo polinomial en el largo (de la representación) de N. El módulo adivinador pasa al segundo módulo, el verificador, las entradas $x = N$ e $y = (P, Q)$. Este segundo módulo multiplica P y Q, verifica si el producto es N, y responde SI (es compuesto) o NO (no es compuesto), según corresponda. Todo esto también puede ser hecho en tiempo polinomial en el largo de N.

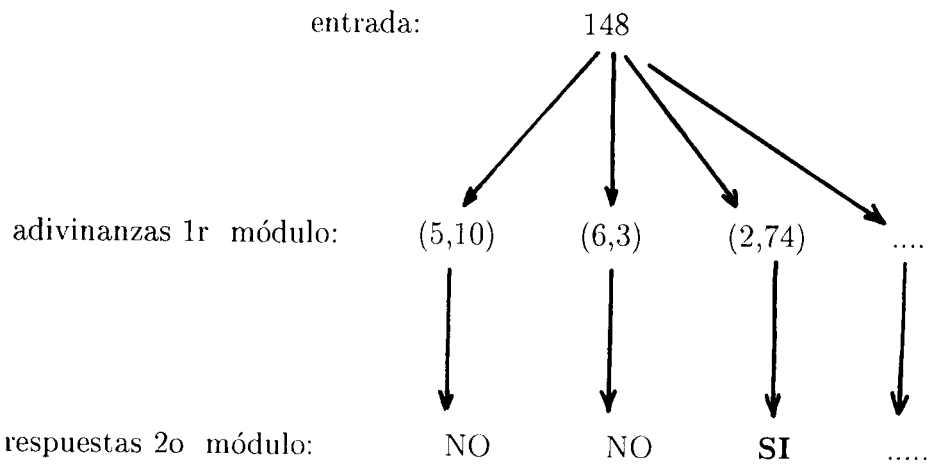
Por ejemplo, si entregamos a este algoritmo el número 148, el primer módulo podría producir los números 12 y 4. El segundo módulo los multiplica, obteniendo 48, que es distinto de 148. En consecuencia, responde NO (es decir, "no es compuesto").

Obviamente, este algoritmo no determinista no ha dado, en este caso, la respuesta correcta. Sin embargo, lo que lo hace un algoritmo de decisión para NUMCOMP es el hecho que existe alguna adivinación (y en consecuencia, alguna computación) que conduce a la respuesta correcta SI, a saber, cuando produce el par de números $(2, 74)$. Con estos números se llega a la respuesta correcta en un tiempo acotado por un polinomio (general, asociado al agoritmo)

correcta en un tiempo acotado por un polinomio (general, asociado al algoritmo) evaluado en el largo de la entrada, es decir, en 3 (el largo del número en base decimal).

Uno se podría sentir tentado a transformar este algoritmo en uno determinista para el mismo fin: basta con esperar o producir todas las posibles adivinanzas de pares (P, Q), con $1 < P, Q < N$, y chequear si algún producto da N. Si es así, respondemos SI, si no, respondemos (correctamente) NO. Esto equivale a seguir todas las posibles computaciones que se producen con el algoritmo no determinista a partir de la entrada N. Es claro que este algoritmo determinista es, en el peor caso, no polinomial en el largo de N, ya que hay que chequear $O(N)$ (o, más precisamente, $O(\sqrt{N})$) adivinanzas para estar seguro de la respuesta de este algoritmo. Esta(s) cantidad(es) es (son) más que polinomial en el largo de N.

En suma, quedarse con una computación de un algoritmo no determinista es peligroso en términos de la veracidad de la respuesta. Por otro lado, tratar de generar un algoritmo determinista a partir de un no determinista por consideración exhaustiva de las posibles computaciones, es, si bien seguro desde el punto de vista de la veracidad de la respuesta, complejo en términos temporales: es fácil y usual pasar de lo polinomial (no determinista) a lo exponencial (determinista).



Algunas Trayectorias de Computación del Algoritmo No Determinista

Ejemplo: Existe un algoritmo no determinista de tiempo polinomial para decidir SAT:

- Entrada: una fórmula proposicional φ en FNC.
- El primer módulo (adivinador) produce (en forma no determinista) una valuación (sucesión de ceros y unos) σ. Finalmente entrega φ y σ al segundo módulo.
- El segundo módulo verifica (en forma determinista) si la valuación satisface la fórmula y responde SI o NO.

Ejemplo: Existe un algoritmo no determinista de tiempo polinomial para decidir PDVV:

- Entrada: una instancia (n, Matriz, b).
- El primer módulo adivina (genera en forma no determinista) una trayectoria simple y cerrada (una permutación de $2, \ldots, n$).
- El segundo módulo calcula el costo de esa trayectoria y lo compara con b y se responde si es menor o no.

■

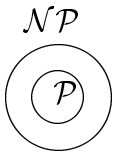
De las definiciones de las clases $\mathcal{P}$ y $\mathcal{NP}$ vemos que la siguiente inclusión es inmediata:

$$\mathcal{P} \subseteq \mathcal{NP}$$

Esto se desprende del hecho que todo algoritmo determinista se puede ver trivialmente como un algoritmo no determinista.

4.4 La Clasificación $\mathcal{P}$ vs. $\mathcal{NP}$

En la clase $\mathcal{NP}$ hemos reunido muchos problemas importantes y naturales con varias características en común. Tenemos la siguiente situación:



La pregunta natural y fundamental es si la inclusión inversa es verdadera: ¿$\mathcal{NP} \subseteq \mathcal{P}$?.

Notar que una repuesta positiva a esta pregunta nos diría que $\mathcal{P} = \mathcal{NP}$, y en consecuencia, que todos los problemas 1.–10. que mencionamos tendrían una solución determinista de tiempo polinomial. Como para ninguno de ellos (ni para muchos otros problemas importantes en la clase $\mathcal{NP}$) tenemos en este momento algoritmos deterministas de tiempo polinomial ni se ha podido probar que los hay (aunque no se exhiban), se considera muy poco probable que existan, y en consecuencia, se conjetura que la inclusión inversa no es verdadera:

Conjetura de Cook* : $\mathcal{P} \subsetneq \mathcal{NP}$, es decir, $\mathcal{P} \neq \mathcal{NP}$.

Este es posiblemente el problema abierto más importante y famoso en ciencia de la computación.

Más aún, Cook estableció bases sólidas para conjeturar que el problema SAT pertenece a $\mathcal{NP} - \mathcal{P}$, es decir, que SAT no tiene solución determinista de tiempo polinomial. Esto por razones más profundas que los ingentes esfuerzos infructuosos de mucha gente buscando algoritmos deterministas de tiempo polinomial para resolverlo. Lo que demostró Cook es que SAT es el (uno de los) problema(s) más difícil(es) de la clase $\mathcal{NP}$.

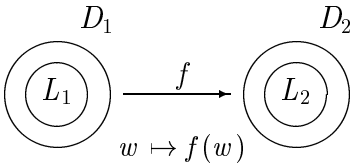
Dado que se trata de comparar dificultad de problemas que están en la clase $\mathcal{NP}$, parece natural referirse al concepto de reducción entre problemas. Más aún, queremos hacer reducciones que nos mantengan dentro de la clase. Una manera de hacer esto es retringiéndose a aquellas reducciones que pueden ser hechas en tiempo polinomial. Estas consideraciones son las que trata la teoría de NP–completidad. SAT resulta ser lo que se llama un "problema NP–completo".

*Por Steve Cook, quien, en 1971, hizo la clasificación que estamos presentando, definió los primeros conceptos relevantes y demostró los primero resultado fundamentales. Su trabajo en esta área le significó ganar el Turing Award, el premio más importante en ciencia de la computación.

Definición: Un problema de decisión L es NP–completo si:

1. $L \in \mathcal{NP}$ y
2. Todo problema de la clase $\mathcal{NP}$ es reducible en tiempo polinomial a L: para todo $L' \in \mathcal{NP}$, se tiene: $L' \leq_{pol} L$.

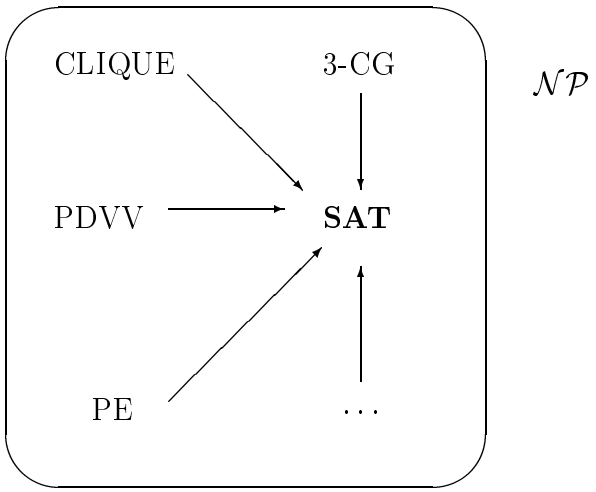
La noción de reducción en tiempo polinomial, denotada con $\leq_{pol}$, es una modificación natural de la noción que ya conocemos de reducción entre problemas de decisión. La función de reducción, además de ser computable, debe serlo en tiempo polinomial:



$$w \in L_1 \Leftrightarrow f(w) \in L_2$$

Para cada instancia w de D_1, $f(w)$ se computa en tiempo acotado por $p(\mid w \mid)$. Donde $p(x)$ es un polinomio fijo, asociado a f.

Teorema (de Cook): SAT es NP–completo.



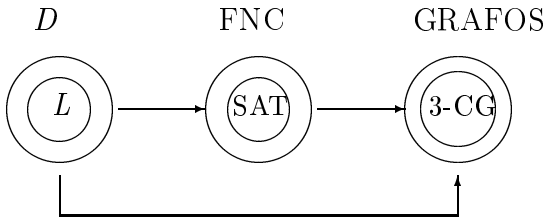
Del teorema de Cook se desprende que SAT es el problema más difícil de la clase $\mathcal{NP}$, ya que todo problema de la clase $\mathcal{NP}$ es reducible en tiempo polinomial

a SAT. En cosecuencia, si hay un algoritmo determinista de tiempo polinomial para SAT, habrá algoritmos deterministas de tiempo polinomial para cada problema de la clase NP, en particular, para todos los otros problemas de decisión que hemos estado considerando.

Este hecho hace poco probable que haya algoritmos deterministas de tiempo para SAT. Notar que si los hubiera, se tendría que $\mathcal{P} = \mathcal{NP}$, lo que contradiría la conjetura de Cook.

Una vez que se tuvo el primer problema NP–completo, en este caso SAT, se demostró que otros problemas también lo son. Esto se hizo explotando la técnica de reducción polinomial*. De este modo se estableció que los problemas PDVV, 3–CG, CH, RV, PE, CLIQUE, MOCHILA, y muchos otros de la clase $\mathcal{NP}$ son NP–completos†.

Para demostrar, por ejemplo, que 3–CG es NP–completo, es decir, que todo problema de la clase $\mathcal{NP}$ es reducible a él en tiempo polinomial, basta con reducir a 3–CG, otro problema que ya sea conocido como NP–completo, por ejemplo, SAT. Esto se debe a la transitividad de la relación $\leq_{pol}$:



Tenemos, entonces, un método general para demostrar que un problema es NP-completo: basta con reducir a él, en tiempo polinomial, otro problema que es NP–completo.

Tenemos así varios problemas NP–completos. Estos son los problemas más difíciles de la clase $\mathcal{NP}$. Si creemos en la conjetura de Cook, tenemos razones contundentes para desalentarnos al saber que nuestro problema favorito es NP–completo. Sobre esta base, sería intratable‡.

---

*A esto contribuyó considerablemente Richard Karp en 1972. Por este trabajo él también ganó el Turing Award.

†No está establecido que NUMCOMP sea NP–completo, probablemente no lo es.

‡Gran parte de la inteligencia artificial y de la teoría de algoritmos tiene que ver con formas de atacar problemas NP–completos. Las alternativas a resolverlos en forma completa eficientemente son: diseñar heurísticas que no tienen asegurado funcionamiemto eficiente o correcto con algunas instancias, métodos que entregan soluciones aproximadas, algoritmos con una componente de aleatoridad que entregan una respuesta con cierta probabilidad baja de error, métodos particulares para instancias de cierto tipo, etc.

Ejemplo: Demostrar que CLIQUE es NP–completo.

Solución: Por la transitividad de las reducciones polinomiales, y porque $CLIQUE \in \mathcal{NP}$ (¿por qué?), basta con demostrar que: $SAT \leq_{pol} CLIQUE$. Es decir, tenemos que encontar una función computable en tiempo polinomial f tal, que si x codifica una fórmula φ en FNC, entonces $f(x)$ codifica un grafo G y un entero k tal, que: φ es satisfacible si y sólo si G tiene un subgrafo completo de tamaño k (la misma construcción que sigue funciona si pedimos "de tamaño $\geq k$").

Veamos cómo construir G, k a partir de φ: primero, k es el número de cláusulas en φ, es decir, φ es de la forma $C_1 \wedge \cdots \wedge C_k$.

Ahora, cada literal en φ (esté repetido o no) aporta un nodo al grafo G, es decir, si C_i tiene m literales, esta clásula aporta m literales (nodos) a G. Con respecto a los arcos, colocamos un arco uniendo a los nodos i y j si y sólo si:

- i y j provienen de diferentes clásulas, y
- i y j (mejor, sus correspondientes literales) no son complementarios, es decir, no es uno la negación del otro.

Por ejemplo, la fórmula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$ da lugar a $k = 3$ y a un grafo con 6 nodos. Por ejemplo, a partir del x_1 de la primera cláusula, se generan arcos sólo hacia $\neg x_2, x_3, \neg x_3$.

Afirmación: G tiene un subgrafo completo de tamaño k si y sólo si φ es satisfacible.

En efecto: Primero demostramos "$\Rightarrow$". Nótese que si i y j están conectados por un arco en G, entonces se les puede dar el valor de verdad 1 en φ, sin conflicto. Si hay un subgrafo completo de tamaño k, entonces φ tiene al menos k literales, todos de cláusulas diferentes, y a cada uno se le puede asignar el valor de verdad 1 en φ, sin entrar en conflictos con los demás. Como φ tiene exactamente k cláusulas, esto da un literal en cada clásula, y así, se puede hacer verdadera la fórmula completa. Esta es una asignación de verdad que satisface la fórmula.

Ahora demostramos la implicación inversa. Supongamos que φ es satisfecha por una asignación σ. Fija σ, cada cláusula C_j debe tener un literal l^j tal, que $\sigma(l^j) = 1$. El conjunto de literales $\{l^1, \ldots, l^k\}$ corresponde a un subgrafo completo en G de tamaño k. Para probar esto, necesitamos sólo establecer que, para i y j arbitrarios, l^i y l^j están conectados por un arco en G. Nótese que:

- l^i y l^j provienen de cláusulas diferentes (por construcción), y
- l^i y l^j no entran en conflicto uno con otro (pues ambos σ les da el valor de verdad 1).

Esto demuestra la afirmación.

Finalmente, hay que mencionar que esta construcción puede ser hecha en tiempo polinomial en $\mid \varphi \mid$. Esto es fácil de ver, pues G tiene tantos nodos como literales tiene φ, digamos n. Determinar cuáles son los arcos a colocar en G toma tiempo n^2.

■

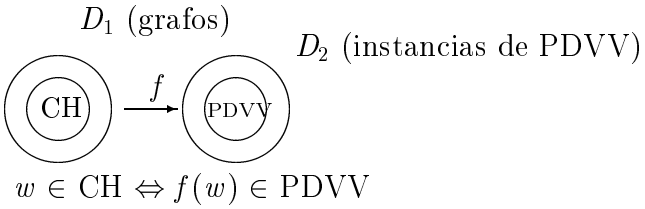
Ejemplo: Demostrar que PDVV es NP–completo usando el hecho que CH es NP–completo.

Hay que probar dos cosas:

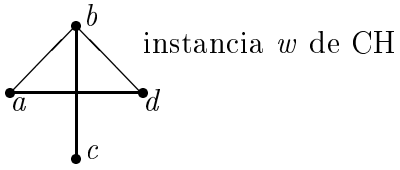
(a) PDVV $\in \mathcal{NP}$. Esto ya fue discutido antes.

(b) Todo problema de $\mathcal{NP}$ es reducible en tiempo polinomial a PDVV.

Por la discusión anterior al ejemplo, basta con establecer que CH $\leq_{pol}$ PDVV, es decir, que el problema de circuito hamiltoniano es reducible en tiempo polinomial al problema del vendedor viajero.



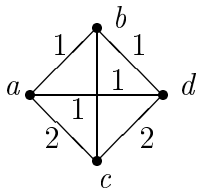
La reducción la indicaremos a través de un ejemplo fácilmente generalizable:



Dado este grafo, la pregunta es: ¿existe en él un circuito hamiltoneano?

Usemos w para denotar (una codificación de) esta instancia. En lugar de decidir directamente para w, reducimos el problema a una instancia $f(w)$ del problema del vendedor viajero.

Instancia $f(w)$ del PDVV:



En $f(w)$ todos los nodos están conectados entre sí y con pesos asignados a los arcos. Además, especificamos como parte de $f(w)$ un presupuesto $b = 4$.

Pregunta: ¿Existe un circuito cerrado y simple, que pasa por todos los nodos, de costo $\leq b$?

La respuesta que obtengamos para esta pregunta se puede traspasar a la pregunta sobre el problema de circuito hamiltoneano original.

Notar que se tiene:

- f es computable en tiempo polinomial.
- $w \in \text{CH} \Leftrightarrow f(w) \in \text{PDVV}$.

■

4.5 ¿Por qué SAT es NP–completo?

Surgen preguntas obvias:

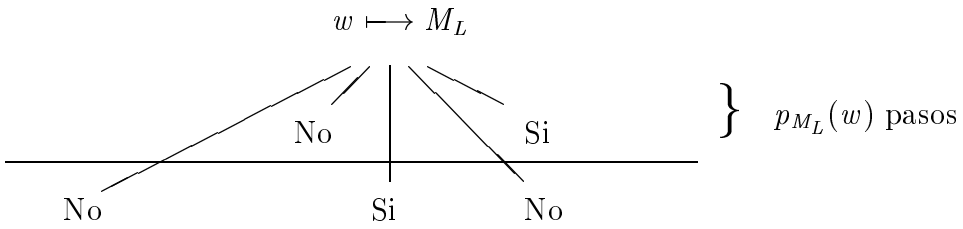
- ¿Por qué SAT es NP–completo?
- ¿Qué hace a este problema, y de paso, a la lógica proposicional, tan especial?
- ¿Cómo es posible que podamos reducir cualquier problema de decisión $(D, L) \in \mathcal{NP}$ a SAT?

- ¿Por qué fue SAT el primer problema cuya NP–completidad fue demostrada?

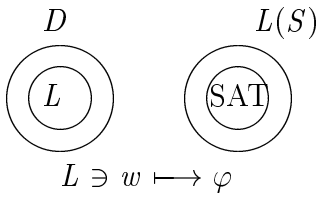
La clave está en que la lógica proposicional provee lenguajes que permiten describir ciertas situaciones y procesos. De hecho, estos lenguajes son lo suficientemente expresivos como para poder describir parte del funcionamiento de algoritmos no deterministas con sus entradas. Veamos esto con mayor detalle.

Sea (D, L) un problema de decisión cualquiera en la clase $\mathcal{NP}$. El es decidible por una MT no determinista M_L *.

Podemos suponer que la máquina tiene un tiempo de ejecución que está acotado superiormente por un polinomio fijo $p_{M_L}(x)$. Más precisamente, la máquina, con entrada $w \in D$, dejará de funcionar después de una cantidad de pasos $\leq p_{M_L}(\mid w \mid)$. Por la definición de algoritmo no determinista de tiempo polinomial para un problema de decisión, tenemos: $w \in L$ si y sólo si la maquina M_L con entrada w tiene alguna computación que termina en SI no después de $\leq p_{M_L}(\mid w \mid)$ pasos.



Todo lo que dijimos, es decir, el funcionamiento de esta máquina M_L con entrada w, acotada en tiempo por $p_{M_L}(\mid w \mid)$, y el criterio de aceptación de palabras w (cuándo pertenece w a L), pueden ser descritos por una fórmula proposicional. Esta descripción nos da precisamente la reducción de L a SAT.



*Las MT no deterministas proporcionan un modelo de algoritmo no determinista que es equivalente al dado anteriormente. Estas tienen relaciones de transición en lugar de funciones de transición, y, en consecuencia, cada configuración instantánea puede tener más de una configuración sucesora.

La fórmula proposicional φ describe el funcionamiento de M_L con entrada w y tiempo acotado por $p_{M_L}(\mid w \mid)$. La fórmula obviamente depende de M, $p(x)$ y w. Como M y $p(x)$ están fijos para todo el problema L, podemos pensar que φ depende sólo de w.

Se tiene: $\quad w \in L \Leftrightarrow \varphi \in SAT$

No vamos a dar la construcción exacta de φ. Sólo mencionemos que φ es computable a partir de w (y de M_L, $p_{M_L}(x)$). Más aún, puede ser construida en tiempo polinomial en $\mid w \mid$. En particular, la fórmula resulta ser corta, es decir de largo acotado por el valor de un polinomio fijo (no necesariamente $p(x)$) en $\mid w \mid$.

Vamos solamente a dar una idea de cómo la lógica proposicional puede ser usada para describir el funcionamiento de una máquina de Turing no determinista concreta M. El conjunto de símbolos de M es finito y fijo, lo mismo su relación de transición. Introducimos entonces variables proposicionales c_{ijk}, que, a pesar de su caracter puramente simbólico, podemos leer o entender como representando la afirmación "la cabeza escritora/lectora de M está leyendo en el instante i, en la celda j el k-ésimo símbolo de M". Entonces, la entrada inicial a la máquina M puede ser descrita a través de una conjunción de la forma $\bigwedge_{j=0}^{n} p_{0jk_j}$ (de largo $n+1$ si la entrada inicial es de largo n).

Podemos introducir otras variables proposicionales t_{iekl}, que podemos leer como "en el tiempo i, estando en el e–ésimo estado y leyendo el k–ésimo símbolo, la máquina aplica la l–ésima tupla de su relación". Con estas variables se puede describir el funcionamiento en tiempo acotado de una máquina de Turing. Se puede ir verificando que las fórmulas involucradas no exceden en largo una cantidad $q(n)$, donde q es un polinomio fijo, dependiente de M, $\mid w \mid$ y $p(x)$.*

Se puede verificar que para hacer las descripciones anteriores, bastan fórmulas de cierto tipo sintáctico. Más precisamente, φ puede estar siempre en forma FNC(3), es decir, en forma normal conjuntiva con a lo más tres literales por disyunción. Por ejemplo, la fórmula $(p \vee \neg r) \wedge (q \vee \neg p \vee s)$ está en FNC(3).

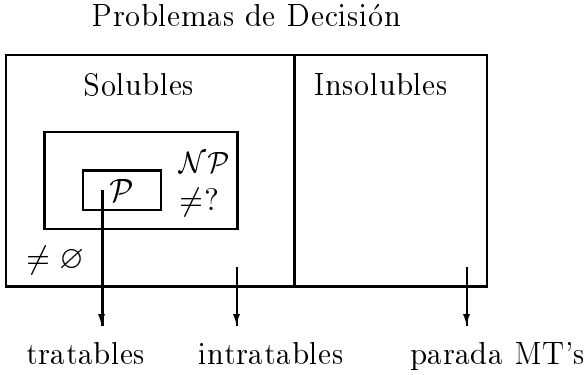
*Notar que hay afirmaciones importantes que no podemos describir, como, "eventualmente la máquina para", para ello necesitaríamos algo como *existe i...* t_{iekl_0}, donde l_0 es el índice correspondiente a una tupla de parada de M, que no es una fórmula de la lógica proposicional. Una afirmación de este tipo sí puede ser hecha en lógica de predicados, y esto marca una diferencia esencial entre las dos lógicas. De hecho, como veremos más adelante, esta diferencia determina la indecidibilidad del problema SAT para la lógica de predicados, en contraste con la decidibilidad de este problema para la lógica proposicional.

De la observación anterior obtenemos que SAT(3), el problema de satisfacibilidad de fórmulas proposicionales en FNC(3), es NP–completo. Es decir, SAT(3) es decidible, está en $\mathcal{NP}$ y podemos reducir el problema de satisfacibilidad de fórmulas proposicionales en FNC al problema de satisfacibilidad de fórmulas en FNC(3).

4.6 Algunas Observaciones Finales

En la clase $\mathcal{P}$ coleccionamos todos los problemas de decisión que consideramos tratables. Si la clase $\mathcal{NP}$ extiende propiamente a la clase $\mathcal{P}$, entonces en la primera habría problemas intratables, y todos los problemas NP–completos caerían en esta categoría. Es posible, si $\mathcal{P} \subsetneqq \mathcal{NP}$, que haya en la diferencia de ambas clases, problemas que no son NP–completos, pero todavía intratables. Un candidato es el problema NUMCOMP, de determinar si un número entero es compuesto o no.

Por otro lado, no todos los problemas de decisión solubles están en $\mathcal{P}$ o $\mathcal{NP}$. Hay algunos que están demostradamente fuera de $\mathcal{NP}$ (veremos uno más adelante). En consecuencia, el panorama que hemos presentado hasta ahora es el siguiente*:



4.7 Ejercicios

1. Demuestre y discuta la siguiente caracterización de los problemas en la clase $\mathcal{NP}$ †:

*Este ha sido refinado considerablemente, sin embargo, el presentar esos resultados, así como las relaciones entre complejidades temporal y espacial, están fuera del alcance de este texto. La llamada "teoría de complejidad computacional" contiene los resultados ya presentados y muchos más.

†Compare con la caracterización similar de los problemas recursivamente enumerables.

$L(\subseteq D) \in \mathcal{NP} \Leftrightarrow$ existe un polinomio $p(x)$ y una relación R $(\subseteq D \times D)$ $\in \mathcal{P}$ tal, que: para todo $x \in D$: $x \in L \Leftrightarrow \exists\, y \in D$ tal que $\mid y \mid \leq p(\mid x \mid)$ y $(x, y) \in R$.

2. Demuestre que el POVV es tan difícil como el PDVV (excepto por tiempo polinomial). Más precisamente, demuestre:

(a) Si el PDVV puede ser resuelto en tiempo polinomial, entonces el problema de "encontrar la longitud del tour más corto" también puede ser resuelto en tiempo polinomial.

Ind.: use búsqueda binaria en el intervalo [0, n max d(i,j)]. ¿Por qué?

(b) Si el PDVV puede ser resuelto en tiempo polinomial, entonces el problema de "encontrar el tour más corto" puede ser resuelto en tiempo polinomial.

Ind.: por (a), se puede encontrar primero la longitud del tour más corto.
3. Demostrar que los siguientes problemas están en la clase $\mathcal{NP}$:

(a) Determinar si existe una solución binaria a un sistema de ecuaciones lineales con coeficientes enteros no negativos (un caso de "programación entera").

(b) Determinar si un grafo tiene un circuito simple (sin repeticiones de nodos) que pase por todos los vértices del grafo (un circuito Hamiltoniano).
4. (a) Demuestre que el problema de determinar si una oración proposicional de la forma: $(q_1 \vee r_1) \wedge \ldots \wedge (q_n \vee r_n)$, donde q_i, r_i son variables proposicionales o negaciones de variables proposicionales, está en la clase $\mathcal{P}$, es decir, existe un algoritmo de decisión, determinista, con tiempo de ejecución acotado superiormente por un polinomio en n.

Ind.: Pensar las cláusulas $q_i \vee r_i$ como implicaciones $\neg q_i \rightarrow r_i$. Verificar que la satisfacibilidad de la fórmula tiene que ver con la existencia de ciclos de la forma: $x_i \rightarrow \ldots \rightarrow \neg x_i \rightarrow \ldots x_i$.

(b) Demuestre que 2–CG ("coloración de grafos con dos colores") es polinomialmente reducible a SAT(2). ¿Es SAT(2) reducible a 2–CG?

(c) Demuestre que 2–CG $\in \mathcal{P}$ usando (a) y (b). Hágalo también directamente usando el hecho que un grafo es 2–colorable si y sólo si no contiene un ciclo de longitud impar (¿por qué?).

5. (a) El problema de decisión de "programación entera", PE, es el siguiente:

Instancia: un sistema de desigualdades lineales $Ax \geq b$ con a_{ij}, $b_j \in \mathbb{Z}$

Pregunta: ¿existe una solución binaria x?

Demuestre que PE es NP–completo.

Ind.: SAT es polinomialmente reducible a PE de la siguiente manera: si φ es la oración proposicional $C_1 \wedge \ldots \wedge C_k$ en FNC y $p_1, \ldots, p_n$ son las variables proposicionales en φ, entonces considere una instancia de PE dada por la matriz $k \times n$ (a_{ij}) y el vector $k \times 1$ (b_i) con $a_{ij} = 1, -1, 0$ dependiendo si p_j aparece en C_i, $\neg p_j$ aparece en C_i, otro caso, resp; y $b_i = 1$ - # de p_j's tales, que $\neg p_j$ aparece en C_i.
6. Demuestre que el problema de optimización de recubrimiento de vértices (¿cuál es?) es reducible en tiempo polinomial al problema de decisión de recubrimiento de vértices: O-RV $\leq_{pol}$ D-RV. Precise y demuestre esta afirmación.

Solución: El problema de optimización de recubrimiento de vértices (encontrar un recubrimiento de tamaño mínimo, es decir, un conjunto de nodos de tamaño mínimo tal, que cada arista tiene al menos un extremo en el subconjunto) se puede resolver en tiempo polinomial si el problema de decisión de recubrimiento de vértices (¿existe recubrimiento de tamaño a lo más k?) se puede resolver en tiempo polinomial.

En efecto, supongamos que $AL_D(G, k)$ es algoritmo para resolver el problema de decisión (para cualquier grafo G y $k \leq n$, donde n es el número de nodos de G). Sea $T(n)$ su tiempo de ejecución. Usaremos este algoritmo como una subrutina que será llamada cierta cantidad de veces, ojalá no una cantidad exponencial de veces en términos de n (en ese caso, no nos serviría).

Primero resolvamos el problema de encontrar el tamaño del recubrimiento óptimo. Esto se puede hacer llamando a $AL_D(G, k)$, con $k = n, n-1, ..., 1$. Esto para cuando el algoritmo da respuesta NO. El paso anterior nos da la respuesta. Esto toma a lo más n pasos. Sea N este número óptimo. La complejidad temporal de este problema es, entonces, $O(n \times T(n))$, que es polinomial (si $T(n)$ lo es).

Ahora queremos encontrar un recubrimiento mínimo. Esto se hace encontrando sus elementos uno a uno. Para cada vértice v, se incluye v en el recubrimiento mínimo si el grafo reducido $G - \{v\}$ tiene un recubrimiento de tamaño a lo más $N - 1$. Si se encuentra tal v, se procede a encontrar recursivamente el recubrimiento de tamaño $N - 1$ para el grafo $G - \{v\}$.

El tiempo de ejecución de este algoritmo es

$$T_1(n) \leq (n \times T(n-1)) + T_1(n-1) \leq (n \times T(n)) + T_1(n-1).$$

Esta recurrencia da tiempo $O(n^2 \times T(n))$. Sumando la complejidad de la primera parte, queda igual. Si $T(n)$ es polinomial, entonces el tiempo total también es polinomial.
7. Demuestre que el problema de Subgrafo Completo (Clique) es NP–completo. Si hace alguna reducción, ilústrela con un ejemplo.

Clique: Dados un grafo (no dirigido) $G = \langle V, E \rangle$ y un entero k, determinar si G tiene un subgrafo completo (o un clique, i.e. un subgrafo donde todos los vértices están conectados entre sí) de tamaño k.
8. ¿Es el siguiente problema problema NP–completo? ¿Por qué?

Diferencia Lógica: Dadas dos fórmulas proposicionales φ_1 y φ_2 en FNC con las mismas variables, determinar si no son lógicamente equivalentes ($\varphi_1 \not\equiv \varphi_2$) (i.e. hay una asignación que les da distinto valor de verdad).
9. (a) ¿Qué tan difícil es el problema de satisfacibilidad, FND, de fórmulas proposicionales en forma normal disyuntiva? (ver sección 2.2.2)

(b) ¿Qué tan complejo puede ser el proceso, y qué tan larga puede ser la fórmula resultante con respecto a la fórmula original, al convertir una fórmula proposicional arbitraria a forma normal disyuntiva?

(c) Responda las mismas preguntas para forma normal conjuntiva.

Capítulo 5

Lógica de Predicados de Primer Orden

En la lógica de predicados de primer orden (LPOP) aparecen lenguajes formalizados más expresivos que los de la lógica proposicional. Primero, veamos, a través de algunos ejemplos, por qué podemos necesitar ese mayor poder expresivo.

Ejemplo 1: Necesitamos una lógica que sancione como válido al siguiente argumento:

"Todos los hombres son mortales.

Sócrates es hombre. En consecuencia,

Sócrates es mortal."

Primero, este argumento no puede ser expresado naturalmente como fórmula (válida) de un lenguaje de la lógica proposicional (LP), ya que:

- En la primera proposición hay una cuantificación universal sobre individuos (los hombres). No tenemos forma de cuantificar el LP, excepto por la posibilidad poco atractiva de introducir un hecho atómico (letra proposicional) para cada posible individuo. Nótese que puede haber infinitos hechos de este tipo, no combinables a la vez en una sola conjunción.
- La validez del argumento anterior también depende de la posibilidad de referirnos, en el lenguaje natural, a los predicados (atributos) "ser mortal", "ser hombre". Esto no se puede hacer en LP, ya que podemos referirnos sólo a proposiciones (afirmaciones) completas que involucran

a esos predicados, y no directamente a todos sus constituyentes, en particular, a los predicados mismos.

Sin embargo, en los lenguajes de la lógica de primer orden para predicados podremos:

- Cuantificar sobre individuos de un dominio de discurso.
- Referirnos a predicados, en particular, a operaciones, es decir, a predicados (relaciones) funcionales.

Informalmente, el argumento anterior podría ser expresado en un lenguaje de esta lógica de la siguiente manera:

$$(\forall\, x\ (Hombre(x) \rightarrow Mortal(x))\ \wedge\ Hombre(Socrates)) \quad \rightarrow \quad Mortal(Socrates)$$

■

Un lenguaje de la lógica de primer orden para predicados queda esencialmente determinado por un conjunto de símbolos inicial, S, elegido por nosotros dependiendo de lo que queramos describir, tal como en LP cuando elegimos variables proposicionales iniciales.

El conjunto de símbolos S será siempre de la forma: $S = \{P, ..., f ....., c, ...\}$, donde:

$P, \ldots$ son símbolos para denotar predicados.

$f, \ldots$ son símbolos para denotar funciones (operaciones).

$c, \ldots$ son nombres o constantes para denotar individuos.

Para construir el lenguaje completo asociado a S, necesitamos, aparte del alfabeto inicial S, los símbolos lógicos y de separación que están en todo lenguaje de la LPOP:

1. $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ (conectivos lógicos)
2. $=$ (símbolo para un predicado binario especial, la igualdad)

3. $\exists$, $\forall$ (cuatificadores existencial y universal, respectivamente)

4. $x_1, x_2, x_3, ...$ (una lista infinita de variables)

5.), ((símbolos de puntuación)

Los símbolos en 1., 2., y 3. se llaman símbolos lógicos, en el sentido que ellos tendrán, después, en el contexto semántico, una interpretación fija, dada por la "logica". Esto no ocurre con los símbolos en S, que tienen interpretación variable según el contexto o dominio.

Las variables están para referirse a individuos del dominio de discurso donde hagamos una interpretación del lenguaje. Se cuantificará usando las variables y, en consecuencia, en los lenguajes de la LPOP se puede hacer sólo cuantificación sobre individuos, por eso sus lenguajes se llaman "de primer orden". Si cuantificaramos sobre, digamos, propiedades de individuos (lo que no está permitido en LPOP), pasaríamos a un orden mayor.

La sintaxis general de la LPOP que veremos en la próxima sección, nos dirá cómo podemos construir lenguajes formales correspondientes a un conjunto de símbolos S dado. Antes veamos otros ejemplos de manera informal.

Ejemplo 2: Consideremos la estructura de los números reales $\mathfrak{R} = \langle \mathbb{R}, <, +, \cdot, 0, 1 \rangle$. Aquí tenemos, aparte del simple conjunto $\mathbb{R}$ de números reales, una estructura sobre este conjunto. Tenemos, además del dominio, una relación de orden, dos operaciones, y dos elementos distinguidos.

Si queremos describir esta estructura, por ejemplo, si queremos decir que "cada número mayor que 0 tiene una raíz cuadrada", nos encontramos en dificultades al pretender usar algún lenguaje de la lógica proposicional.

Podríamos pensar en algunas posibilidades ingenuas, como la de usar una sola letra proposicional para denotar a toda esta proposición. Sin embargo, esta solución no parece muy razonable, pues es demasiado burda e indescomponible en los elementos importantes que aparecen involucrados en la afimación original que pretende denotar. Su falta de sutileza le impide ser combinada con otras afirmaciones que hablan también sobre la relación de orden, el cero, la raíz cuadrada, etc., con el propósito de concluir nuevo conocimiento.

Otra alternativa ingenua sería la de introducir una letra proposicional p_r por cada número real r que denota la proposición "si r es mayor que cero, entonces tiene raíz cuadrada", pero esto requeriría de una conjunción innumerable $\bigwedge_r p_r$, sobre todos los reales $r \in \mathbb{R}$, que no está permitida en lenguajes de la LP.

Sin embargo, con lógica de predicados, podremos escribir la aseveración original en la forma $\forall\, x(0 < x \rightarrow \exists\, y\ x = y \cdot y)$.

■

Ejemplo 3 (Bases de Datos): Consideremos la siguiente base de datos relacional:

Tiene Aeropuerto	Vuelo Directo	Conexion de Vuelo
SCL	SCL - MIAMI	SCL - MIAMI
MIAMI	SCL - RIO	SCL - HONG KONG
RIO	....	...
HONG KONG	SCL - FRANKFURT	
....	RIO - SCL	
	MIAMI - HONG KONG	
	...	

Una base de datos (BD) es un **modelo** de la realidad, es decir, una abstracción o representación simplificada de lo que consideramos o percibimos como una parte de la realidad.

En el ejemplo, tenemos más que un simple conjunto de individuos (`SCL`, `MIAMI`, `HONG KONG`, etc), tenemos, además, una **estructura** en ese conjunto que está indicada a través de distintos atributos (predicados, relaciones) aplicables a los individuos. Las tablas de la BD indican qué individuos tienen los atributos o están en relación. Por ejemplo, `SCL` tiene el atributo de "tener aeropuerto", el par (`SCL`,`MIAMI`) tiene el atributo (con dos argumentos) de "estar conectados por vuelo directo", etc. En este sentido, una base de datos relacional no es diferente en espíritu de una estructura algebraica como $\mathfrak{R}$, la de los números reales.

Para describir este modelo (la BD), podemos usar un lenguaje $\mathcal{L}$ de la lógica de predicados de primer orden. Si elegimos los predicados $TA(\cdot)$, $VD(\cdot,\cdot)$, $CV(\cdot,\cdot)$, para denotar a las tablas "`Tiene Aeropuerto`", "`Vuelo Directo`", "`Conexion de Vuelo`", respectivamente; y denotamos, en $\mathcal{L}$, los individuos (como `SCL`) con los mismos nombres, podemos hacer, entre otras cosas, las siguientes afirmaciones:

1. $TA(MIAMI)$, $VD(SCL, MIAMI)$, ...
2. $\forall\, x, y(VD(x, y) \rightarrow VD(y, x))$
3. $\forall\, x \neg VD(x, x)$
4. $\forall\, x \forall\, y(VD(x, y) \rightarrow CV(x, y))$
5. $\forall\, x \forall\, y(\exists\, z(CV(x, z) \wedge VD(z, y)) \rightarrow CV(x, y))$

Una observación que salta a la vista es la siguiente: en lugar de almacenar explicítamente en la BD toda la información atómica (esencialmente lo que habríamos descrito en 1.), podemos almacenar implícitamente mucha información si incorporamos al contenido de la BD los hechos generales o reglas 2.–5. . En particular, bastaría saber que hay un vuelo directo de `SCL` a `MIAMI` para obtener, usando la regla 2., que también hay un vuelo directo de `MIAMI` a `SCL`. Del mismo modo, podríamos deducir conexiones de vuelo entre ciudades usando información atómica y las reglas 4. y 5.

La simple observación anterior crea los fundamentos para el desarrollo de las llamadas "bases de datos deductivas", que almacenan hechos atómicos y reglas para deducir nueva información explícita. Una parte importante de las bases de datos deductivas tiene que ver con la forma de manejar, deductiva y computacionalmente, la información representada explícitamente en la BD, es decir, los hechos atómicos originales y las reglas. En resumen, la pregunta es la siguiente: ¿cúal es la información implícita en la BD deductiva y cómo la obtenemos? Volveremos más adelante sobre este punto.

■

5.1 Sintáxis de Lenguajes de la LPOP

La lógica de predicados de primer orden permite construir distintos lenguajes formales, dependiendo del conjunto de símbolos básico con que se parta. Todos ellos, sin embargo, tienen una manera uniforme de ser construidos. Es este proceso de construcción el que queremos definir a continuación.

Partimos con un conjunto de símbolos fijo $S = \{P, ..., f, ..., c, ...\}$. Como mencionamos en la sección anterior, en S hay listas de símbolos para predicados, operaciones e individuos, respectivamente. Estas listas pueden ser finitas (eventualmente vacías) o infinitas.

La única exigencia adicional sobre estos símbolos es que cada símbolo para predicado y operación tiene una aridad prescrita asociada. Por ejemplo, el símbolo P podria tener aridad fija 2, es decir, P denotaría un predicado binario (relacion binaria, atributo aplicable a pares de individuos). El símbolo f podría tener aridad 3, es decir, denotaría una operación (función) de 3 argumentos. A veces, indicaremos las aridades escribiendo explícitamente el número de argumentos de la siguiente manera: $P(\cdot,\cdot)$, $f(\cdot,\cdot,\cdot)$.

Ejemplo 1: Consideremos el conjunto de símbolos $S_1 = \{Hombre(\cdot), Mortal(\cdot), Socrates\}$. Aquí hay dos (símbolos para) predicados unarios y un nombre para individuo. Hay que destacar que en S hay sólo símbolos, es decir, no hay a priori ninguna interpretación o semántica definida para ellos (aunque puede haber una subentendida obvia). Desde este punto de vista, el nombre 'Socrates' puede ser posteriormente asignado a (interpretado como) cualquier individuo en un dominio de interpretación del lenguaje. No tiene por qué ser asignado al filósofo (individuo) Sócrates. En este sentido, desde el punto de vista descriptivo, el conjunto de símbolos $S_2 = \{H(\cdot), M(\cdot), Leo\}$ cumpliría la misma función, tanto para describir el universo de habitantes de la antigua Grecia, como para referirnos a cualquier otro dominio.

El argumento del ejemplo 1. de la sección anterior podría igualmente ser expresado sobre la base de este último conjunto de símbolos de la siguiente manera:

$$(\forall\, x(H(x) \to M(x)) \wedge H(Leo)) \to M(Leo).$$

■

Ejemplo 2: Para hablar (formalmente) sobre la aritmética de números naturales, en otras palabras, para describir la estructura $\mathfrak{N} = \langle \mathbb{N}, <, +, \cdot, 0, 1\rangle$, podríamos elegir el siguiente conjunto de símbolos aritméticos: $S_{ar} = \{<, +, \cdot, 0, 1\}$, en el cual: $<$ es predicado binario, $+$ y $\cdot$ son operaciones binarias, 0 y 1 son nombres para individuos distinguidos*.

El conjunto S_{ar} contiene sólo símbolos, los que usamos usualmente en el álgebra. Posteriormente, estos símbolos podrían ser interpretados en un dominio concreto, al pasar a la semántica, como una relación binaria, operaciones

*En realidad, deberíamos distinguir entre los símbolos del metalenguaje matemático y los del lenguaje formal, por ejemplo, usando, en el segundo caso, algo como $+'$ en lugar del símbolo matemático usual $+$. Sin embargo, no haremos distinción, esperando que el lector sepa distinguir ambas situaciones.

binarias y elementos distinguidos, respectivamente. No hay nada que, a priori, nos obligue a interpretar al símbolo $<$ como una relación de orden.

Más aún, podríamos agregar otros símbolos a este alfabeto aritmético, como un predicado unario P para referirnos, por ejemplo, a números primos o a números pares (ambos son atributos aplicables a un individuo a la vez) si es que hacemos una interpretación de los símbolos en el dominio de números enteros.

■

Retomemos la construcción de los lenguajes de la LPOP. Para describir la construcción, fijemos el conjunto de símbolos S, que escribimos genéricamente así: $S = \{P, \ldots, f, \ldots, c, \ldots\}$.

Las proposiciones del lenguaje formal hablan sobre objetos, también formales, que corresponden, en el contexto semántico, a individuos de los dominios de interpretación. Estos objetos formales son los **términos** del lenguaje. Para todos los efectos prácticos, se puede pensar en los términos del álgebra, por ejemplo, en $x \cdot (y + 1)$. Ellos son objetos que manipulamos formalmente y que son potencialmente interpretables como números (enteros, reales, complejos, ...). De hecho, en la posibilidad de manejarlos formalmente, sin acarrear permanentemente su significado o interpretación (o valor), radica el poder del álgebra.

Ahora, en el contexto de un lenguaje formal cualquiera, pero tal como en el álgebra clásica, podemos hablar sobre individuos usando términos construidos con variables, nombres para individuos, y operaciones aplicadas a individuos. La definición general de término de un lenguaje de la LPOP se hace por inducción estructural.

Definición: (S fijo) Son **términos** del lenguaje de la LPOP basado en S, todas las sucesiones de símbolos construidas aplicando una cantidad finita de veces las siguientes reglas:

1. Toda variable es un término.
2. Todo nombre para individuo $c(\in S)$ es término.
3. Si f es operación n–aria ($f \in S$) y $t_1, ..., t_n$ son términos, entonces $f(t_1, \ldots, t_n)$ es término.

Ejemplo 3:

(a) Dado $S_1 = \{Hombre(\cdot), Mortal(\cdot), Socrates\}$, son términos, entre otros: $Socrates, x_3, x_{20}$.

(b) Dado $S_2 = \{H(\cdot), M(\cdot), Leo\}$, son términos, entre otros: Leo, x_2, x_{100}. Al igual que en (a), aquí los predicados no aportan nada a la construcción de términos.

(c) Dado $S_{ar} = \{<, +, \cdot, 0, 1\}$, son términos, entre otros : $x_2, 0, x_9, 1, \cdot(x_1, +(x_2, 1)$. El último término lo escribimos, por simplicidad, en la forma $x_1 \cdot (x_2 + 1)$.

■

Denotemos con $T(S)$ al conjunto de términos del lenguaje basado en el conjunto de símbolos S. Los términos no afirman nada, no son proposiciones potenciales, tan sólo denotan individuos de posibles dominios de interpretación. Para hacer afirmaciones necesitamos las fórmulas del lenguaje.

Por ejemplo, las siguientes van ser fórmulas correspondientes a los conjuntos de símbolos presentados antes: $\forall\, x_3(H(x_3) \to M(x_3))$, $\forall\, x(x < y \to x + z < y + z)$, $\forall\, y(y < 0 \wedge y \cdot y < 0)$.*

Las fórmulas también serán definidas por inducción estructural. Para ello, fijemos el conjunto de símbolos S.

Definición: Las fórmulas del lenguaje de la LPOP basado en S se obtienen mediante una cantidad finita de aplicaciones de las siguientes reglas:

1. Si t_1 y t_2 son términos, entonces $t_1 = t_2$ es una fórmula.
2. Si P es símbolo de predicado n–ario y $t_1, \cdots, t_n$ son términos, entonces $P(t_1, t_2, ..., t_n)$ es una fórmula.
3. Si φ es fórmula, entonces $\neg\varphi$ también es fórmula.
4. Si φ y ψ son fórmulas, entonces $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \to \psi)$, y $(\varphi \leftrightarrow \psi)$ también son fórmulas.
5. Si φ es fórmula y x es una variable, entonces $\exists\, x\varphi$ y $\forall\, x\varphi$ son fórmulas.

*Por simplicidad usamos $x, y, \ldots$ como variables en lugar de las variables de la lista oficial $x_1, x_2, \ldots$.

Las fórmulas correspondientes a los casos 1. y 2. se llaman **fórmulas atómicas**, y son las fórmulas más simples que podemos formar.

Ejemplo 4: Si $S = \{H, M, Socrates\}$, entonces son fórmulas, entre otras:

(a) $(\forall\, x(H(x) \to M(x)) \wedge H(Socrates)) \to M(Socrates))$

(b) $x = Socrates$

(c) $\neg(Socrates = Socrates)$

(d) $\exists\, y(H(y) \wedge \neg M(y))$

Ejemplo 5: Si $S_{ar} = \{<, +, \cdot, 0, 1\}$, con las fórmulas del lenguaje formal podemos expresar propiedades de los números naturales:

(a) $\forall\, x \forall\, y \forall\, z(x < y \wedge 0 < z \to x \cdot z < y \cdot z)$

(b) $\forall\, x\ x + 0 = x$

(c) $\forall\, x((0 < x \vee 0 = x) \wedge \exists\, y\ x < y)$

■

Denotamos con $L(S)$ al lenguaje de la LPOP basado en el conjunto de símbolos S. Podemos identificar $L(S)$ con el conjunto de términos y fórmulas construidas de acuerdo con las dos definiciones precedentes.

Ejemplo 6: Surge una pregunta natural a propósito de la definición de fórmula de un lenguaje de primer orden y del ejemplo de los números naturales. Esta tiene que ver con el conocido principio de inducción que nos permite definir conceptos asociados a números naturales y demostrar propiedades generales de los números naturales. ¿Cómo expresamos este principio de inducción en el lenguaje de primer orden $L(\{<, +, \cdot, 0, 1\})$? Una manera de escribirlo sería a través de la fórmula:

$$\forall\, X((X(0) \wedge \forall\, x(X(x) \to X(x + 1)) \to \forall\, xX(x)).$$

En esta fórmula, X sería una variable para propiedades (predicados) unarias, para cuantificar sobre subconjuntos del dominio. Esta no es una fórmula admisible del lenguaje de primer orden, de hecho, es una fórmula de "segundo orden", en el sentido que usa variables para cuantificar sobre objetos de segundo orden, como lo son los subconjuntos del dominio, en contraste, con

objetos de primer orden, que son los individuos (elementos) del dominio. Veremos más adelante una forma de expresar el principio de inducción a través de fórmulas de primer orden. Sin embargo, también veremos que esa forma alternativa no es tan expresiva como la fórmula de segundo orden que ya dimos. Más aún, veremos que el lenguaje de primer orden es intrínsecamente menos expresivo que el lenguaje de segundo orden, que permite cuantificaciones sobre subconjuntos, cuando se trata de formalizar el principio de inducción.

■

Debido a la definición inductiva de términos y fórmulas, es posible definir conceptos asociados a estos objetos y demostrar propiedades generales de ellos usando inducción estructural.

Ejemplo 7: Queremos definir el conjunto $VL(\varphi)$ de **variables libres** de una fórmula φ, es decir, de las variables que aparecen en la fórmula fuera del alcance de un cuantificador. Por ejemplo, en el caso de la fórmula $\varphi: \ \forall x \forall w(\exists y R(x,y) \rightarrow \exists z R(u,v)) \wedge R(x,z)$, la definición debería entregarnos el conjunto $VL(\varphi) = \{x, z, u, v\}$.

La definición inductiva es la siguiente:

1. Si φ es atómica, entonces $VL(\varphi) :=$ el conjunto de todas las variables que aparecen en φ.
2. Si φ es $\neg\psi$, entonces $VL(\varphi) := VL(\psi)$.
3. Si φ es $(\psi * \chi)$, donde $*$ es cualquiera de los conectivos binarios, entonces $VL(\varphi) := VL(\psi) \cup VL(\chi)$.
4. Si φ es $\forall x\psi$ o $\exists x\psi$, entonces $VL(\varphi) := VL(\psi) - \{x\}$.

De hecho, sobre la base de esta definición inductiva, cuyas declaraciones 1. – 4. constituyen una especificación del conjunto VL, podemos diseñar de forma inmediata algoritmos para encontrar las variables libres de una fórmula arbitraria, es decir, producir implementaciones computacionales correspondientes a la especificación.

■

Una fórmula φ sin variables libres, es decir, tal, que $VL(\varphi) = \varnothing$, se llama **oración**. Intuitivamente, una oración corresponde a una proposición cerrada,

con sentido completo. No tiene grados de libertad debido a las variables no cuantificadas.

Dejaremos momentaneamente hasta aquí la sintaxis de los lenguajes de primer orden. Más adelante veremos un sistema formal deductivo para lenguajes de la LPOP. Ahora trataremos la semántica de estos lenguajes.

5.2 Semántica de Lenguajes de Primer Orden

Tal como lo hicimos con la lógica proposicional, queremos dar significado a los símbolos de un lenguaje, definir cuándo una fórmula es verdadera, y cuándo una fórmula es consecuencia lógica de un conjunto de fórmulas.

Veamos algunos ejemplos:

(a) ¿Cuál es el significado o interpretación de la fórmula

$$\forall x(\exists y(R(x,y) \rightarrow \exists z R(u,v)) \wedge R(x,z)?$$

¿Cuál es su valor de verdad?

(b) ¿Es verdadera la fórmula $\exists y(VD(SCL,y) \wedge VD(y,HK))$ en la base de datos del ejemplo 3 del comienzo del capítulo? Aquí, en contraste con el ejemplo anterior, tenemos un contexto concreto en el cual nos gustaría evaluar la fórmula (o, en la terminología de bases de datos, la "consulta" sobre vuelos de Santiago de Chile a Hong Kong con un cambio de avión).

(c) ¿Es verdadera la fórmula $\forall x(0 < x \rightarrow \exists y\ y \cdot y = x)$ en la estructura aritmética de los números naturales?

Para responder a estas preguntas es necesario hacer algunas precisiones. La situación es más compleja que en lógica proposicional. Hay que dar, primero, significado a los símbolos no lógicos (por ejemplo, a los símbolos del conjunto $S = \{R(\cdot,\cdot)\}$ en (a)); y también a los símbolos lógicos: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists$. Estos últimos tendrán un significado fijo y esperado, dado por "la lógica". También hay que dar significado a las variables; más generalmente, a los términos del lenguaje.

5.2.1 Estructuras

Comencemos por dar significado a los símbolos no lógicos del lenguaje. Sea $S = \{R, \ldots, f, \ldots, c, \ldots\}$ el conjunto de símbolos básicos. Estos símbolos se interpretan en estructuras.

Por ejemplo, si $S_{ar} = \{<, +, \cdot, 0, 1\}$ es el conjunto de símbolos para hablar sobre los números naturales, una interpretación natural para estos símbolos es provista por la estructura de los números naturales:

$$\mathfrak{N} = \langle \mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}} \rangle.$$

Los símbolos $<, +, \cdot, 0, 1$ del lenguaje objeto se interpretan como los objetos matemáticos que son denotados en el metalenguaje (matemático usual) a través de $<^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}$, respectivamente.

Notemos aquí que:

1. $<^{\mathbb{N}}$ es relación binaria sobre $\mathbb{N}$, es decir, $<^{\mathbb{N}} \subseteq \mathbb{N} \times \mathbb{N}$
2. $+^{\mathbb{N}}$ y $\cdot^{\mathbb{N}}$ son operaciones binarias sobre $\mathbb{N}$, es decir, $+^{\mathbb{N}}, \cdot^{\mathbb{N}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
3. $0^{\mathbb{N}}, 1^{\mathbb{N}} \in \mathbb{N}$, es decir, son elementos de $\mathbb{N}$

Definición: Dado $S = \{R, .., f, ..., c, ...\}$, una **estructura** compatible con (o para) S es una tupla $\mathfrak{A} = \langle A, R^A, ..., f^A, ..., c^A, ... \rangle$, en la cual:

- A es un conjunto no vacío llamado el **universo** o **dominio** de la estructura.
- Si R es símbolo para predicado n–ario, entonces R^A es una relación n–aria sobre A, es decir, $R^A \subseteq A^n$, donde A^n es el producto cartesiano de A con sí mismo, n veces.
- Si f es símbolo para operación n–aria, entonces f^A es una operación (función) n–aria sobre A, es decir, $f^A : A^n \to A$.
- Si c es un símbolo para constante (una constante), entonces c^A es un elemento de A, un individuo particular del dominio: $c^A \in A$.

Decimos que $R^A, ..., f^A, ..., c^A, ...$ son las interpretaciones de $R, ..., f, ..., c, ...$ en $\mathfrak{A}$.

Ejemplo: Sea $S = \{H(\cdot), M(\cdot), Socrates\}$. Una estructura compatible con este lenguaje es:

$\langle$ *el conjunto de habitantes de Grecia AC, el subconjunto de hombres,*

el subconjunto de habitantes mortales, el filósofo Sócrates $\rangle$.

En esta estructura, las fórmulas $\forall x(H(x) \to M(x))$ y $H(Socrates)$ deberían ser verdaderas (cuando definamos el concepto de "verdad").

Otra estructura compatible con el lenguaje es:

$\langle$*el conjunto de personas en la sala, el subconjunto de personas mayores de 22 años,*

subconjunto de personas con nota no inferior a 5 en la interrogación 1, el profesor $\rangle$.

En esta estructura, la primera fórmula debería ser falsa, y la segunda, verdadera.

■

Ejemplo: Consideremos $S_{ar} = \{<, +, \cdot, 0, 1\}$. Una estructura compatible con S_{ar} es:

$$\mathfrak{R}' = \langle \mathbb{R}, =^{\mathbb{R}}, (\cdot) - (\cdot), (\cdot)^{(\cdot)}, \pi, e \rangle,$$

donde:

- $=^{\mathbb{R}}$ es la igualdad en $\mathbb{R}$. Esta sería la interpretación para $<$ en $\mathfrak{R}'$.
- $(\cdot) - (\cdot)$ es la resta en $\mathbb{R}$ (para interpretar el símbolo $+$).
- $(\cdot)^{(\cdot)}$ es la exponenciación (para interpretar el símbolo $\cdot$).
- π y e son elementos de $\mathbb{R}$ (las interpretaciones para los símbolos 0 y 1).

Esta interpretación para S_{ar} no es tal vez la natural o subentendida, sin embargo, es totalmente aceptable de acuerdo con nuestra definición.

■

Este último ejemplo nos muestra que un mismo lenguaje tiene muchas interpretaciones posibles. De todos modos, la lógica tiene que dar cuenta de esta diversidad de interpretaciones posibles. Más aún, muchos de los problemas que trata la lógica simbólica, si es que no todos, provienen de este hecho. El concepto de verdad lógica, tiene que ver con que una aseveración formalizada sea verdadera en cualquier contexto, en cualquier interpretación imaginable.

La lógica formal nos proporcionará una definición adecuada y precisa de esta noción.

Ejemplo: Una base de datos extensional puede ser vista como una estructura. Toda estructura es un modelo de (parte de) una realidad, una abstracción o simplificación. Esto es particularmente claro en el caso de bases de datos. Una ventaja importante, en comparación con el ejemplo anterior de los números reales, es que hay una representación computacional de la estructura. La estructura está representada a través de sus tablas, cuyas extensiones (contenidos) están almacenados en el computador explícitamente.

Más precisamente, la base de datos de vuelos del ejemplo inicial puede ser vista como una estructura con un universo no vacío, relaciones definidas sobre ese dominio, y elementos distinguidos. El universo es el conjunto de todos los individuos que aparecen en la base de datos (SCL, MIAMI, ...,). Las relaciones (binarias) son: "Tiene Aeropuerto", "Vuelo Directo", "Conexión de Vuelo". Los individuos distinguidos pueden (y suelen) ser todos los del universo: SCL, MIAMI, ...

En el ejemplo inicial ya vimos un lenguaje de la lógica de predicados apropiado para describir esa base de datos. Ese lenguaje es compatible con la base de datos. En él introdujimos nombres (símbolos) para las relaciones, en este caso, $TA(\cdot)$, $VD(\cdot,\cdot)$, $CV(\cdot,\cdot)$, y nombres para los indivi-duos: $\overline{SCL}, \overline{MIAMI}, \ldots$ (para distinguirlos de los individuos). Con estos símbolos se construye un lenguaje de la lógica de predicados.

Ejemplo: Sea $S = \{P\}$, donde P es predicado unario. Una estructura compatible con S es $\langle \mathbb{Z}, \mathbb{P} \rangle$, donde $\mathbb{P}$ es el conjunto de números pares.

5.2.2 Interpretación de Términos

En el contexto sintáctico hemos procedido hasta ahora del siguiente modo: dado un conjunto de símbolos S, construimos el conjunto $T(S)$ de términos del lenguaje, y, en seguida, el lenguaje completo $L(S)$. Ahora, en el paralelo semántico, hemos dado interpretaciones sólo a los símbolos de S. Quisiéramos interpretar los términos de $T(S)$. Si el término es una constante $c \in S$, o de la forma $f(c)$, entonces, dada una interpretación $\mathfrak{A} = \langle A, f^A, c^A \rangle$ para S, podríamos interpretarlos, respectivamente como c^A y $f^A(c^A)$. Sin embargo, ¿cómo interpretamos el término $f(x)$, cuando x es una variable? Necesitamos dar valores en A a las variables del lenguaje. En consecuencia, con las estructuras podemos dar interpretaciones a los símbolos de S, pero no a todos los

términos de lenguaje, no a los que contienen variables.

Para poder dar interpretaciones a todos los términos de $T(S)$, partimos de una estructura

$$\mathfrak{A} = \langle A, R^A, ..., f^A, ..., c^A, .. \rangle$$

que da significado a los símbolos de S, y damos significado (valores) a las variables a través de una **asignación**. Esta es una función del conjunto de variables al dominio de la estructura:

$$\beta : \{x_1, x_2, ...\} \longrightarrow A,$$

es decir, β da valores, que están en el dominio de la estructura, a las variables del lenguaje.

Ejemplo: Consideremos nuevamente los símbolos aritméticos $S = \{<, +, \cdot, 0, 1\}$, y la estructura:

$$\mathfrak{N} = \langle \mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}} \rangle.$$

Una posible asignación es:

$$\beta_0 : \{x_1, x_2, ...\} \longrightarrow \mathbb{N},$$

definida por: $\beta_0(x_i) := 2i \quad (\in \mathbb{N})$. Por ejemplo, se tiene $\beta_0(x_7) := 14$.

¿Cuál es la interpretación de los términos de acuerdo con la estructura escogida y esta asignación? ¿Por ejemplo, del término $x_7 + ((1 + 1) \cdot x_3)$? La definición general debería darle la siguiente interpretación:

$$14 +^{\mathbb{N}} ((1^{\mathbb{N}} + 1^{\mathbb{N}}) \cdot^{\mathbb{N}} 6) = 26.$$

Definición: Una **interpretación** para $L(S)$ es una par $\mathcal{I} = (\mathfrak{A}, \beta)$, donde $\mathfrak{A}$ es una estructura compatible con S y β es una asignación sobre A.

Una interpretación, entonces, está formada por dos objetos semánticos. Con $\mathcal{I}$ podemos interpretar los términos de $T(S)$. La definición inductiva de la interpretación de un término está basada en la estructura de términos:

1. $\mathcal{I}(c) := c^A$ (de este tipo de términos da cuenta la estructura)
2. $\mathcal{I}(x) := \beta(x)$ (esto está determinado por la asignación)
3. $\mathcal{I}(f(t_1, ..., t_n)) := f^A(\mathcal{I}(t_1), ..., \mathcal{I}(t_n))$ (éste es el caso mixto y propiamente inductivo)

Ejemplo: Consideremos $S_{ar} = \{<, +, \cdot, 0, 1\}$ y la interpretación $\mathcal{I}_0 = (\mathfrak{N}, \beta_0)$ con:

$$\mathfrak{N} = \langle \mathbb{N}, <, +, \cdot, 0, 1 \rangle,$$
$$\beta_0 : x_i \longmapsto 2i, \quad i = 1, 2...$$

Según la definición general de interpretación de términos, tendríamos:

$$\begin{aligned} \mathcal{I}(x_7 + ((1+1) \cdot x_3)) &:= \beta_0(x_7) + (1+1) \cdot \beta_0(x_3) \\ &= 14 + 2 \cdot 6 \\ &= 26 \quad (\in \mathbb{N}). \end{aligned}$$

■

Siempre la interpretación de un término es un elemento del dominio de la estructura.

La especificación de interpretación de términos se puede hacer ejecutable a través de un programa que evalúe términos con respecto a una interpretación. Para esto necesitamos, además, una re-presentación computacional de la interpretación. Esto se puede hacer, en particular, cuando la interpretación está basada en una estructura proveniente de una base de datos provista de operaciones.

5.2.3 Noción de Verdad de Fórmulas

El siguiente paso es decir cuándo una fórmula es verdadera en una interpretación. Hasta ahora hemos manejado este concepto en forma solamente intuitiva. En esta definición daremos, de paso, significado a los símbolos lógicos de lenguaje.

Ejemplo: Consideremos nuevamente $S_{ar} = \{<, +, \cdot, 0, 1\}$ e $\mathcal{I}_0 = (\mathfrak{N}, \beta_0)$, con $\mathfrak{N} = \langle \mathbb{N}, <, +, \cdot, 0, 1 \rangle$ y $\beta_0 : x_i \longmapsto 2i, \quad i = 1, 2...$

La fórmula $\exists\, x_1(x_3 = (1+1) \cdot x_1)$ debería ser verdadera con respecto a $\mathcal{I}_0$, sin embargo, la fórmula $\neg\, \exists\, x_3(x_3 < x_1)$ debería ser falsa.

Ejemplo: Consideremos nuevamente la base de datos de vuelos del ejemplo 3 al comienzo del capítulo. Nos gustaría determinar si podemos volar de Santiago de Chile a Hong Kong con una sola escala intermedia. Para ello, podemos hacer la consulta a través de la fórmula:

$$\exists\, x(\, VD(SCL, x) \wedge VD(x, HK)).$$

Evaluar esta consulta con respecto a la base de datos es determinar si la fórmula es verdadera en la base de datos, cuando esta última es vista como una estructura en el sentido de la lógica, y posiblemente, en el caso afirmativo, obtener un valor para la variable x.

La evaluación de consultas expresadas como fórmulas se puede programar y ejecutar, ya que, por un lado, tendremos una definición de verdad de fórmulas susceptible de ser manejada computacionalmente en varios aspectos; y, por otro, usualmente tenemos una representación computacional de la estructura. Esta observación está en la base del diseño de lenguajes de consulta declarativos para bases de datos.

En esencia, un lenguaje –como SQL (Structured Query Language)– es un lenguaje formal que proviene de la lógica de predicados (del cálculo relacional, para respetar la terminilogía de bases de datos), y que permite hacer consultas a bases de datos, sin tener que especificar cómo obtener la información, como es el caso cuando se hace consultas a través del lenguje del álgebra relacional, que es un lenguaje procedural (o imperativo).

■

Consideremos un lenguaje de primer orden $L(S)$. Queremos definir la relación $\mathcal{I} \models \varphi$, donde $\mathcal{I} = (\mathfrak{A}, \beta)$ es una interpretación compatible con S y φ es una fórmula de $L(S)$. Esta relación se lee: "$\mathcal{I}$ satisface (hace verdadera a, es modelo de) φ".

La definición de verdad está basada en la "teoría de la correspondencia" de Alfred Tarski, según la cual, una aseveración es verdadera en un mundo (una realidad) si lo que expresa está en correspondencia con ese mundo. En nuestro caso, las aseveraciones son expresadas en un lenguaje formal y el mundo está representado por una interpretación.

Definición: Definimos $\mathcal{I} \models \varphi$ por inducción estructural en la fórmula φ de $L(S)$, para una interpretación $\mathcal{I} = (\mathfrak{A}, \beta)$ compatible con $L(S)$ arbitraria:

1. Si φ es fórmula atómica de la forma $t = s$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad \mathcal{I}(t) = \mathcal{I}(s).$$

$\mathcal{I}(t)$ y $\mathcal{I}(s)$ son las interpretaciones de t y s según $\mathcal{I}$, por lo tanto, son elementos de A, el dominio de $\mathfrak{A}$. En $\mathcal{I}(t) = \mathcal{I}(s)$ aparece el símbolo de igualdad del metalenguaje que denota a la igualdad en A. Es en esta parte de la definición donde estamos especificando que el símbolo de igualdad del lenguaje objeto sea interpretado como la relación de igualdad en el dominio de interpretación; es decir, si éste es A, como la relación $\{(a, a) \mid a \in A\}$, la "diagonal" de $A \times A$.

2. Si φ es atómica de la forma $R(t_1, t_2, ..., t_n)$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad (\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in R^A,$$

es decir, si y sólo si la tupla de intepretaciones de los términos pertenece a la relación R^A, que interpreta al símbolo de relación R según $\mathfrak{A}$.

3. Si φ es de la forma $\neg\psi$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad \text{no (es cierto que) } \mathcal{I} \models \psi.$$

Aquí estamos dando su significado al símbolo de negación $\neg$.

4. Si φ es de la forma $(\psi \wedge \chi)$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad \mathcal{I} \models \psi \ \text{ y } \ \mathcal{I} \models \chi.$$

5. Si φ es de la forma $(\psi \vee \chi)$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad \mathcal{I} \models \psi \ \text{ o } \ \mathcal{I} \models \chi.$$

6. Si φ es de la forma $(\psi \to \chi)$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad (\mathcal{I} \models \psi \ \Rightarrow \ \mathcal{I} \models \chi).$$

El símbolo "$\Rightarrow$" es del metalenguaje y asigna valores de verdad tal como en la lógica proposicional, es decir, según la tabla de verdad:

ψ	χ	$\psi \Rightarrow \chi$
V	V	V
V	F	F
F	V	V
F	F	V

7. Si φ es de la forma $(\psi \leftrightarrow \chi)$, entonces

$$\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad (\mathcal{I} \models \psi \ \Leftrightarrow \ \mathcal{I} \models \chi).$$

8. Si φ es de la forma $\exists x\ \psi$, entonces

   $\mathcal{I} \models \varphi \quad :\Leftrightarrow$ existe $a \in A$ tal, que $\mathcal{I} \models \psi$, asignando a x el valor a (y a las otras variables, sus valores según β).

   Esta especificación usualmente se escribe así:

   $\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad$ existe $a \in A$ tal, que $\mathcal{I}\frac{x}{a} \models \psi$, donde $\mathcal{I}\frac{x}{a}$ es la nueva interpretación $(\mathfrak{A}, \beta\frac{x}{a})$, cuya nueva asignación está definida por:

$$\beta\tfrac{x}{a}(y) := \begin{cases} \beta(y) & \text{si } y \neq x \\ a & \text{si } y = x \end{cases}$$

   Es decir, la nueva asignación coincide con β en los valores de todas las variables, excepto, posiblemente, en el valor de x, a la cual asigna el valor a.

   Aquí estamos dando su significado al cuantificador existencial $\exists$.

9. Si φ es de la forma $\forall x\ \psi$, entonces

   $\mathcal{I} \models \varphi \quad :\Leftrightarrow \quad$ para todo $a \in A$, $\mathcal{I}\frac{x}{a} \models \psi$.

■

La definición de la verdad de una fórmula de un lenguaje de la lógica de predicados de primer orden, significó un gran avance en el desarrollo de la semántica para lenguajes formalizados, de hecho, se considera un paradigma a seguir en las definición de una semántica precisa para otros tipos de lenguajes, por ejemplo, para lenguajes de otras lógicas y lenguajes de programación. La definición se debe a Alfred Tarski, y se remonta al año 1930. En ella se da el significado a los símbolos lógicos de los lenguajes de primer orden.

En 1. vemos que la interpretación del símbolo $=$ es fija. En toda estructura $\mathfrak{A} = (A, \ldots)$, se interpreta como la relación de igualdad sobre el universo A. Más adelante veremos cómo liberarnos de esta restricción si nos parece conveniente hacerlo.

Ejemplo: Consideremos $S_{ar} = \{<, +, \cdot, 0, 1\}$, la estructura $\mathfrak{R} = \langle \mathbb{R}, <, +, \cdot, 0, 1\rangle$, y la asignación

$$\begin{aligned} \beta \ : \ & \{x_1, x_2, ...\} \longrightarrow R \\ & x_i \longmapsto \frac{i+1}{2} \end{aligned}$$

Con ellas formamos la interpretación $\mathcal{I} = (\mathfrak{R}, \beta)$.

Veamos, de acuerdo con la definición anterior, si la fórmula $\exists\, x_6\ (x_6 + x_8 < 0)$ es verdadera en la interpretación escogida:

$$\mathcal{I} \models \exists\, x_6\ (x_6 + x_8 < 0) \quad \Leftrightarrow \quad \text{existe} \quad r \in \mathbb{R} \text{ tal, que :}$$

$$(\mathfrak{R}, \beta\frac{x_6}{r}) \models x_6 + x_8 < 0.$$

Es decir, si y sólo si existe $r \in \mathbb{R}$ tal, que: $r + \frac{8+1}{2} < 0$. Como esto es cierto, la fórmula original es verdadera en la intepretación dada.

■

Ejemplo: Veamos si es verdadera la fórmula $\forall\, x_3\ (0 < x_3 \ \rightarrow\ x_8 \cdot x_8 = x_3)$ en la estructura $\langle \mathbb{R}, <^{\mathbb{R}}, \cdot^{\mathbb{R}}, 0^{\mathbb{R}} \rangle$, con la asignación: $\beta : x_i \mapsto \sqrt{i}$.

Sea $\mathcal{R}'$ la estructura. De acuerdo con la definición de verdad de Tarski, la fórmula es verdadera en $\mathcal{R}'$ con la asignación β si y sólo si: para todo número real r, $(\mathcal{R}', \beta\frac{x_3}{r}) \models 0 < x_3 \ \rightarrow\ x_8 \cdot x_8 = x_3)$. Y esto si y sólo si, para todo número real r, $0 < r \ \Rightarrow\ \sqrt{8} \cdot \sqrt{8} = r$ (Todo esto en la estructura $\mathcal{R}'$. Omitimos los super $\mathbb{R}$). Finalmente, esto es cierto si y sólo si, para todo número real r, $0 < r \ \Rightarrow\ 8 = r$ (en la estrutura). Esta afirmación es falsa.

■

El proceso de evaluación de la fórmula podría ser realizado parcialmente a través de un algoritmo general basado en la definición inductiva de verdad de Tarski. Sin embargo, al final, en el último paso, fue necesario recurrir a nuestro conocimiento sobre la estructura de los reales (hay números reales menores que -4.5). Si quisiéramos una evaluación totalmente computacional, sería necesario contar con una representación computacional de la interpretación, como en el caso de bases de datos, o bien, con conocimiento declarativo sobre la estructura expresado a través de axiomas, que nos permitiera deducir automáticamente la información necesaria (en el caso del ejemplo, que hay números menores que -4.5).

Un problema que obviamente surge en el proceso de evaluación de fórmulas con respecto a una interpretación, tiene que ver con el manejo de los cuantificadores, que, en el caso del ejemplo, se refieren a los elementos del conjunto innumerable de los números reales. ¿Cómo tomamos en cuenta todos sus elementos? Notar que si tenemos una base de datos relacional usual, vista como

una estructura, entonces sí tenemos una representación explícita de la estructura, de hecho, con universo finito, y podemos evaluar una fórmula implementando computacionalmente la definición de Tarski (esto no necesariamente es lo más eficiente).

En el caso de la estructura $\mathfrak{R}$, dado que no tenemos una representación computacional explícita de ella, nos preguntamos si podemos usar alternativamente una descripción de la estructura en un lenguaje que sí pueda ser manejado computacionalmente. De este modo pasamos a una axiomatización expresada en términos de un conjunto de oraciones $AX \subseteq L(\{<, +, \cdot, 0, 1\})$, y nos preguntamos si φ se puede demostrar (o deducir) a partir de AX en lugar de preguntar y evaluar directamente si $\mathfrak{R} \models \varphi$. Al respecto, surgen varias preguntas:

(a) ¿Qué relación hay entre AX y $\mathfrak{R}$?

(b) ¿Qué relación hay entre $\mathfrak{R} \models \varphi$ y "φ se deduce de AX"?

(c) ¿Qué significa "deduce"?

(d) ¿Cuáles podrían ser los axiomas en el conjunto AX?

La pregunta (a) tiene que ver con poder **especificar** la estructura (de datos o tipo de datos, en el lenguaje computacional) $\mathfrak{R}$ por medio de AX. ¿Hay un conjunto de axiomas lo suficientemente descriptivos, y a la vez, manejables computacionalmente, que caractericen a la estructura, ojalá excepto por isomorfismos? Preguntas de este tipo serán tratadas en general, y en contextos específicos también, más adelante.

Para bases de datos, Raymond Reiter propone* hacer una "reconstrucción lógica" de las bases de datos relacionales, más precisamente, él entrega un método general para llevar el conocimiento almacenado en una base de datos relacional a un conjunto de oraciones de un lenguaje de la LPOP ad hoc. El conocimiento declarativo almacenado en estos axiomas es tal, que una consulta φ es verdadera en la base de datos si y sólo si ella es deducible a partir de los axiomas. Con esto es posible reemplazar "evaluación" por "deducción".

Volvamos a algunas consideraciones finales sobre la definición de Tarski. Si, en la relación $\mathcal{I} \models \varphi$, la fórmula φ es una oración, entonces su valor de verdad

*En "A Logical Reconstruction of Relational Database Theory", en 'On Conceptual Modelling', Brodie & Mylopoulos (eds.), Springer, 1984.

con respecto a $\mathcal{I} = (\mathfrak{A}, \beta)$ depende sólo de la estructura $\mathfrak{A}$ y no de la asignación β. En términos más generales, el valor de verdad de una fórmula con respecto a una interpretación depende sólo de la estructura y de los valores que se asigne a las variables libres de la fórmula. Esto puede ser precisado y demostrado fácilmente. Por ejemplo, $(\mathfrak{R}, \gamma) \models \forall\, x_7\, \exists\, x_3(\neg 0 < x_7 \vee x_3 \cdot x_3 = x_7)$,independientemente de γ, la asignación de valores a las variables.

Debido a lo anterior, basta indicar los valores que se asigna a las variables libres de una fórmula:

$$\mathfrak{A} \models \varphi\ [a_1, \ldots, a_n]$$

Aquí:

- $\mathfrak{A}$ es una estructura.
- φ es una fórmula con n variables libres, esto a veces lo denotamos así $\varphi(y_1, \ldots, y_n)$.
- $a_1, ..., a_n$ son los valores en A para las n variables libres $y_1, \ldots, y_n$, en un orden subentendido.

Por ejemplo, $\mathfrak{R} \models \exists\, x\ (x \cdot x = y)\ [2]$. Es decir, la fórmula es verdadera en la estructura $\mathfrak{R}$, dando a la variable libre y el valor 2.

Ejemplo: Volvamos otra vez a la base de datos de vuelos. Recordemos que evaluar consultas en una base de datos relacional corresponde, desde el punto de vista de la lógica, a verificar la verdad de una fórmula (la consulta) en una estructura (la base de datos). Si hay variables libres en la fórmula (consulta), entonces se busca individuos en el dominio de la estructura que satisfacen la fórmula (si se puede). Esto se puede hacer siguiendo el procedimiento recursivo que proviene de la definición de verdad de Tarski.

Por ejemplo, en el lenguaje de la lógica de predicados que introdujimos para describirla, podemos hacer, entre otras, las siguientes consultas y afirmaciones:

- ¿Cuales conexiones directas hay desde Santiago de Chile?

  Esta consulta se expresa a través de la fórmula $\psi(x) : \ VD(\overline{SCL}, x)$. Más precisamente, estamos preguntando por el conjunto de todos los posibles valores para x en la base de datos $\mathcal{BD}$ (mejor, en el universo de ella vista como estructura) que hacen verdadera la fórmula en la base de datos. Es decir, preguntamos por el conjunto $\{b \mid \mathcal{BD} \models \psi[b]\}$.

- ¿Cuales son las alternativas para ir de Miami a Frankfurt, teniendo a Hong–Kong como última escala antes de Frankfurt?

  Esta consulta puede ser planteada así: $CV(\overline{MIAMI}, x) \wedge VD(x, \overline{HONGKONG}) \wedge VD(\overline{HONGKONG}, \overline{FRANKFURT})$. La parte de esta consulta que involucra a CV puede ser respondida recurriendo a la definición recursiva de CV en términos de VD que vimos al presentar por primera vez la base de datos de vuelo*. Otra manera de plantear la consulta es simplemente a través de $CV(\overline{MIAMI}, \overline{HONGKONG}) \wedge VD(\overline{HONGKONG}, \overline{FRANKFURT})$, teniendo en cuenta que aún queda por determinar cómo se va de Miami a Hong Kong, es decir, hay que procesar la primera parte de la consulta.

- Cada vez que hay una conexión entre dos ciudades, éstas deben tener aeropuerto:

  Esta no es una consulta, sino una afirmación que puede ser escrita así: $\forall x \forall y \ (CV(x, y) \ \rightarrow TA(x) \wedge TA(y))$. Esta afirmación tiene al menos dos usos o roles. Primero, se puede usar como una "restricción de integridad" (o restricción de consistencia), que impide ingresar un par en la tabla "Conexión de Vuelo", sin ingresar a la vez las ciudades en la tabla "Tiene Aeropuerto". Segundo, se puede usar para deducir o ramificar nuevo conocimiento: si se tiene una par en la tabla "Conexión de Vuelo", se puede gatillar los ingresos correspondientes en la tabla "Tiene Aeropuerto", o bien, se puede deducir, a partir de la aparición de una tupla en la tabla "Conexión de Vuelo" y la regla anterior, que hay aeropuerto en las ciudades involucradas.

■

La definición de satisfacción tiene una extensión natural al caso en que se tiene un conjunto de fórmulas: una interpretación $\mathcal{I}$ (estructura $\mathfrak{A}$) satisface un conjunto de fórmulas (oraciones) $\sum \subseteq L(S) \ :\Leftrightarrow \ \mathcal{I} \models \varphi$, para todo $\varphi \in \sum$.

*Esa definición hay que interpretarla en forma procedural, al estilo de programación en lógica, pues se puede demostrar que la clausura transitiva de una relación, en este caso, CV es la clausura transitiva de VD, no puede ser definida en el sentido de la lógica clásica, mediante fórmulas de primer orden.

Ahora, tal como en el caso de la lógica proposicional, estamos en condiciones de definir la noción central de la lógica de predicados de primer orden, la de **consecuencia lógica**:

$$\Sigma \models \varphi \quad :\Leftrightarrow \quad \text{toda interpretación que satisface a } \Sigma \text{ también satisface a } \varphi.$$

Notar que ésta es una relación semántica entre dos objetos sintácticos: entre un conjunto de fórmulas y una fórmula individual.

Esta definición modela la noción intuitiva de "φ se concluye de (o es implicada por) las premisas en Σ". Veremos más adelante que una definición de consecuencia lógica no siempre toma la forma de arriba. Para modelar otras formas de razonamiento, por ejemplo, con sentido común, se ha propuesto "lógicas" que modelan otras nociones de "consecuencia lógica".

Ejemplo: Volviendo al ejemplo inicial, es fácil verificar que

$$\{\forall x\ (Hombre(x) \rightarrow\ Mortal(x)), Hombre(Socrates)\}\ \models\ Mortal(Socrates).$$

■

Finalmente, tal como en el caso de la lógica proposicional, podemos tener fórmulas que son verdaderas en cualquier interpretación compatible con el lenguaje de la fórmula. A estas fórmulas las llamamos "universalmente válidas" (o simplemente, "válidas"). Reservamos el término "tautología" para las fórmulas proposicionales verdaderas para toda valuación. Por ejemplo, $\forall x\ x = x$ y $P(c) \leftarrow \exists x\ P(x)$ son universalmente válidas (¡verificarlo!). Como en el caso proposicional, se tiene: φ es universalmente válida si y sólo si $\models \varphi$.

Ejemplo: La fórmula $\forall x\ 0 < x \cdot x$ no es universalmente verdadera. Para ver esto, basta con exhibir una estructura en la cual la fórmula sea falsa. Por ejemplo, la estructura $\mathcal{Z}' = \langle \mathbb{Z}, <, +, 0 \rangle$, es decir, con $\cdot$ interpretada como la suma en $\mathbb{Z}$ no hace verdadera la oración.

5.2.4 Ejercicios

1. Determine, justificando, si las siguientes fórmulas, por separado, son satisfacibles (verdaderas en alguna interpretación) o (universalmente) válidas: $1 < 2 \rightarrow \exists x(x < 2)$, $P(a,b) \rightarrow \exists x P(x,b)$, $\exists x(x < 1)$, $1 < 2 \wedge \forall x \neg(x < 2)$, $\forall x \forall y \forall z(x < y \wedge y < z \rightarrow x < z)$.

2. Considere el lenguaje de primer orden basado en en el conjunto de símbolos $S = \{<, t, a\}$ ($<$ es predicado binario; t es operación unaria; a es constante). Determine si las siguientes oraciones son verdaderas, satisfacibles, falsas en la estructura: $\mathfrak{A} = \langle\{0,1,2\}, \{(0,1),(1,2),(2,0)\}, ((\cdot)+2) \bmod 3, 0\rangle$: $\forall x \exists y(t(x) < t(y))$, $\forall x(t(x) < t(y))$, $\forall x((t(x) < t(y)) \to \neg x = y)$.

3. Sea P un predicado binario. Sea φ la fórmula:

$$\forall x \exists y P(x,y) \ \wedge \ \forall x \forall y_1 \forall y_2((P(x,y_1) \wedge P(x,y_2)) \to y_1 = y_2) \wedge$$
$$\forall x_1 \forall x_2 \forall y((P(x_1,y) \wedge P(x_2,y)) \to x_1 = x_2) \ \wedge \ \neg \forall y \exists x P(x,y).$$

(a) Demuestre que hay una estructura infinita $\mathfrak{A}$ (es decir, con universo infinito A) en la cual φ es verdadera.

(b) Demuestre que φ es falsa en toda estructura finita.

Ind.: Piense en P como una función. ¿Por qué?

4. (a) Dé una oración de primer orden ϕ_n que sea satisfecha sólo por estructuras con al menos n ($\in \mathbb{N}$) individuos ($n \geq 1$). (El número de individuos de una estructura es el número de elementos de su universo.)

(b) Dé una oración de primer orden que sea satisfecha sólo por estructuras con exactamente n elementos.

5. Sea ϕ_n la oración de 4(a), con n fijo. Demuestre que la siguiente oración es verdadera en todas las estructuras compatibles con el conjunto de símbolos $\{f\}$ (f operación unaria):

$$\phi_n \wedge \forall x \forall y(f(x) = f(y) \to x = y)) \to \forall x \exists y(f(x) = y).$$

¿Qué dice la oración?

6. Escriba una oración del lenguaje de primer orden basado en $\{R\}$ (R es un predicado binario) que exprese el hecho que cuando tenemos una relación de equivalencia y dos elementos están relacionados con un tercero, entonces también están relacionados entre sí.

7. Dé una demostración informal de que la siguiente oración es universalmente válida, es decir, verdadera en toda estructura compatible con ella:

$$(\forall x P(x, f(g(x))) \wedge \forall x \forall y \forall z((P(x,y) \wedge P(y,z)) \to P(x,z)))$$
$$\to \ \forall x \exists y P(x, f(g(f(y)))).$$

8. Sea $S_{ar} = \{<, +, *, 0, 1\}$ el conjunto de símbolos para la aritmética. Escriba programas en SCHEME y PASCAL que entreguen el conjunto de variables libres de cualquier fórmula φ de $L(S)$. Compare ambos programas con la especificación de $VL(\varphi)$ que se dio en este capítulo.

9. Sean $S_{ar} = \{<, +, \cdot, 0, 1\}$, $\mathfrak{R} = \langle \mathbb{R}, <, +, \cdot, 0, 1\rangle$, $\mathfrak{R}' = \langle \mathbb{R}, \mathbb{R} \times \mathbb{R}, +, \cdot, 0, 1\rangle$, y β una asignación con $\beta(x_i) = 3$, para todo i.

(a) Demuestre o refute: $(\mathfrak{R}, \beta) \models \exists\, x_2\ x_2 \cdot x_2 = 1$.

(b) Sea $\varphi:\ \ \forall\, x_5(\neg x_5 = 0 \to \exists\, x_1\ x_5 \cdot x_1^3 + (x_5 + x_5) \cdot x_1 + x_5 = 0)$. Demuestre o refute:
$(\mathfrak{R}, \beta) \models \varphi, \quad (\mathfrak{R}', \beta) \models \varphi$.

10. Sea $S = \{R(\cdot, \cdot)\}$. Demuestre o refute:
(a) $\{\forall\, x \exists\, y\ R(x, y)\} \models \exists\, y \forall\, x\ R(x, y)$.
(b) $\{\exists\, y \forall\, x R(x, y)\} \models \forall\, x \exists\, y\ R(x, y)$.

12. Sean $S = \{f(\cdot), c\}$ y
φ_1: $\forall\, x \forall\, y\ (f(x) = f(y) \to x = y)$,
φ_2: $\forall\, x\ \neg f(x) = c$,
φ_3: $\forall\, x\ (\neg x = c \to \exists\, y\ f(y) = x)$.

Dé estructuras $\mathfrak{A}$, $\mathfrak{A}_1$, $\mathfrak{A}_2$ y $\mathfrak{A}_3$, compatibles con S, tales, que: $\mathfrak{A} \models \{\varphi_1, \varphi_2, \varphi_3\}$, $\mathfrak{A}_1 \models \{\neg\varphi_1, \varphi_2, \varphi_3\}$, $\mathfrak{A}_2 \models \{\varphi_1, \neg\varphi_2, \varphi_3\}$, $\mathfrak{A}_3 \models \{\varphi_1, \varphi_2, \neg\varphi_3\}$.

11. Considere la siguiente base de datos:

Usable	Vuelo	Máquina
	#83	B727
	#83	B747
	#84	B727
	#84	B747
	#109	B707

Certificado	Piloto	Máquina
	Simmons	B707
	Simmons	B727
	Barth	B747
	Hill	B727
	Hill	B747

(a) Reconstruya esta base de datos como una estructura para lenguajes de la LPOP. Indique cuál es el lenguaje.

(b) Formule las siguientes hechos y consultas en un lenguaje de la LPOP apropiado para describir la base de datos:

1. Hill puede pilotear en el vuelo #83? ¿Qué significa evaluar esta consulta en la reconstrucción estructural de la base de datos que hizo?
2. ¿En qué vuelos puede volar el piloto Simmons?
3. Queremos una lista de vuelos con sus posibles pilotos (no nos interesa la máquina).
4. En la tabla "certificado" sólo aparecen tuplas de individuos que son pilotos y máquinas, y en ese orden.

Capítulo 6

Demostraciones Formales

6.1 Introducción

El hecho $\sum \models \varphi$, que leemos "φ es consecuencia lógica de $\sum$", también lleva la carga intuitiva de "φ es *demostrable* a partir de las hipótesis en $\sum$". De hecho, si pensamos en $\sum$ como un conjunto de axiomas, y en φ, como en un teorema de la teoría con axiomas $\sum$, nos damos cuenta que en matemática usual, al demostrar teoremas, lo que hacemos es establecer hechos semánticos del tipo $\sum \models \varphi$, verificando que cada vez que son verdaderos los axiomas de $\sum$, también el "teorema" φ, lo es. Sin embargo, este hecho rara vez se establece mostrando explícita y exhaustivamente que cada estructura que hace verdadera a $\sum$, también hace verdadera a φ. Más bien, esto se hace a través de "demostraciones" que tienen fuertes componentes "deductivas".

Es natural preguntarse cuáles son esos componentes deductivos, cuáles son admisibles en una demostración matemática, y, más radicalmente, si es posible reformular la noción semántica de consecuencia lógica en términos puramente deductivos. Como estos procesos deductivos procesan de alguna manera la información expresada a través de la fórmulas en $\sum$, φ, y otras intermedias, obteniendo nuevas fórmulas a partir de ellas, la última pregunta se puede, en el fondo, replantear en términos de la existencia de una contraparte sintáctica para la noción semántica.

Volvemos entonces al tema de la modelación dentro de la lógica de primer orden de las nociones intuitivas de "deducción" y "demostración". Esto lo haremos a nivel de lenguaje objeto, a nivel sintáctico, formal, en el sentido que las demostraciones serán ciertas sucesiones finitas de fórmulas. Para ganar mayor intuición sobre el problema, veremos primero algunas situaciones familiares de la matemática usual (pensar, por ejemplo, en un curso de álgebra lineal o álgebra abstracta, donde se desarrollan la "teoría de espacios vectoriales" o la "teoría de grupos").

Las demostraciones usuales de la matemática parten de proposiciones, van produciendo nuevas proposiciones mediante aplicaciones de reglas lógicas, y llegan a una proposición final después de un número finito de pasos "lógicos". Sin embargo, no está claro qué es precisamente una demostración matemática. Después de que el profesor del curso de álgebra termina una demostración, usualmente nos quedamos convencidos de que el presunto teorema en realidad lo es. Sin embargo, si nos ponemos a pensar más en profundidad en lo que significa la demostración a la que fuimos expuestos, tendremos problemas en caracterizarla exactamente, a pesar de que los pasos son "tan lógicos". Además, hay factores psicológicos, como la credibilidad del profesor o, más importante aún, el hecho que usualmente es en los mismos cursos de matemática, al ser ejercitados en hacer demostraciones, que aprendemos esa "lógica de las demostraciones". De hecho, es cuestionable si la traemos desde antes de empezar los primeros cursos de matemáticas, en los cuales se nos presenta demostraciones por primera vez. Tal vez es una lógica adquirida: aprendemos, tarde en nuestras vidas, las formas de razonamiento que nos permiten entender y hacer demostraciones matemáticas.

En fin, pretendemos modelar el concepto de demostración matemática, que, para todos, representa el paradigma deductivo. De hecho, es natural querer incorporar la capacidad de hacer demostraciones a un sistema computacional "inteligente", porque, por una lado, esta capacidad parece ser algo propio de un agente inteligente, y porque, por otro, percibimos en esa actividad ciertos procesos susceptibles de ser mecanizados.

Veamos un ejemplo de la matemática usual, más precisamente, de la teoría de grupos. Usaremos los axiomas y proposiciones informalmente, tal como en un curso de matemática típico.

Ejemplo: Consideremos los **axiomas de la teoría de grupos**:

(1) Para todo a, b, c : $a * (b * c) = (a * b) * c$ (asociatividad).

(2) Para todo a : $a * e = a$ (e es neutro por la derecha).

(3) Para todo a, existe b : $a * b = e$ (existencia de inverso).

Por definición, un **grupo** es una estructura $\mathfrak{G} = (G, *, e)$ que satisface los axiomas (1), (2), (3). Aquí:

- G es un conjunto no vacío.

- $*$ es una operación binaria sobre G.
- e es un elemento distinguido de G.

Dados los axiomas de la teoría, nos aparecen los "teoremas de la teoría de grupos", por ejemplo,

Teorema:* Si $(G, *, e)$ es un grupo, entonces:

$$\text{para todo } g \in G: \quad e * g = g. \qquad (4)$$

■

El teorema dice que e es neutro por la izquierda también. Más precisamente, que todo grupo satisface la proposición: "para todo g : $e * g = g$". Es decir, que toda estructura que satisface (1), (2), (3) también satisface (4). En otras palabras, que "(4) es concecuencia lógica de (1), (2), (3)":

$$\{(1), (2), (3)\} \models (4).$$

Como vemos, en matemática usual planteamos relaciones de consecuencia lógica entre axiomas y teoremas. Cuando demostramos un teorema, en realidad, lo que hacemos es demostrar que se cumple una relación semántica de consecuencia lógica. Veamos esto en el ejemplo a través de la "demostración" del teorema:

Demostración†: Sea $(G, *, e)$ un grupo arbitrario, es decir, una estructura que satisface los axiomas (nótese esta partida semántica).

Sea $g \in G$ un elemento arbitrario (por el "para todo" de (4)). Entonces, por (3), existe $g' \in G$ tal, que:

$$g * g' = e. \qquad (\alpha)$$

Por (3), existe g'' tal, que:

$$g' * g'' = e. \qquad (\beta)$$

*Este es un teorema que hemos importado de un curso de álgebra y que usaremos como motivación para temas de la lógica.

†Damos una demostración como la haríamos en un curso de álgebra.

Se tiene:

$$\begin{aligned}
g &= g * e && \text{por (2)} \\
&= g * (g' * g'') && \text{por } (\beta) \\
&= (g * g') * g'' && \text{por (1)} \\
&= e * g'' && \text{por } (\alpha) \\
&= (e * e) * g'' && \text{por (2)} \\
&= (e * (g * g')) * g'' && \text{por } (\alpha) \\
&= (e * g) * (g' * g'') && \text{por 2 veces (1)} \\
&= (e * g) * e && \text{por } (\beta) \\
&= e * g. && \text{por (2)}
\end{aligned}$$

Q.E.D.

Esta demostración amerita algunas observaciones:

1. A pesar de que hemos usado estructuras concretas, la demostración es bastante simbólica; estamos manipulando axiomas y propiedades. Naturalmente nos preguntamos si no se puede hacer en forma totalmente simbólica.

2. En la demostración usamos desaprensivamente la relación de igualdad (=) y sus propiedades, por ejemplo, simetría (notar que hemos demostrado que $g = e * g$ y no que $e * g = g$, pero nos damos por satisfechos) y sustitutividad de iguales por iguales en ciertas expresiones.

3. No cuestionamos el uso de estas propiedades de la igualdad porque las consideramos parte de nuestra "lógica". Sin embargo, en nuestra modelación de la noción de demostración, tendremos que dar cuenta de eso.

4. También es natural preguntarse por qué es correcta la demostración que dimos. Posiblemente, si la demostración fuera totalmente simbólica, sería más fácil caracterizar la corrección de la demostración y eventualmente podríamos poner a un programa computacional a verificar esa corrección.

En la demostración anterior partimos con los axiomas y, después de un proceso deductivo, llegamos finalmente a la proposición que queríamos demostrar. Esto contrasta con las demostraciones por contradicción, en las cuales agregamos a las hipótesis (los axiomas) la negación de lo que queremos probar, y

demostramos una contradicción. Esto se refleja en el hecho semántico que vimos en lógica proposicional y que también es válido en lógica de primer orden:

$$\sum \models \varphi \iff \sum \cup \{\neg\varphi\} \text{ es insatisfacible.}$$

Por otro lado, intuimos "$\sum \cup \{\neg\varphi\}$ es insatisfacible" como "$\sum \cup \{\neg\varphi\}$ es inconsistente", y ésta como "$\sum \cup \{\neg\varphi\}$ es contradictoria ". Sin embargo, estas intuiciones tienen que ver con la existencia de una contradicción que puede ser demostrada, deducida, a partir de, en este caso, $\sum \cup \{\neg\varphi\}$.

Para modelar este tipo de demostraciones por contradicción, presentamos, en el caso de la lógica proposicional, las refutaciones por resolución (demostraciones por resolución de una contradicción)*. Sin embargo, la noción formalizada de demostración que veremos a continuación permite modelar naturalmente tanto demostraciones por contradicción como demostraciones directas (como las del teorema). En ese sentido es un modelo más amplio que el correspondiente a resolución (aunque no necesariamente más efectivo desde el punto de vista computacional). Además, es un modelo más extendible, en el sentido que se puede adaptar más fácil y naturalmente a otras lógicas (que modelan otras formas de procesos deductivos distintos de los matemáticos) para las cuales no hay naturalmente algo parecido a resolución. Por último, el sistema deductivo que presentaremos admite fórmulas generales, en contraste con resolución, que admite sólo cláusulas.

Nuestra modelación formal del concepto de demostración está basado en los sistemas deductivos de David Hilbert. Hemos elegido este tipo de sistema deductivo por su naturalidad. Sin embargo, hay muchos otros sistemas deductivos.

6.2 Un Sistema Deductivo para la Lógica Proposicional

Comenzaremos con un sistema formal deductivo para la lógica proposicional. Esto se debe a que podemos ver a la lógica de predicados como una extensión de la lógica proposicional, en consecuencia, el sistema formal para la primera también puede ser visto como una extensión del sistema formal para la segunda.

Para cada lenguaje proposicional tendremos un sistema formal deductivo. Sean $L(P)$ un lenguaje proposicional, $\sum(\subseteq L(P))$ un conjunto de fórmulas,

*Hay una versión de resolución para la lógica de primer orden que veremos más adelante. Esta versión es particularmente útil y fundamental en muchos demostradores mecánicos de teoremas y en programación en lógica, en particular, en PROLOG.

y $\varphi(\in L(P))$, una fórmula particular.

Definición: Una demostración (deducción, derivación) formal de φ a partir de $\sum$ es una sucesión finita de fórmulas de $L(P)$

$$\begin{array}{c} \varphi_1 \\ \varphi_2 \\ . \\ . \\ . \\ . \\ \varphi_{n-1} \\ \varphi_n \end{array}$$

tal, que:

- cada $\varphi_i \in L(P)$;
- φ_n es φ;
- cada φ_i es:
  1. elemento de $\sum$ o
  2. axioma lógico o
  3. se obtiene a partir de fórmulas precedentes en la sucesión, digamos φ_j , φ_l $(j \ , \ l \ < \ i)$ mediante una aplicación de la regla de deducción.

Notar que una demostración formal es un objeto sintáctico, formal, simbólico. En ella pueden aparecer hipótesis (las de $\sum$), fórmulas que son axiomas lógicos (de la lógica pura, sin hipótesis particulares), y fórmulas que se obtienen a través de un paso deuctivo. Por supuesto, debemos especificar cuáles son los axiomas lógicos y cuál es la regla de deducción.

Los Axiomas Lógicos de L(P): Intuitivamente, son algunas de las fórmulas que son verdaderas "por lógica", y no como producto de ciertas premisas o hipótesis particulares. Ellos forman una lista infinita de fórmulas, sin embargo, en lugar de dar esta lista infinita, daremos tres mecanismos generadores, o esquemas, de axiomas lógicos.

Los axiomas lógicos son las instancias de cualquiera de los tres esquemas siguientes:

1. $\varphi \to (\psi \to \varphi)$
2. $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
3. $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$

En estos esquemas, $\varphi, \psi, \chi, ...$ denotan fórmulas de $L(P)$, es decir, son metavariables para fórmulas de $L(P)$. Reeplazando $\varphi, \psi, \chi, ...$ por fórmulas concretas se obtiene axiomas lógicos concretos.

Ejemplo: $S = \{p, q\}$. Reeplacemos φ por $(p \to q)$ y ψ por q en el esquema 1. Con esto se obtiene el axioma lógico concreto: $(p \to q) \to (q \to (p \to q))$.

Podemas ver que esta fórmula es una tautología.

■

Es fácil convencerse, a través de tablas de verdad, de que cualquier axioma lógico obtenido de los esquemas es una tautología. Sin embargo, directamente de los esquemas no obtenemos todas las tautologías de $L(P)$. Esto no se debe sólo a que hay ciertos símbolos lógicos $(\wedge, \vee, \leftrightarrow)$ que no aparecen en ellos, ya que éstos son definibles a partir del conjunto funcionalmente completo $\{\to, \neg\}$. De hecho, tampoco se obtienen directamente otras tautologías que contienen sólo $\to$ y $\neg$. La idea es obtener estos últimos usando la regla de deducción. De este modo, a través de los tres esquemas y la regla de deducción se podrá obtener todas las tautologías*, y en consecuencia, en las demostraciones formales tendremos todas ellas a nuestra disposición. Nuestro conjunto de axiomas lógicos es austero y manejable a la vez, donde por "manejable" entendemos "computacionalmente manejable", ya que los axiomas lógicos que ellos generan tienen una forma sintáctica bien determinada.

La Regla de Deducción:

$$\frac{\varphi \to \psi \quad \varphi}{\psi}$$

Regla de Modus Ponens o de
Eliminación del Condicional

*No se espera que el lector quede completamente convencido de este hecho en este punto. Esto es algo que abordaremos nuevamente más adelante.

En esta regla, φ y ψ son metavariables para fórmulas de $L(P)$. La regla puede ser entendida de la siguiente forma: "si en algún instante (de una demostración) se dispone de fórmulas de las formas $(\varphi \to \psi)$ y φ, entonces se puede inferir (saltar a, concluir, deducir, generar) la fórmula ψ.

Nosotros usamos esta regla en muchos de nuestros raciocinios. Un ejemplo es el siguiente:

"Si Sócrates es hombre, entonces es mortal"
"Sócrates es hombre"

"Sócrates es mortal"

La regla de deducción es formal, sintáctica; y aporta el elemento dinámico a los procesos deductivos, ya que permite ir generando explícitamente nuevo conocimiento (formalizado) que está implícito en conocimiento previo. En contraste, los axiomas lógicos o de una teoría $\sum$ representan conocimiento estático.

Cabe preguntarnos si la regla de Modus Ponens (MP) es aceptable en algún sentido. Creemos que sí. Por una lado, parece intuitiva en el sentido que $(\varphi \to \psi)$ y φ "implican" ψ. Además, modela cierto tipo de deducciones que efectuamos a menudo. Por otro lado, la regla es **correcta**, es decir –y ésta es una definición– "preserva la verdad" en el siguiente sentido:

$$(\varphi \to \psi)\ ,\ \varphi \models \psi,$$

es decir, la conclusión ψ obtenida con aplicación de la regla MP es consecuencia lógica de las premisas $(\varphi \to \psi)$ y φ.

Así, tenemos sistemas formales deductivos para los lenguajes de la lógica proposicional. Estos son los llamados sistemas de Hilbert*. Con ellos tenemos, además, completa nuestra definición de demostración formal.

Notación: Cuando existe una demostración formal de una fórmula φ a partir de un conjunto de fórmulas $\sum$, escribimos:

$$\sum \vdash \varphi.$$

*Por su creador, el matemático alemán David Hilbert. Hay otros tipos de sistemas deductivos. De hecho, el primer tipo sistema formal deductivo para lenguajes proposicionales fue propuesto por Gottlob Frege ($\sim$ 1890).

Esto significa, entonces, que hay una sucesión de fórmulas de la forma

φ_1
.
.
.
φ_i
.
.
.
φ

donde cada φ_i (incluyendo a φ) pertenece a $\sum$ o es axioma lógico, o es deducible con la regla MP de fórmulas anteriores. En este caso, decimos que φ es un **teorema** de $\sum$.

Notación: Si $\sum$ es el conjunto vacío $\varnothing$ de fórmulas, escribimos:

$$\vdash \varphi.$$

Esto significa que φ se puede deducir sin premisas o hipótesis especiales, sino sólo a partir de axiomas lógicos y la regla de deducción. En este caso, decimos que φ es un **teorema** (de la lógica proposicional).

Ejemplo: $\vdash (p \to p)$.

En efecto, una demostración formal (si premisas) de $(p \to p)$ es la siguiente:

1. $((p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p)))$ axioma lógico obtenido del esquema 2. con: $\varphi \equiv p$, $\psi \equiv (p \to p)$, $\chi \equiv p$.
2. $p \to ((p \to p) \to p)$ axioma lógico del esquema 1.
3. $((p \to (p \to p)) \to (p \to p))$ MP aplicado a 1. y 2.
4. $p \to (p \to p)$ axioma lógico del esquema 1.
5. $p \to p$ MP aplicado a 3. y 4.

■

Ejemplo: $\underbrace{\{p, q \to (p \to r)\}}_{\sum} \vdash (q \to r)$.

Hay que dar una demostración formal de $(q \to r)$ a partir de $\sum$:

1. p hipótesis de $\sum$
2. $(q \to (p \to r))$ hipótesis de $\sum$
3. $p \to (q \to p)$ axioma lógico del esquema 1.
4. $q \to p$ MP con 1. y 3.
5. $(q \to (p \to r)) \to ((q \to p) \to (q \to r))$ axioma lógico del esquema 2.
6. $(q \to p) \to (q \to r)$ MP con 2. y 5.
7. $(q \to r)$ MP con 4. y 6.

■

Algunas Observaciones:

- Vemos que las demostraciones son objetos sintácticos.
- "$\vdash$" es una relación entre un conjunto de fórmulas y una fórmula. Decimos que es una relación sintáctica porque no está definida, como la relación de consecuencia lógica "$\models$", en términos de esos objetos semánticos que son las asignaciones de verdad.
- Este sistema deductivo es poco natural, en el sentido que es difícil demostrar, **en** él, teoremas (tan simples como en los ejemplos anteriores), pero es fácil demostrar cosas **sobre** él, es decir, **metateoremas**, que son teoremas (en el sentido matemático usual) sobre el sistema deductivo.
- Veremos más adelante sistemas deductivos más apropiados para un uso computacional.

En el primer ejemplo obtuvimos $\vdash (p \to p)$. Notar que $(p \to p)$ es una tautología. Una pregunta natural es la siguiente: ¿es cierto que siempre $\vdash \varphi \Rightarrow \models \varphi$? Intutivamente, dado que cada axioma lógico es una tautología y que la regla de deducción preserva la verdad, todo teorema de la lógica proposicional debería ser tautología. Esto será demostrado más adelante.

Más generalmente, veremos:

$$\sum \vdash \varphi \quad \Rightarrow \quad \sum \models \varphi.$$

Esto nos dice que el sistema deductivo es **correcto**, es decir, si una fórmula se obtiene mediante una demostración formal a partir de $\sum$, entonces ella es consecuencia lógica de $\sum$. Con el sistema deductivo no se puede deducir "cosa raras" que no sean "implicadas" por las hipótesis $\sum$.

Otra pregunta que naturalmente se nos hace presente es la siguiente: ¿es verdad que toda tautología es un teorema, es decir, puede ser demostrada formalmente con el sistema deductivo? O, más generalmente, ¿es verdad que $\sum \models \varphi \quad \Rightarrow \quad \sum \vdash \varphi$? También veremos que esta pregunta tiene una respuesta positiva, es decir, que los 3 esquemas de axiomas lógicos (que generan infinitos axiomas lógicos) y la regla de deducción bastan para capturar completamente la noción semántica de consecuencia lógica. Diremos que el sistema formal es **completo**.

6.2.1 Ejercicios

Todos los ejercicios son sobre lógica proposicional.

1. Construir demostraciones formales correspondientes a:

(a) $\vdash (p \to (p \to p))$
(b) $\{p\} \vdash p \to p$
(c) $\{p, p \to q\} \vdash q$
(d) $\{p\} \vdash q \to p$
(e) $\{p \to q, q \to r\} \vdash p \to r$
(d) $\{p \to (q \to r), q\} \vdash p \to r$
(f) $\vdash \neg\neg p \to p$
(g) $\vdash p \to \neg\neg p$
(h) $\{p \to \neg q\} \vdash (q \to \neg(p)$
(i) $\{\neg p \to p\} \vdash p$
(j) $\vdash (p \to q) \to ((\neg p \to q) \to q))$
(k) $\vdash (p \to (\neg q \to F)) \to (p \to q)$ (F es una contradicción, por ejemplo, $\neg(p \to p)$)

2. Demostrar (sin pasar a la versión semántica) que:

(a) $\Sigma \cup \{\chi\} \vdash \varphi$ y $\Sigma \cup \{\neg\chi\} \vdash \varphi$ implican $\Sigma \vdash \varphi$ (Demostración por Casos).

(b) $\Sigma \cup \{\neg\varphi\} \vdash F$ implica $\Sigma \vdash \varphi$ (Demostración por Contradicción). Aquí, F es una contradicción, por ejemplo, la del ejercicio 1.(k).

Ind.: use (1j)

3. Podemos aumentar el sistema formal deductivo introduciendo una nueva regla de deducción, la de demostración por casos (DC):

$$\frac{\chi \to \varphi \qquad \neg\chi \to \varphi}{\varphi}$$

(a) Demostrar que esta regla es correcta.

(b) Demostrar que DC es redundante, es decir, toda demostración que se haga usando la regla DC puede ser traducida a una demostración que no la usa. (Decimos que es una regla "derivable").

Ind.: lo puede inspirar el ejercicio 2.

(c) Dé un ejemplo de una demostración formal que sea más corta con DC que sin ella. Ilustre el algoritmo de compilación que dio en (b).

4. Consideremos la "regla de abducción" para la lógica proposicional:

$$\frac{\varphi \to \psi, \quad \psi}{\varphi}$$

Esta regla es usada, por ejemplo, en diagnóstico: inferimos una explicación, φ, a partir de una evidencia, ψ.

(a) Demostrar que esta regla no es correcta, i.e. la conclusión de la regla no es consecuencia lógica de los antecedentes de la regla.

(b) Con esta regla, la noción de demostración formal deja de ser correcta. Precise en qué sentido.

(c) Por (b), deja de haber la correspondencia que había entre "consecuencia lógica" ($\models$) y "deducción" ($\vdash$). Con esta correspondencia que había, podemos obtener, de la parte semántica, muchas propiedades de la noción $\vdash$. ¿Cúales de las propiedades que aparecen en la sección 2.7 y los ejercicios 2.7.1 siguen siendo válidas y cuáles no? ¡Demuestre o refute!

(d) En los casos en que se pierda propiedades con respecto a la lógica proposicional usual, discuta qué consecuencias puede tener eso para hacer raciocinio de diagnóstico. Por ejemplo, ¿qué consecuencias tiene la pérdida de la monotonía para esta lógica?

6.2.2 Propiedades del Sistema Formal Deductivo

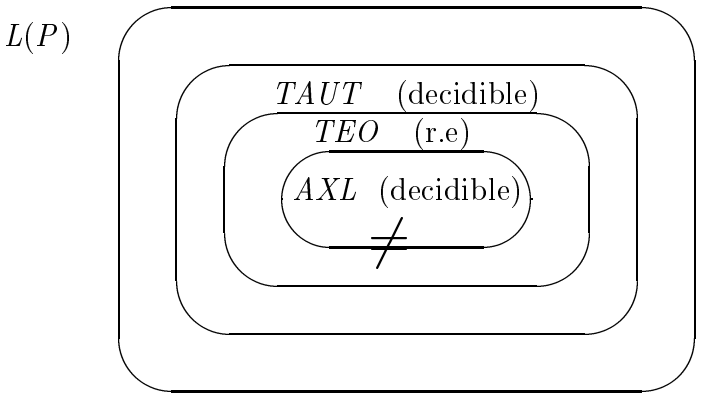
Fijemos un lenguaje proposicional $L(P)$. Los axiomas lógicos de $L(P)$ forman un conjunto decidible AXL. En efecto, dada una fórmula arbitraria $\varphi \in L(P)$, basta chequear si ella toma la forma sintáctica de alguno de los tres esquemas. Luego, también es recursivamente enumerable (r.e)*.

Sea $TEO := \{\varphi \in L(P) \mid \vdash \varphi\}^*$, el conjunto de fórmulas demostrables formalmente (sin permisas). Este conjunto es r.e.: basta con ir aplicando "sistemáticamente" la regla de deducción junto con la generación "sistemática" de axiomas lógicos a través de los 3 esquemas.

El teorema de corrección que veremos a continuación nos dice que:

$$TEO \subseteq TAUT := \{\ \varphi \in L(P)\ \mid \models \varphi\ \}.$$

En consecuencia, tenemos la siguiente situación:



Otro hecho interesante sobre los aspectos computacionales del sistema deductivo es el siguiente: es posible chequear automáticamente si una sucesión finita de fórmula es una demostración legal a partir de un conjunto decidible de premisas, es decir, hay un algoritmo para chequear la legalidad de presuntas demostraciones formales. En otras palabras, podríamos poner a un programa computacional a verificar la corrección de demostraciones formales. Sin embargo, esto no nos dice que hay un algoritmo para producir demostraciones formales. De hecho, el área de demostración mecánica de teoremas es una de las más difíciles de la lógica matemática y la inteligencia artificial.

---

*Supondremos que el conjunto de símbolos P es decidible o recursivamente enumerable.

*A veces, usamos las notaciones $:=$ y $:\Leftrightarrow$ para indicar que lo que está a la izquierda del símbolo se está definiendo en términos de lo que está al lado derecho de él.

Teorema (de corrección):[†] Sean $\sum \subseteq L(P)$ y $\varphi \in L(P)$. Se tiene:

$$\sum \vdash \varphi \quad \Rightarrow \quad \sum \models \varphi.$$

Demostración: Primero algunas consideraciones preliminares. "$\sum \vdash \varphi$" significa que existe una demostración formal:

φ_1
$\vdots$
$\varphi_n \quad (\equiv \varphi)$,

con $\varphi_i \in \sum$ o $\varphi_i \in AXL$ o φ_i obtenible mediante MP de las fórmulas precedentes.

Notar que si truncamos una demostración en, digamos, φ_j , lo que tenemos es una demostración formal de φ_j a partir de $\sum$:

φ_1
$\vdots$
φ_j

Hay que probar que: $\sum \vdash \varphi \ \Rightarrow \ \sum \models \varphi$. Esto lo haremos por "inducción en el largo de la demostración formal" de φ, es decir, demostraremos que, para todo $n \in \mathbb{N}$, si φ tiene una demostración de largo n a partir de $\sum$, entonces $\sum \models \varphi^*$. Con esto estamos listos pues toda demostración formal tiene algún largo finito n.

La inducción:

Primer caso: $n = 1$, es decir, la demostración formal es de la forma $\varphi_1 \ (\equiv \varphi)$. Entonces hay sólo dos posibilidades: $\varphi_1 \in AXL$ o $\varphi_1 \in \sum$.

Como todo axioma lógico es tautólogia, tenemos: $\models \varphi$. Por monotonía podemos concluir: $\sum \models \varphi$.

En el otro caso, $\varphi_1 \in \sum$; y se tiene trivialmente que $\sum \models \varphi$.

[†]Este es un teorema "sobre" el sistema formal deductivo y no "del" sistema tal como se definió antes. La demostración de él es como la de cualquier teorema de la matemática tradicional.

*Haremos inducción por "curso de valores" en n (para φ arbitraria), probando que la propiedad se traspasa, de todos los números naturales predecesores de n, a n. ¿Por qué no inducción usual (que, de hecho, es equivalente a inducción por curso de valores), pasando de $n-1$ a n?

Paso inductivo: Aquí tenemos la siguiente

Hipótesis de Inducción: toda fórmula ψ que tiene una demostración formal a partir de $\sum$ de largo menor que n es consecuencia lógica de $\sum$, es decir, $\sum \models \psi$.

Supongamos que $\sum \vdash \varphi$, con una demostración de largo n:

φ_1
$\vdots$
$\varphi_n (\equiv \varphi)$

Hay varias posibilidades para φ_n:

- Surgió como axioma lógico en la demostración; éste es el caso básico: $\sum \models \varphi_n$, es decir, $\sum \models \varphi$.
- Surgió como premisa, es decir, $\varphi_n \in \sum$; entonces trivialmente $\sum \models \varphi_n$.
- Surgió por aplicación de MP con φ_l , φ_k con l , $k < n$.

Veamos este último caso: φ_l , φ_k están, cada una, al final de un segmento inicial propio de la demostración formal:

φ_1 $\quad$ φ_1
$\vdots$ $\quad$ $\vdots$
φ_l $\quad$ φ_k

En consecuencia, $\sum \vdash \varphi_l$ y $\sum \vdash \varphi_k$. Por hipótesis de inducción, $\sum \models \varphi_l$ y $\sum \models \varphi_k$. Además,

$\varphi_l \quad (\equiv (\varphi_k \to \varphi))$
φ_k
———————

φ

Es decir, la fórmula φ se obtiene con MP de φ_l y φ_k. Entonces basta con chequear que MP preserva la verdad con respecto a $\sum$. Más precisamente, supongamos que σ es una valuación arbitraria tal, que $\sigma \models \sum$. Entonces hay que probar que $\sigma \models \varphi$.

Se tiene:

$$\sum \models \varphi_l \quad \Rightarrow \quad \sigma \models (\varphi_k \to \varphi) \qquad (1)$$

$$\sum \models \varphi_k \quad \Rightarrow \quad \sigma \models \varphi_k \qquad (2)$$

De (1) y (2) se obtiene inmediatamente que $\sum \models \varphi$.

■

Hemos demostrado: $\sum \vdash \varphi \quad \Rightarrow \quad \sum \models \varphi$. Nos preguntamos si se puede invertir la dirección de esta implicación.

Teorema (de completidad): $\sum \models \varphi \quad \Rightarrow \quad \sum \vdash \varphi$.

No daremos la demostración*. Sin embargo, este importante teorema amerita varias observaciones:

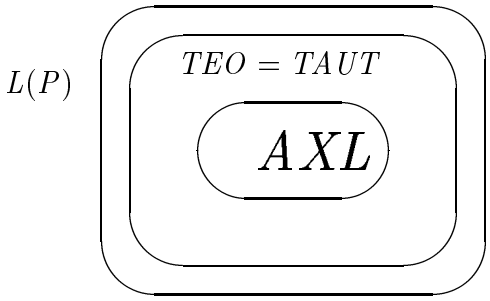
- Esta propiedad del sistema deductivo se llama "completidad", y habla del poder (la fuerza, el alcance) del sistema deductivo.
- Nos dice que toda consecuencia lógica de un conjunto de premisas (un concepto semántico) se puede demostrar formalmente a partir de las premisas (un concepto sintáctico), es decir, el sistema deductivo da cuenta por completo de la semántica.
- Nada verdadero con respecto a un conjunto de premisas queda fuera del alcance del sistema deductivo.
- El sistema formal deductivo (SFD) cubre todos los casos (de verdad con respecto a un conjunto de premisas).
- Este teorema, en combinación con el teorema de corrección, nos dice que el SFD no sólo genera todas las consecuencias semánticas de un conjunto de premisas, sino, además, solamente éstas (no genera casos erróneos).
- Los dos teoremas juntos nos dicen que hay perfecta coincidencia entre semántica y sintáxis, entre consecuencia lógica y deducción formal.

Así tenemos: $\sum \models \varphi \iff \sum \vdash \varphi$. En particular: $\models \varphi \iff \vdash \varphi$.

Luego: $TEO = TAUT$.

Con toda esta información podemos actualizar la situación anterior:

*Esta se puede encontrar el libro de Kleene [29] y, casi entera, en el libro de Reeves y Clarke [33].



Además, ahora sabemos que TEO es decidible (antes teníamos sólo r.e.).

Volvamos al sistema formal deductivo, y supongamos que $\sum \vdash \varphi$. Por definición de demostración formal, existe un subconjunto **finito** $\sum_0$ de $\sum$ tal, que $\sum_0 \vdash \varphi$. Por otro lado,

$\sum$ es inconsistente

$\iff \sum \models F$ donde F es una fórmula contradictoria (fija, por ejemplo, $\neg \forall x\ x = x$).

$\iff \sum \vdash F$

$\iff \sum_0 \vdash F$ donde $\sum_0$ finito $\subseteq \sum$ (por la finitud de las demostraciones formales)

$\iff \sum_0 \models F$

$\iff$ existe $\sum_0 \subseteq \sum$, finito e inconsistente.

Hemos demostrado:

Teorema (de finitud):* Sea $\sum \subseteq L(P)$. Se tiene: $\sum$ es insatisfacible $\iff$ existe un subconjunto **finito** $\sum_0$ de $\sum$ que es insatisfacible.

■

Este teorema semántico habría sido difícil de demostrar sin la correspondencia entre semántica y sintaxis.

*En la literatura también se le conoce como "teorema de compacidad".

Otra versión del teorema es la siguiente:

Teorema: Sea $\sum \subseteq L(P)$. Se tiene: $\sum$ es satisfacible $\iff$ todo subconjunto **finito** de $\sum$ es satisfacible.

■

Esta versión es particularmente útil para demostrar que conjuntos infinitos de fórmulas son satisfacibles.

Corolario: $\sum \models \varphi \Leftrightarrow$ existe $\sum_0 \subseteq \sum$, finito, con $\sum_0 \models \varphi$.

■

Finalmente, a propósito de completidad, recordemos que en la sección 2.8 establecimos que la regla de resolución para la lógica proposicional es completa en el siguiente sentido: para un conjunto de cláusulas $\mathcal{C}$,

$\mathcal{C}$ es inconsistente si y sólo si hay una refutación por resolución a partir de $\mathcal{C}$.

Es decir, tenemos una completidad para cierto tipo de fórmulas y cierto tipo de demostraciones, a saber, cláusulas y refutaciones, respectivamente.

6.2.3 Ejercicios

1. Demuestre la siguiente versión generalizada del teorema de corrección y completidad: $\Sigma \models \varphi$ si y sólo si $\Sigma \vdash \varphi$, usando la versión para tautologías de este teorema, es decir, $\models \varphi$ si y sólo si $\vdash \varphi$, y el teorema de compacidad.

2. Sea $\Sigma \subseteq L(P)$ un conjunto recursivamente enumerable de fórmulas proposicionales. Supon-gamos que para cada fórmula $\varphi \in L(P)$: $\Sigma \models \varphi$ o $\Sigma \models \neg\varphi$. Demuestre que el conjunto de fórmulas de $L(P)$ que son consecuencias lógicas de Σ es decidible.

3. Sea Σ un conjunto infinito de fórmulas. Esta situación no es impensable en ciencia de computación: los axiomas en Σ podrían ser generados mediante un procedimiento de enumeración recursiva o por un esquema para generar axiomas. Suponga que los axiomas en Σ son axiomas que se pueden enumerar: $\varphi_1, \varphi_2, \varphi_3, \ldots$. Demuestre que Σ es satisfacible si y sólo si para cada n,

$\{\varphi_1, \ldots, \varphi_n\}$ es satisfacible.

4. Demuestre que $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$ es satisfacible si y sólo si para infinitos n, $\varphi_1 \wedge \ldots \wedge \varphi_n$ es satisfacible.

5. Un conjunto de fórmulas Λ es un sistema de axiomas para un conjunto de fórmulas Σ si Λ y Σ son satisfechos por exactamente las mismas valuaciones de verdad. El caso más interesante es cuando Σ es infinito, pero Λ es finito. Decimos en ese caso que Σ es finitamente axiomatizable.

Supongamos que $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$, donde, para cada $n \geq 1$, $\models (\varphi_{n+1} \rightarrow \varphi_n)$ y $\not\models (\varphi_n \rightarrow \varphi_{n+1})$. Demuestre que Σ no es finitamente axiomatizable.

6. (a) Sea $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$ un conjunto de fórmulas. Demuestre que Σ es satisfacible si y sólo si para $n = 1, 2, \ldots,$ $\{\varphi_1, \ldots, \varphi_n\}$ es satisfacible.

(b) Usando la parte (a), determine si el conjunto de fórmulas $\Sigma = \{p_1 \vee p_2, \neg p_2 \vee \neg p_3, p_3 \vee p_4, \neg p_4 \vee \neg p_5, \ldots\}$ es satisfacible.

6.3 Un Sistema Deductivo para LPOP

Antes de dar el sistema formal deductivo para la la lógica de primer orden de predicados, veremos algunos ejemplos que nos permitirán ganar cierta intuición sobre las componentes que debería tener tal sistema.

Ejemplos: En los ejemplos tendremos a la teoría vacía, $\sum = \varnothing$, como teoría base.

1. Se tiene: $\models P(x) \rightarrow (Q(x) \rightarrow P(x))$.

Decimos que la fórmula $P(x) \rightarrow (Q(x) \rightarrow P(x))$ es **universalmente verdadera** (reservamos la noción de "tautología" para la lógica proposicional), en el sentido que toda interpretación $(\mathfrak{A}, \beta)$, con $\mathfrak{A} = \langle A, P^A, Q^A, \cdots \rangle$ y $\beta : \{x_1, x_2, \cdots\} \rightarrow A$, satisface la fórmula.

También decimos que $P(x) \rightarrow (Q(x) \rightarrow P(x))$ es, en la lógica de predicados, una **instancia de una tautología** de la lógica proposicional, por ejemplo, de $p \rightarrow (q \rightarrow p)$.

Podemos verificar que una interpretación arbitraria $(\mathfrak{A}, \beta)$ (compatible con el lenguaje) satisface la fórmula, es decir, que

$$(\mathfrak{A}, \beta) \models P(x) \rightarrow (Q(x) \rightarrow P(x)), \qquad (*)$$

notando que la aseveración $(*)$ es equivalente a:

$$P^A(\beta(x)) \;\Rightarrow\; (Q^A(\beta(x)) \;\Rightarrow\; P^A(\beta(x))). \quad (**)$$

Recordar que $P^A(\beta(x))$ es la interpretación de $P(x)$ en A, "$\Rightarrow$" es símbolo del metalenguaje, y $\beta(x)$ es un elemento de A.

En $(**)$ tenemos una aseveración de la forma $p \Rightarrow (q \Rightarrow p)$ que es verdadera (tal como lo sancionamos en la lógica proposicional).

2. Se tiene: $\models (P(c) \vee \neg P(c))$.

Aquí tenemos la misma situación que en el ejemplo anterior.

3. Se tiene: $\models \forall\, x(P(x) \vee \neg P(x))$.

La fórmula $\forall\, x\ (P(x) \vee \neg P(x))$ no es exactamente una instancia de tautología, pero sí una **generalización universal** de una instancia de tautología, en el sentido que hemos antepuesto cuantificadores universales a una instancia de tautología.

4. Se tiene: $\models (\forall\, xP(x) \rightarrow P(c))$.

Esta fórmula

- es universalmente verdadera,
- no es instancia de tautología,
- no es una generalización universal de instancia de tautología.

La fórmula $(\forall\, xP(x) \rightarrow P(c))$ podría axiomatizar, a nivel objeto, el "significado" del cuantificador $\forall$ o, por lo menos, parte de su "significado".

5. $\not\models \forall\, x(P(x) \rightarrow P(c))$.

Es decir, $\forall\, x(P(x) \rightarrow P(c))$ no es universalmente verdadera. Por ejemplo, $\langle \mathbb{N}, P^{\mathbb{N}}, c^{\mathbb{N}} \rangle \not\models \forall\, x(P(x) \rightarrow P(c))$, cuando $P^{\mathbb{N}} = \{0, 2, 4, \ldots\}$ y $c^{\mathbb{N}} = 3$. Sin embargo, la fórmula es satisfacible.

6. Se tiene: $\models (\forall\, xP(x) \rightarrow \forall\, yP(y))$.

7. Se tiene: $\models (\forall\, xP(x) \rightarrow \exists\, yP(y))$.

Esto se obtiene del hecho que, en esta lógica, los universos de las estructuras son no vacíos.

8. Se tiene: $\models (P(c) \rightarrow \exists\, yP(y))$.

Aquí, podríamos hacer un comentario similar al de **4.**, pero para el cuantificador existencial.

9. $\not\models (\forall x \neg P(x) \rightarrow P(c))$.

De hecho, la fórmula es contradictoria, es decir, no satisfacible.

10. $\models \forall x \; x = x$.

■

Ejercicio: Verifíquese con cuidado las aseveraciones hechas en **1.** – **10.**.

En lógica de predicados, tenemos hasta ahora sólo nociones semánticas, como la de "fórmula universalmente verdadera" o la de "consecuencia lógica", etc. Queremos presentar una contraparte sintáctica, deductiva, de estas nociones, la cual tendrá que dar cuenta de situaciones que no teníamos en lógica proposicional, por ejemplo, de lo que ocurre con las fórmulas de los ejemplos **4.**, **8.**, y **10.** anteriores. Estas fórmulas son universalmente verdaderas, pero no por herencia de la lógica proposicional, sino como producto de la interpretación de los cuantificadores, del símbolo de igualdad, en el último caso..

Como dijimos más arriba, podríamos incorporar al SFD, es decir, a nivel objeto, el "significado" de los cuantificadores (o de parte de sus "significados"), introduciendo las fórmulas en **4.** y **8.** al sistema como axiomas lógicos.

Un Sistema Formal Deductivo para un Lenguaje $L(S)$ de la LPOP:

El sistema formal deductivo para el lenguaje de primer orden $L(S)$ está formado por **axiomas lógicos** y **reglas de deducción**. Algunos de ellos deberán dar cuenta de la parte proposicional, y serán esencialmente los que ya teníamos para la lógica proposicional. Además, tendrá que haber axiomas lógicos y/o reglas que den cuenta de fórmulas universalmente válidas como **4.**, **8.** y **10.**.

Además de lo anterior, habrá que dar cuenta de la presencia del símbolo "=", que es interpretado, en forma fijada por la lógica, como la relación de igualdad. Es decir, hay que axiomatizar (especificar) la igualdad de manera formal, a nivel de lenguaje objeto, para forzarla a tener el significado subentendido.

A) Axiomas Lógicos:

Son axiomas lógicos las siguientes fórmulas:

1. Las obtenidas de los siguientes esquemas:

   (a) $(\varphi \to (\psi \to \varphi))$

   (b) $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$

   (c) $(\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$

Aquí, φ, ψ, χ denotan fórmulas arbitrarias de $L(S)$ (ya no sólo proposicionales como antes).

2. Las cuantificaciones universales (generalizaciones) de fórmulas de $L(S)$ de las formas:

   (a) $\forall\, x\varphi(x) \to \varphi(t)$

   (b) $\varphi(t) \to \exists\, x\varphi(x)$

Estos dos esquemas dan cuenta directamente de los ejemplos **4.** y **8.** anteriores. En ellos t denota un término arbitrario.

3. Axiomas para la igualdad:

Como motivación, recordemos que en el teorema de la teoría de grupos usual (no formalizada) que vimos en la sección 6.1 hicimos ciertos manejos con la igualdad –por ejemplo, reemplazo de iguales por iguales– que, de alguna forma, tienen que quedar capturados en nuestro sistema formal deductivo. Es decir, nuestro manejo lógico usual de la igualdad, como una relación especial de equivalencia con propiedad de sustitutividad, no puede quedar fuera del sistema deductivo, si es que éste quiere capturar la práctica matemática usual.

Estos son los axiomas para la igualdad:

   (a) $\forall\, x(x = x)$
   Este axioma da cuenta directamente del ejemplo **10.** de más arriba.

   (b) $\forall\, x\, \forall\, y(x = y \to y = x)$

   (c) $\forall\, x\, \forall\, y\, \forall\, z(x = y \wedge y = z \;\to\; x = z)$

   (d) Para $P \in S$, predicado n–ario, el axioma:

$$\forall\, x_1 \ldots x_n y_1 \ldots y_n\, (P(x_1, \ldots, x_n) \wedge x_1 = y_1 \wedge \cdots \wedge x_n = y_n \;\longrightarrow\; P(y_1, \ldots, y_n))$$

(e) Para cada $f \in S$, operación n–aria, el axioma:

$$\forall\, x_1 \ldots x_n\, y_1 \ldots y_n\, (x_1 = y_1 \wedge \cdots \wedge x_n = y_n \;\rightarrow\; f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)).$$

Ejemplos:

1. Si $R(\cdot,\cdot)$, $c \in S$, entonces son axiomas lógicos, entre otras, las fórmulas: $\forall\, y(\forall\, xR(x,y) \rightarrow R(c,y))$ y $\forall\, x(R(c,x) \rightarrow \exists\, yR(y,x))$.

2. Los axiomas lógicos para la igualdad dependen de S, pero tienen una manera uniforme de ser construidos a partir de S. Por ejemplo, si $S = \{R(\cdot,\cdot)\}$, entonces hay sólo un axioma de igualdad:

$$\forall\, x\, \forall\, y\, \forall\, u\, \forall\, v(R(x,y) \;\wedge\; x = u \;\wedge y = v \;\rightarrow\; R(u,v)).$$

3. Si $S = \{<, +\}$, entonces hay dos axiomas de igualdad:

$$\forall\; x\; y\; u\; v\; (x < y \;\wedge\; x = u \;\wedge\; y = v \;\rightarrow\; u < v) \quad \text{y}$$

$$\forall\; x\; y\; u\; v\; (x = u \;\wedge\; y = v \;\rightarrow\; x + y = u + v).$$

Observaciones:

- Una alternativa a los axiomas lógicos en 1. es poner como axiomas lógicos a todas las instancias, en $L(P)$, de tautologías de la lógica proposicional. Por ejemplo, si $R \in S$, podríamos poner como axioma lógico, de manera inmediata, a $\forall\, xR(x,y) \vee \neg\, \forall\, xR(x,y)$, que es instancia de $\varphi \vee \neg\varphi$.

- Los axiomas (a) – (c) en 3. dicen que "=" es una relación de equivalencia. Sin embargo, la igualdad es una relación de equivalencia especial: en todas las propiedades podemos reemplazar objetos iguales por iguales. De eso dan cuenta los "axiomas de sustitución" (d) y (e).

- A veces, para acortar las demostraciones formales, se reemplaza (d) y (e) por un "esquema de sustitución":

  $$\forall\, x_1 \ldots x_n\, y_1 \ldots y_n(\varphi(x_1, \ldots, x_n) \wedge x_1 = y_1 \wedge \cdots \wedge x_n = y_n \;\rightarrow\; \varphi(y_1, \ldots, y_n)).$$

  En él, $\varphi(x_1, \ldots, x_n)$ denota una fórmula arbitraria de $L(S)$ que contiene las variables $x_1, \ldots, x_n$. Con distintos n's y φ's obtenemos distintos axiomas lógicos.

B) Reglas de Deducción:

(a) Modus Ponens:

Esta regla es igual que en el caso proposicional:

$$\frac{\varphi \to \psi \ , \ \ \varphi}{\psi}$$

En esta regla y las que siguen, todas las fórmulas involucradas son de $L(S)$. Sin embargo, la manera de construir reglas para distintos $L(S)$'s es uniforme.

(b) Reglas de Generalización:

Nuevamente, en busca de motivación, observemos la demostración no formalizada del teorema de la teoría de grupos dado en la sección 6.1. En ella, partimos trabajando con un elemento arbitrario g, pero fijo –genérico– del grupo. Al final de la demostración llegamos a $g = e \circ g$, y la dimos por terminada satisfactoriamente. Es decir, quedo tácito (y aceptado) el hecho que, al haber demostrado la propiedad deseada para ese g genérico, teníamos establecido que $\forall x \ (e \circ x = x)$. La razón es que en las hipótesis del teorema (en los axiomas de grupo), no hay nada dicho en particular sobre el elemento g, y sin embargo, demostramos algo sobre g. En consecuencia, lo mismo podría ser demostrado para cualquier elemento del grupo. En otras palabras, podríamos generalizar lo demostrado.

Esta idea queda capturada por los axiomas siguientes:

(b1) cuando y no aparece libre en φ:

$$\frac{\varphi \to \psi(y)}{\varphi \to \forall x \psi(x)}$$

Como anticipamos, podemos explicar intuitivamente la condición "cuando y no aparece libre en φ" de la siguiente manera: si de la proposición φ, que no menciona explícitamente a y, se concluye que y tiene una propiedad ψ, entonces para cualquier individuo x se puede concluir la propiedad ψ.

(b2) cuando y no aparece libre en φ:

$$\frac{\psi(y) \to \varphi}{\exists x \psi(x) \to \varphi}$$

Esta regla se puede motivar como la anterior: si, a partir de ψ, y con un individuo y que satisface a ψ y que no aparece en φ se puede demostrar φ, entonces basta con la existencia de un individuo que satisface a ψ para demostrar φ.

La definición de **demostración formal** dentro de este SFD es como antes, con la única diferencia que ahora los axiomas lógicos y las reglas de deducción son las que acabamos de presentar.

Notemos que, tal como anunciamos al comienzo del capítulo, el sistema formal deductivo para la lógica de primer orden (en realidad, hay uno para cada lenguaje de primer orden) se puede ver como una extensión del sistema formal deductivo para la lógica proposicional.

Ejemplo: Sea $S = \{R(\cdot,\cdot)\}$, y consideremos los siguientes axiomas, no lógicos, sino propios de una teoría específica:

$$\sum = \begin{cases} Ax_1: & \forall x R(x,x) \\ Ax_2: & \forall x \ \forall y (R(x,y) \to R(y,x)) \\ Ax_3: & \forall x \ \forall y \ \forall z \ (R(x,y) \ \wedge \ R(y,z) \to R(x,z)) \end{cases}$$

Aquí, $\sum \subseteq L(\{R\})$, y contiene los axiomas para relaciones de equivalencia. Con esto queremos decir que cualquier estructura $\mathfrak{A} = \langle A, R^A, \cdots \rangle$ que satisface a $\sum$ es tal, que R^A es relación de equivalencia sobre A. También podemos decir que, por definición, una relación de equivalencia es una relación que satisface los axiomas en $\sum$.

Es fácil ver que:

$$\sum \ \models \ \forall x \ \forall y \ (\exists z (R(x,z) \ \wedge \ R(y,z)) \to R(x,y))).$$

Esto quiere decir que, como consecuencia lógica de los axiomas para relaciones de equivalencia, se obtiene la propiedad φ de que dos elementos relacionados con un tercero son equivalentes entre sí. Esta afirmación, de naturaleza semántica, se puede demostrar en la manera matemática usual: usando primero simetría, y después, transitividad:

$$(a \overset{R}{\to} c) \ \text{ y } \ (b \overset{R}{\to} c) \ \Rightarrow \ (a \overset{R}{\to} c) \ \text{ y } \ (c \overset{R}{\to} b) \ \Rightarrow \ (a \overset{R}{\to} b)$$

QED.

La proposición φ es lógicamente equivalente a:

$$\forall x \ \forall y \ \forall z \ (R(x,z) \ \wedge \ R(y,z) \rightarrow R(x,y)).$$

Veamos una demostración formal del teorema representado por la oración $\varphi : \forall x \forall y(\exists z(R(x,z) \wedge R(y,z)) \rightarrow R(x,y))$, correspondiente a la demostración semántica.

Antes de dar esa demostración conviene motivar el procedimiento general observando la demostración del estilo usual que acabamos de dar. También es útil observar la demostración del teorema de grupos de la sección 6.1: Primero se genera elementos arbitrarios y fijos (a, b, c en el caso anterior, y g en el teorema de grupos), con los cuales se trabaja hasta llegar a lo que se quiere demostrar, pero instanciado en esos elementos arbitrarios. En seguida, como esos elementos arbitrarios no aparecen explícitamente en las hipótesis del teorema, se generaliza. En la parte intermedia, cuando se está trabajando con los elementos arbitrarios, hay que arreglárselas para poder usar Modus Ponens, lo que puede requerir introducir largas fórmulas que se irán simplificando por eliminación del condicional.

Esta es la demostración formal. La primera parte genera los elementos arbitrarios, en este caso, representados por variables libres. La última parte es la generalización del resultado obtenido para las variables libres:

1. $\forall x \forall y \forall z(R(x,y) \wedge R(y,z) \rightarrow R(x,z))$

[explicación: axioma Ax_3]

2. $Ax_3 \rightarrow (R(x,z) \wedge R(z,y) \rightarrow R(x,y))$

[axioma lógico 2.(a)]

3. $R(x,z) \wedge R(z,y) \rightarrow R(x,y)$

[de 1. y 2. con regla (a)]

4. $(R(x,z) \wedge R(z,y) \rightarrow R(x,y)) \rightarrow (R(z,y) \rightarrow (R(x,z) \rightarrow R(x,y)))$

[axioma lógico; una instancia de la tautología proposicional $(p \wedge q \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$]

5. $R(z,y) \rightarrow (R(x,z) \rightarrow R(x,y))$

[de 3. y 4. con regla (a)]

6. $\forall x \forall y(R(x,y) \leftrightarrow R(y,x))$

[axioma Ax_2]

7. $Ax_2 \rightarrow (R(z,y) \leftrightarrow R(y,z))$

[axioma lógico 2.(a)]

8. $R(z,y) \leftrightarrow R(y,z)$

[de 6. y 7. con regla (a)]

9. $(R(z,y) \leftrightarrow R(y,z)) \rightarrow \{(R(z,y) \rightarrow (R(x,z) \rightarrow R(x,y))) \rightarrow (R(y,z) \rightarrow (R(x,z) \rightarrow R(x,y)))\}$

[axioma lógico; una instancia de la tautología proposicional $(p \leftrightarrow q) \rightarrow \{(p \rightarrow r) \rightarrow (q \rightarrow r)\}$]

10. $(R(z,y) \rightarrow (R(x,z) \rightarrow R(x,y))) \rightarrow (R(y,z) \rightarrow (R(x,z) \rightarrow R(x,y)))$

[de 8. y 9. con regla (a)]

11. $R(y,z) \rightarrow (R(x,z) \rightarrow R(x,y))$

[de 5. y 10. con regla (a)]

12. $(R(y,z) \rightarrow (R(x,z) \rightarrow R(x,y))) \rightarrow (R(x,z) \wedge R(y,z) \rightarrow R(x,y))$

[axioma lógico; una instancia de la tautología proposicional $(p \rightarrow (q \rightarrow r)) \rightarrow (q \wedge p \rightarrow r)$]

13. $R(x,z) \wedge R(y,z) \rightarrow R(x,y)$

[de 11. y 12. con regla (a)]

14. $\exists z(R(x,z) \wedge R(y,z)) \rightarrow R(x,y)$

[de 13. con regla (b2), notar que z no está libre en $R(x,y)$]

15. $\forall x \forall y(\exists z(R(x,z) \wedge R(y,z)) \rightarrow R(x,y))$

[de 14. con regla (b1), con φ vacía] [Fin de la demostración]

Podríamos expandir el último paso, 15., con tal de no aplicar la regla con una fórmula vacía, lo que puede parecer poco convincente. En lugar de 15. pongamos:

15'. $(\exists z(R(x,z) \wedge R(y,z)) \rightarrow R(x,y)) \rightarrow (T \rightarrow (\exists z(R(x,y) \wedge R(y,z)) \rightarrow R(x,y)))$

[instancia de la tautología proposicional $p \to (q \to p)$. T es cualquier fórmula universalmente válida, digamos, una instancia de una tautología proposicional, e.g. $R(u,u) \to R(u,u)$]

16. $T \to (\exists\, z(R(x,z) \wedge R(y,z)) \to R(x,y)$

[de 14. y 15'. con regla (a)]

17. $T \to \forall\, x \forall\, y(\exists\, z(R(x,z) \wedge R(y,z)) \to R(x,y))$

[regla (b1)]

18. T

[T fue elegido como axioma lógico]

19. $\forall\, x \forall\, y(\exists\, z(R(x,z) \wedge R(y,z)) \to R(x,y))$

[de 17. y 18. con regla (a)] [Fin de la demostración]

■

6.3.1 Algunas Propiedades del Sistema Deductivo

Es fácil verificar que la generalización universal de cualquier axioma lógico es una fórmula universalmente verdadera. En particular, también lo es la **clausura universal**, que es una generalización universal que no deja variables libres. Por ejemplo, la oración $\forall\, x\ (\forall\, yR(x,y)\ \vee\ \neg \forall\, yR(x,y))$ es la clausura universal del axioma lógico $(\forall\, yR(x,y)\ \vee\ \neg \forall\, yR(x,y))$, y es verdadera en cualquier estructura compatible con $L(\{R\})$.

Las reglas de deducción son correctas en el sentido que la clausura universal de cualquier fórmula que ha sido obtenida con una aplicación de una regla de deducción, es consecuencia lógica de la(s) fórmula(s) a la(s) cual(es) se aplicó la regla. Algunas definiciones equivalentes de la noción de corrección de una regla, ilustradas a través de la regla (b1), son las siguientes:

- La regla

$$\frac{\varphi \to \psi(y)}{\varphi \to \forall\, x\psi(x)}$$

(con la restricción sobre y) es correcta en el sentido que:

$$\models (\varphi \to \psi(y)) \quad \Rightarrow \quad \models (\varphi \to \forall\, x\psi(x)).$$

- La regla (b1) es correcta en el sentido que, para toda estructura $\mathfrak{A}$:

$$(\mathfrak{A}, \beta) \models (\varphi \to \psi(y)) \qquad \text{para toda asignación } \beta.$$

$$\Downarrow$$

$$(\mathfrak{A}, \delta) \models (\varphi \to \forall\, x \psi(x)) \qquad \text{para toda asignación } \delta.$$

La definición de corrección de una regla es más compleja en el caso de la lógica de predicados que en el de la lógica proposicional. La razón es que ahora tenemos, posiblemente, variables libres en las fórmulas*.

El lector debería verificar que las reglas (b1) y (b2), sin la restricción sobre la variable libre, dejan de ser correctas.

Debido a la validez universal de las generalizaciones de axiomas lógicos y la corrección de las reglas de deducción, también tenemos para la lógica de predicados el:

Teorema (de Corrección): Si $\sum \subseteq L(S)$ y $\varphi \in L(S)$ son un conjunto de oraciones y una oración, respectivamente, entonces:

$$\sum \vdash \varphi \quad \Rightarrow \quad \sum \models \varphi.$$

También tenemos la otra dirección:

Teorema (de Completidad de Gödel, 1930)†: Con las mismas hipótesis que en el teorema anterior:

$$\sum \models \varphi \quad \Rightarrow \quad \sum \vdash \varphi.$$

Tal como en el caso de la lógica proposicional, podemos obtener fácilmente el:

Teorema (de Compacidad): Sea $\sum \subseteq L(S)$, un conjunto de oraciones.

$\sum$ es satisfacible $\Leftrightarrow$ todo subconjunto finito $\sum_0$ de $\sum$ es satisfacible.

En capítulos que siguen veremos aplicaciones del teorema de compacidad en la demostración de la existencia de modelos para conjuntos infinitos de oraciones.

---

*Una discusión detallada de este sistema de Hilbert para la lógica de predicados, en particular, del concepto de corrección de una regla, puede ser encontrado en el libro de Kleene [29].

†No daremos la demostración de este teorema. El lector puede consultar el libro de Kleene [29].

6.3.2 Ejercicios

1. (a) Demuestre que si x no aparece libre en φ, entonces:

$\models (\forall x\psi \vee \varphi) \leftrightarrow \forall x(\psi \vee \varphi)$
$\models (\forall x\psi \wedge \varphi) \leftrightarrow \forall x(\psi \wedge \varphi)$
$\models (\exists x\psi \vee \varphi) \leftrightarrow \exists x(\psi \vee \varphi)$
$\models (\exists x\psi \wedge \varphi) \leftrightarrow \exists x(\psi \wedge \varphi)$

(b) Demuestre que se necesita la condición en x.

(c) Haga las demostraciones formales correspondientes.

2. (a) Demuestre que $\models \forall x(\psi \wedge \varphi) \leftrightarrow (\forall x\psi \wedge \forall x\varphi)$ y que $\models \exists x(\psi \vee \varphi) \leftrightarrow (\exists x\psi \vee \exists x\varphi)$

(b) Haga las demostraciones formales correspondientes.

3. Demuestre que si x no aparece libre en φ, entonces $\models \forall x(\psi \rightarrow \varphi) \leftrightarrow (\exists x\psi \rightarrow \varphi)$. Haga también la demostración formal.

4. Demuestre que $\models \neg \forall x\psi \leftrightarrow \exists x\neg\psi$ y que $\models \neg \exists x\psi \leftrightarrow \forall x\neg\psi$. Haga también la demostración formal.

5. Sea Σ el conjunto de axiomas de la teoría de grupos. Sea φ la fórmula $\forall x(e \circ x = x)$. Demuestre que $\Sigma \vdash \varphi$ dando una demostración formal en el sistema deductivo de Hilbert * (se supone que los axiomas para la igualdad son parte de los axiomas lógicos).

Ind.: Pude inspirarse en la demostración informal de $\Sigma \models \varphi$ que dimos en la sección 6.1. En el caso de este lenguaje, los axiomas para la igualdad son los mismos que se dio para relaciones de equivalencia (con "=" en lugar de "R") más el siguiente axioma de substitución: $\forall xyzw(x = y \wedge z = w \rightarrow \circ(x, z) = \circ(y, w))$.

6. Dé los axiomas lógicos para la igualdad para el lenguaje basado en el conjunto de símbolos $S = \{<, +, *, 0, 1\}$, para el álgebra con orden.

7. Demuestre que los axiomas lógicos de LPOP son lógicamente válidos y que las reglas de deducción dadas son correctas.

*En el próximo capítulo daremos una demostración basada en resolución de primer orden de otro teorema elemental de la teoría de grupos.

8. Demuestre, con un contraejemplo, que las reglas de generalización de LPOP no son correctas sin la restricción en la variable.

9. Demuestre que la "regla de inducción":

$$\frac{\varphi(t_1), \varphi(t_2), \ldots, \varphi(t_n)}{\forall\, x \varphi(x)}$$

no es correcta.

10. Demostrar que no existe ninguna oración de la lógica de predicados de primer orden que sea verdadera exactamente en las estructuras finitas, es decir, con universo finito. En otras palabras, el concepto de "finitud" no es expresable en esa lógica.

Ind.: Razone por contradicción, y use el ejercicio 4 de la sección 5.2.4 y el teorema de compacidad.

11. Demuestre que $\models \forall\, x \exists\, y(f(x) = y)$, es decir, nuestra lógica considera sólo funciones totales, es decir, definidas en todo el universo en cuestión. Haga también la demostración formal correspondiente.

6.4 Completidad y un Ejemplo de Bases de Datos

Los temas de la completidad y corrección surgen siempre que diseñamos sistemas formales o computacionales, como sistemas deductivos, programas, algoritmos, sistemas de reglas para definir objetos, etc. Se trata de probar que el sistema formal hace "todo lo que queremos que haga" (completidad) y "nada más" (corrección), cuando hay una definición precisa (no necesariamente formal) de lo que queremos que haga (entregue, especifique) el sistema. Por ejemplo, si queremos establecer que un algoritmo es correcto y completo, debemos asegurarnos, respectivamente, de que: (1) no entregue resultados erróneos para entrada alguna, y (2) entregue siempre el resultado correcto para todas las entradas (en un dominio preespecificado).

A continuación, veremos un ejemplo de un sistema formal en el área de bases de datos relacionales, más precisamente, de un sistema formal de reglas para derivar dependencias funcionales entre atributos de una base de datos relacional (BDR). Para ello, debemos comenzar con una breve introducción a algunos conceptos básicos del área.

Consideremos una modelación conceptual de (una parte de) la realidad a través de entidades, relaciones y atributos. La figura A6.1 muestra un modelo Entidad/Relación:

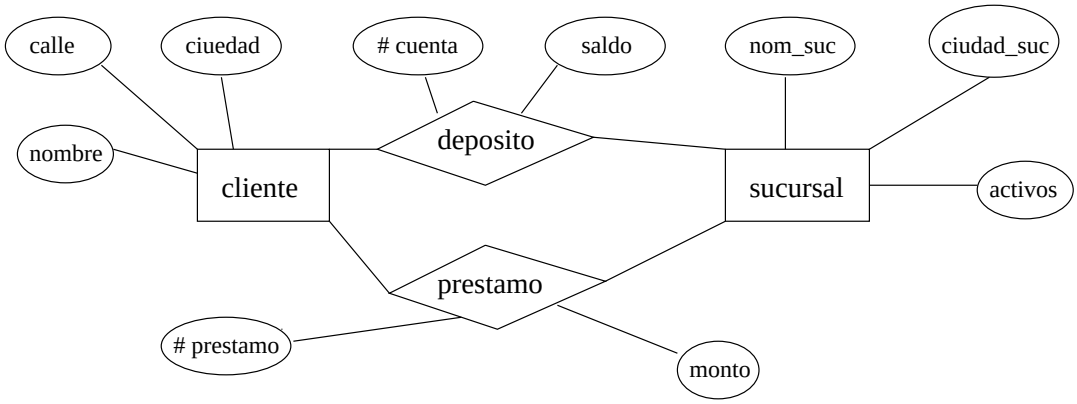


Figura A6.1: Un modelo Entidad/Relación.

A partir de este modelo E/R se puede crear un modelo relacional definiendo las relaciones a través del siguiente esquema de la BD:

SUC(nom–suc,activos,ciudad–suc)

CLIENTE(nombre,calle,ciudad–cl)

DEP(nom–suc,# cuenta,nombre,saldo)

PREST(nom–suc,# préstamo, nombre,monto)

En la práctica, estas relaciones van a estar realizadas, concretadas, a través de tablas de tuplas de datos. Estas realizaciones o extensiones de las relaciones irán variando en el tiempo de acuerdo con las transacciones y actualizaciones. Sin embargo, hay ciertos hechos que van a persistir a pesar de las actualizaciones, por ejemplo, ciertas **dependencias funcionales**, es decir las dependencias de ciertos atributos como funciones de otros atributos en el esquema de la base de datos.

En nuestro ejemplo tenemos las siguientes dependencias funcionales

nom–suc $\longrightarrow$ activos , ciudad–suc

nombre $\longrightarrow$ calle , ciudad–cl

$$\#\text{ cuenta} \longrightarrow \text{saldo , nombre–suc}$$

$$\#\text{ préstamo} \longrightarrow \text{monto , nombre–suc}$$

Esta notación indica que el (los) atributo(s) de la derecha depende(n), como función, del atributo de la izquierda. Usamos aquí, como es usual en matemática, el símbolo "$\longrightarrow$" para denotar funciones. En principio, no tiene nada que ver con el símbolo de implicación que hemos estado usando hasta ahora.

Por ejemplo, no puede haber dos tuplas en una tabla correspondiente a la relación **DEP** con el mismo " # cuenta" y distintos valores para los atributos "saldo" y/o "nombre sucursal". Estas dependencias funcionales son casos especiales de **restricciones de integridad**, que son restricciones que se deben satisfacer independientemente de los contenidos concretos y actuales (extensiones) de las tablas. Ellas sirven, entre otras cosas, para verificar consistencia de los datos.

Dado un esquema de BD, las realizaciones (conjuntos de tablas en correspondencia con el esquema), es decir, las bases de datos extensionales, pueden "satisfacer" (hacer verdaderas) o no las dependencias funcionales. Puede haber ciertas dependencias funcionales que son "consecuencia" de otras dependencias funcionales. A ambas nociones, "satisfacción de una restricción de integridad" y "consecuencia", se les puede dar un sentido semántico preciso tal como en la lógica de predicados. Esto se debe a que, como hemos tratado de enfatizar en secciones anteriores, podemos ver a las bases de datos extensionales como estructuras de la lógica de predicados. Las dependencias funcionales pueden ser vistas como ciertas expresiones formales del tipo $X \longrightarrow Y \cdots Z$, donde $X, Y, \cdots, Z$ denotan conjuntos de atributos.

Estamos tal como en lógica de predicados en su parte semántica. De hecho, podemos usar la notación:

$$\underbrace{X \to Y}_{\text{una DF}}, \cdots, \; Z \to U \;\models\; L \to M.$$

Esto significa que toda BD extensional concreta (conjunto de tablas) que satisface las dependencias funcionales $X \to Y$, $\cdots$, $Z \to U$, también satisface la dependencia funcional $L \to M$.

Una pregunta que nos surge naturalmente es la siguiente: ¿Dado un conjunto de DF's, qué otras DF's son consecuencia de ellas? Armstrong (1974) propuso un sistema formal de reglas para derivar DF's a partir de otras.

Reglas de Armstrong:

- $X \to X$ (se puede ver como un axioma lógico)
- $\underline{X \to Y}$

  $XZ \to Y$
- $X \to Y$

  $\underline{X \to Z}$

  $X \to YZ$
- $\underline{X \to YZ}$

  $X \to Y$
- $X \to Y$

  $\underline{Y \to Z}$

  $X \to Z$
- $X \to Y$

  $\underline{YZ \to W}$

  $XZ \to W$

Estas reglas son puramente formales y permiten derivar mecánicamente DF's a partir de otras dadas. Podemos usar, como en la lógica formal, el símbolo "$\vdash$" para denotar derivación formal. De hecho, hay algoritmos establecidos para determinar "todas" las DF's provenientes de un conjunto dado de DF's. ¿Qué significa "todas"? Las DF's fueron definidas en forma semántica, y también las nociones de satisfacción y consecuencia. Como las reglas de Amstrong y su uso son formales, sintácticos, corresponde entonces preguntarse por la corrección y completidad del sistema de reglas de derivación de DF's de Amstrong. Se tiene:

El sistema es correcto: Toda DF $X \to Y$ derivable con las reglas de Amstrong de un conjunto de DF's $\sum$ es también consecuencia semántica de $\sum$:

$$\sum \vdash X \to Y \quad \Rightarrow \quad \sum \models X \to Y.$$

El sistema es completo:

$$\sum \models X \to Y \quad \Rightarrow \quad \sum \vdash X \to Y\,.$$

Este teorema es de Armstrong y Fagin, y se puede demostrar "reduciendo" este sistema de DF's y reglas de derivación a una porción de la lógica proposicional con un sistema formal deductivo que resulta ser completo.

6.4.1 Ejercicios

1. Derive la siguiente regla para obtener dependencias funcionales en bases de datos relacionales usando las reglas de Armstrong:

$$\frac{Y \subseteq X}{X \to Y}$$

2. Considere la tabla:

$\underline{ABCD}$
$a_1\, b_1\, c_1\, d_1$
$a_2\, b_2\, c_1\, d_1$
$a_1\, b_1\, c_1\, d_2$
$a_3\, b_3\, c_2\, d_3$

Esta tabla satisface la dependencia funcional: $A \to B$. Demuestre usando los axiomas de Armstrong que también satisface las DF's: $AB \to B$, $AC \to B$, $AD \to B$, $ABC \to B$, $ABD \to B$, $ACD \to B$, $ABCD \to B$.

6.5 Indecidibilidad de la Lógica de Predicados

Hasta ahora hemos visto un desarrollo paralelo de la lógica proposicional y la lógica de predicados. Todos los teoremas que teníamos para la primera los hemos encontrado también en la segunda. Por ejemplo, en ambas lógicas tenemos el siguiente caso particular del teorema de corrección y completidad: $\models \varphi \quad \Leftrightarrow \quad \vdash \varphi$, es decir, una fórmula φ es universalmente verdadera si y sólo si es un teorema de la lógica.

En el caso proposicional hay, además, un algoritmo para testear validez universal de una fórmula arbitraria φ. También hay un algoritmo para testear

satisfacibilidad de fórmulas. Nos preguntamos si existen tales algoritmos para la LPOP.

En el caso de una fórmula proposicional, para chequear su satisfacibilidad o validez universal, bastaba con examinar exhaustivamente todas las posibles valuaciones, y había una cantidad finita de ellas. Este algoritmo no tiene una extensión natural al caso de la LPOP: tendríamos que chequear una cantidad infinita de interpretaciones, de hecho, infinitas con universos infinitos.

Fijemos el lenguaje de la lógica de predicados $L(S)$, y consideremos el siguiente:

Problema: ¿Hay algún algoritmo para decidir si una fórmula arbitraria de $L(S)$ es universalmente válida? Es decir, ¿es el siguiente problema decidible?

$$VU := \{\varphi \in L(S) \mid \models \varphi\}.$$

Este problema es relativo a un conjunto de símbolos, sin embargo, esto no es muy importante. Podríamos fijar el lenguaje, eligiendo $S = \{R_1, \ldots, f_1, \ldots, c_1, \ldots\}$.

Sabemos que el problema de validez universal para la lógica proposicional es siempre decidible, aunque difícil. Sin embargo, para la lógica de predicados la respuesta a la pregunta planteada es negativa: VU es indecidible (Alonso Church, Alan Turing, $\sim$ 1936).

- Este es el llamado "Teorema de Indecibilidad de la Lógica de Predicados" de Church.
- Excepto para conjuntos de símbolos S muy simples, el problema es siempre indecidible. Basta con que en S haya un predicado binario.
- El teorema de Church es muy importante y marca una gran diferencia cualitativa entre lógica proposicional y lógica de predicados. El teorema nos dice que la segunda tiene limitaciones computacionales intrínsecas. $TAUT$, el problema correspondiente de la lógica proposicional, es decidible (pero difícil).
- Por "reducción" se obtiene que también el problema de satisfacibilidad

  $$SAT := \{\varphi \in L(S) \mid \varphi \text{ es satisfacible}\}$$

  es indecidible.

A pesar de estas dificultades computacionales intrínsecas de la lógica proposicional, no está todo perdido. Por un lado, sabemos que hay una contraparte sintáctica, formal, de la noción de consecuencia lógica y de validez universal. Por otro, si nos restringimos a fórmulas de cierto tipo sintáctico, se puede conseguir que los dos problemas de decisión anteriores sean solubles.

Es natural preguntarse por qué se da esta diferencia tan grande entre ambas lógicas. Intuitivamente, la diferencia se produce por la mayor expresividad de los lenguajes de la lógica de predicados. En lo que sigue, queremos precisar esta intuición.

Es posible describir en un lenguaje de la LPOP el funcionamiento de cualquier máquina de Turing (MT). La descripción es tal, que, a cada MT $\mathcal{M}$ se puede asociar una **oración** $\varphi_{\mathcal{M}}$ de la lógica de predicados con la siguiente propiedad:

$$\mathcal{M} \text{ para (con la cinta vacía)} \iff \models \varphi_{\mathcal{M}}.$$

La oración $\varphi_{\mathcal{M}}$ describe el funcionamiento de $\mathcal{M}$.

Lo que tenemos entonces es una "reducción" del problema de parada de una máquina de Turing al problema de validez universal de una fórmula de la lógica de predicados. Esta reducción nos permite obtener de inmediato la indecidibilidad de la validez universal, ya que, como sabemos, el problema de la parada (HP) es indecidible:

$$\text{HP} \leq \{\varphi \in L(S) \mid \varphi \text{ es oración y } \models \varphi\} =: \text{VU}$$

$$\Downarrow$$

$$\text{HP es indecidible} \implies \text{VU también es indecidible.}$$

En contraste, si recordamos el teorema de Cook, en lógica proposicional podíamos describir el funcionamiento de MT's con tiempo de ejecución acotado a priori por un polinomio fijo, evaluado en el largo de la entrada. La mayor expresividad de los lenguajes de la lógica de predicados nos permite describir el funcionamiento de máquinas de Turing, sin restricción en el tiempo de ejecución, que de hecho, puede ser infinito.

Vamos a considerar ahora la enumerabilidad recursiva de los problemas SAT y VU. Recordemos que en lógica proposicional, los problemas "$\models \varphi$" y SAT

son recursivamente enumerables (r.e). Esto se obtuvo del hecho general de que todo problema decidible es recursivamente enumerable. Como para la lógica de predicados no tenemos la decidibilidad de los problemas SAT y VU, no podemos concluir, como antes, que ellos son r.e., podrían serlo o no.

¿Hay un algoritmo que genere todas las fórmulas de $L(S)$ que son universalmente válidas?

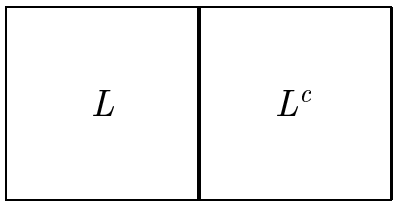
Como $\models \varphi \iff \vdash \varphi$, basta con ir aplicando "sistemáticamente" todos los axiomas lógicos y reglas de deducción para ir obteniendo fórmulas universalmente válidas. En consecuencia, VU es recursivamente enumerable. De aquí se obtiene, además, que SAT^c, es decir, el problema de decidir si una fórmula no es satisfacible, también es recursivamente enumerable (pero obviamente no decidible).

De paso obtenemos un ejemplo de un problema no decidible, pero recursivamente enumerable. Otro ejemplo es HP, el "halting problem" o problema de parada de una máquina de Turing.

Nos falta responder la pregunta por la enumerabilidad recursiva de SAT, el problema de satisfacibilidad de fórmulas de la lógica de predicados.

El problema de satisfacibilidad no es recursivamente enumerable. Este resultado se puede obtener fácilmente del siguiente

Teorema (de S. Kleene): Sean L y L^c un problema de decisión y su complemento con respecto a un dominio de instancias D, respectivamente.



D (dominio de instancias)

Entonces: L y L^c r.e. $\implies$ L y L^c decidibles.

Demostración: Necesitamos un algoritmo para decidir L (la decidibilidad del complemento se obtiene entonces de inmediato). La estrategia es el procesamiento en paralelo.

Sea $w \in D$. Queremos decidir si $w \in L$ o no. Para ello, echamos a funcionar los algoritmos para enumeración recursiva de L y L^c por separado:

$\underline{L}$	$\underline{L^c}$
w_1	w_1'
w_2	w_2'
w_3	w_3'
$\vdots$	$\vdots$

Después de un tiempo finito, w va a aparecer en una y sólo una de las listas. Según dónde aparezca, respondemos SI o NO.

■

Como corolario del teorema de Kleene obtenemos que SAT no es r.e.: si SAT fuera r.e., entonces, dado que SAT^c es recursivamente enumerable, tendríamos que SAT es decidible, lo que no es posible.

6.5.1 Ejercicios

1. Demuestre detalladamente que el problema SAT^c es recursivamente enumerable.

2. Demuestre que el problema $\{\Sigma \mid \Sigma \subseteq L_0(S)$ y Σ es consistente $\}$ es indecidible. Es decir, el problema de decidir la consistencia de conjuntos de axiomas de primer orden no es soluble computacionalmente.

6.6 Resolución de Primer Orden

En la sección 2.8 presentamos la regla de resolución proposicional. En esta sección presentaremos la regla de resolución de primer orden, que será usada en el capítulo 9. En estricto rigor, el teorema de completidad de Gödel sobre el sistema formal de la lógica de predicados de primer orden nos asegura que esta regla no es necesaria para hacer demostraciones formales. Sin embargo, su introducción en ciencia de computación, por parte de J.A. Robinson en 1965, cambió radicalmente el uso de la lógica formal en sistemas computacionales, como mecanismo deductivo, al hacerlo más claro, fácil y eficiente.

En lugar de cláusulas proposicionales, ahora trabajaremos con cláusulas de la forma $L_1(\bar{x}) \vee \cdots \vee L_n(\bar{x})$, donde los literales $L_i(\bar{x})$ son, ahora, fórmulas atómicas cualesquiera de la lógica de predicados o negaciones de ellas. Como antes, el caso $n = 0$ nos da la cláusula vacía, $\Box$. Ademas, se subentiende que

las variables $\bar{x}$ en la cláusula están cuantificadas universalmente al inicio de la cláusula.

Con el siguiente ejemplo veremos que la aparición de variables nos obliga a introducir un elemento adicional a la regla que ya teníamos en el caso proposicional.

Ejemplo: Dado el lenguaje $L(\{P(\cdot), Q(\cdot), R(\cdot), S(\cdot), f(\cdot), c\})$, supongamos que tenemos las cláusulas $P(x) \vee Q(y) \vee S(x)$ y $\neg P(f(z)) \vee R(c)$ (por la cuantificación universal tácita de variables podemos suponer que las cláusulas no tienen variables en común). Para poder resolver estas dos cláusulas por medio de los literales $P(x)$ y $\neg P(f(z))$, primero tenemos que dar valores a las variables de modo que se conviertan en literales (exactamente) complementarios. Estos "valores" pueden, en realidad, ser términos cualesquiera, en particular, pueden contener variables libres. Una posibilidad es dar a x el valor $f(z)$ (también la cuantificación universal nos permite hacer esto), con lo cual nos quedamos con las cláusulas $P(f(z)) \vee Q(y) \vee S(f(z))$ y $\neg P(f(z)) \vee R(c)$. Ahora, podemos resolver y obtenemos la cláusula resolvente $Q(y) \vee S(f(z)) \vee R(c)$, en la cual las variables también están cuantificadas universalmente. Esta cláusula puede ser usada en nuevas resoluciones.

Notar que podríamos dar a las variables otros valores que también permitirían la resolución, por ejemplo, podríamos dar a x el valor $f(c)$ y a z el valor c, con lo cual las cláusulas a resolver serían $P(f(c)) \vee Q(y) \vee S(f(c))$ y $\neg P(f(c)) \vee R(c)$. La resolvente en este caso es $Q(y) \vee S(f(c)) \vee R(c)$. Notemos que esta cláusula es menos apta para hacer nuevas resoluciones, ya que tiene menos variables libres que la resolvente anterior. En consecuencia, lo ideal es ser lo menos restrictivo (o más general) posible al dar valores a las variables para hacer la resolución. Este requerimiento será puesto como parte de la regla.

■

El proceso de dar valores a las variables para conseguir literales complementarios, se llama "unificación". Este concepto puede ser aplicado en cualquier contexto en que se quiera hacer idénticas dos o más expresiones formales que involucren variables, por medio de la asignación de valores a las variables. Nosotros querremos la unificación más general.

Más precisamente, un **unificador** para un conjunto de expresiones $\mathcal{E} = \{E_1, \ldots, E_n\}$ que involucran variables y simbolos de un conjunto S es una función $\theta : Var \to T(S)$, del conjunto de variables a los términos del lenguaje tal, que $E_1\theta \equiv \cdots \equiv E_n\theta$. Aquí, $E_i\theta$ denota la expresión obtenida de E_i al reemplazar **simultáneamente** en ella las apariciones de variables por sus valores según θ; y

"$\equiv$" entre dos expresiones simbólicas significa que éstas coinciden exactamente como sucesiones de caracteres.

Ahora, un **unificador más general** (umg) θ para el conjunto de expresiones $\mathcal{E}$ es un unificador para $\mathcal{E}$ tal, que el efecto de cualquier otro unificador σ sobre $\mathcal{E}$ se puede reproducir aplicando primero θ, y aplicando, en seguida, otra sustitución, digamos γ. Es decir, $\mathcal{E}\sigma = \mathcal{E}\theta\gamma$. La aplicación de la segunda substitución da cuenta de la posible "sobrevaloración" de las variables que se mencionó más arriba, y el unificador más general, aplicado primero, da cuenta de las mínimas asignaciones de valores a las variables para que las expresiones se hagan sintácticamente iguales.

Presentando las cláusulas en notación conjuntista, la **regla de resolución** se ve ahora de la siguiente forma:

$$\frac{\{L_1, \cdots, L_n\} \qquad \{L'_1, \cdots, L'_m\}}{(\{L_1, \cdots, L_n\}\theta - \{L_n\theta\}) \cup (\{L'_1, \cdots, L'_m\}\theta - \{L'_m\theta\})}$$

donde $L_n\theta$ y $L'_m\theta$ son literales complementarios, y, en consecuencia, θ es un unificador de las fórmulas atómicas correspondientes a L_n y L'_m. En esta regla exigimos, además, que θ sea un unificador más general.

La aplicación de θ en la resolvente corresponde a la reasignación de valores a las variables co-rrespondiente a la unificación que se hizo de L_n y L'_m.

Por último, es natural preguntarse cuándo puede uno razonar a partir de cláusulas de primer orden. Recordemos que en lógica proposicional, siempre es posible llevar una base de conocimiento escrita con fórmulas arbitrarias a una escrita en términos de cláusulas proposicionales, de tal modo, que la segunda es lógicamente equivalente a la original. De hecho, ya discutimos que basta con que ambos conjuntos de fórmulas sean equiconsistentes.

En el caso en que tenemos una base de conocimiento escrita con oraciones de primer orden arbitrarias*, también es posible pasar, de manera mecánica, a una base de conocimiento escrita con cláusulas de primer orden como las que ya vimos en esta sección. La diferencia con el caso proposicional es que

*Es poco interesante el caso en que hay variables libres en las fórmulas, así es que supondremos que la base de conocimiento contiene sólo oraciones.

no tenemos la seguridad de obtener conjuntos de fórmulas lógicamente equivalentes, pero sí, equiconsistentes. Ya sabemos que esto es suficiente, pues nuestro propósito al usar resolución es determinar inconsistencia, la cual, si se da, se preserva en la transformación del conjunto original.

El método consiste en eliminar los cuantificadores, quedándose con fórmulas de apariencia proposicional, que después podemos llevar a cláusulas a través de la forma normal conjuntiva. El método es como sigue:

- Cuando se tiene una fórmula de la forma $\forall x_1 \ldots x_n \varphi$, donde φ no tiene cuantificadores, se elimina los cuantificadores universales, dejando las variables correspondientes aparentemente libres. En realidad, se subentiende que están universalmente cuantificadas al principio de la cláusula, todas las variables que se vean libres.

  Por ejemplo, la fórmula $\forall x \forall y\ (P(x) \vee (Q(y) \wedge \neg P(y)))$ se pasa a $(P(x) \vee (Q(y) \wedge \neg P(y)))$. Se entiende que las variables x e y están universalmente cuantificadas. Después podemos pasar a las cláusulas: $(P(x) \vee Q(y))$ y $(Q(x) \vee \neg P(y))$, con la misma convención sobre las variables.

- Una fómula de la forma $\exists x\ \varphi(x)$ se transforma en $\varphi(c)$, donde c es una **nueva** constante individual (nuevo nombre), que no aparece en ninguna otra de las fórmulas. Ese nombre nuevo es un testigo de la existencia del x, pero descomprometido con respecto a tomar un valor conocido. Obviamente las dos fórmulas son equiconsistentes, pero no necesariamente lógicamente equivalentes.

  Por ejemplo, la fórmula $\exists x\ P(x, y)$ se transforma en $P(d, y)$, donde d es un nombre nuevo.

- Más generalmente, una fórmula de la forma $\forall x \exists y\ \varphi(x, y)$ se transforma en $\varphi(x, f(x))$, donde f es un nuevo símbolo de operación, ya que, primero, el y depende en principio del x del cuantificador universal, y segundo, hay que dejar libre ese x. Es decir, pasamos primero a $\forall x \varphi(x, f(x))$, y despue's, a $\varphi(x, f(x))$.

  Este proceso de introducción de nuevas constantes y funciones para eliminar cuantificadores existenciales se llama "skolemización" (por el lógico Thoraf Skolem).

  Por ejemplo, la fórmula $\forall x \forall y \exists z\ R(x, y, z)$ se transforma en $R(x, y, g(x, y))$, donde g es un nuevo símbolo de función binaria.

Con este método para eliminar cuantificadores, más el paso a forma clausal vía la forma normal conjuntiva, siempre podemos pasar de un conjunto de

fórmulas de primer orden arbitrario, a uno formado por cláusulas de primer orden. Sin embargo, como se vio en el primer item en el método de eliminación de cuantificadores, antes de aplicar el método de skolemización, es necesario tener las fórmulas escritas en la llamada "forma normal prenex", bajo la cual, la fórmula empieza con un prefijo de cuantificadores (de cualquiera de los dos tipos), seguido de una fórmula sin cuantificadores (en consecuencia, con apariencia proposicional). Por ejemplo, la fórmula $\exists x \forall y \exists z\ (P(x) \vee \neg Q(y) \vee R(z))$ está en forma normal prenex, pero no lo está la fórmula $\exists x \forall y (P(x) \vee \neg Q(y) \vee \exists z R(z))$.

Toda fórmula de primer orden puede ser llevada a una fórmula en forma normal prenex que es lógicamente equivalente a la original. Esto se puede hacer de manera mecánica con un algoritmo basado en los resultados de los primeros ejercicios de la sección 6.3.2. Para esto puede ser necesario renombrar variables.

A continuación demostraremos un teorema de la teoría de grupos mediante resolución de primer orden.

Ejemplo: Consideremos los axiomas de la teoría de grupos
$\forall x \forall y \forall z\ x \circ (y \circ z) = (x \circ y) \circ z$,
$\forall x \exists y\ y \circ x = e$
$\forall x\ e \circ x = x$.
Queremos usar resolución de primer orden para probar el siguiente teorema de cancelación:

$$\forall x \forall y \forall z\ x \circ y = x \circ z\ \rightarrow\ y = z.$$

Primero, tenemos que skolemizar la teoría e incluir los axiomas de igualdad, ya que el símbolo de igualdad aparece como tal. Se tiene los siguientes axiomas:

$$x \circ (y \circ z) = (x \circ y) \circ z \qquad (1)$$

$$f(x) \circ x = e \qquad (2)$$

$$e \circ x = x \qquad (3)$$

$$\neg x = y\ \vee\ \neg y = z\ \vee\ x = z \qquad (4)$$

$$\neg x = y\ \vee\ y = x \qquad (5)$$

$$x = x \qquad (6)$$

$$\neg x = y\ \vee\ \neg z = w\ \vee\ x \circ z = y \circ w \qquad (7)$$

$$\neg x = y\ \vee\ f(x) = f(y) \qquad (8)$$

Los axioms (1) – (3) son las versiones clausales de los axiomas originales; (4) – (7) corresponden a los de transitividad, simetría, reflexividad para la igualdad,

y de funcionalidad (o de sustitución) para $\circ$, respectivamente. El axioma (8) es el axioma de funcionalidad para la nueva función de Skolem, f.

Ahora, hay que skolemizar la negación del teorema a demostrar. Los siguientes son los pasos de la skolemización:

$$\neg \forall x \forall y \forall z \; \neg x \circ y = x \circ z \;\vee\; y = z$$

$$\exists x \exists y \exists z \; \neg(\neg x \circ y = x \circ z \;\vee\; y = z)$$

$$\neg(\neg a \circ b = a \circ c \;\vee\; b = c)$$

$$a \circ b = a \circ c \;\wedge\; \neg b = c$$

De esta última fórmula obtenemos las dos cláusulas:

$$a \circ b = a \circ c \qquad (9)$$

$$\neg b = c \qquad (10)$$

Hemos introducidos nuevas constantes de Skolem, a, b, c. A partir de esta base de conocimiento debemos producir una refutación por resolución. Hay que recordar que, después de hacer una resolución, conviene hacer un renombre de variables, introduciendo variables frescas, para poder así hacer más fácil y claramente las resoluciones posteriores *.

Antes de seguir con la refutación, se da como ejercicio al lector el dar una demostración usual, la que daría en un curso de álgebra, ya que, a pesar de los detalles formales, sintácticos, de una refutación formal, los pasos de una demostración usual por contradicción están presentes en esta demostración formal, y, de hecho, nos han ayudado a guiarla.

Partiremos resolviendo (4) con (10), con las sustituciones $\frac{x}{b}, \frac{z}{c}$. (A veces, indicaremos las substituciones usadas a la derecha de la fórmula resultante de la resolución (la resolvente). En las substituciones así indicadas, la variable aparece arriba de la raya de fracción, y el término asignado a ella, abajo.)

$$\neg x = y \;\vee\; y = z \;\vee\; x = z \qquad (4)$$
$$\neg b = c \qquad (10)$$

$$\neg b = y \;\vee\; \neg y = c \qquad (11) \;\; \frac{x}{b}, \frac{z}{c}$$

---

*En realidad, este renombre se puede hacer cuando se quiera, pues las variables están universalmente cuantificadas delante de las cláusulas.

Por otro lado,

$$\neg u = y \ \vee \ y = u \qquad (5)$$

$$e \circ x = x \qquad (3)$$

$$x = e \circ x \qquad (12) \qquad \frac{u}{e \circ x}, \frac{y}{x}$$

$$\neg b = y \ \vee \ \neg y = c \qquad (11)$$

$$\neg e \circ b = c \qquad (13) \quad \frac{x}{b}, \frac{y}{e \circ b}$$

$$\neg x = y \ \vee \ \neg y = z \ \vee \ x = z \qquad (4)$$

$$\neg e \circ b = y \ \vee \ \neg y = c \qquad (14) \quad \frac{x}{e \circ b}, \frac{z}{c}$$

$$e \circ x = x \qquad (3)$$

$$\neg e \circ b = e \circ c \qquad (15) \quad \frac{x}{c}, \frac{y}{e \circ c}$$

$$\neg x = y \ \vee \ y = z \ \vee \ x = z \qquad (4)$$

$$\neg e \circ b = y \ \vee \ \neg y = e \circ c \qquad (16) \quad \frac{x}{e \circ b}, \frac{z}{e \circ c}$$

Por otro lado,

$$\neg x = y \ \vee \ \neg z = w \ \vee \ x \circ z = y \circ w \qquad (7)$$

$$z_1 = z_1 \qquad (6)$$

$$\neg x = y \ \vee \ x \circ z_1 = y \circ z_1 \qquad (17) \quad \frac{w}{z_1}, \frac{z}{z_1}$$

$$f(x_1) \circ x_1 = e \qquad (2)$$

$$(f(x_1) \circ x_1) \circ z_1 = e \circ z_1 \qquad (18) \quad \frac{x}{f(x_1) \circ x_1}, \frac{y}{e}$$

$$\neg x = y \ \vee \ y = x \qquad (5)$$

$$e \circ z_1 = (f(x_1) \circ x_1) \circ z_1 \qquad (19) \quad \frac{x}{f(x_1) \circ x_1) \circ z_1}, \frac{y}{e \circ z_1}$$

$$\neg e \circ b = y \ \vee \ \neg y = e \circ c \qquad (16)$$

$$\neg (f(x_1) \circ x_1) \circ b = e \circ c \qquad (20) \quad \frac{z_1}{b}, \frac{y}{f(x_1) \circ x_1) \circ b}$$

$$\neg x = y \ \vee \ y = z \ \vee \ x = z \qquad (4)$$

$$\neg(f(x_1) \circ x_1) \circ b = y \ \vee \ \neg y = e \circ c \quad (21)$$
$$(f(y_1) \circ y_1) \circ z_1 = e \circ z_1 \qquad (18)$$

$$\neg(f(x_1) \circ x_1) \circ b = (f(y_1) \circ y_1) \circ c \quad (22) \ \frac{z_1}{c}$$
$$\neg x = y \ \vee \ y = z \ \vee \ x = z \qquad (4)$$

$$\neg(f(x_1) \circ x_1) \circ b = y \ \vee \ \neg y = (f(y_1) \circ y_1) \circ c \qquad (23)$$

Por otro lado,

$$\neg x = y \ \vee \ y = x \qquad (5)$$
$$x \circ (y \circ z) = (x \circ y) \circ z \qquad (1)$$

$$(x \circ y) \circ z = x \circ (y \circ z) \qquad (24)$$
$$\neg(f(x_1) \circ x_1) \circ b = y_2 \ \vee \ \neg y_2 = (f(y_1) \circ y_1) \circ c \quad (23)$$

$$\neg f(x_1) \circ (x_1 \circ b) = (f(y_1) \circ y_1) \circ c \qquad (25)$$
$$\neg x = y \ \vee \ y = z \ \vee \ x = z \qquad (4)$$

$$\neg f(x_1) \circ (x_1 \circ b) = y \ \vee \neg y = (f(y_1) \circ y_1) \circ c \qquad (26)$$
$$x \circ (y_2 \circ z) = (x \circ y_2) \circ z \qquad (1)$$

$$\neg f(x_1) \circ (x_1 \circ b) = f(y_1) \circ (y_1 \circ c) \quad (27)$$

Por otro lado,

$$x_1 = x_1 \qquad (6)$$
$$\neg x = y \ \vee \ \neg z = w \ \vee \ x \circ z = y \circ w \qquad (7)$$

$$\neg z = w \ \vee \ x_1 \circ z = x_1 \circ w \qquad (28)$$
$$\neg f(x_2) \circ (x_2 \circ b) = f(y_1) \circ (y_1 \circ c) \quad (27)$$

$$\neg x_2 \circ b = x_2 \circ c \qquad (29) \qquad \frac{x_1}{f(x_2)}, \frac{z}{x_2 \circ b}, \frac{y_1}{x_2}, \frac{w}{x_2 \circ c}$$
$$a \circ b = a \circ c \qquad (9)$$

□

6.6.1 Ejercicios

1. ¿Por qué es necesario, al aplicar una substitución (de variables por términos), hacer el reemplazo **simultáneo** de las variables? Ilustre con un ejemplo.

2. Dé ejemplos de unificadores más generales y de cómo otros unificadores se pueden "dividir" por un unificador más general. ¿Por qué se puede pensar en división?.

3. Dé un algoritmo general para pasar una fórmula arbitraria de primer orden a un conjunto de cláusulas equiconsistente con ella. Para ello, primero, dé un algoritmo para pasar una fórmula a forma normal prenex, depués, uno de skolemización, y, por último, el de aplicación de la forma normal conjuntiva.

4. Dé un conjunto de cláusulas equiconsistente con la fórmula $\forall x \exists y \neg \forall z (P(x) \vee (Q(z) \wedge K(u) \wedge \neg \exists u (\neg T(u) \vee S(y))))$.

Ind.: Puede ser necesario renombrar variables de modo de satisfacer las restricciones sobre la variable que aparece en los ejercicios en la sección 6.3.2.

5. Dados los axiomas de la teoría de grupos: $\forall x \forall y \forall z \; x \circ (y \circ z) = (x \circ y) \circ z$, $\forall x \; x \circ e = x$, $\forall x \exists y \; x \circ y = e$. Demuestre formalmente mediante mediante resolución de primer orden los teoremas $\forall x \; e \circ x = e$ y $\forall x \exists y \; y \circ x = e$.

Ind.: Dé demostraciones matemáticas usuales de los teoremas, e inspírese en ellas para reproducirlas como refutaciones por resolución.

Capítulo 7

Teorías

¿Qué es una teoría? Intuitivamente, una teoría es un cúmulo de información expresada a través de aseveraciones que es libre de contradicciones y cerrada con respecto a conclusiones que de ella salen (lo que se pude deducir de ella ya está considerado dentro). Podemos modelar esta noción por medio de la lógica de predicados*.

Definición: Un conjunto de oraciones $\sum \subseteq L(S)$ es una **teoría** si y sólo si (1) es consistente (es decir, tiene un modelo o no se puede derivar contradicciones a partir de ella), y (2) es cerrado con respecto a consecuencia lógica (o, equivalentemente, derivabilidad), es decir, para toda oración φ, $\sum \models \varphi \quad \Rightarrow \quad \varphi \in \sum$.

A continuación veremos distintas maneras de obtener teorías. Como en esta parte estaremos considerando primordialmente oraciones, vamos a denotar al conjunto de oraciones del lenguaje $L(S)$ por medio de $L_0(S)$ †.

1. Mediante un sistema de axiomas (no necesariamente lógicos) o premisas:

Sea $AX \subseteq L_0(S)$ un conjunto de oraciones, que tomaremos como axiomas para una teoría. Definimos:

$$Th(AX) := \{\varphi \in L_0(S) \quad | \quad AX \models \varphi\},$$

es decir, coleccionamos en $Th(AX)$ los teoremas que se pueden deducir de los axiomas.

Es fácil verificar que si el conjunto AX es consistente, entonces $Th(AX)$ es una teoría.

*También tiene sentido y utilidad considerar teorías en el marco proposicional, pero, para mayor generalidad, las trataremos haciendo referencia a la lógica de predicados.

†Más generalmente, podríamos denotar con $L_n(S)$ el conjunto de fórmulas de $L(S)$ que tienen sus variables libres entre $x_0, \ldots, x_{n-1}$, las n primeras variables de la lista oficial.

Ejemplo: Sea $S = \{\circ, e\}$, y consideremos el siguiente conjunto de axiomas:

$$AX_g := \{\forall x\ y\ z\,(x\circ(y\circ z) = (x\circ y)\circ z), \forall x\,(x\circ e = x), \forall x\ \exists y\,(x\circ y = e)\}.$$

AX_g contiene los axiomas para la teoría de grupos, y es consistente, pues hay grupos, que son sus modelos. Entonces,

$$Th(AX_g) := \{\varphi \in L_0(\{\circ, e\}) \quad | \quad AX_g \models \varphi\}$$

es la "teoría de grupos".

Por ejemplo, $\forall x(e \circ x = x) \in Th(AX_g)$, es decir, $\forall x(e \circ x = x)$ es un teorema de la teoría de grupos.

■

Notemos que la teoría de grupos es **finitamente axiomatizable**, es decir, la teoría se puede obtener a partir de un conjunto finito de axiomas. Esta situación es deseable, pues podemos, para algunas tareas, concentrarnos en unos pocos axiomas, en lugar de todo un conjunto enorme de oraciones (la teoría).

2. Partimos con un mundo y lo describimos en un cierto lenguaje:

Sea $\mathfrak{A}$ una estructura compatible con un lenguaje $L(S)$. Formamos **la teoría de la estructura** $\mathfrak{A}$:

$$Th(\mathfrak{A}) := \{\varphi \in L_0(S) \quad | \quad \mathfrak{A} \models \varphi\}.$$

$Th(\mathfrak{A})$ contiene todas las proposiciones (formalizadas) que son verdaderas en la estructura $\mathfrak{A}$. Es fácil verificar que, para toda estructura $\mathfrak{A}$, la teoría correspondiente es realmente un teoría de acuerdo con la definición.

Ejemplo: Sea $S_{ar} = \{+, \cdot, 0, 1\}$, el conjunto de símbolos aritméticos, y consideremos la estructura $\mathfrak{N} = \langle \mathbb{N}, +, \cdot, 0, 1\rangle$.

Llamamos **aritmética** a la teoría de $\mathfrak{N}$, es decir, a la colección de todas las aseveraciones que son verdaderas en $\mathfrak{N}$:

$$Th(\mathfrak{N}) := \{\varphi \in L_0(S_{ar}) \quad | \quad \mathfrak{N} \models \varphi\}.$$

Nos preguntamos si $Th(\mathfrak{N})$ es finitamente axiomatizable. Si no lo fuera, podríamos ser menos exigentes, pero todavía aspirando a contar con un conjunto manejable de axiomas para la teoría, es decir, con un conjunto decidible o recursivamente enumerable de axiomas. En otros términos, la pregunta es: ¿existe un $AX \subseteq L_0(S_{ar})$, interesante en algún sentido computacional particular, tal, que $Th(AX) = Th(\mathfrak{N})$? Una pregunta de este tipo es usual en la práctica científica: se trata de buscar principios (pocos) que expliquen todo lo observado en el dominio de discurso o inspección.

Otra pregunta frecuente y natural cuando nos enfrentamos a una teoría, es por su decidibilidad. ¿Es $Th(\mathfrak{N})$ decidible? Si $Th(\mathfrak{N})$ fuera decidible, podríamos, en principio, dejar de hacer matemática (aritmética, en este caso) en la forma usual, es decir, demostrando o refutando (usualmente a través de contraejemplos) presuntos teoremas o conjeturas. Bastaría usar el algoritmo general para decidir si una conjetura, expresada en el lenguaje aritmético, pertenece a $Th(\mathfrak{N})$ o no, es decir, si es verdadera o falsa en la estructura de los números naturales. Los teoremas de la aritmética serían las oraciones con respuesta "SI". Esto parece en extremo atractivo.

■

3. Partimos con una clase de estructuras $(\mathfrak{A}_i)_{i\in I}$ compatibles con un lenguaje $L(S)$, y formamos la teoría de esta clase de estructuras, coleccionando en ella las oraciones que son verdaderas en todas las estructuras $\mathfrak{A}_i$:

$$Th((\mathfrak{A}_i)_{i\in I}) := \{\varphi \in L_0(S) \mid \mathfrak{A}_i \models \varphi, \text{ para todo } i \in I\}.$$

Ejemplo: La teoría de espacios vectoriales es el conjunto de todas las proposiciones que son verdaderas para todos los espacios vectoriales. Una manera alternativa de definir esta teoría es a través del primer método ya visto para generar teorías, a saber, por medio de los axiomas para espacios vectoriales conocidos del álgebra lineal. Esto no es sorprendente, ya que, de hecho, un espacio vectorial se define como una estructura que satisface esos axiomas. Obviamente, esta teoría es finitamente axiomatizable.

■

Del mismo modo, la teoría de grupos puede ser vista como el conjunto de oraciones $\varphi \in L(\{\circ, e\})$ que son verdaderas en todos los grupos $(G, \star, \mathbf{e})$. Como mencionamos antes, esta teoría es finitamente axiomatizable por los tres axiomas ya vistos.

7.1 Axiomatización de la Igualdad

Hasta ahora tenemos a la igualdad, $=$, como un símbolo lógico, con interpretación fija, a saber, la "diagonal" del producto cartesiano del dominio de discurso. Esto se refleja en varios aspectos de la lógica:

1. En las estructuras no se da interpretación libre al símbolo "$=$", de hecho, no explicitamos su interpretación en la estructura (como hacemos con los otros símbolos no lógicos).
2. La definición (de Tarski) de verdad de fórmulas atómicas de la forma $t_1 = t_2$.
3. La aparición de los axiomas lógicos relativos a la igualdad en el sistema formal deductivo.

Por un lado, la igualdad es difícil de manejar computacionalmente. Por otro, no simpre es necesaria su presencia. En consecuencia, una alternativa usual en ciencia de la computación es no considerar a la igualdad como símbolo lógico, es decir, presente en todo lenguaje con interpretación fija y sistema deductivo. En la práctica, esto significa:

- eliminar los puntos 2. y 3. del desarrollo de la lógica mencionados más arriba, y,
- si se necesita, agregar "$=$" a S y axiomatizar su significado introduciendo la "**teoría de la igualdad**".

Entonces, para recuperar la lógica tal como ha sido presentada hasta ahora, bastaría con considerar (como complemento a alguna otra teoría con la que se está trabajando) la teoría de la igualdad con los axiomas que ya vimos y que dicen que $=$ es una relación de equivalencia con propiedad de sustitutividad (una relación de equivalencia con propiedad de sustitutividad se llama "relación de congruencia").

Si, por motivos computacionales, se quiere recuperar axiomáticamente "parte" de la igualdad, se puede considerar algunos u otros axiomas relativos a la igualdad. Por ejemplo, en programación en lógica, y en PROLOG, en particular, se acostumbra introducir sólo el axioma de reflexión $\forall\, x\ (x = x)$, ya que esa igualdad es una identidad sintáctica: los objetos que son iguales son aquellos que son idénticos como sucesiones de caracteres.

7.2 Ejercicios

1. Verificar que los tres métodos para generar teorías que presentamos al comienzo de este capítulo realmente generan teorías (en el sentido de la definición).

2. Considere la siguiente especificación formal en el lenguaje de primer orden $L(\{\leq\})$:

1. $\forall x \; x \leq x$
2. $\forall x \forall y \; (x \leq y \; \wedge \; y \leq x \quad \rightarrow \quad x = y)$
3. $\forall x \forall y \forall z \; (x \leq y \; \wedge \; y \leq z \quad \rightarrow \quad x \leq z)$
4. $\forall x \forall y \; (x \leq y \; \vee \; y \leq x)$

Los axiomas 1.–3. dicen que $\leq$ es un orden parcial (o semi orden). 1.–4. dicen que $\leq$ es un orden total (o lineal).

(a) ¿Qué significa y cómo establecería (de manera general) que el axioma 4. no es redundante con respecto a 1.–3?

(b) Demuestre (ahora en específico) que el axioma 4. no es redundante con respecto a 1.–3.

(c) ¿Cómo establecería que la proposición $\forall x \forall y \forall z \; ((x \leq y \wedge \neg x = y \wedge y \leq z) \quad \rightarrow \quad (x \leq z \wedge \neg x = z))$ sí es redundante con respecto 1.–3.?

3. Ya vimos que la teoría de relaciones de equivalencia puede ser axiomatizada de la siguiente manera:

(a) $\forall x R(x, x)$

(b) $\forall x \forall y (R(x, y) \rightarrow R(y, x))$

(c) $\forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$

Demuestre que la teoría de clases de equivalencia no tiene axiomas redundantes, o, equivalentemente, que cada uno de los axiomas es independiente de los otros dos, exhibiendo un modelo para cada par de ellos que no sea un modelo del tercero. Además, justifique el método en forma precisa.

4. Demostrar que toda teoría recursivamente axiomatizable (es decir, axiomatizable por un conjunto decidible de axiomas) es recursivamente enumerable.

¿Qué pasa si el conjunto de axiomas es finito? ¿Y qué, si el conjunto de axiomas es r.e.?

7.3 Teorías Completas

Intuitivamente, una teoría completa nos proporciona conocimiento completo. Toda proposición expresada en el mismo lenguaje de la teoría queda determinada en su valor de verdad con respecto a la teoría.

Definición: Una teoría $\sum \subseteq L_0(S)$ es **completa** si, para cada oración $\varphi \in L_0(S)$,

$$\sum \models \varphi \quad \text{o} \quad \sum \models \neg\varphi.$$

Es decir, cada proposición es consecuencia lógica de la teoría $\sum$ (o pertenece a $\sum$, según la condición de clausura deductiva impuesta por la definicíon de teoría) o su negación es consecuencia lógica de la teoría (pero no ambas).

El problema del conocimiento completo es motivo de intenso estudio en inteligencia artificial, en particular, en el área de representación lógica de conocimiento. La razón es que habitualmente los seres humanos y los sistemas computacionales no tienen conocimiento completo en las situaciones que deben enfrentar. También en matemática es frecuente tener teorías incompletas; y es más bien excepcional disponer de teorías completas.

Ejemplo: Consideremos la teoría de grupos $\sum_g$. Es decir, $\sum_g = \{\varphi \in L_0(\{\circ, e\}) \mid AX_g \models \varphi\}$, donde $AX_g = \{\forall xyz\ (x \circ (y \circ z) = (x \circ y) \circ z)\ , \forall x\ (x \circ e = x)\ , \forall x\ \exists y\ (x \circ y = e)\}$.

La teoría $\sum_g$ es incompleta. En efecto, consideremos la oración particular $\psi := \forall x\ \forall y\ (x \circ y = y \circ x)$ que expresa la propiedad de conmutatividad de la operación $\circ$.

Se tiene:

1. $\sum_g \not\models \psi$
2. $\sum_g \not\models \neg\psi$

Para probar 1., basta probar que $\sum \cup \{\neg\psi\}$ es consistente, es decir, que hay grupos (que satisfacen $\sum_g$), donde la operación no es conmutativa (que satisfacen $\neg\psi$). Un ejemplo es el grupo S_3 de las permutaciones de tres elementos, digamos 1, 2 y 3.

S_3 tiene $3! = 6$ elementos:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \quad , \quad \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad , \quad \ldots$$

La operacióni, $\circ$, es la composición de permutaciones, por ejemplo:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} := \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

El elemento neutro es la permutación identidad:

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

Se puede demostrar que S_3, así definido, es un grupo, es decir, satisface $\sum_g$ (basta con verificar que $S_3 \models AX_g$), sin embargo, $S_3 \models \neg\psi$, ya que la operación no es conmutativa. Un contraejemplo es el siguiente:

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \quad \text{y}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

Es decir,

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \neq \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Para probar 2., basta probar que $\sum \cup \{\psi\}$ es consistente, es decir, que hay grupos conmutativos (abelianos). Hay muchos, un ejemplo es $\mathfrak{Z} = \langle \mathbb{Z}, +, 0 \rangle$.

Así, hemos comprobado que la conmutatividad queda indeterminada a la luz de la teoría de grupos. En consecuencia, podemos extender consistentemente la teoría de grupos, e ir completándola parcialmente, al agregar a ella ya sea la oración ψ, o la oración $\neg\psi$. En el primer caso, obtendríamos la teoría de grupos conmutativos (o abelianos).

■

Ejemplo: La teoría de los números naturales $Th(\mathfrak{N})$, con $\mathfrak{N} = \langle \mathbb{N}, +, \cdot, 0, 1 \rangle$, es decir, la aritmética, es completa: toda oración φ es verdadera o falsa (y en consecuencia, su negación es verdadera) en la estructura $\mathfrak{N}$.

■

Este es un hecho general: toda teoría proveniente de una estructura es completa. Esto no necesariamente es cierto si hay más de una estructura. Por ejemplo, la teoría de los grupos S_3 y $\mathfrak{Z}$ (simultáneamente) no es completa, pues hay aseveraciones que no son simultánemanete verdaderas ni simultáneamente falsas en ambos grupos, por ejemplo, aquella sobre la conmutatividad.

Ejemplo: Las bases de datos relacionales pueden ser vistas como teorías incompletas: típicamente almacenan explícitamente sólo conocimiento atómico positivo, por ejemplo, que hay un vuelo directo de Santiago a Buenos Aires, pero no dice que no hay un vuelo directo de Santiago a Moscú. La "suposición del mundo cerrado", una suposición a metanivel, nos permite completar (cierto tipo de) conocimiento, ya que nos dice, de manera adicional, que si no está dicho en la base de datos que hay un vuelo directo de Santiago a Moscú, entonces no lo hay.

La suposición del mundo cerrado puede ser vista como un mecanismo para extender una teoría de modo de generar una nueva teoría con conocimiento más completo.

■

Hay diversos maneras para extender teorías incompletas a teorías completas. En general, toda teoría incompleta tendrá muchas teorías completas que la extienden (la contienen).

7.3.1 Ejercicios

1. Verifique cuidadosamente que el conjunto de oraciones (compatibles con, y) verdaderas en una estructura fija es una teoría completa.

2. Verifique que la estructura, S_3, de las permutaciones de tres elementos, con la operación de composición y la identidad como neutro, es un grupo (no conmutativo).

3. Esboce un método para generar las extensiones completas de una teoría.

7.4 El Lenguaje que Distingue

En matemática es común querer clasificar los modelos de ciertas teorías según ciertas características cuantitativas, llamadas "invariantes", de tal modo, que los modelos (estructuras) que comparten esos invariantes sean isomorfos, es

decir, estructuralmente iguales. Es así como se ha logrado clasificar todos los modelos de la teoría de grupos conmutativos generados por una cantidad finita de elementos.

La noción de isomorfismo es muy fuerte. Dos estructuras isomorfas son prácticamente indistinguibles: quiere decir que hay una función biyectiva de un universo al otro que preserva todas las propiedades estructurales. En matemática, dos estructuras isomorfas se consideran prácticamente iguales. En particular, tienen las mismas propiedades matemáticas.

Una noción más débil que la de isomorfismo es la noción de **equivalencia elemental**. Esta noción usualmente es relativa a un lenguaje fijo de la lógica de primer orden: dos estructuras $\mathfrak{A}$ y $\mathfrak{B}$, compatibles con un lenguaje $L(S)$, son elementalmente equivalentes (y esto se denota así $\mathfrak{A} \equiv_{L(S)} \mathfrak{B}$) si son indistinguibles en el lenguaje, más precisamente, si para toda oración $\varphi \in L(S)$: $\mathfrak{A} \models \varphi \Leftrightarrow \mathfrak{B} \models \varphi$. En otros términos, si las disticiones entre dos estructuras, si las hay, se pueden expresar sólo en el lenguaje $L(S)$, y estas estructuras son elementalmente equivalentes, entonces no podremos distinguirlas.

Dos estructuras isomorfas son siempre elementalmente equivalentes, pero la afirmación recíproca no necesariamente es verdadera. La equivalencia elemental puede ser vista como un substituto débil del isomorfismo.

Tal como con la noción de isomorfismo, se ha logrado clasificar según equivalencia elemental a todos los modelos de algunas teorías de primer orden. Esto también se hace a través de la determinación de ciertos invariantes numéricos. Por ejemplo, existen clasificaciones según equivalencia elemental de los modelos de la teoría (de primer orden) de grupos conmutativos, sin embargo, no existe una clasificación de ellos según tipos de isomorfismo.

Es fácil demostrar que si una teoría es completa, entonces todos sus modelos son elementalmente equivalentes (tal como los modelos de una teoría matemática categórica son todos isomorfos). Es decir, una teoría completa no permite distinguir (en su lenguaje) a dos de sus modelos. Desde este punto de vista, la clasificación según equivalencia elemental de los modelos de una teoría (no necesariamente completa) puede ser vista como una clasificación de todas las extensiones completas de la teoría.

7.4.1 Ejercicios

1. Formule de manera precisa la noción de isomorfismo entre dos estructuras (compatibles con un mismo lenguaje).

2. Demuestre que dos estructuras isomorfas son elementalmente equivalentes.

3. Demuestre que una teoría es completa si y sólo si todos sus modelos son elementalmente equi-valentes.

7.5 Teorías Decidibles

Retomemos otro problema interesante sobre teorías:

Problema: Dada una teoría $\sum \subseteq L_0(S)$, ¿es $\sum$ decidible? Aquí nos preguntamos si existe un algoritmo que, aplicado a una oración arbitraria $\varphi \in L_0(S)$, responde:

SI si $\sum \models \varphi$ (o $\varphi \in \sum$)
NO si $\sum \not\models \varphi$ (o $\varphi \notin \sum$).

Ejemplo: Por el teorema de Church, la teoría en un lenguaje $L(S)$ con el conjunto vacío de axiomas, es decir, el conjunto de oraciones universalmente verdaderas de $L(S)$, es indecidible.

Ejemplo: Consideremos la estructura $\mathfrak{N}' = \langle \mathbb{N}, \sigma, 0 \rangle$, donde σ es la función sucesor, que envía cada número n a $n+1$. Formemos $Th(\mathfrak{N}') \subseteq L_0(\{\sigma, 0\})$. Esta teoría es completa y, además, decidible*.

Ejemplo: Consideremos ahora la estructura $\mathfrak{N} = \langle \mathbb{N}, +, \cdot, 0, 1 \rangle$. La teoría $Th(\mathfrak{N}) \subseteq L_0(\{+, \cdot, 0, 1\})$ es completa, pero indecidible. Este es un teorema de Kurt Gödel.

Una consecuencia metodológica de este resultado es que, dado que no hay un algoritmo para chequear conjeturas, hay que seguir haciendo aritmética como hasta ahora.

Ejemplo: La teoría de grupos es indecidible. Este es un teorema de Alfred Tarski. Sin embargo, la teoría de grupos conmutativos es decidible. Este es un teorema de Wanda Smielev.

■

Otro problema que teníamos era el siguiente:

*Una demostración detallada de la decidibilidad se encuentra en el libro Enderton [20].

Problema: Dada una teoría $\sum \subseteq L_0(S)$, ¿es $\sum$ finitamente axiomatizable. Aquí nos preguntamos si existe $AX \subseteq L_0(S)$, finito, tal, que $\sum = Th(AX) := \{\varphi \in L_0(S) \mid AX \models \varphi\}$.

¿Es, por ejemplo, $Th(\mathfrak{N})$ finitamente axiomatizable? Podemos responder a esta pregunta recurriendo al siguiente:

Teorema: Si una teoría $\sum$ es finitamente axiomatizable y completa, entonces es decidible.

Demostración: Sea AX, finito, tal, que $Th(AX) = \sum$. Un algoritmo de decisión para $\sum$ es el siguiente: dada una oración arbitraria φ, nos preguntamos: $¿\varphi \in \sum?$

Para responder, usemos "sistemáticamente" los axiomas lógicos, las reglas de deducción y los axiomas pertenecientes a AX (una cantidad finita), para ir produciendo demostraciones formales. Recopilando todas las últimas lineas de demostraciones formales, obtenemos todos los elementos de $\sum$.

Como $\sum$ es completa, φ va a aparecer o $\neg\varphi$ va a aparecer, en algún momento, al final de una demostración formal. Entonces respondemos según el caso.

■

Del teorema anterior podemos concluir, de inmediato, que $Th(\mathfrak{N})$ no es finitamente axiomatizable.

Observando la demostración del teorema, nos damos cuenta de que es posible debilitar la hipótesis de que el conjunto de axiomas AX sea finito. Basta con que los axiomas de AX puedan ser usados sistemáticamente. En consecuencia, es suficiente que AX sea decidible o recursivamente enumerable. En consecuencia, $Th(\mathfrak{N})$ no cuenta con un conjunto r.e. de axiomas.

Estamos entonces en la siguiente situación:

- La aritmética no puede ser axiomatizada razonablemente* en el lenguaje $L(\{+, \cdot, 0, 1\}))$.
- Sin embargo, nosotros hacemos aritmética, y no sólo "mirando" la estructura aritmética, sino también deductivamente.

*Lo menos que le podemos pedir a un conjunto de axiomas para una teoría es que sea decidible, o al menos, recursivamente enumerable.

Tenemos al menos dos alternativas:

1. No limitarnos a lenguajes de primer orden.
2. Axiomatizar sólo parte de la aritmética, es decir, sólo parte de la teoría de $\mathfrak{N}$.

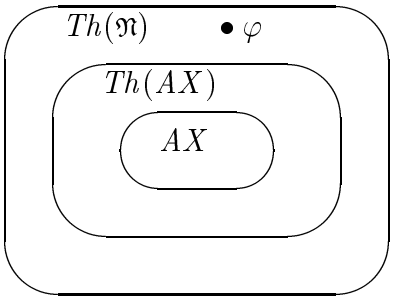
Antes de explorar estas alternativas, miremos con más cuidado los resultados sobre "no decidibilidad" y "no axiomatizabilidad recursiva" (es decir, mediante un conjunto recursivo, o decidible, de axiomas) de la teoría $Th(\mathfrak{N})$. De hecho, tenemos un resultado más fuerte que el teorema de la indecidibilidad de $Th(\mathfrak{N})$; a saber, el famoso **Primer Teorema de Incompletidad de Kurt Gödel**:

Teorema: Sea $AX \subseteq L_0(\{+, \cdot, 0, 1\})$ arbitrario (pero consistente). Si AX es decidible, entonces existe una oración $\varphi \in L_0(\{+, \cdot, 0, 1\})$ tal, que:

- $\mathfrak{N} \models \varphi$, pero
- $AX \not\models \varphi$.

■

El teorema dice que cualquier sistema de axiomas razonable que pensemos para la aritmética es insuficiente, incompleto, para capturarla:



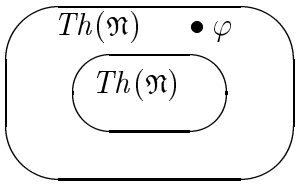
Notemos que si AX es decidible y $AX \subseteq Th(\mathfrak{N})$, entonces $Th(AX)$ es incompleta, ya que, para la oración φ del teorema, $\varphi \notin Th(AX)$ y $\neg\varphi \notin Th(AX)$.

Del teorema de incompletidad de Gödel se obtiene fácilmente los resultados que teníamos antes sobre la indecidibilidad y no axiomatizabilidad finita de $Th(\mathfrak{N})$:

(a) $Th(\mathfrak{N})$ no es recursivamente (ni finitamente) axiomatizable, ya que cualquier conjunto recursivo de axiomas para ella genera una teoría incompleta, pero $Th(\mathfrak{N})$ es completa.

(b) $Th(\mathfrak{N})$ no es decidible (no lo hemos probado y, de hecho, lo usamos para obtener el punto (a) anterior, antes de tener el teorema de Gödel).

En efecto: si $Th(\mathfrak{N})$ fuera decidible, podríamos tomar trivialmente $AX := Th(\mathfrak{N})$, es decir, $Th(\mathfrak{N})$ se podría axiomatizar a sí misma. Como $Th(Th(\mathfrak{N})) = Th(\mathfrak{N})$, obtenemos fácilmente una contradicción. Gráficamente:



El primer teorema de Gödel se demuestra codificando o representando de manera aritmética, en el lenguaje objeto, ciertos conceptos sintácticos del metalenguaje, tales como fórmula, demostración formal, demostrabilidad, consistencia, etc. De esta manera se obtiene oraciones aritméticas (pero usualmente con contenido metamatemático) que son verdaderas en la teoría de los naturales, pero indecidibles. Demostraciones de este interesante e importante teorema pueden ser encontradas en el libro de Enderton y en el artículo de Uspensky.

La posibilidad de codificar aritméticamente metaconceptos sintácticos como los recién mencionados permite obtener el **Segundo Teorema de Incompletidad de Gödel**, que dice, esencialmente, que en cualquier teoría formal que sea lo suficientemente expresiva como para construir la aritmética (por ejemplo, la aritmética misma o la teoría de conjuntos), no se puede demostrar internamente, en la misma teoría, la consistencia de ésta. Para demostrar su consistencia, tendríamos que salirnos a otra teoría que comprenda a la anterior y que podamos suponer consistente. En particular, si consideramos el hecho que la matemática se puede construir dentro de la teoría de conjuntos, entonces estarímos imposibilitados de demostrar la consistencia de la matemática con elementos y métodos de la matemática.

Volviendo al primer teorema de incompletidad de Gödel, vemos que, en definitiva, no tenemos ninguna esperanza de hacer aritmética en forma deductiva a partir de un sistema manejable de axiomas. Ni tampoco de decidir sobre presuntos teoremas de manera algorítmica y general. Estos resultados hablan

de las limitaciones de los métodos computacionales.

En las próximas secciones exploraremos las dos alternativas ya planteadas:

1. No limitarnos a la lógica de primer orden.
2. Desarrollar deductivamente sólo parte de la aritmética.

7.5.1 Ejercicios

1. Demostrar que toda teoría completa y recursivamente enumerable es decidible.

2. Demostrar que la aritmética, es decir, la teoría de $(\mathbb{N}, +, *, 0, 1)$, no es recursivamente enumerable.

3. (a) Defina la suma de naturales en términos de la función sucesor σ usando recursión.

(b) Haga lo mismo para la multiplicación en términos de la suma.

(c) ¿Cree Ud. que es posible definir la suma y la multiplicación en términos de la función σ usando fórmulas del lenguaje de primer orden con conjunto de símbolos $\{\sigma, 0\}$?

Ind.: ¿Qué sabe sobre la teoría de $(\mathbb{N}, \sigma, 0)$?

7.6 Aritmética de Segundo Orden

Usaremos el lenguaje de segundo orden $L_{II}(\{+, \cdot, 0, 1\})$. Esto significa que ahora podemos hacer cuantificaciones sobre predicados y funciones. Consideremos los siguientes axiomas:

(a) $\forall x \ \neg\, x + 1 = 0$
$\forall x \ \forall y \ \ (x + 1 = y + 1 \ \rightarrow \ x = y)$

(b) $\forall x \ \ x + 0 = x$
$\forall x \ \forall y \ \ x + (y + 1) = (x + y) + 1$

(c) $\forall x \ \ x \cdot 0 = 0$
$\forall x \ \forall y \ \ x \cdot (y + 1) = x \cdot y + x$

(d) $\forall X\ ([X(0) \wedge \forall x\ (X(x) \to X(x+1))] \to \forall x\ X(x))$

Estos axiomas son la base de AP_{II}, la **Aritmética de Peano**. Los roles de estos axiomas son los siguientes:

(a) es la axiomatización de la función sucesor; y dice que el 0 no es un sucesor y que la función es uno a uno.

(b) es la definición recursiva de la suma en términos de la función sucesor.

(c) es la definición recursiva del producto en términos de la suma.

(d) es el axioma de inducción de segundo orden, que presenta una cuantificación de segundo orden, $\forall X$, sobre subconjuntos del dominio. Este axioma permite demostrar que existen funciones únicas que satisfacen las ecuaciones de recursión anteriores. Además, restringe la extensión del dominio $\mathbb{N}$ a una minimal: en $\mathbb{N}$ encontramos sólo los elementos 0 y los que se obtienen a partir de él mediante aplicaciones sucesivas de la función sucesor (sumas de 1).

Esta es una axiomatización finita y muy potente, en el siguiente sentido:

Teorema (Dedekind) Toda estructura $\mathfrak{A}$ que satisface AP_{II} es esencialmente la estructura $\mathfrak{N} = \langle N, +, \cdot, 0, 1\rangle$, más precisamente,

$$\mathfrak{A} \models AP_{II} \quad \Longrightarrow \quad \mathfrak{A} \cong \mathfrak{N}.$$

- El símbolo $\cong$ indica que las estructuras son isomorfas, es decir, que existe una biyección entre los dominios que preserva las propiedades estructurales.
- Que dos estructuras sean isomorfas significa que, en esencia, son la misma estructura, pero se obtiene una a partir de la otra a través de un renombramiento de sus constituyentes.
- El teorema de Dedekind es fácil de probar. La demostración usa fuertemente el axioma de inducción de segundo orden.
- El teorema dice que la especificación AP_{II} es **categórica**, es decir, todos sus modelos son isomorfos. En consecuencia, ella caracteriza plenamente a la estructura que pretende especificar, en este caso, $\mathfrak{N}$. ¡Este es el ideal de una especificación!

De la categoricidad de AP_{II} se desprende, además, que $Th(AP_{II}) := \{\varphi \in L_0(S_{ar}) \mid AP_{II} \models \varphi\}$ es una teoría completa.

La teoría de primer orden $Th(\mathfrak{N})$ no es categórica, es decir, si juntamos todo lo que observamos sobre la estructura $\mathfrak{N}$, expresado en el lenguaje aritmético de primer orden, no logramos capturarla. Más precisamente, demostraremos que hay modelos de $Th(\mathfrak{N})$, es decir, estructuras $\mathfrak{A} = \langle A, +^A, \cdot^A, 0^A, 1^A\rangle$ que satisfacen la teoría $Th(\mathfrak{N})$, pero que no son isomorfos a $\mathfrak{N}$. Cualquier subconjunto de $Th(\mathfrak{N})$ proporciona sólo una especificación parcial de $\mathfrak{N}$.

Con lenguajes de segundo orden sí obtenemos una especificación categórica de $\mathfrak{N}$. Pareciera entonces que no hay razones para limitarnos a lenguajes de primer orden. Sin embargo,

- No todos los teoremas agradables que tenemos para los lenguajes de primer orden son válidos para lenguajes de segundo orden.
- Demostradamente no existe ningún sistema formal deductivo completo para la lógica de segundo orden (aunque sí hay sistemas deductivos correctos).
- El problema VU de validez universal para lógica de segundo orden no es recursivamente enumerable.
- No es válido el teorema de compacidad para la lógica de segundo orden.

7.6.1 Ejercicios

1. Intente demostrar el teorema de Dedekind. En él, el axioma de inducción de segundo orden juega un papel esencial.

2. Considere los siguiente axiomas para la teoría de $\mathfrak{N}'$ (ver sección anterior): $\forall x \neg \sigma(x) = 0$, $\forall x \forall y\, (\sigma(x) = \sigma(y) \to x = y)$, $\forall X\, ([X(0) \wedge \forall x\, (X(x) \to X(\sigma(x)))] \to \forall x\, X(x))$ (el axioma de inducción de segundo orden). Demuestre que hay una única función $suma^{\mathbb{N}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, que satisface la definición recursiva: $\forall x\, (suma(x, 0) = x)$, $\forall x \forall y\, (suma(x, \sigma(y)) = \sigma(suma(x, y)))$.

Ind.: El axioma de inducción es esencial en esta demostración.

3. Escriba los axiomas para cuerpo ordenado completo (los que satisface el

cuerpo ordenado de los números reales, como vio en su curso de álgebra) usando un lenguaje de segundo orden*.

Ind: Por si no recuerda, la "ordenación completa" está dada a través del axioma del supremo que dice que todo subconjunto de los reales que está acotado superiormente tiene un supremo, es decir, una cota superior mínima.

4. Dé una fórmula de segundo orden φ_{fin} que defina a las estructuras finitas, es decir, que sea satisfecha exactamente por estructuras con dominio finito.

Ind.: Básese en el ejercicio 3. de la sección 5.2.4 y use variables para relaciones binarias (o funciones unarias).

5. Demuestre que el teorema de compacidad no puede ser válido para la lógica de segundo orden.

Ind.: Compare con el ejercicio 10 de la sección 6.3.2.

7.7 Modelos No Estándar de la Aritmética

Como anunciamos, vamos a demostrar que existen modelos de la aritmética de primer orden que no son isomorfos a $\mathfrak{N}$, de hecho, que tienen números "infinitos". Al ser éstos modelos de la teoría de $\mathfrak{N}$, pero estructuralmente distintos del modelo que da origen a la teoría, a saber, el modelo estándar $\mathfrak{N}$, los llamamos "modelos no estándar de la aritmética". La demostración de la existencia de tales modelos usa el teorema de compacidad de la lógica de primer orden.

Para simplificar la exposición vamos a agregar un nuevo símbolo al lenguaje aritmético, el predicado binario $<$ intepretado como relación de orden usual entre números naturales. Esto no es algo esencial (pero ayuda a la exposición), pues el orden entre naturales se puede definir mediante una fórmula de primer orden a partir de los otros constituyentes del lenguaje:

$$\forall x \ \forall y \ (x < y \ \leftrightarrow \underbrace{\exists z \ (\neg(0 = z) \wedge x + z = y)}_{\text{sin "<"}}).$$

En consecuencia, si queremos eliminar el símbolo $<$, volviendo a la situación original, basta reemplazar las apariciones de fórmulas del tipo $t < s$ por el correspondiente lado derecho del axioma de definición de $<$.

*Esta teoría también es categórica. En consecuencia, $\mathfrak{R}$ es, en esencia, el único cuerpo ordenado completo.

Trabajaremos entonces con la estructura $\mathfrak{N}^< = \langle \mathbb{N}, <, +, \cdot, 0, 1 \rangle$. Formamos su teoría $Th(\mathfrak{N}^<) = \{\varphi \in L_0(\{<, +, \cdot, 0, 1\}) \mid \mathfrak{N}^< \models \varphi\}$. Como antes, la teoría es completa, indecidible y no recursivamente axiomatizable. La seguimos llamando "aritmética".

Queremos un modelo no estándar de $Th(\mathfrak{N}^<)$, es decir, un modelo no isomorfo a $\mathfrak{N}$ que, además, tenga elementos infinitamente grandes. Para esto especificamos nuestros deseos:

$$\Sigma := Th(\mathfrak{N}^<) \cup \{0 < \eta \ , \ 1 < \eta \ , \ 1 + 1 < \eta \ , \ 1 + 1 + 1 < \eta \ , \ \cdots\}.$$

Aquí η es una nueva constante del lenguaje, un nombre para individuo; y $\sum \subseteq L_0(\{<, +, \cdot, 0, 1, \eta\})$.

Demostremos que $\sum$ tiene un modelo $\mathfrak{A}$. Por el teorema de compacidad, basta verificar que cada subconjunto finito $\sum_0$ de $\sum$ tiene un modelo.

Si $\sum_0$ es finito y está contenido en $\sum$, entonces, para algún $n \in \mathbb{N}$, $\sum_0$ está contenido en:

$$\sum\nolimits_0{}' := Th(\mathfrak{N}^<) \cup \{0 < \eta \ , \ 1 < \eta \ , \ 1 + 1 < \eta \ , \cdots, \ \underbrace{1 + \cdots + 1}_{n \text{ veces}} < \eta\}.$$

Basta con verificar que $\sum_0'$ tiene un modelo. La estructura $\langle \mathfrak{N}^<, \ n + 1 \rangle$, en la cual todos los símbolos antiguos se interpretan como en $\mathfrak{N}^<$, y η, como $n + 1$, es un modelo de $\sum_0'$.

Hemos demostrado que $\sum$ es consistente. Sea $\mathfrak{A} = \langle A, <^A, +^A, \cdot^A, 0^A, \ 1^A, \eta^A \rangle$ un modelo de $\sum$. Entonces, en particular, $\mathfrak{A} \models Th(\mathfrak{N}^<)$, es decir, $\mathfrak{A}$ es modelo de la aritmética. Además, por la segunda parte de la axiomatización $\sum$, se tiene:

$$0^A <^A \eta^A \ , \ 1^A <^A \eta^A \ , \ 1^A +^A 1^A <^A \eta^A \ , \ \cdots.$$

Por estas desigualdades decimos que η^A es un número infinito: es mayor que cualquier número natural finito, o, más precisamente, mayor que cualquier elemento de la estructura $\mathfrak{A}$ que se obtenga sumando una cantidad finita de 1^A's.

¿Cómo se ve la estructura $\mathfrak{A}$? Un segmento inicial de A se ve como en la figura A7.1.

Las siguientes proposiciones son verdaderas en $\mathfrak{N}^<$, y entonces pertenecen a $Th(\mathfrak{N}^<)$:

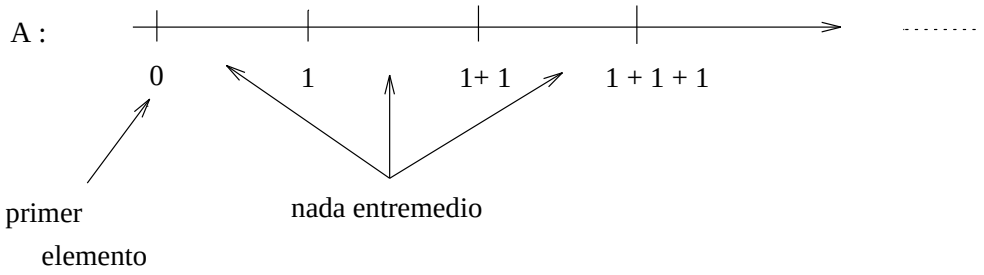


Figura A7.1:

$$\left.\begin{array}{l} \forall x\ (x = 0\ \vee\ 0 < x) \\ \neg\,\exists x\ (0 < x\ \wedge\ x < 1) \\ \neg\,\exists x\ (1 < x\ \wedge\ x < 1 + 1) \\ \vdots \\ \forall x\ \exists y\ (x < y\ \wedge y = x + 1\ \wedge\ \neg\,\exists z\ (x < z\ \wedge\ z < y)) \end{array}\right\} \subseteq Th(\mathfrak{N}^{<})$$

Luego, $\mathfrak{A}$ también las satisface. Podemos ir completando la imagen de la estructura, que ahora se ve como en la figura A7.2.

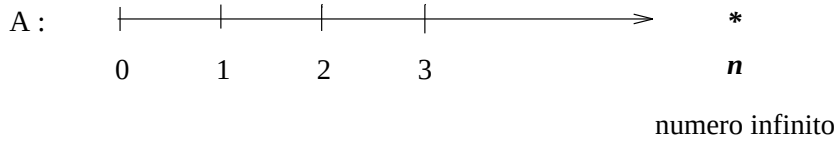


Figura A7.2:

¿Algo más? Bueno, también pertenecen a $Th(\mathfrak{N}^{<})$ las proposiciones:

$$\left.\begin{array}{l} \forall x\ (\neg x = 0\ \rightarrow\ x - 1 < x < x + 1) \\ \forall x\ \neg\,\exists z\ (x < z\ \wedge\ z < x + 1) \end{array}\right\} \subseteq Th(\mathfrak{N}^{<})$$

Luego, también son verdaderas en $\mathfrak{A}$. Sabemos más sobre la estructura $\mathfrak{A}$; y ahora la podemos ver como en la figura A7.3.

Obviamente, esta estructura no es isomorfa a $\mathfrak{N}^{<}$. Sin embargo, al ser ambas estructuras modelos de una teoría completa, son indistinguibles en el lenguaje aritmético de primer orden $L(\{<, +, \cdot, 0, 1\})$, es decir, satisfacen exactamente las mismas oraciones aritméticas. Más precisamente, para toda oración $\varphi \in$

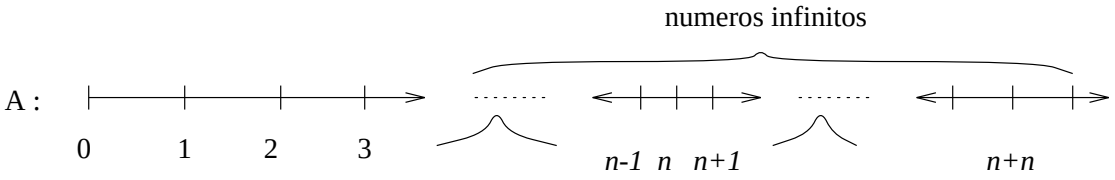


Figura A7.3: Un modelo no estándar para la artmética

$L_0(\{<,+,\cdot,0,1\})$, se tiene:

$$\mathfrak{N}^{<} \models \varphi \iff \mathfrak{A} \models \varphi.$$

Si $AX \subseteq Th(\mathfrak{N}^{<})$ es un conjunto consistente de axiomas para, posiblemente, parte de la aritmética, entonces AX es verdadero en todos los modelos de la aritmética $Th(\mathfrak{N}^{<})$, tanto en el subentendido o estándar $\mathfrak{N}^{<}$, como en los otros no isomorfos a $\mathfrak{N}^{<}$, en particular, en el modelo no estándar anterior. En consecuencia, razonar deductivamente a partir de AX es razonar simultáneamente sobre todos los modelos de AX.

Las consideraciones anteriores nos hablan de las limitaciones expresivas de los lenguajes de primer orden. Sin embargo, en el lenguaje de segundo orden $L_{II}(\{<,+,\cdot,0,1\})$ sí son distinguibles las estructuras $\mathfrak{N}^{<}$ y $\mathfrak{A}$: el axioma de inducción es satisfecho por $\mathfrak{N}^{<}$, pero no por $\mathfrak{A}$, que no tiene una extensión minimal, es decir, no tiene como elementos sólo aquellos que está forzado a tener (el 0 y sus sucesores finitos). No es sorprendente entonces que el teorema de compacidad, con el cual obtuvimos la estructura $\mathfrak{A}$, no sea válido para la lógica de segundo orden.

7.7.1 Ejercicios

1. Demuestre que el modelo estándar de la aritmética es elementalmente equivalente al modelo no estándar presentado en la sección anterior.

2. Muestre que dos modelos elementalmente equivalentes no necesariamente son isomorfos.

3. Llamamos "análisis" a la teoría de primer orden $Th(\mathfrak{R})$ de la estructura de los números reales $\mathfrak{R} = (\mathbb{R}, <, +, *, 0, 1)$.

(a) Demuestre que hay un modelo no estándar del "análisis", es decir, no

isomorfo a $\mathfrak{R}$, en el que hay números "infinitesimales" c con $0 < c < n^{-1}$, para todo $n \in \mathbb{N}$ (en realidad, para todo entero positivo finito del modelo no estándar).

(b) Demuestre que en este modelo también existe la raíz cuadrada de cualquier elemento no negativo.

(c) Demuestre que el axioma del supremo no puede ser expresado en lógica de primer orden.

7.8 Aritmética de Primer Orden

Ahora trataremos la segunda alternativa axiomática a la aritmética completa $Th(\mathfrak{N})$. Esta consiste en axiomatizar, en primer orden, sólo una parte de la aritmética. Hay varias posibilidades, sin embargo, una muy atractiva consiste en mantenernos lo más cerca posible de la axiomatización de Peano de segundo orden AP_{II}, pero usando sólo el lenguaje de primer orden $L(S_{ar})$.

Como en AP_{II}, sólo el último axioma, el de inducción, es de segundo orden, nos quedaremos con los primeros axiomas, (a) – (c) en la sección 7.6, y cambiaremos sólo el último.

El axioma de inducción de segundo orden

$$\forall X\ ([X(0) \wedge \forall x\ (X(x) \to X(x+1))] \to \forall x\ X(x))$$

tiene una cuantificación universal sobre todos los posibles subconjuntos del dominio de discurso. Reemplazemos esta cuantificación por una que considere sólo todos los subconjuntos (propiedades unarias) que son definibles o describibles en el lenguaje de primer orden (o propiedades expresables en el lenguaje de primer orden). Para ganar más intuición veamos un par de ejemplos. Aquí el lenguaje de primer orden es $L(\{+, \cdot, 0, 1\})$.

Ejemplo: (a) El conjunto de los números pares es definible en este lenguaje (o la propiedad de "ser par" es expresable en el lenguaje):

$$\{2, 4, 6, \cdots\} \ = \ \{n \in \mathbb{N} \ \mid \ \mathfrak{N} \models \varphi[n]\},$$

donde $\varphi(x)$ es la fórmula $\exists\, y\ (\neg(y = 0)\ \wedge\ x = (1+1) \cdot y)$ de $L(\{+, \cdot, 0, 1\})$, con una variable libre x, y el n en $\varphi[n]$ indica que él es el valor asignado a x.

(b) El conjunto de los números primos es definible mediante la fórmula:

$$\psi(x): \ \ \neg\, \exists\, y\ \exists\, z\ (x = y \cdot z\ \wedge\ (\neg(y = 1)\ \wedge\ \neg(z = 1)))\ \wedge\ \neg(x = 1).$$

Entonces:

$$\{2, 3, 5, \cdots\} \;=\; \{n \in \mathbb{N} \;\mid\; \mathfrak{N} \models \psi[n]\}.$$

■

Volviendo al problema de restringir la cuantificación universal en el axioma de inducción, vemos que podemos identificar subconjuntos definibles en el lenguaje de primer orden con fórmulas del lenguaje de primer orden con una variable libre. Como en el lenguaje de primer orden no podemos cuantificar sobre todas las fórmulas del lenguaje, hacemos aparecer explícitamente cada posible fórmula del lenguaje con una variable libre en el axioma de inducción. Esto conduce a infinitos axiomas, pues hay infinitas fórmulas de primer orden con una variable libre. Sin embargo, todos estos axiomas siguen un mismo esquema.

Más precisamente, reemplazamos el axioma de segundo orden:

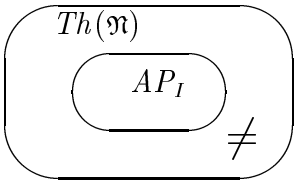
$$\forall\, X\ ([X(0) \wedge \forall\, x\ (X(x) \to X(x+1))] \to \forall\, x\ X(x)),$$

por el **esquema de inducción**: para cada $\varphi(x) \in L(\{+, \cdot, 0, 1\})$, el siguiente es un axioma:

$$[\varphi(0)\ \wedge\ \forall\, x\ (\varphi(x) \to \varphi(x+1))] \to \forall\, x\ \varphi(x).$$

Vemos que el esquema es un generador de axiomas de primer orden para la nueva teoría (de primer orden) para la aritmética. La nueva teoría, con los axiomas de primer orden de la aritmética de segundo orden de Peano, más el esquema de inducción, se llama **Aritmética de Peano de Primer Orden**, y la denotamos con AP_I.

AP_I tiene una cantidad infinita enumerable de axiomas, pues hay una cantidad infinita enumerable de fórmulas en $L(\{+, \cdot, 0, 1\}))$, que podrían ser reemplazadas en el esquema de inducción. De hecho, la lista de axiomas de AP_I es recursivamente enumerable, más aún, es decidible. En otras palabras, la aritmética de Peano de primer orden es recursivamente axiomatizable. En consecuencia, por el teorema de incompletidad de Gödel, AP_I es incompleta.



Tal como hay modelos no estándar de $Th(\mathfrak{N})$, también hay modelos no estándar de AP_I, es decir, estructuras $\mathfrak{A}$ tales, que: $\mathfrak{A} \models AP_I$, pero $\mathfrak{A} \ncong \mathfrak{N}$. Sin embargo, teníamos que, para toda estructura $\mathfrak{A}$:

$$\mathfrak{A} \models AP_{II} \implies \mathfrak{A} \cong \mathfrak{N}.$$

En consecuencia, AP_I es una axiomatización definitivamente más débil que la axiomatización de segundo orden AP_{II}. Es decir, el esquema de primer orden es más débil que el axioma de segundo orden: la xiomatización de segundo orden es tan expresiva que determina, excepto por isomorfismo, un único modelo; sin embargo, la axiomatización correspondiente de primer orden, por ser menos expresiva, deja abierta la posibilidad de que haya modelos no isomorfos, es decir, esencialmente distintos.

Podemos concluir de lo anterior que hay subconjuntos de $\mathbb{N}$ que no son definibles en el lenguaje de primer orden. Esto no es sorprendente, pues

- hay a lo más una cantidad infinita enumerable de subconjuntos definibles (al haber a los más una cantidad infinita enumerable de fórmulas aritméticas con una variable libre que no son lógicamente equivalentes);
- hay $\mid P(\mathbb{N}) \mid$ (cardinalidad del conjunto potencia de $\mathbb{N}$) subconjuntos de $\mathbb{N}$.
- $\mid P(\mathbb{N}) \mid > \mid \mathbb{N} \mid$.

Sin embargo, aunque AP_I no sea tan expresiva como AP_{II}, para la mayoría de los efectos prácticos es suficiente, y tiene la ventaja de poder ser usada con un sistema formal deductivo completo.

7.8.1 Ejercicios

1. (a) Dado el esquema generador de fórmulas:

$$[\varphi(0) \wedge \forall x(\varphi(x) \rightarrow \varphi(x+1))] \rightarrow \forall x \varphi(x),$$

que remplazaría al principio de inducción de segundo orden (φ es una metavariable para fórmulas con una variable libre), escriba un programa que genere instancias (fórmulas) concretas a partir del esquema. Aplíquelo (también hágalo a mano) en los casos en que φ:

- define al conjunto de números primos

- define al conjunto de números pares
- define al conjunto de números que son potencias de 2
- define al conjunto de números de la sucesión de Fibonnaci (¿existe tal fórmula?)

(b) ¿Cuántas fórmulas concretas se puede generar a partir del esquema? ¿Por qué?

(c) ¿En qué medida el esquema generador es un sustituto del principio de inducción de segundo orden?

2. Reescriba los axiomas de cuerpo ordenado completo que dio en la sección 7.6.1 usando un esquema de primer orden que reemplace al axioma (de segundo orden) del supremo.

7.9 Complejidad de Teorías Formalizadas

Hasta ahora hemos visto como deseable que una teoría sea decidible por sus posibles efectos prácticos: para determinar si presuntos teoremas son parte de la teoría, bastaría echar a funcionar un algoritmo que nos va a entregar la respuesta SI o NO. Sin embargo, no nos hemos cuestionado cuánto tiempo nos puede tomar esperar la respuesta. Para que el algoritmo general sea de utilidad práctica, quisiéramos que éste fuera de tiempo de ejecución al menos polinomial.

Nuevamente surgen las preguntas que vimos en la sección sobre complejidad de problemas de decisión. En nuestro contexto actual, nos preguntamos, para una teoría $\sum$ decidible, si hay algún algoritmo de decisión de tiempo polinomial en el largo de las oraciones de entrada. También nos podemos preguntar por la complejidad temporal de algoritmos concretos para $\sum$. Veamos algunos ejemplos:

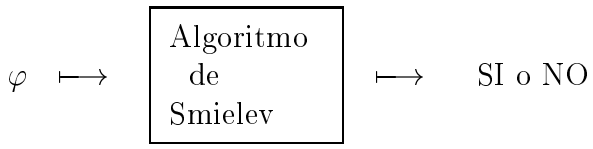
Ejemplo: Sean AX_g el conjunto de los tres axiomas para la teoría de grupos, y $AX_{gc} := AX_g \cup \{\forall\, x\, \forall\, y(x \circ y = y \circ x)\}$, los axiomas para la teoría de grupos conmutativos. Ya dijimos anteriormente que la teoría de grupos conmutativos

$$Th(AX_{gc}) = \{\varphi \in L_0(\{\circ, e\}) \;\mid\; AX_{gc} \vdash \varphi\}$$

es decidible.

¿Qué tan difícil es decidir, para una oración arbitraria φ, si $\varphi \in Th(AX_{gc})$?

El algoritmo de decisión original para $Th(AX_{gc})$, el algoritmo de Smielev (1949), es superexponencial en el largo de la entrada:



Cuando $\mid \varphi \mid = n$, el número de pasos que da el algoritmo en el peor caso es al menos

$$\underbrace{2^{2^{\cdot^{\cdot^{2^{c \cdot n}}}}}}_{n-1 \ \ exponentes} .$$

El año 1988 se obtuvo una mejora substancial con un algoritmo cuyo tiempo de decisión en el peor caso es:

$$\underbrace{2^{2^{\cdot^{\cdot^{2^{c \cdot n}}}}}}_{8 \ \ exponentes} ,$$

es decir, todavía varias veces exponencial, pero considerablemente mejor que antes. No se sabe si hay algún algoritmo de decisión polinomial para esta teoría.

■

Ejemplo: Sea $\mathfrak{R} = \langle \mathbb{R}, <, +, \cdot, 0, 1 \rangle$, la estructura de los números reales. Formamos la teoría de primer orden de los reales:

$$Th(\mathfrak{R}) = \{\varphi \in L_0(\{<, +, \cdot, 0, 1\}) \ \mid \ \mathfrak{R} \models \varphi\}.$$

Tarski demostró que esta teoría es decidible al establecer que $Th(\mathfrak{R})$ es recursivamente axiomati-zable. Como, por construcción, esta teoría es completa, y, además, es de axiomatizabilidad recursiva, se obtiene, de inmediato, su decidibilidad (ver resultados de la sección 7.5). Los axiomas que dio Tarski para $Th(\mathfrak{R})$ son los siguientes:

- los de cuerpo ordenado;
- "todo elemento positivo tiene raiz cuadrada": $\forall\, x\ (0 < x \rightarrow \exists\, y\ (y \cdot y = x))$;

- "todo polinomio de grado impar tiene una raíz":

  $\forall\, x_0\ x_1 \cdots x_{2n+1}\ (x_{2n+1} \neq 0 \rightarrow\ \exists\, y\ (x_0\ +\ x_1 \cdot y\ +\ x_2 \cdot y \cdot y\ +\ \cdots\ +\ x_{2n+1} \cdot y^{2n+1} = 0))$,

  $$n = 1, 2, 3, \cdots.$$

Este conjunto infinito y decidible de axiomas AX_r genera deductivamente $Th(\mathfrak{R})$, es decir, $Th(\mathfrak{R}) = Th(AX_r)$.

Tarski, además, dio un algoritmo de decisión para la teoría de los reales. Su algoritmo es superexponencial en tiempo de ejecución. Es natural preguntarse si hay algún algoritmo más eficiente. Ojalá, uno de tiempo polinomial. En 1974, Fischer y Rabin borraron toda esperanza de obtener un algoritmo de tiempo polinomial.

Teorema (Fischer & Rabin): Para cualquier algoritmo de decisión para $Th(\mathfrak{R})$, determinista o no determinista, existen una constante c e infinitas oraciones φ tales, que el tiempo para decidir sobre estas últimas con el algoritmo es al menos $2^{c \cdot |\varphi|}$.

En consecuencia, la teoría de los números reales es teóricamente decidible, pero prácticamente indecidible. Este teorema establece una cota inferior de complejidad para el problema de decisión de la teoría de los números reales: todo algoritmo de decisión para esta teoría da, en el peor de los casos, una cantidad de pasos que es al menos exponencial en el largo de la entrada*.

Podemos notar la enorme cantidad de información que nos proporciona un resultado de este tipo: una cota inferior para la complejidad de un problema de decisión nos dice que no hay esperanzas de obtener un algoritmo mejor más allá de cierto límite. En la teoría de complejidad computacional, resultados sobre cotas inferiores son apetecidos y, en general, difíciles de obtener, ya que se refieren a todos los posibles algoritmos. En contraste, resultados sobre cotas superiores del tipo "El algoritmo Al tiene un tiempo para resolver el problema de decisión que no supera los $f(\mid w \mid)$ pasos, para los peores casos de w", se obtiene a través de algoritmos concretos (en este caso, Al). El problema de decisión resulta ser no más difícil que lo que le cuesta a Al resolverlo, todo esto en el peor caso. Notemos que en al tener una cota superior de complejidad,

*Esto no impide que, a pesar de este resultado negativo categórico, se busque algoritmos más eficientes que los ya existentes para decidir la teoría de los números reales, pues este problema tiene importancia práctica en "robótica".

tenemos una cuantificación existencial sobre el conjunto de todos los algoritmos que resuelven el problema dentro del tiempo dado por la cota. Mejorar cotas superiores significa, entonces, obtener algoritmos concretos más eficientes.

Como consecuencia del teorema se obtiene:

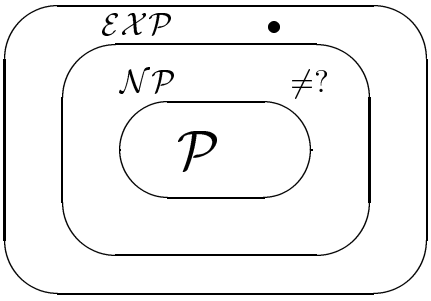
Corolario: Para cualquier sistema de axiomas AX para $Th(\mathfrak{R})$, existen infinitas oraciones $\varphi \in Th(\mathfrak{R})$ tales, que toda demostración formal de φ a partir de AX:

$$\varphi_1 \cdot \cdots \cdot \varphi_m \quad (\equiv \ \varphi)$$

tiene largo $m \geq 2^{c \cdot |\varphi|}$.

■

De paso tenemos un problema de decisión soluble que probadamente está fuera de la clase $\mathcal{NP}$:



$\mathcal{EXP}$ es la clase de problemas de decisión solubles en tiempo exponencial, y
• es el problema de decisión de $Th(\mathfrak{R})$.

7.9.1 Ejercicios

1. Demuestre que hay infinitas oraciones que son verdaderas en la estructura $\mathfrak{R}$ de los números reales para las cuales las demostraciones formales a partir de los axiomas de Tarski para la teoría de los reales, van a ser de largo exponencial en el largo de la oración.

Ind.: Use el hecho que el teorema de Fischer y Rabin también vale para algoritmos de decisión no deterministas, y el hecho que el uso deductivo y formal de un sistema de axiomas (sea el de Tarski o no) se puede ver como la aplicación de un algoritmo no determinista.

Capítulo 8

Lenguajes con Varias Especies

En aplicaciones de la lógica matemática a la misma matemática, y a ciencia de computación, en especial, a inteligencia artificial, surge la necesidad de hacer ligeramente más expresivos los lenguajes de primer orden, incorporando la posibilidad de referirse a individuos de tipos o especies distintas. A estos lenguajes los llamaremos "lenguajes con varias especies"*. Tal vez, el ejemplo más claro y típico sobre la necesidad de referirse a individuos de especies distintas que existen al mismo nivel de lenguaje, es el de la teoría de espacios vectoriales. En este caso, necesitamos hablar sobre vectores y escalares, y distinguirlos, sin privilegiar a ninguna de las dos especies sobre la otra.

En contraste con los lenguajes de primer orden usuales, en aquellos con varias especies tenemos, además de todo los que tienen los primeros, un conjunto T de tipos o especies, digamos, $\varnothing \neq T = \{t_1, \ldots, t_r\}$.

Sea S un conjunto de símbolos para lenguajes de primer orden como hasta ahora. A cada símbolo de S, se asocia un tipo o una tupla de tipos:

- Si $R \in S$ es símbolo de relación n–aria, entonces su tipo es una n–tupla de tipos (la tupla de los tipos de sus argumentos):
$$\mathit{tipo}(R) = (t_{i_1}\ ,\ \ldots\ldots\ ,\ t_{i_n})\ \in\ T^n.$$
- Si $f \in S$ es símbolo de función n–aria, entonces su tipo es una $(n+1)$–tupla (los n tipos de sus argumentos más el tipo del valor de la función):
$$\mathit{tipo}(f) = (t_{i_1}\ ,\ \ldots\ldots\ ,\ t_{i_n}\ ,\ t_{i_{n+1}})\ \in\ T^{n+1}.$$
- Si $c \in S$ es una constante individual, entonces su tipo es un elemento de T (el del individuo que ella denota):
$$\mathit{tipo}(c) = (t),\ \text{con } t\ \in\ T.$$

*En inglés se les conoce por "many–sorted languages".

Ejemplo: Podemos expresar la teoría de espacios vectoriales usando un lenguaje con dos especies. Aquí, $T = \{1, 2\}$. Escalares son de tipo 1, y vectores de tipo 2. Aquí, $S = \{+_v, *, +_e, \cdot_e, 0, 1, \mathbb{O}\}$, donde $+_v$ denota la suma de vectores; $*$, la multiplicación de un vector por un escalar; $+_e$ y $\cdot_e$, la suma y producto de escalares; 0 y 1, el cero y uno escalares; y $\mathbb{O}$, el vector cero. Se tiene:

$tipo(0) = tipo(1) = (1)$
$tipo(+_e) = tipo(\cdot_e) = (1, 1, 1)$
$tipo(+_v) = (2, 2, 2)$
$tipo(*) = (1, 2, 2)$
$tipo(\mathbb{O}) = (2)$.

■

Ahora los símbolos tienen asociado un tipo, además de la aridad que ya tenían, de hecho, la aridad puede ser leída del tipo. Además de los símbolos de S, hay una colección de variables $x_0^t, x_1^t, \ldots$, para cada tipo t. El tipo de x_i^t es (t). Los términos del lenguaje se construyen respetando los tipos.

Ejemplo (continuación): En el lenguaje formal para espacios vectoriales tenemos, además, dos tipos de variables: $x^1,\ y^1, \ldots$ y $x^2,\ y^2, \ldots$. Como dijimos, hay que respetar los tipos al formar términos, por ejemplo, $0\ +_v\ 1$ no es un término porque 0 y 1 son constantes de tipo (1), pero $+_v$ es operación de tipo $(2, 2, 2)$, y, en consecuencia, exige que sus argumentos sean de tipo (2). El tipo de un término complejo, obtenido con aplicaciones de operaciones, es el tipo del valor que entrega la última aplicación, es decir, es un elemento de T. Por ejemplo, el término $x^1 * (\mathbb{O} +_v x^2)$ es de tipo (2).

■

La definición de fórmula no ofrece grandes complicaciones. La diferencia con los lenguajes que teníamos antes se produce a nivel de fórmulas atómicas:

- Sean τ_1 y τ_2 dos téminos. Entonces $\tau_1 = \tau_2$ es fórmula si $tipo(\tau_1) = tipo(\tau_2)$.
- $R(\tau_1,\ \ldots\ ,\ \tau_n)$ es fórmula si $tipo(R) = (tipo(\tau_1),\ \ldots\ ,\ tipo(\tau_n))$.

Hasta ahora hemos considerado sólo la sintaxis de los lenguajes con varias especies. Para dar la semántica, debemos partir con las estructuras con varias

especies. Estas son de la forma:

$$\mathfrak{A} = \langle \underbrace{A^1 \ , \ \ldots \ , \ A^r}_{\text{conjuntos no vacios}} \ , \ R^{\mathfrak{A}} \ , \ \ldots \ , \ f^{\mathfrak{A}} \ , \ \ldots \ , \ c^{\mathfrak{A}} \ , \ \ldots \rangle. \qquad (*)$$

Si $tipo(R) = (j_1, \ldots, j_n)$, entonces $R^{\mathfrak{A}} \subseteq A^{j_1} \times \cdots \times A^{j_n}$. (Estamos suponiendo, por simplicidad, que los tipos son $1, \ldots, r$.)

Ejemplo (continuación): Los espacios vectoriales son estructuras de la forma:

$$\mathfrak{V} = \langle E, \ V, \ +_e^E, \ \cdot_e^E, \ +_v^V, \ *^{E,V}, \ 0^E, \ 1^E, \ \mathbb{O}^V \rangle,$$

donde, por ejemplo, $*^{E,V} : E \times V \to V$, y $\mathbb{O}^V \in V$.

Un ejemplo concreto de espacio vectorial es el de los pares de números reales sobre el cuerpo de los reales. En notación estructural, este espacio de ve así:

$$\langle \mathbb{R}, \mathbb{R}^2, \ \underbrace{+^{\mathbb{R}}, \ \cdot^{\mathbb{R}}, \ 0, \ 1}_{\text{sólo tipo 1}} \ , \ \underbrace{+^{\mathbb{R}^2}, (0,0)}_{\text{tipo 2}} \ , \ \underbrace{*_{esc}}_{\text{ambos tipos}} \ \rangle.$$

■

Una estructura con varias especies como $(*)$ determina una estructura para cada especie. Por ejemplo, la estructura correspondiente al espacio vectorial $\mathbb{R}^2$ del ejemplo anterior determina las estructuras $\mathfrak{V}^1 := \langle \mathbb{R}, \ +^{\mathbb{R}}, \ \cdot^{\mathbb{R}}, \ 0, \ 1 \rangle$ y $\mathfrak{V}^2 := \langle \mathbb{R}^2, \ +^{\mathbb{R}^2}, \ (0,0) \rangle$, que sólo involucran a individuos y operaciones de los tipos 1 y 2, respectivamente. Notar que la interpretación del símbolo $*$, que es de tipo mixto, no aparece en ninguna de estas dos estructuras.

A veces, las estructuras con varias especies son escritas así:

$$\mathfrak{A} = \langle \mathfrak{A}^1, \ \ldots\ldots, \ \mathfrak{A}^r, \ (K^{\mathfrak{A}}) \rangle,$$

donde:

- cada $\mathfrak{A}^t$ es una estructura para interpretar símbolos que involucran sólo al tipo t.
- $(K^{\mathfrak{A}})$ es una colección de interpretaciones de símbolos $K \in S$ que son de tipo mixto.

Por ejemplo, el espacio vectorial de pares de reales puede ser representado como la estructura $\mathfrak{V} = \langle \mathfrak{V}^1,\ \mathfrak{V}^2,\ *_{esc} \rangle$.

A pesar de que es bastante cómodo trabajar con lenguajes con varias especies, veremos, a conti-nuación, que, en realidad, éstos no son esencialmente más expresivos que los lenguajes de primer orden que teníamos antes. Más precisamente, veremos cómo transformar un lenguaje con varias especies a uno ordinario, sin perder poder expresivo. Esta transformación es útil también, para demostrar propiedades de los lenguajes con varias especies, apelando a propiedades ya demostradas para los lenguajes ordinarios.

El método de transformación es el de **reducción o eliminación de tipos**: se introduce en el lenguaje nuevos predicados unarios $T_1, \ldots, T_r$, uno por cada tipo básico. Intuitivamente, cada T_t será interpretado como el dominio de individuos de tipo t, es decir, como el dominio de la estructura $\mathfrak{A}^t$.

De una estructura con varias especies

$$\mathfrak{A} = \langle A^1,\ \ldots\ ,\ A^r,\ R^{\mathfrak{A}},\ \ldots\ ,\ f^{\mathfrak{A}},\ \ldots\ ,\ c^{\mathfrak{A}},\ \ldots \rangle,$$

pasamos a una estructura ordinaria

$$\mathfrak{A}' = \langle A^1\ \cup\ \ldots\ \cup\ A^r,\ A^1,\ \ldots\ ,\ A^r,\ R^{\mathfrak{A}}, \ldots\ldots \rangle.$$

En ella,

- $A^1\ \cup\ \cdots\ \cup\ A^r$ es un nuevo dominio único,
- $A^1, \ldots, A^r$ son interpretaciones para los nuevos predicados $T_1, \ldots,\ T_r$,
- los símbolos antiguos de S se interpretan como antes.

Por ejemplo, de la estructura con dos especies $\mathfrak{V} = \langle \mathbb{R},\ \mathbb{R}^2,\ \ldots\ldots \rangle$, podemos pasar a la estructura ordinaria $\mathfrak{V}' = \langle \mathbb{R} \cup \mathbb{R}^2,\ \mathbb{R},\ \mathbb{R}^2,\ \ldots\ldots \rangle$. En su dominio coexisten números reales y pares de números reales. Una manera de distinguir entre ellos a nivel de lenguaje, es a través de los predicados para cada tipo. Si en el lenguaje hemos introducido los nuevos predicados E y V para referirnos a escalares y vectores, respectivamente, entonces sus interpretaciones en $\mathfrak{V}'$ son $\mathbb{R}$ y $\mathbb{R}^2$, respectivamente.

Las fórmulas del lenguaje con varias especies deben ser reescritas en el nuevo lenguaje de primer orden ordinario que contiene, además de los símbolos de

S, los nuevos predicados $T_1, \ldots, T_r$. En este lenguaje tenemos sólo un tipo de variables, en consecuencia, el proceso de transformación debe eliminar las variables tipificadas. Para ello se recurre a los nuevos predicados para tipos. Esto se hace según las siguientes reglas:

- $\forall x^s \ \varphi$, donde x^s es variable del tipo s, se transforma en $\forall x \ (T_s(x) \rightarrow \varphi')$. Aquí x es variable de tipo único y φ' es la transformación recursiva de φ.
- $\exists x^s \ \varphi$ se transforma en $\exists x \ (T_s(x) \wedge \varphi')$.

Con esta transformación tenemos un lenguaje de primer orden común y corriente, sin especies. Por ejemplo, la fórmula

$$\forall x^1 \forall x_1^2 \forall x_2^2 \ (x^1 * (x_1^2 \ +_v \ x_2^2) = x^1 \ * \ x_1^2 \ +_v \ x^1 \ * \ x_2^2)$$

se transforma en la fórmula

$$\forall x \forall y \forall z \ (E(x) \ \wedge \ V(y) \ \wedge \ V(z) \ \rightarrow \ x * (y +_v z) = x * y \ +_v \ x * z).$$

Es fácil demostrar que la fórmula original es verdadera en una estructura con varias especies siempre y cuando la fórmula transformada es verdadera en la estructura ordinaria correspondiente*.

8.1 Ejercicios

1. Escriba de manera formal los axiomas para espacios vectoriales en un lenguaje con dos especies. En seguida, reescríbalos de acuerdo con la eliminación de tipos.

2. Consideremos un dominio consistente de individuos y stacks formados con esos individuos (en consecuencia, tenemos coexistiendo dos especies distintas). Para describir ese dominio tenemos:

*Nótese que en la fórmula inmediatamente anterior, todavía aparece la operación $+_v$, a pesar de la eliminación de tipos. Para que la transformación esté de acuerdo con toda lo que hemos visto sobre lógica de predicados sin especies, en particular, que las funciones son todas totales, es necesario, reinterpretar funciones como $+_v$ de tal modo, que estén definidas en todo el nuevo dominio común. Esto se puede hacer de manera arbitraria para aquellos argumentos nuevos, en este caso, para los escalares. Esto no es un problema, pues las fórmulas transformadas tienen sus cuantificadores relativizados a los tipos adecuados, y es para esos tipos que se quiere afirmar ciertas propiedades.

- un predicado unario *EstaVacio* que es verdadero cuando el stack está vacío
- la operación unaria *top* que indica el individuo que está arriba en el stack
- la operación unaria *pop* que entrega el stack sin su elemento de arriba
- la operación binaria *push* que entrega el stack con un individuo adicional arriba
- la constante *stackNulo* que denota al stack vacío

Axiomatize la teoría de stacks usando lógica de predicados de primer orden. Hágalo con y sin especies. La axiomatización debe ser lo más completa posible, pero sin redundancias. ¿Qué se quiere decir con estas dos últimas calificaciones? Precise en términos lógicos.

8.2 El Cálculo de Situaciones

El cálculo de situaciones fue concebido por John McCarthy con el propósito de representar conocimiento en inteligencia artificial relativo mundos que están en evolución, pasando por diferentes estados, como resultado de la ejecución de ciertas acciones identificadas con nombres.

Pareciera ser que, al haber evolución, necesitamos referirnos explícitamente al tiempo. Esta idea se ha materializado a través de ciertas lógicas para raciocinio temporal. En ellas

- Hay representación explícita del tiempo.
- Eventos y acciones están asociadas a tiempos (en los cuales se producen o ejecutan).
- Una cosa ocurre antes que otra si el tiempo en que ocurre la primera es anterior al tiempo en que ocurre la segunda.
- Aunque no ocurra nada, el tiempo pasa.
- Esto acarrea problemas. ¿Qué consideramos: tiempo discreto, continuo, denso, intervalos, instantes, etc.?

- Por ejemplo, si pensamos en el tiempo como la recta real, para hacer raciocinio temporal, habría que agregar una axiomatización formal de la recta real, la que no es fácil de manejar desde el punto de vista computacional (recordar el teorema de Fischer & Rabin).

Sin embargo, McCarthy parte de otra base:

- A veces, no es de interés ni utilidad tener una representación explícita del tiempo para re-presentar conocimiento y razonar sobre mundos en evolución.
- Para muchos propósitos estamos más interesados en cadenas de eventos, que en sucesiones de instantes.
- En el contexto de planificación automática en inteligencia artificial, por ejemplo, lo que nos importa son los cambios en el estado del mundo que se producen por acciones específicas, con el propósito de determinar de manera automática una sucesión de acciones que lleve el estado del mundo a uno deseado*.
- El mundo se concibe como estando en un cierto estado, y este estado puede cambiar sólo cuando un agente ejecuta una acción.
- Podemos razonar en términos de sucesiones de estados, donde la aparición de uno posterior a otro en la sucesión, significa que se obtuvo del anterior a través de una sucesión de acciones concatenadas. Esto lo podemos visualizar como en la figura A8.1.

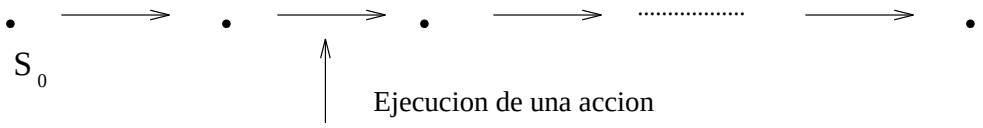


Figura A8.1:

- El transcurso del tiempo está representado implícitamente, de hecho, no necesitamos pensar en tiempo (pero puede ayudar a la intuición).

*Esta actividad se llama "planning", en inglés.

- De este modo, no tiene importancia si el tiempo es discreto, continuo, denso, tampoco el cuantificarlo para saber cuánto dura una acción o por cuánto tiempo se prolonga un estado en el tiempo[†].

Más formalmente, el cálculo de situaciones es un lenguaje de primer orden con varias especies. Lo más típico es tener los tipos "objeto", "situación" (o "estado") y "acción", que denotamos con "*ob*", "*sit*", "*acc*".

Como en todo lenguaje de primer orden, tenemos predicados, operaciones y constantes, todas con tipos asociados:

- S_0 es constante individual de tipo *sit* y denota la **situación inicial**.
- Los **fluentes** son predicados de tipo $(ob, \ldots, ob,\ sit)$, es decir, el último argumento es de tipo "situación".

  Por ejemplo, el predicado $Color(x, y, s)$ es un fluente que denota el hecho que el objeto x tiene color y (otro objeto) en el estado s. En otros téminos, los fluentes son propiedades de objetos que dependen del estado del mundo. De este modo, el predicado, con los mismos objetos, puede tomar distintos valores de verdad según la situación. En ellos está presente el dinamismo y el cambio.
- Las **acciones** son operaciones de tipo $(ob,\ \cdots, ob) \longrightarrow acc$.*

  Por ejemplo, la operación *sobre* es una acción y es de tipo $(ob,\ ob) \longrightarrow acc$. Ella tiene dos parámetros, y $sobre(x, y)$ es un término de tipo "acción" que denota la acción de poner el objeto x sobre el objeto y.

  Vemos que las acciones pueden ser parametrizadas. Cuando no lo son, tenemos constantes de tipo acción. Por ejemplo, *cantar* (sin especificar quién ni qué) podría ser una constante de tipo *acc*.
- Una operación distinguida importante es "*ejecutar*"[†]. Ella es de tipo $(acc,\ sit) \longrightarrow sit$. En consecuencia, $ejecutar(a, s)$ es un término de tipo "situación", y denota la situación que resulta de ejecutar la acción a en la situación s.

  Por ejemplo, $ejecutar(sobre(x, y), s)$ es el término que denota la situación que resulta de poner x sobre y en la situación s.

[†]Sin embargo, últimamente se ha hecho extensiones del cálculo de situaciones con el propósito de agregar tiempo y duración a las acciones y eventos.

*Deberíamos escribir de tipo $(ob,\ \cdots, ob,\ acc)$, pero esta notación no enfatiza el hecho que una acción envía una tupla de objetos (los parámetros de la acción) a algo de tipo "acción".

[†]En la literatura en inglés se acostumbra denotarla con "*do*" o "*result*".

- Además, hay **variables** para cada una de las especies:

  $x^1,\ x^2 \cdots\cdots$ (de tipo "objeto"),
  $s^1,\ s^2 \cdots\cdots$ (de tipo "situación"),
  $a^1,\ a^2 \cdots\cdots$ (de tipo "acción").

De este modo, tenemos un lenguaje de primer orden con al menos tres especies*. En representación de conocimiento es útil manejarlo como lenguaje con varias especies, sin hacer reducción de tipos.

Las interpretaciones o estructuras serán entonces de la forma:

$$\mathfrak{M} = \langle O, S, A,\ \mathit{ejecutar}^{\mathfrak{M}}, S_0^{\mathfrak{M}}, P^{\mathfrak{M}}, \cdots \rangle,$$

donde,

- O es el dominio de objetos
- S es el dominio de situaciones, en particular, $S_0^{\mathfrak{M}} \in S$
- A es el dominio de acciones
- $\mathit{ejecutar}^{\mathfrak{M}}$ es una función que va de $A \times S$ a S
- Si P es un fluente con $n+1$ argumentos (el último de tipo situación), entonces $P^{\mathfrak{M}}$ es una colección $(R_s(\cdot, \ldots, \cdot))_{s \in S}$ de relaciones n-arias sobre objetos en O, es decir, para cada situación $s \in S$, $R_s \subseteq O^n$, y nos dice cuáles n-tuplas de objetos de O tienen la propiedad P (en la situación s).

En interesante observar que objetos, acciones y situaciones coexisten como individuos a un mismo nivel en el lenguaje y en las estructuras. En particular, es posible cuantificar sobre cada uno de ellos, manteniéndonos dentro del lenguaje de primer orden. Esta facilidad se ha mostrado como extrordinariamente útil en representación de conocimiento y "planning" en el contexto de la inteligencia artificial; y en especificación de transacciones en bases de datos relacionales. Esto último no es sorprendente si pensamos que las bases de datos son mundos en evolución que pasan por sucesivos estados, como resultado de las transacciones de actualización que efectuamos sobre ellas. Usualmente estamos más interesados en los cambios, que en los tiempos en que éstos se producen.

*Podría ser que nos interese subdividir el dominio de objetos en subdominios, por ejemplo, diferenciando objetos materiales de los colores.

8.3 Una Aplicación a Bases de Datos

A continuación veremos un ejemplo muy simplificado de especificación formal de la dinámica de una base de datos bibliográfica por medio del cálculo de situaciones.[†].

Por simplicidad, cada libro que puede ser comprado a una editorial o que aparece en la base de datos inicial se supondrá incluido en la tabla global, constante, *BooksInPrint*, que puede ser interpretada como un predicado. En un ejemplo más realista, se podría interpretar como un fluente porque cambia en el tiempo (o de estado en estado). Inicialmente hay libros *Clasificados* y *NoClasificados*; y estas propiedades son fluentes. Los que están clasificados tienen un número de identificación *id*, que incluye un número *ISBN* y el número de *copias* (de modo que copias diferentes de un mismo libro tienen distinto *id*). Los libros *NoClasificados* son los que hemos comprado, pero no han sido clasificados aún. Finalmente, hay un fluente (tabla) *Stock* que registra el número de ejemplares de cada libro clasificado.

Predicado (tabla estática): $BooksInPrint(isbn, titulo, autor, editoral, año, edicion)$.

Fluentes (tablas dinámicas): $Clasificados(isbn, id, s)$, $Stock(isbn, cant, s)$, $NoClasificados(isbn, copias, s)$, $LibroPerdido(id, s)$, $Agotado(isbn, s)$.

Acciones de actualización (transacciones atómicas): $eliminarLibro(id)$, $clasificarLibro(isbn, id)$, $solicitarCompra(isbn, copias)$. Abajo se describe cada acción:

eliminarLibro Disminuye el *Stock* en uno. Cuando la cantidad *cant* es 0, elimina el libro de las tablas *Clasificados*, *Stock* y *LibroPerdido*.

clasificarLibro Asigna un *id* al libro en *NoClasificados* y elimina el libro de *NoClasificados*, y lo inserta en *Clasificados*; también actualiza el *Stock* de acuerdo con el número de *copias*.

[†]Esta especificación se ajusta a una proposición general de Ray Reiter para especificaciones de bases de datos en el cálculo de situaciones. Ver R. Reiter, "On Specifying Database Updates", J. Logic Programming, 1995.

solicitarCompra Agrega un ítem de *BooksInPrint* que no está *Agotado* en *NoClasificados*.

Los efectos de las transacciones atómicas han sido descritos de manera informal. Más adelante daremos cuenta de de ellos en nuestra especificación formal, a través de los "axiomas de estado sucesor". Antes de eso, daremos los axiomas que especifican cuándo las transacciones atómicas son posibles de ejecutar. Introducimos un predicado binario, Pos, de tipo (acc, sit):

Precondiciones para Acciones:*

$$Pos(eliminarLibro(id), s) \equiv LibroPerdido(id, s)$$

$$Pos(solicitarCompra(isbn, copias), s) \equiv (\exists\ titulo, autor, editorial, año, edicion$$
$$BooksInPrint(isbn, titulo, autor, editorial, año, edicion) \wedge copias > 0)$$

$$Pos(clasificarLibro(isbn, id), s) \equiv (\exists\, copias \quad NoClasificados(isbn, copias, s)$$
$$\wedge\ \neg\ \exists\, isbn'\ \ Clasificados(isbn', id, s))$$

Axiomas de Estado Sucesor: Los axiomas de estado sucesor proveen las condiciones necesarias y suficientes para que un fluente se haga verdadero en un estado sucesor, es decir, alcanzado después de ejecutar una acción arbitraria. Para que esas condicones sean válidas, también deben ser satisfechas las precondiciones para la ejecución de esa acción. Habrá un axioma por cada fluente. Por ejemplo, para *Clasificados*, *LibroPerdido* y *Agotado* tenemos:

$$\forall\, a, s\ [\ Pos(a, s) \rightarrow Clasificados(isbn, id, ejecutar(a, s)) \equiv$$
$$(a = clasificarLibro(isbn, id) \vee (Clasificados(isbn, id, s) \wedge$$
$$a \neq eliminarLibro(id)))\ ]$$

$$\forall\, a, s\ [\ Pos(a, s) \rightarrow LibroPerdido(id, ejecutar(a, s)) \equiv (LibroPerdido(id, s) \wedge$$
$$a \neq eliminarLibro(id))\ ]$$

*En estos axiomas y los que sigan, las variables que no aparecen explícitamente cuantificadas deben entenderse como cuantificadas universalmente al principio de la fórmula.

$\forall\, a, s\ [\ Pos(a, s)\ \rightarrow\ Agotado(isbn, ejecutar(a, s))\ \equiv\ Agotado(isbn, s)\ ]^{\dagger}$

También hay axiomas de estado sucesor para *Stock* y *NoClasificados*, pero los omitimos aquí.

Notar que los efectos de las acciones, descritos de manera informal anteriormente, están presentes en los axiomas de estado sucesor. Lo más interesante es que estos axiomas, además, aseguran que los únicos cambios que se pueden dar son los que corresponden a esos efectos, todo lo demás permanece constante de un estado a otro[‡].

La especificación incluye, además,

Axiomas de Nombres Unicos para Transacciones Atómicas:
$\forall\, x \forall\, y \forall\, z\ solicitarCompra(x, y) \neq eliminarLibro(z)$, etc.

Axiomas de Nombres Unicos para Estados:
$\forall\, a, s\ S_0 \neq ejecutar(a, s), \forall\, a, s, a', s'\ (ejecutar(a, s) = ejecutar(a', s')\ \rightarrow\ a = a' \wedge s = s')$.

Por supuesto, la especificación contiene, además, una especificación de la base de datos inicial, es decir, de los predicados y de los fluentes en el estado S_0.

Dado que tenemos una especificación de la dinámica de una base de datos, es natural preguntarse por el estatus de las restricciones de integridad, que son propiedades que deben ser satisfechas en cada estado legal de la base de datos. Este tipo de especificaciones están dadas sobre el supuesto de que las restricciones de integridad deben ser consecuencias lógicas (demostrables a partir) de la especificación presente.

Para precisar la idea de estado legal de la base de datos, se define un predicado unario de "legalidad", *Legal*, de tipo *sit*. Intuitivamente, un estado es legal si

---

[†]Aquí el fluente *Agotado* aparece estático. La razón es lo simplificado de nuestro ejemplo. Si introdujéramos otras acciones, por ejemplo, una que informe que un libro se agotó, tendríamos un axioma más interesante para este fluente.

[‡]Esto resuelve de una manera elegante y económica el llamado "problema del marco" que veremos en más detalle en el capítulo 10, y que consiste, en esencia, en poder especificar de manera económica todo lo que **no** cambia cuando se ejecuta una acción. Como usualmente son pocas las cosas que cambian y muchas las que no cambian, el especificar lo que no cambia es, en principio, largo, tedioso, y va en contra del sentido común. Sin embargo, al computador hay que darle, de alguna manera, ese conocimiento de sentido común o darle algún mecanismo que le permita obtenerlo.

se puede llegar a él desde la situación inicial, S_0, por medio de la ejecución de una sucesión finita de acciones atómicas que son posibles (en el correspondiente estado de ejecución). Nótese la similitud con los números naturales, que son obtenidos a partir del cero mediante la ejecución de la acción "sumar 1". No es extraño que este predicado se defina mediante un axioma de inducción (de segundo orden), que dice que *Legal* es el menor conjunto de estados que contiene a S_0, y que, cada vez que contiene un estado s, entonces también contiene al estado $ejecutar(a, s)$, cuando la acción a es posible en s:

$$\forall P\ [\ P(S_0) \wedge \forall\, a, s(P(s) \wedge Pos(a, s) \rightarrow P(ejecutar(a, s))) \ \rightarrow$$
$$\forall\, s(Legal(s) \rightarrow P(s))\ ].$$

Las restricciones de integridad serán, entonces, fórmulas de la forma

$$\forall\, s(Legal(s) \rightarrow \varphi(s)),$$

donde φ es la propiedad que nos interesa satisfacer en todo estado legal de la base de datos.

En nuestro ejemplo, una restricción de integridad deseable dice que el predicado *Clasificados* es funcional en el segundo argumento, es decir, que no hay dos libros con distinto *ISBN* que tengan el mismo *id*:

$$\forall\, s\ [Legal(s) \rightarrow (Clasificados(isbn1, id, s)\ \wedge\ Clasificados(isbn2, id, s)\ \rightarrow$$
$$isbn1 = isbn2)].$$

De acuerdo con nuestro enfoque, esta restricción debería ser consecuencia lógica (en el sentido usual) de la especificación que dimos, incluyendo el axioma de inducción (y si no lo es, debería intentarse cambiar la especificación, que no está dando cuenta de la restricción deseada).

En esta especificación de la dinámica de una base de datos, hemos usado todo el poder de la cuantificación de primer orden sobre situaciones y acciones. Además del poder expresivo que nos da este lenguaje del cálculo de situaciones, contamos con una semántica para la especificación que es muy clara y está muy bien estudiada, a saber, la que nos provee la lógica de primer orden. Más aún, esta especificación es susceptible de ser analizada con métodos matemáticos y formales. En particular, algunas tareas de análisis y otras se obtención de nuevo conocimiento a partir de la especificación pueden ser automatizadas, usando, por ejemplo, demostradores mecánicos de teoremas.

8.3.1 Ejercicios

Los ejercicios se refieren a la especificación de la base de datos bibliográfica dada en esta sección.

1. (a) Demuestre que la ejecución de la acción *solicitarCompra* no cambia el valor de verdad del fluente *Clasificados* desde el estado de ejecución al estado sucesor.

(b) ¿De qué manera está usando los axiomas de nombres únicos para acciones en esta demostración?

2. Demuestre, por inducción en los estados (legales), la restricción de integridad indicada.

3. Dé los axiomas de estado sucesor que faltan en la especificación.

4. Invente nuevas acciones y modifique la especificación de manera correspondiente.

Capítulo 9

Programación en Lógica

por Raymond Reiter*

9.1 Introducción

Ordinariamente, una colección de fórmulas lógicas es vista como una representación *estática* o *declarativa* de las verdades sobre algún mundo o estado de cosas. Como tal, éstos son hechos puramente descriptivos. Para el científico de la computación, una de las características más intrigantes de la lógica de primer orden es que a tales formulas se les puede dar una interpretación *procedural*, es decir, ellas se pueden ver como *programas ejecutables*. De hecho, un lenguaje de programación, llamado PROLOG, está basado justamente en este punto de vista. PROLOG (por PROgramming in LOGic) es usado ampliamente para aplicaciones de programación no–numéricas en Inteligencia Artificial, bases de datos, matemática simbólica, comprensión de lenguaje natural, etc. Además, no es como ningún otro lenguaje de programación y es, en consecuencia, de interés para el científico de la computación de manera completamente independiente de sus aplicaciones.

Básicamente, un programa en PROLOG es una colección de fórmulas de primer orden de cierta forma restringida. La entrada para tal programa es un teorema para ser demostrado usando estas fórmulas como premisas. Los resultados de ejecutar el programa son obtenidos como efectos laterales de una demostración del teorema. En consecuencia, el interprete de PROLOG es un demostrador de teoremas.

El propósito de este capítulo es precisar estas ideas, investigar algunas de las

*Department of Computer Science, University of Toronto, Toronto M5S 1A4, Canada.

características más interesantes de PROLOG como lenguaje de programación, y explorar algunas de sus aplicaciones. En el camino el lector puede sufrir un impacto cultural al ver girar sus conceptos sobre la naturaleza de la programación en 180^o. Esto sólo puede ser bueno, aunque un poco inconfortable.

9.2 Demostraciones Descendentes

Empecemos por representar en lógica de primer orden las características espaciales de una colección de bloques sobre una mesa, como en la figura A9.1.

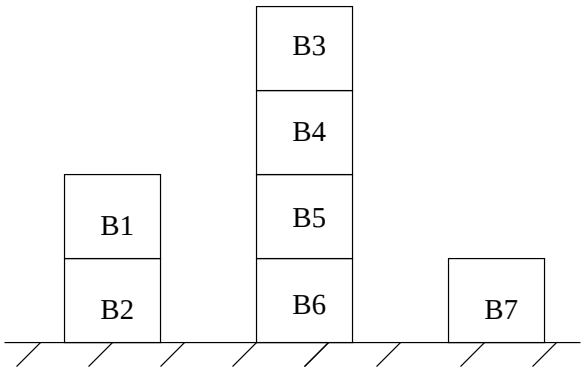


Figura A9.1:

Las relaciones espaciales de interés serán:

$Encima(x, y)$ - El bloque x está encima del (tocando al) bloque y.
$Arriba(x, y)$ - En la pila de bloques que contiene al bloque y, el bloque x está arriba del (no necesariamente tocando al) bloque y.
$IzquierdaDe(x, y)$ - El bloque x está a la izquierda del bloque y.
$DerechaDe(x, y)$ - El bloque x está a la derecha del bloque y.

Suponemos que, inicialmente, todas las instancias de la relación *Encima* están especificadas, y que la relación *IzquierdaDe* está especificada por aquellos pares adyacentes de bloques que están sobre la mesa:

$$\left.\begin{array}{ll} Encima(B1, B2) & Encima(B3, B4) \\ Encima(B4, B5) & Encima(B5, B6) \\ IzquierdaDe(B2, B6) & IzquierdaDe(B6, B7) \end{array}\right\} \quad (9.0)$$

Ahora, deseamos especificar axiomas que nos permitirán las relaciones existentes entre todos los otros pares de bloques en la escena. Por ejemplo, queremos ser capaces de determinar que $Arriba(B3, B6)$, $IzquierdaDe(B1, B7)$,

$DerechaDe(B3, B1)$, etc. Una de tales fórmulas es la siguiente, donde por brevedad omitimos cuantificadores universales, de modo que todas las variables se entienden como cuantificadas universalmente:

$$Arriba(x, y) \ \& \ IzquierdaDe(y, z) \ \rightarrow \ IzquierdaDe(x, z) \quad (9.1)$$
$$Arriba(y, z) \ \& \ IzquierdaDe(x, z) \ \rightarrow \ IzquierdaDe(x, y) \quad (9.2)$$
$$IzquierdaDe(x, y) \ \& \ IzquierdaDe(y, z) \ \rightarrow \ IzquierdaDe(x, z) \quad (9.3)$$
$$IzquierdaDe(x, y) \ \rightarrow \ DerechaDe(y, x) \quad (9.4)$$
$$Encima(x, y) \ \rightarrow \ Arriba(x, y) \quad (9.5)$$
$$Encima(x, y) \ \& \ Arriba(y, z) \ \rightarrow \ Arriba(x, z) \quad (9.6)$$

Ahora, usando las fórmulas (9.0) – (9.6), podemos, usando resolución, demostrar varias cosas sobre la figura A9.1, por ejemplo, que $Arriba(B3, B6)$. En lugar de eso, vamos a proponer un procedimiento de prueba diferente de resolución para establecer tales teoremas. De hecho, este nuevo procedimiento de prueba sólo se verá distinto de resolución. Como veremos más tarde éste realmente corresponde a un tipo especial de estrategia de resolución lineal.

Entonces, consideremos el tratar de establecer $Arriba(B3, B6)$ – nuestra *meta* de más alto nivel*. Este no es un hecho de (9.0), de modo que no podemos alcanzar esta meta usando estas fórmulas. ¿Qué fórmulas podrían contribuir a alcanzar esta meta? Las únicas fórmulas que involucran al predicado *Arriba* al lado derecho de un signo de implicación son (9.5) y (9.6). Concentremonos en (9.5). Esta puede ser leída así: Si usted puede alcanzar la *submeta* $Encima(x, y)$ para valores de las variables x e y, usted habrá alcanzado la meta $Arriba(x, y)$. O, equivalentemente: para alcanzar la meta $Arriba(x, y)$, es suficiente alcanzar la submeta $Encima(x, y)$.

De manera similar, (9.6) nos dice: para alcanzar la meta $Arriba(x, z)$, es suficiente alcanzar la submeta $Encima(x, y)$ & $Arriba(y, z)$.

Luego, podemos usar las fórmulas (9.5) y (9.6) para *reducir* nuestra meta original $Arriba(B3, B6)$ al problema de establecer una de dos posibles submetas, como en la figura A9.2, donde hemos etiquetado las ramas con aquellos números de fórmulas que conducen a las submetas.

Nótese cómo se obtiene estas submetas. Hemos determinado aquellas fórmulas cuyos lados derechos unifican con la meta original, aplicamos la substitución unificadora a los lados izquierdos, y establecimos los lados izquierdos correspondientes como nuevas posibles submetas.

---

*top level *goal*. (Todas las notas al pie de página en este capítulo son de L. Bertossi)

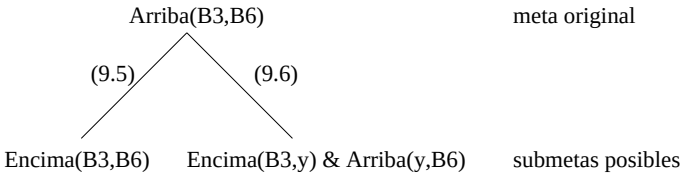


Figura A9.2:

Nuestro problema ahora es establecer una de las dos posibles submetas de la figura A9.2. El enfoque natural es repetir el proceso de generación de submetas para generar submetas de éstas en forma adicional. Entonces, concentrémonos primero en la submeta izquierda, $Encima(B3, B6)$ de la figura A9.2. Esta no es una de las fórmulas (9.0). Tampoco unifica con con el lado derecho de ninguna de las fórmulas (9.1) - (9.6). Esta submeta *falla*. Luego, debemos concentrarnos en la submeta derecha, $Encima(B3, y)$ & $Arriba(y, B6)$, de la figura A9.2. Esto puede ser establecido así:

1. estableciendo $Encima(B3, y)$ para algún valor de la variable y, y entonces
2. estableciendo $Arriba(y, B6)$ para ese valor de y.

Ahora, $Encima(B3, y)$ puede ser establecida usando (9.0) con $y = B4$, dejándonos con la tarea de establecer $Arriba(B4, B6)$ como en la figura A9.3.

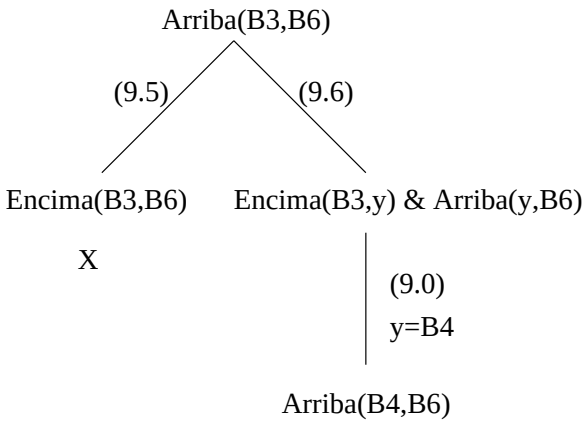


Figura A9.3:

El patrón general debería ahora estar claro. La figura A9.4 muestra el árbol meta–submeta completo bajando hasta el primer nivel donde se obtiene una solución exitosa.

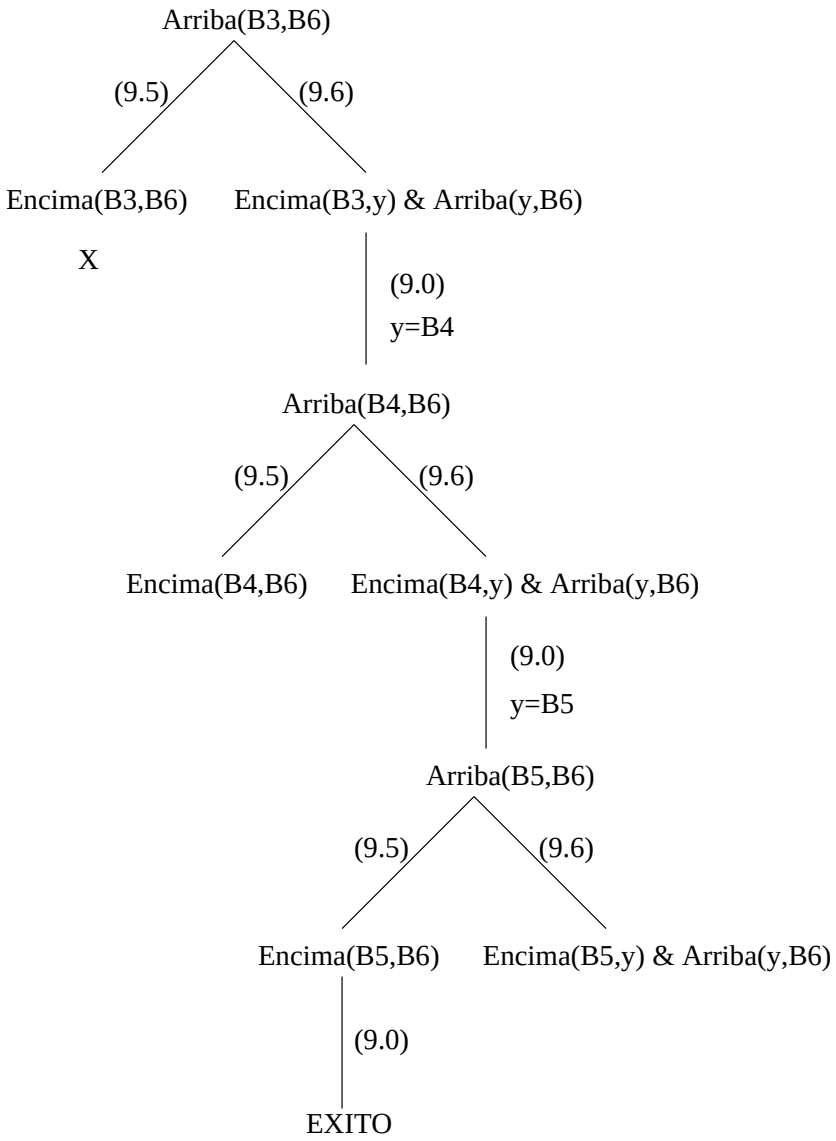

Arriba(B3,B6)
(9.5)
(9.6)
Encima(B3,B6)
Encima(B3,y) & Arriba(y,B6)
X
(9.0)
y=B4
Arriba(B4,B6)
(9.5)
(9.6)
Encima(B4,B6)
Encima(B4,y) & Arriba(y,B6)
(9.0)
y=B5
Arriba(B5,B6)
(9.5)
(9.6)
Encima(B5,B6)
Encima(B5,y) & Arriba(y,B6)
(9.0)
EXITO

Figura A9.4:

Como ejemplo adicional, la figura A9.5 muestra el árbol meta–submeta completo para una demostración de $DerechaDe(B7, B2)$. Los nodos en el árbol fueron generados de un nivel a la vez en el orden indicado por los números a lado de cada nodo, hasta que un nodo exitoso fue obtenido. Hay varios puntos notables en ese árbol:

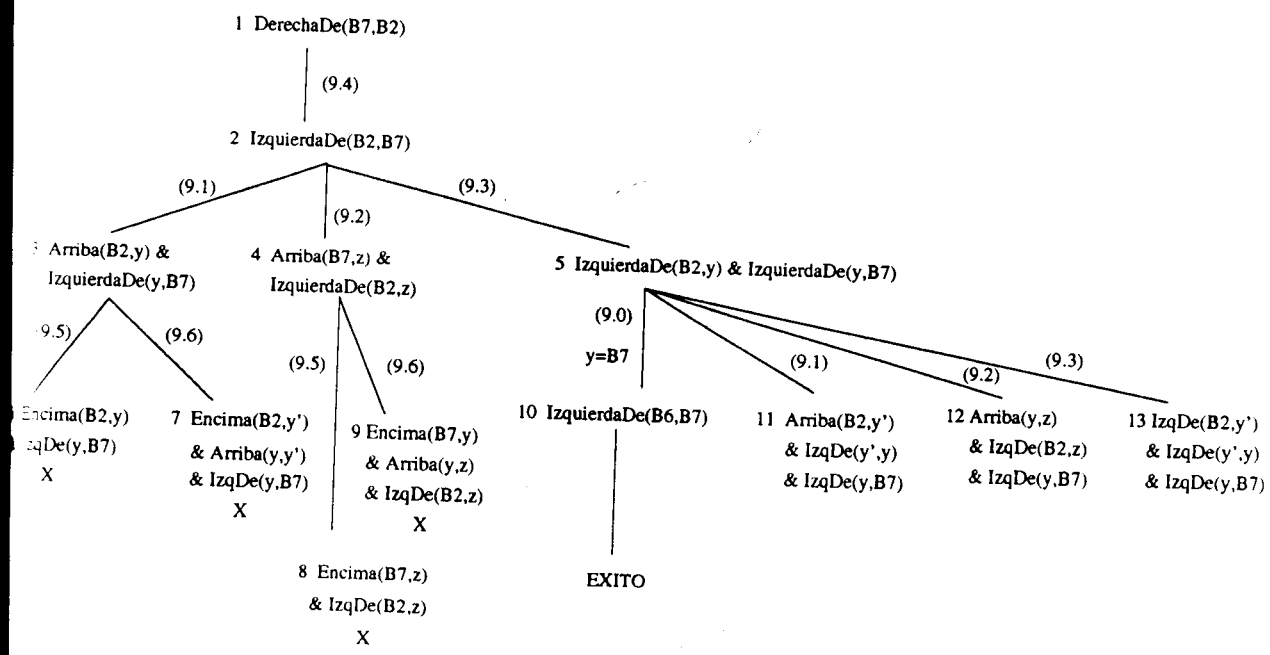


Figura A9.5:

1. Cuando se procesa una meta *conjuntiva*, es decir, una fórmula de la forma A_1 & $A_2 \cdots$ & A_n con $A_1, \ldots, A_n$ literales y con $n \geq 2$, es el *primer* literal A_1 que tratamos de solucionar. Luego, al establecer el nodo 9 como submeta del nodo 4, nos focalizamos sobre el primer literal del nodo 4, $Arriba(B7, x)$. En seguida, determinamos su submeta $Encima(B7, y)$ & $Arriba(y, z)$, usando la formula (9.6). Finalmente reemplazamos $Arriba(B7, x)$ del nodo 4 por su submeta para obtener el nodo 9, $Encima(B7, y)$ & $Arriba(y, z)$ & $IzquierdaDe(B2, y)$. Similarmente, procesando el nodo 9, nos focalizamos sobre su primer literal, $Encima(B7, y)$ y notamos que éste falla, de modo que el nodo falla. Este ejemplo ilustra que una convención a la cual vamos a adherirnos siempre: *Cuando se procesa un nodo con una meta conjuntiva, trátese de solucionar su primer literal. Cualquier submeta de este primer literal reemplaza ese literal al establecer la submeta del nodo original.*

2. Considérese el proceso de establecer el nodo 12 como submeta del nodo 5. Como se describió arriba, primero determinamos la submeta $Arriba(y, x)$ & $IzquierdaDe(B2, y)$ del primer literal, $IzquierdaDe(B2, y)$, del nodo 5. Pero esta submeta es equivalente a $IzquierdaDe(B2, z)$ & $Arriba(y, z)$, de modo que el nodo 12 también podría ser etiquetado con $IzquierdaDe(B2, z)$ & $Arriba(y, z)$ & $IzquierdaDe(y, B7)$. Para resolver tales posibles ambigüedades, adoptamos la siguiente convención:

   Al generar una submeta de un nodo, reténgase el orden de los literales que aparecen al lado izquierdo de un signo de implicación de cualquier axioma usado en la creación de la submeta.

3. Nótese que el nodo 7, una de las submetas del nodo 3, involucra una nueva variable y'. Esta submeta fue obtenida del nodo 3, usando el axioma (9.6), que es $Encima(x, y)$ & $Arriba(y, z) \rightarrow Arriba(x, z)$. Este axioma involucra una variable y, tal como el nodo 3. Pero estas dos y's no tienen nada en común; ellas son simplemente variables a las cuales se les dio el mismo nombre. Para evitar confundirlas, se renombra la y del axioma (9.6) con y'. Usando esta versión renombrada del axioma (9.6) se obtiene el nodo 7 como submeta del nodo 3.

En este punto es apropiado formalizar las ideas que hemos desarrollado. Para empezar, seremos precisos sobre la clase de axiomas que vamos a permitir. Una fórmula se llama *cláusula de Horn positiva* si no tiene cuantificadores, y tiene la forma P, o la forma P_1 & $\cdots$ & $P_n \rightarrow P$, donde $P, P_1, \ldots, P_n$ son todas fórmulas atómicas. Una fórmula de Horn positiva puede tener, y

usualmente tendrá, variables libres. Estas variables se entienden como cuantificadas universalmente. Por ahora, nos preocuparemos sólo de axiomas que son todos cláusulas de Horn positivas, como en el ejemplo del mundo de los bloques. Además, los únicos teoremas que vamos a considerar para probar a partir de los axiomas serán conjunciones T_1 & $\cdots$ & T_n de fórmulas atómicas T_i. Los teoremas pueden contener variables libres. La interpretación de tales variables se discutirá más tarde.

Ahora podemos formalizar el proceso de generación de submetas en la forma que sigue:

Sea G_1 & G_2 & $\cdots$ & G_n la meta actual. (Inicialmente, éste será el teorema a demostrar.)

1. Supongamos que la fórmula atómica G es un axioma sin variables en común con las de G_1 & G_2 & $\cdots$ & G_n. Si G tiene tales variables en común, renómbrese la variables de G en forma apropiada. Supongamos que G unifica con G_1 con $umg\,\sigma$. Entonces $(G_2$ & $\cdots$ & $G_n)\sigma$ es una submeta de G_1 & G_2 & $\cdots$ & G_n. Si $n = 1$, entonces EXITO es una submeta.

2. Supongamos que la cláusula de Horn positiva A_1 & $\cdots$ & $A_m \rightarrow G$ es un axioma cuyas variables han sido renombradas si es necesario para que difieran de las de G_1 & $\cdots$ & G_n. Supongamos también que G unifica con G_1 con $umg\,\sigma$. Entonces $(A_1$ & $\cdots$ & A_m & G_2 & $\cdots$ & $G_n)\sigma$ es una submeta de G_1 & G_2 & $\cdots$ & G_n.

3. G_1 & G_2 & $\cdots$ & G_n no tiene otras submetas excepto las determinadas por 1. y 2.

Diremos que la sucesión de conjunciones $C_1, C_2, \ldots, C_n$ es una demostración descendente* de C_1 a partir de un conjunto de axiomas de Horn positivos si:

1. $C_n =$ EXITO
2. Para $i = 2, \ldots, n$, cada C_i es una submeta de C_{i-1}.

En el ejemplo, de la figura 9.5, la sucesión $DerechaDe(B7, B2)$, $IzquierdaDe(B2, B7)$, $IzquierdaDe(B2, y)$ & $IzquierdaDe(B6, B7)$, EXITO es una demostración

*Top–Down Proof, en inglés

descendente de $DerechaDe(B7, B2)$ a partir de los axiomas de Horn positivos (9.0) – (9.6). Uno puede pensar una demostración descendente como una trayectoria en el árbol desde el nodo superior (top), el teorema a ser demostrado, hasta el nodo EXITO.

Las demostraciones descendentes formalizan un proceso psicológico natural de reducción de problemas. Uno comienza con un problema a ser resuelto (es decir, un teorema a ser demostrado). En seguida uno se pregunta si este problema puede ser reducido a un subproblema cuya solución condiciría a una solución del problema original. Determinar tales posibles subproblemas es justamente el proceso de generación de submetas. En seguida se repite este proceso con el subproblema, generando subproblemas adicionales, hasta que se alcanza subproblemas cuyas soluciones son "obvias". En el caso de demostraciones descendentes, soluciones "obvias" son aquellas que involucran axiomas que son fórmulas atómicas.

La reducción de problemas, o la demostración descendente, forma la base de muchos sistemas resolvedores de problemas en Inteligencia Artificial. Esta yace en el corazón del primer programa computacional para hacer problemas de geometría de colegio (H. Gelernter). Es una componente medular de programas más recientes para hacer matemáticas avanzadas como teoría de conjuntos, topología y análisis (W. Bledsoe). Es el principal mecanismo de sistemas expertos para diagnóstico médico (MYCIN), para interpretación de muestras geológicas para predecir depósitos de minerales (PROSPECTOR), y para realizar análisis químicos (DENDRAL). Y, como veremos, las demostraciones descendentes caracterizan al intérprete para el lenguaje de programación PROLOG.

9.2.1 Ejercicios

1. Demuestre que un conjunto de cláusulas de Horn positivas es consistente.

Ind.: Elija una estructura que haga a cada átomo verdadero.

9.3 Buscando Demostraciones Descendentes

Es claro de los ejemplos anteriores que en general es necesario *buscar* una demostración descendente de una fórmula meta. No toda rama que sale de un nodo en el árbol va a conducir a una demostración exitosa. Como usualmente no sabemos por adelantado qué rama va a ser exitosa, debemos estar preparados para explorarlas todas de manera sistemática. La forma particular en que

vamos eligiendo las ramas a explorar define una *estrategia de búsqueda*.

En los ejemplos anteriores, hemos estado usando una estrategia de búsqueda particular llamada *busqueda primero en anchura* *. Según esa estrategia, se empieza con el nodo meta, que es el único nodo en el nivel 1 del árbol. En seguida se genera todas sus posibles submetas, lo que define los nodos de nivel 2 en el árbol. A continuación, para cada nodo de nivel 2, se genera todos sus posibles nodos submetas (si los hay). La colección de todos tales nodos define el nivel 3 del árbol, etc. En general, se genera todos los nodos de nivel i en el árbol antes de generar los de nivel $i + 1$. Este proceso termina cuando se genera un nodo exitoso.

La búsqueda primero en anchura tiene dos ventajas:

1. Si existe una demostración descendente, ésta será encontrada.
2. La primera tal demostración está garantizada de ser una demostración del largo más corto.

La búsqueda primero en anchura tiene dos desventajas:

1. El árbol generado con esta estrategia crece exponencialmente, de modo que para demostraciones descendentes largas se requiere considerable memoria computacional.
2. En la eventualidad que un nodo exitoso es uno que está entre los nodos más a la izquierda, los nodos más a la derecha habrán sido inútiles. La figura A9.5 es un modesto ejemplo de esta situación. Los nodos 11, 12 y 13 no habría sido necesario generarlos.

Por estas desventajas, y también porque puede ser eficientemente implementada, se usa a menudo una estrategia diferente, llamada *búsqueda primero en profundidad*†. Aquí la idea es generar el árbol de posibilidades generando primero las ramas más a la izquierda. Si alguna vez se alcanza un nodo fallido, entonces se hace *backtracking* al nodo previo y se genera una rama diferente a partir de él, si es posible. Si se ha intentado con todas las ramas a partir de ese nodo previo y han fallado, entonces se hace backtracking al nodo previo a ése y se repite el proceso. En ciencia de computación a menudo se hace referencia

*breadth first search, en inglés

†depth first search, en ingl' es

a esta manera de generar el árbol de posibilidades como *travesía pre-orden*[‡] del árbol. La figura A9.6 indica cómo una búsqueda primero en profundidad generaría el nodo EXITO para el ejemplo de la figura A9.5. Las operaciones de backtracking están indicadas con flechas punteadas, y el orden en que son generadas las ramas está indicado por números en las ramas.

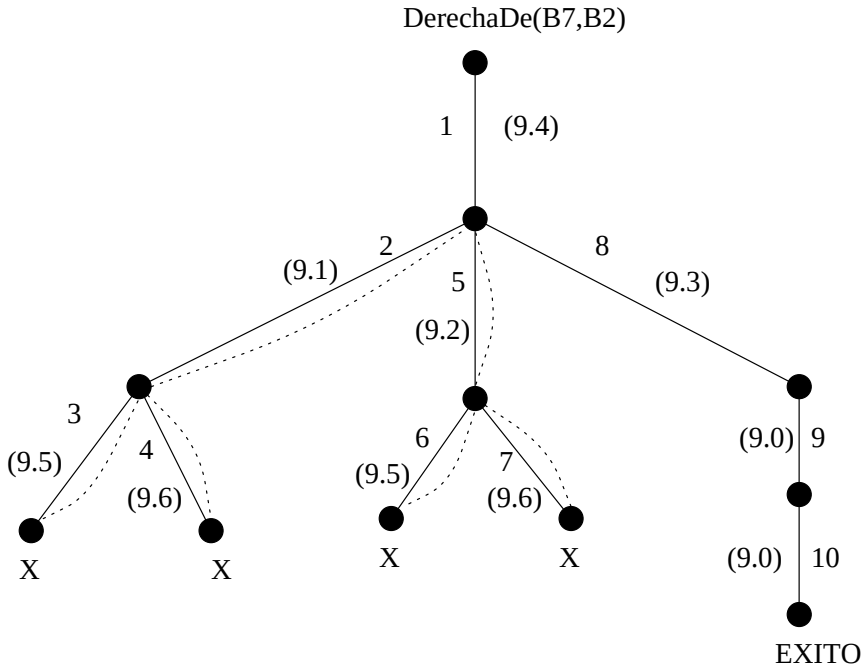


Figura A9.6:

El árbol de la figura A9.6 fue generado con una convención implícita que ahora hacemos expícita. Consideremos, por ejemplo, el nodo 2 de la figura A9.5. Hay tres ramas a partir de ese nodo, correspondientes a la generación de submetas usando los axiomas (9.1), (9.2), y (9.3). En la genración de árbol primero en profundidad de la figura A9.6, elegimos generar primero la rama correspondiente al axioma (9.1). Cuando esta rama finalmente falla, elegimos generar a continuación la rama del axioma (9.2), y finalmente, la rama de (9.3). Podríamos igualmente haber generado esas ramas en el orden (9.2), (9.1), (9.3) o, en general, en cualquiera de las 6 permutaciones posibles de los tres axiomas. El orden particular que elegimos fue el orden en que los axiomas se escribieron al principio. De aquí en adelante, ésta será la convención que usaremos al generar los árboles primero en profundidad. Más precisamente, adoptamos la siguiente convención:

[‡]pre–order traversal, en inglés

Considérese cualquier nodo del árbol.

1. Lístese todos los axiomas que pueden contribuir a la generación de una submeta de ese nodo en el orden en que fueron escritos cuando se propuso por primera vez los axiomas.

2. Ahora, en una búsqueda primero en profundidad, para generar la primera rama de un nodo, úsese el primer axioma de la lista. Si esta rama finalmente falla, de modo que hacemos backtracking a este nodo, genérese la segunda rama usando el segundo axioma en la lista, etc.

Debería estar claro ahora que, bajo la convención de orden anterior, un teorema tendrá un único árbol de búsqueda primero en profundidad para su demostración descendente. Como ejemplo final, la figura A9.7 da un árbol de búsqueda primero en profundidad para una demostración descendente de $IzquierdaDe(B1, B5)$.

Por la convención anterior, los árboles de búsqueda primero en profundidad son sensibles al orden en que se escribe los axiomas. Ordenes diferentes pueden entregar árboles de búsqueda diferentes para el mismo teorema, de modo que la eficiencia con que se encuentra demostraciones descendentes puede ser sensible al orden de los axiomas. Como veremos, PROLOG hace búsqueda primero en profundidad para demostraciones que usan esta convención de orden, de modo que en interés de la ejecución eficiente, el programador de PROLOG debe estar conciente del orden en que presenta los axiomas.

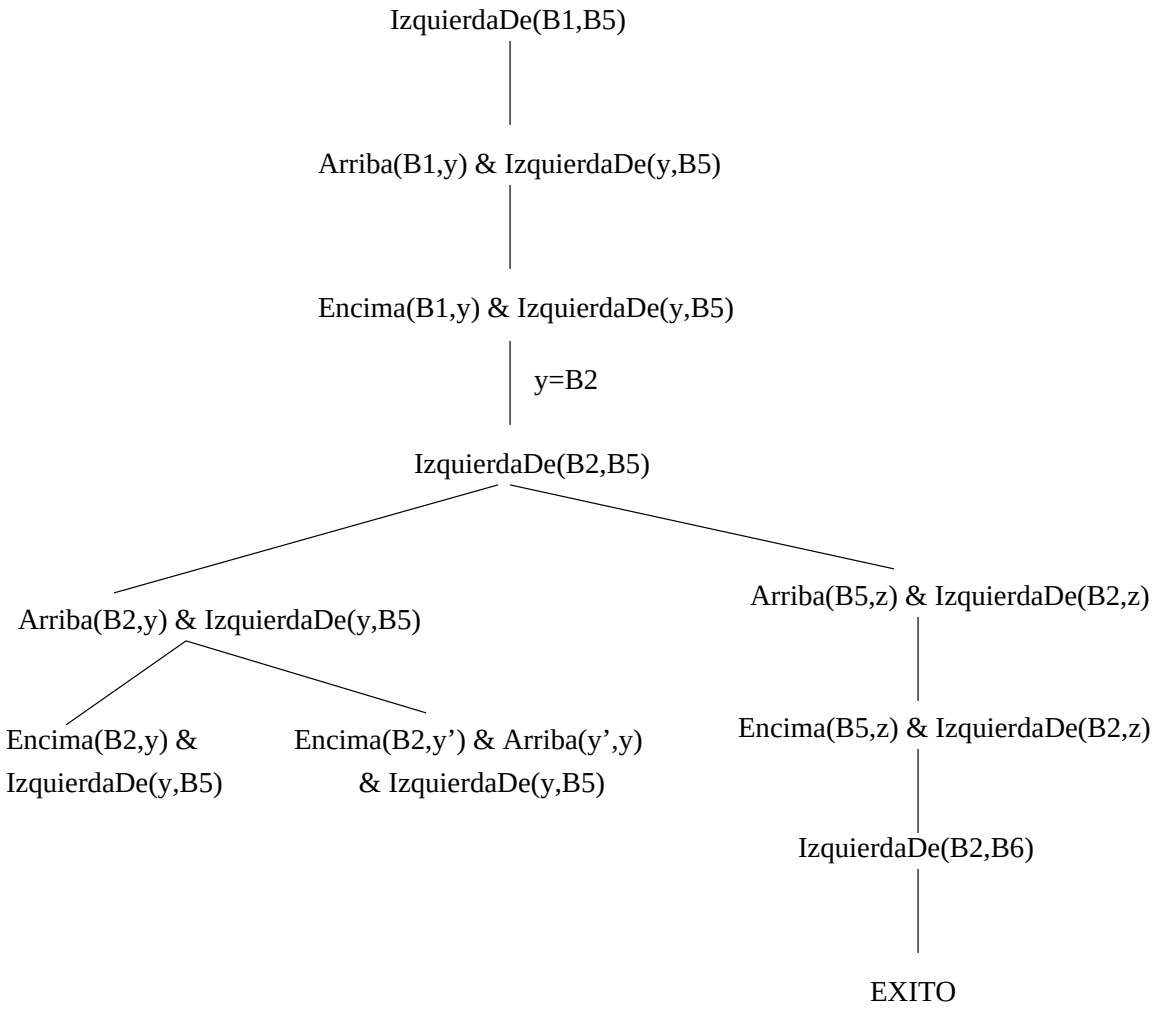
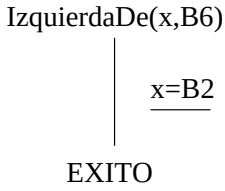

IzquierdaDe(B1,B5)
Arriba(B1,y) & IzquierdaDe(y,B5)
Encima(B1,y) & IzquierdaDe(y,B5)
y=B2
IzquierdaDe(B2,B5)
Arriba(B2,y) & IzquierdaDe(y,B5)
Arriba(B5,z) & IzquierdaDe(B2,z)
Encima(B2,y) & IzquierdaDe(y,B5)
Encima(B2,y') & Arriba(y',y) & IzquierdaDe(y,B5)
Encima(B5,z) & IzquierdaDe(B2,z)
IzquierdaDe(B2,B6)
EXITO

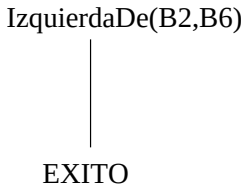
Figura A9.7:

9.4 Teoremas con Variables

Hasta ahora, todos teoremas de ejemplo para los cuales hemos dado demostraciones descendentes no han contenido variables. Ellos han sido aseveraciones sobre relaciones que se dan entre bloques individuales definidos, como $B1$, $B6$, etc. Luego, al probar, por ejemplo, que $IzquierdaDe(B1, B6)$, podemos responder "sí" a la pregunta "¿Está $B1$ a la izquierda de $B6$?". Supongamos ahora que deseamos responder preguntas como "¿Cuál bloque está a la izquierda de $B6$?". Podemos ver esto como el problema de determinar un valor para la variable x tal, que $IzquierdaDe(x, B6)$ se cumple. Esto sugiere que pongamos a la fórmula $IzquierdaDe(x, B6)$ como teorema, y que tratemos de determinar una demostración descendente de esto:



Nótese que la demostración forzó a la variable a x a tomar un valor particular, $B2$. Esto significa que, en lugar de probar el teorema original, $IzquierdaDe(x, B6)$, podríamos haber probado el mismo teorema, con $B2$ reemplazando a x, en exactamente la misma forma:



Luego, $B2$ es un *ejemplo* de un x para el cual $IzquierdaDe(x, B6)$ es demostrable.

Como un ejemplo adicional, consideremos el responder a la pregunta "¿Para qué bloques x e y está x arriba de y y a la izquierda de $B7$?". Ponemos a $Arriba(x, y)$ & $IzquierdaDe(y, B7)$ como teorema, el cual tiene una demostración descendente (figura A9.8). Esta demostración forzó a las variables x e y del teorema a tomar los valores $B1$ y $B2$, respectivamente. En consecuencia, esencialmente la misma demostración funciona para el teorema original con las variables x e y reemplazadas por $B1$ y $B2$ (figura A9.9). Nótese que hay varias otras demostraciones del teorema original, por ejemplo, la de la figura A9.10.

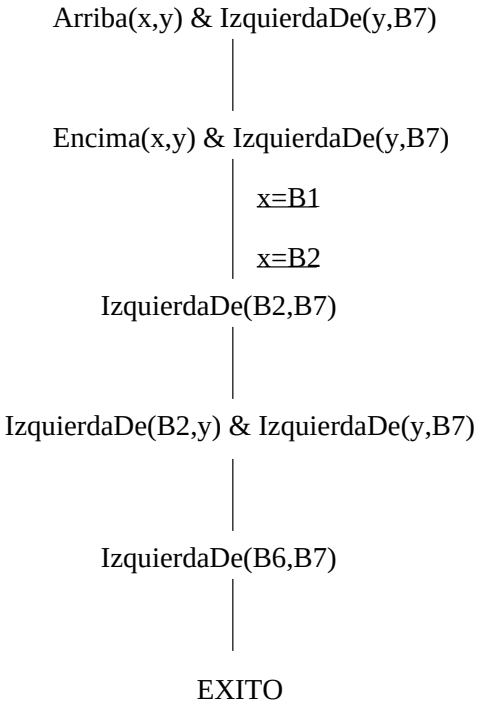


Figura A9.8:

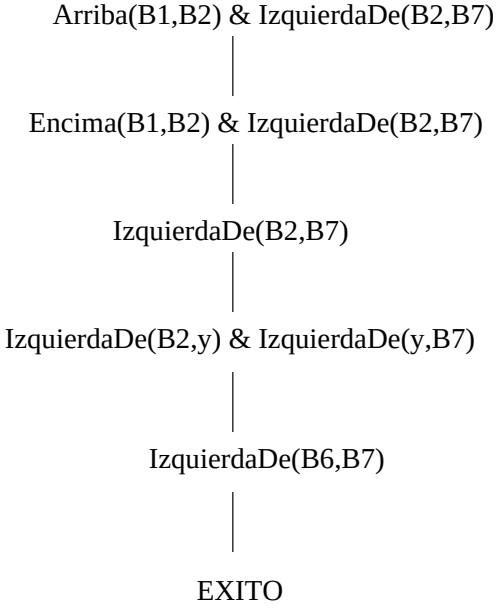


Figura A9.9:

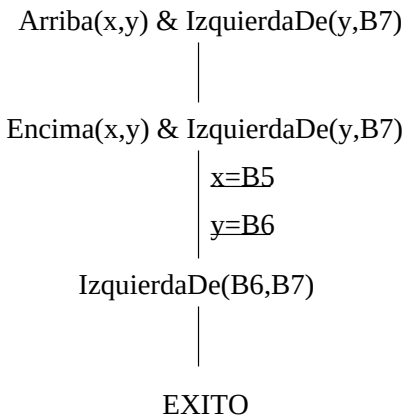


Figura A9.10:

Cualquiera de estas demostraciones va a proporcionar un *ejemplo* de un par x, y tal, que $Arriba(x, y)$ & $IzquierdaDe(y, B7)$ se cumple. Las dos demostraciones de arriba entregan dos ejemplo diferentes de x e y, a saber, $x = B1, y = B2$ y $x = B5, y = B6$.

Debería estar claro ahora que aquí hay un principio general. Si $T(x, y, \ldots)$ es un teorema cuya demostración fuerza a las variables $x, y, \ldots$ a tomar los valores $a, b, \ldots$, entonces $T(a, b, \ldots)$ también tiene una demostración. Entonces, $a, b, \ldots$ son ejemplos de $x, y, \ldots$ para los cuales $T(a, b, \ldots)$ se cumple.*. Esta simple idea de demostrar teoremas con variables para obtener ejemplos de cosas resulta ser extremadamente poderosa. Como veremos, es uno de los mecanismos centrales para realizar computaciones en PROLOG.

9.5 PROLOG

El lenguaje de programación PROLOG se originó con la observación de que una búsqueda primero en profundidad de demostraciones descendentes puede ser vista como un intérprete para un lenguaje de programación cuyos programas consisten de sucesiones de cláusulas de Horn positivas y que tales cláusulas

*El poner un teorema a demostrar como meta superior (top) en el árbol equivale a agregar a la base de conocimiento la negación de él, y llegar a EXITO equivale a llegar a una inconsistencia, a la cláusula vacía. En consecuencia, poner una fórmula con variables del tipo $T(x, y, \ldots)$ como meta superior en el árbol, equivale a agregar a la base de conocimiento la oración $\forall\, xy \cdots\ \neg\, T(x, y, \ldots)$, es decir, $\neg\, \exists\, xy \cdots\ T(x, y, \ldots)$. Esto es por la cuantificación universal tácita de las variables en las cláusulas que aparecen en una demostración. En consecuencia, el método presentado equivale a demostrar el teorema existencial $\exists\, xy \cdots\ T(x, y, \ldots)$; y los valores obtenidos para $x, y, \ldots$ son casos concretos de tal cuantificación existencial.

pueden ser vistas como procedimientos. Un programa en PROLOG puede ser ejecutado con un llamado de nievel superior† de la forma $T_1 \ \& \ \cdots \ \& \ T_n$, que representa un teorema cuya demostración descendente estamos buscando. El intérprete de PROLOG hace una búsqueda primero en profundidad de tal demostración, el resultado de la cual es ya sea falla, o éxito. En el último caso, PROLOG también entrega aquellos valores que las variables del teorema, si las hay, estuvieron forzadas a tomar en la demostración exitosa. Normalmente, tales valores para las variables son vistos como la salida de la ejecución del programa, ya que esos son ejemplos de valores de las variables para los cuales vale el teorema.

Para ver cómo, en PROLOG, una cláusula de Horn positiva puede ser vista como un procedimiento, consideremos una meta $G_1 \ \& \ \cdots \ \& \ G_n$ y el proceso de establecer una submeta de esta meta.

1. Si la fórmula atómica G es un axioma y G unifica con G_1 con $umg\ \sigma$, entonces $(G_2 \ \& \ \cdots \ \& \ G_n)\sigma$ es una submeta de $G_1 \ \& \ \cdots \ \& \ G_n$. PROLOG ve a $G_1 \ \& \ \cdots \ \& \ G_n$ como una sucesión de llamados a procedimientos a ser ejecutados en orden de izquierda a derecha. Después de que se ejecuta G_1, PROLOG ejecuta la sucesión de llamados a procedimientos $G_2\sigma \ \& \ \cdots \ \& \ G_n\sigma$. Luego, podemos ver el axioma G como un procedimiento que fue invocado por el llamado G_1. Tal procedimento es invocado por una coincidencia de patrones exitosa*, es decir, éste unifica con su llamador G_1. G es un tipo de procedimiento particularmente simple porque una vez invocado, no tiene nada más que hacer; hasta ahí llega.

2. Si $A_1 \ \& \ \cdots \ \& \ A_m \ \rightarrow \ G$ es una cláusula de Horn positiva y G unifica con G_1 con $umg\ \sigma$, entonces $(A_1 \ \& \ \cdots \ \& \ A_m \ \& \ G_2 \ \& \ \cdots \ \& \ G_n)\sigma$ es una submeta de la meta $G_1 \ \& \ \cdots \ \& \ G_n$. De nuevo, PROLOG ve a la meta $G_1 \ \& \ \cdots \ \& \ G_n$ como una sucesión de llamados a procedimientos para ser ejecutados en orden de izquierda a derecha. Después de que se ejecuta G_1, PROLOG ejecuta la sucesión de llamados a procedimientos $A_1\sigma \ \& \ \cdots \ \& \ A_m\sigma \ \& \ G_2\sigma \ \& \ \cdots \ \& \ G_n\sigma$. Luego, podemos ver el axioma $A_1 \ \& \ \cdots \ \& \ A_m \ \rightarrow \ G$ como un procedimiento que fue invocado por el el llamado a procedimiento G_1. G se llama *la cabeza* del procedimiento, y $A_1 \ \& \ \cdots \ \& \ A_m$ es el cuerpo del procedimiento. Tal procedimiento es invocado por unificación exitosa de su cabeza G con el primer llamado a procedimiento G_1 de la meta

†top level call, en inglés

*successful pattern match, en inglés

actual G_1 & $\cdots$ & G_n, con *umg* σ. La sucesión de llamados a procedimientos resultante es $A_1\sigma$ & $\cdots$ & $A_m\sigma$ & $G_2\sigma$ & $\cdots$ & $G_n\sigma$. Como éstos son ejecutados de izquierda a derecha, es el cuerpo instanciado $A_1\sigma$ & $\cdots$ & $A_m\sigma$ el que se ejecuta primero, antes que el resto de los literales meta originales. Si este cuerpo de procedimiento instanciado se ejecuta exitosamente, el procedimiento entrega un resultado exitoso. El efecto de un llamado a procedimiento exitoso G_1 es que a algunas de las variables compartidas de G_1 y G_2 & $\cdots$ & G_n se les asignará valores de modo que después de que el procedimiento tiene éxito, la sucesión restante de literales meta a ser ejecutados tendrá la forma $G_2\,\mu$ & $\cdots$ & $G_n\,\mu$, para algún unificador μ que especifica los valores de esas variables.

Podemos resumir nuestra interpretación de las demostraciones descendentes como un proceso de invocación de procedimientos, con cláusulas de Horn positivas jugando el papel de procedimientos de la manera que sigue:

1. Una meta G_1 & $\cdots$ & G_n se ve como una sucesión de llamados a procedimientos a ser ejecutados de izquierda a derecha.

2. El llamado a procedimento G_1 tiene éxito si, ya sea

   a) G_1 unifica con un procedimiento G, o

   b) G_1 unifica con la cabeza G de un procedimiento A_1 & $\cdots$ & $A_m \rightarrow G$ con *umg* σ y la sucesión de llamados a procedimiento $A_1\sigma$ & $\cdots$ & $A_m\sigma$ tiene éxito.

3. Una sucesión de dos o más llamados a procedimiento G_1 & $\cdots$ & G_n tiene éxito si el llamado a procedimiento G_1 tiene éxito, y la sucesión de llamados a procedimiento $G_2\,\mu$ & $\cdot$ & $G_n\,\mu$ tiene éxito, donde el unificador μ especifica aquellos valores que las variables en G_1 & $\cdots$ & G_n están forzadas a tomar por la ejecución exitosa de G_1.

Uno puede pensar las variables de un procedimiento A_1 & $\cdots$ & $A_m \rightarrow G$ como sus parámetros formales. Estos parámetros están ligados en tiempo de invocación al procedimiento por medio del *umg* σ determinado por cuando G unifica con el primer literal en la meta actual. El cuerpo $A_1\sigma$ & $\cdots$ & $A_m\sigma$ del procedimento instanciado es ejecutado a continuación. Pero hay una extraña y maravillosa diferencia entre ligazón de parámetros en PROLOG y lenguajes de programación convencionales: los parámetros de un procedimiento no

necesitan estar completamente ligados en tiempo de invocación. Por ejemplo, la ejecución del llamado a procedimento $Q(a, z)$ con el procedimiento $P(x, y) \rightarrow Q(x, y)$ entrega como meta el cuerpo de procemiento instanciado $P(a, y)$, donde el parámetro formal x está ligado al valor a, pero el parámetro y, estando no ligado, permanece totalmente descomprometido a un valor. Es posible que al ejecutar $P(a, y)$, el parámetro y finalmente tome un valor, pero cualquier valor tal será el resultado de la ejecución del proce-dimiento, y no, como en lenguajes de programación convencionales, el resultado de un llamado al procedimiento. En un sentido muy real, un procedimiento puede ser ejecutado sin tener valores iniciales para todos sus parámetros formales.

Otra distinción importante entre PROLOG y lenguajes de programación convencionales es la forma en que se llama a los procedimientos. En la mayoría de los lenguajes de programación, un llamado a procedimiento consiste en un nombre de procedimiento junto con algunos parámetros actuales, o argumentos, para ser pasados al procedimiento nombrado que debe ser único. En PROLOG, un llamado a procedimiento consiste de un *patrón* formado por un predicado con sus argumentos y *cualquier procedimiento cuya cabeza se identifica** *con ese patrón (por unificación) será aplicable.* Por esta razón, uno se refiere a esto por *invocación de procedimientos dirigida por patrón*†, especialmente en los círculos de inteligencia artificial.

Tales invocaciones puede ser bastante complejas cuando se usa símbolos de función, ya que para ellas los patrones pueden ser complejos. Por ejemplo, consideremos el llamado $P(f(b, u), g(u), h(v, v))$ y el procedimiento $Q(x, y, z, w) \rightarrow P(f(x, g(y)), g(w), h(x, z))$. Esto entrega el cuerpo de procedimiento instanciado $Q(b, y, b, g(y))$. Como veremos en la sección siguiente, tales patrones complejos que involucran símbolos de función surgen frecuentemente en programación en lógica cuando se computa con estructuras de datos.

Sabemos que en lógica, a diferencia con lenguajes de programación convencionales, *se puede aplicar más de un procedimiento a un llamado a procedimiento dado.* Por ejemplo, en nuestro mundo de los bloques, el llamado a procedimiento $IzquierdaDe(B2, y)$ puede invocar cualquiera de cuatro procedimientos diferentes. Esto hace a tales programas *no deterministas*; ellos pueden, en principio, intentar ejecutar una meta en más de una forma. Esto corresponde al hecho que encontrar una demostración descendente para un teorema requiere de búsqueda. El intérprete de PROLOG elimina este no determinismo al aplicar una búsqueda primero en profundidad de demostraciones descendentes, usando los procedimientos en el orden en que aparecen en

*matches, en inglés

†pattern directed procedure invocation

el programa. Luego, al ejecutar un llamado a procedimiento, PROLOG trata de satisfacer este llamado usando el primer procedimiento aplicable en el programa. Si el uso de este procedimiento finalmente lleva a falla, de modo que hay backtracking a este punto en la búsqueda de una demostración, PROLOG trata el siguiente procedimiento aplicable, etc.

Resumamos las características esenciales de PROLOG:

1. Un programa en PROLOG es una sucesión de procedimientos. Cada uno de tales proce-dimientos es una cláusula de Horn positiva.

2. La entrada, o meta de nivel superior, a un programa en PROLOG es una conjunción $T_1 \ \& \ \cdots \ \& \ T_n$ de fórmulas atómicas. De manera abstracta, esto se ve como un teorema, el programa se ve como una colección de axiomas, y estamos buscando una demostración descendente de la entrada a partir de los axiomas. Aquellos valores tomados por las variables de $T_1 \ \& \ \cdots \ \& \ T_n$ como resultado de su demostración descendente son vistos normalmente como la salida de la ejecución del programa.

3. El intérprete de PROLOG es simplemente un demostrador de teoremas descendente que usa una estartegia de búsqueda primero en profundidad.

Dado que un procedimiento de PROLOG es llamado por una identificación de patrones (pattern match) en la cabeza de la cláusula, es en esta cabeza de cláusula en lo que más nos fijamos. Por esta razón, los procedimientos de PROLOG se escriben "de atrás para adelante" (backwards) - primero la cabeza, seguida del cuerpo del procedimiento, como aquí:

$$Arriba(x, z) \ \leftarrow \ Encima(x, y) \ \& \ Arriba(y, z).$$

El orden usual de izquierda a derecha en el cuerpo del procedimiento prevalece. Luego, $Encima(x, y)$ se ejecutaría primero, seguido de $Arriba(y, z)$. En el resto de este capítulo adoptaremos esta notación "de atrás para adelante" para las cláusulas siempre que estemos tratando con procedimientos de PROLOG.

9.6 Estructuras de Datos en Lógica

Hasta ahora, el lenguaje de programación basado en lógica que hemos creado es bastante débil; no tiene ninguna de nuestras estructuras de datos favoritas, como números, arreglos o listas. Pero nótese que en el ejemplo del mundo

de los bloques con el que hemos estado tratando no hace uso de símbolos de función. Es una visión elegante de la programación en lógica el que *símbolos de función puedan ser usados para definir estructuras de datos.* Empezaremos ilustrando cómo las estructuras de datos llamadas listas pueden ser definidas y manipuladas en lógica.

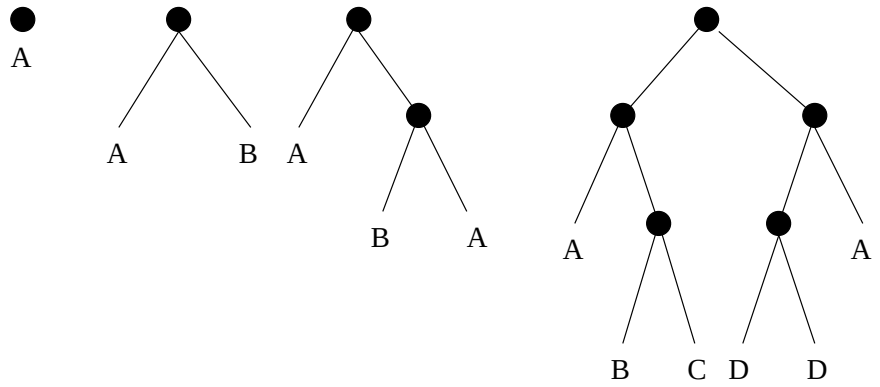


Figura A9.11: Ejemplos de árboles binarios

Para ver cómo, consideremos primero cómo la lógica puede representar *árboles binarios* como los de la figura A9.11. El lenguaje de programación LISP representa tales árboles usando un símbolo de función binaria llamado *cons*. Los cuatro árboles de la figura A9.11 son representados por A, $cons(A, B)$, $cons(A, cons(B, A))$ y $cons(cons(A, cons(B, C)), cons(cons(D, D), A))$, respectivamente. La regla es simple:

Un nodo hoja (es decir, uno del cual no salen ramas) es representado por por su etiqueta. Un nodo no hoja es representado por $cons(t_1, t_2)$, donde t_1 y t_2 representan los subárboles izquierdo y derecho del nodo, respectivamente.

En LISP, una clase de árbol binario, llamada *lista*, juega un papel particularmente prominente. Para definir una lista, supongamos que tenemos disponible una constante distinguida, llamada *NIL*. Luego, podemos definir una lista en la forma que sigue:

1. *NIL* es una lista, llamada la lista *vacía*.

2. Si l es una lista y b representa un árbol binario (tal vez, una lista), entonces $cons(b, l)$ es una lista. Debería estar claro que una lista no vacía típica tiene la forma $cons(b_1, cons(b_2, \cdots cons(\ b_n, NIL)\cdots))$, donde $b_1, \ldots, b_n$ representan árboles binarios.

Ahora, la notación "*cons*" de arriba es particularmente torpe para representación de árboles binarios y listas. Dado que tales estructuras de datos surgen muy frecuentemente, se usa una notación humanamente más legible, introduciendo una función infija *cons* denotada por "$\cdot$". En consequencia, se tiene $b_1 \cdot b_2$ en lugar de $cons(b_1, b_2)$. Con esta notación, los cuatro árboles de la figura A9.11 se representan por A, $A \cdot B$, $A \cdot (B \cdot A)$, $(A \cdot (B \cdot C)) \cdot ((D \cdot D) \cdot A)$. Podemos mejorar esta notación todavía más introduciendo una notación para eliminar paréntesis cuando no puede surgir ambigüedad. Así, $b_1 \cdot b_2 \cdot \cdots \cdot b_n$ será usada en lugar de $b_1 \cdot (b_2 \cdot (b_3 \cdot \cdots \cdot (b_{n-1} \cdot b_n) \cdots))$. Luego, los cuatro árboles de la figura A9.11 pueden ser representados por A, $A \cdot B$, $A \cdot B \cdot A$, $(A \cdot B \cdot C) \cdot (D \cdot D) \cdot A$.

Con esta notación mejorada, una lista no vacía típica, que en la notación *cons* completa se veía como $cons(b_1, cons(b_2, \cdots cons(b_n, NIL) \cdots))$, ahora se verá como $b_1 \cdot b_2 \cdot \cdots \cdot b_n \cdot NIL$.

La tabla siguiente nos da algunas listas tanto en notación abreviada punto, como en la notación *cons* completa correspondiente. Nótese que todavía son necesarios algunos paréntesis para evitar ambigüedad.

Notación punto	**Notación cons**
$A \cdot NIL$	$cons(A, NIL)$
$A \cdot B \cdot C \cdot NIL$	$cons(a, cons(B, cons(C, NIL)))$
$(A \cdot B \cdot NIL) \cdot A \cdot NIL$	$cons(cons(A, cons(B, NIL)), cons(A, NIL))$
$(A \cdot B) \cdot ((B \cdot NIL) \cdot NIL) \cdot NIL$	$cons(cons(A, B), cons(cons(cons(B, NIL), NIL), NIL))$

Con la notación punto simplificada, una lista no vacía típica se escribe en la forma $b_1 \cdot b_2 \cdot b_3 \cdot \cdots \cdot b_n \cdot NIL$. De este modo, una lista puede ser vista como representando a la sucesión de elementos $b_1, b_2, \ldots, b_n$. Nótese que *NIL* *no* es el último elemento de esta sucesión, sino b_n. Para ver por qué esto es así, considérese la lista vacía *NIL*. Queremos que ésta denote a la sucesión nula, es decir, la sucesión sin elementos. Pero si, digamos, la lista $b \cdot NIL$ representara la sucesión de dos elementos b, NIL, entonces la lista *NIL* debe representar la sucesión con un elemento *NIL*, mientras que queremos que la lista *NIL* represente sucesión sin elementos.

En resumen, la lista $b_1 \cdot b_2 \cdot \cdots \cdot b_n \cdot NIL$ representa una sucesión $b_1, b_2, \ldots, b_n$. $b_1, b_2, \ldots, b_n$ se llaman los *elmentos* de la lista. b_1 es su primer elemento, b_2, su segundo, ..., b_n, su último. Los b's pueden ser constantes o pueden representar árboles binarios. En la mayoría de las aplicaciones, los b's serán constantes u

otras listas, como en la tabla anterior. La lista *NIL* representa la lista vacía.

Ahora nuestra representación de listas requiere de un único símbolo de función binaria, *cons*, y una única constante, *NIL*. Pero símbolos de función y constantes son provistos por los lenguajes de la lógica de primer orden. *En consecuencia, una lista es simplemente un cierto tipo de término de la lógica de primer orden.* Ahora que podemos representar la estructura de datos lista en lógica, podemos realizar la misma clase de computaciones sobre listas como son hechas el lenguajes para procesamiento de listas como LISP. Por ejemplo, el siguiente es un programa en PROLOG para $MEMBER(x, y)$,que tiene éxito si x es un elemento de la lista y, y falla en otro caso:

$$MEMBER(x, x \cdot y)$$
$$MEMBER(x, y \cdot z) \quad \leftarrow \quad MEMBER(x, z)$$

La primera cláusula dice que x en un lemento de cualquier lista cuyo primer elemento es x. La segunda dice que si x es miembro de una lista z, entonces x es miembro de cualquier lista obtenida al agregar un elemento al principio de z. La figura A9.12 muestra el árbol de búsqueda primero en profundidad para una demostración de $MEMBER(a, b \cdot c \cdot a \cdot NIL)$.

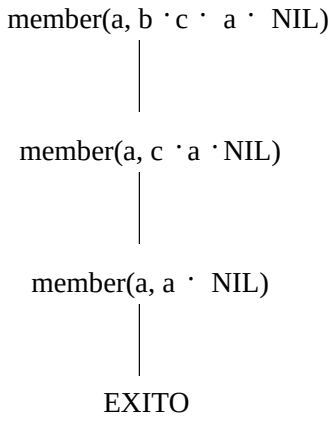


Figura A9.12:

Al estudiar la figura A9.12, recuérdese que la notación punto es sólo una abreviatura conveniente para la notación *cons* completa. Luego, la meta superior es, en realidad, $MEMBER(a, cons(b, cons(c, cons(a, NIL))))$, y ésta unifica con la cabeza de la segunda cláusula, que es $MEMBER(x, cons(y, z))$. El *umg* es entonces $\frac{x}{a}, \frac{y}{b}, \frac{z}{cons(c,cons(a,NIL))}$, en notación punto, $\frac{x}{a}, \frac{y}{b}, \frac{z}{c \cdot a \cdot NIL}$. La figura A9.13

nos muestra la computación de $MEMBER(a, b \cdot c \cdot NIL)$, que, por supuesto, falla.

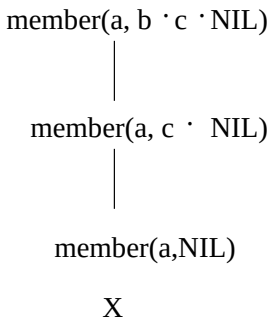


Figura A9.13:

Como segundo ejemplo, consideremos $SUBLIST(x, y)$, que tiene éxito cuando x es una sublista de y, y falla en otro caso. Una sublista de una lista se obtiene eliminando cero o más elementos del frente de la lista. Luego, nos gustaría que : $SUBLIST(a \cdot b \cdot NIL, a \cdot b \cdot NIL)$, $SUBLIST(a \cdot NIL, b \cdot a \cdot NIL)$ y $SUBLIST(NIL, a \cdot b \cdot NIL)$ tengan éxito, y que $SUBLIST(a \cdot b \cdot NIL, NIL)$ y $SUBLIST(a \cdot NIL, a \cdot b \cdot NIL)$, fallen.

El programa para esto es:

$$\begin{aligned} SUBLIST(x, x) & \\ SUBLIST(x, y \cdot z) \quad & \leftarrow \quad SUBLIST(x, z) \end{aligned}$$

A continuación, consideremos el computar la concatenación de dos listas. Que $APPEND(x, y, z)$ denote que la lista z es el resultado de anteponer lista x al frente la lista y. Por ejemplo, la anteposición de $a \cdot b \cdot NIL$ a $c \cdot d \cdot NIL$ entrega la lista $a \cdot b \cdot c \cdot d \cdot NIL$.

Un programa para esto es:

$$\begin{aligned} APPEND(NIL, x, x) & \\ APPEND(x \cdot y, z, x \cdot w) \quad & \leftarrow \quad APPEND(y, z, w) \end{aligned}$$

La figura A9.14 muestra el árbol de búsqueda primero en profundidad para computar el resultado de anteponer $a \cdot b \cdot NIL$ a $c \cdot d \cdot NIL$.

Nótese que el teorema tiene una variable x en él que actua como variable *de salida*. Luego, el valor que x es forzado a tomar por la demostración será el resultado de concatenar los dos primeros argumentos de $APPEND$. En la

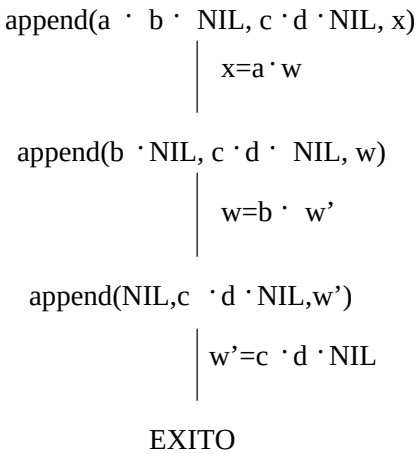


Figura A9.14:

figura A9.14, este resultado se obtiene "resolviendo las ecuaciones" $x = a \cdot w$, $w = b \cdot w'$, $w' = c \cdot d \cdot NIL$, para entregar $x = a \cdot b \cdot c \cdot d \cdot NIL$. Esta contabilidad es realizada automáticamente por el intérprete de PROLOG, que, al ejecutar exitosamente el llamado a procedimiento $APPEND(a \cdot b \cdot NIL, c \cdot d \cdot NIL, x)$, informaría al usuario que $x = a \cdot b \cdot c \cdot d \cdot NIL$.

Al discutir el programa para *APPEND*, hemos estado suponiendo que los dos dos primeros argumentos actuan como entradas y que el tercer argumento actua como salida. Luego, hemos estado suponiendo, como fue el caso en la figura A9.14, que dos listas de entradas serán especificadas como los dos primeros argumentos de *APPEND*, y que el tercero una variable de salida cuyos valores queremos computar como la concatenación de las dos listas de entrada. Pero no hay nada sagrado, o necesario, en este punto de vista. Suponganos, en lugar de eso, que pedimos computar $APPEND(a \cdot b \cdot NIL, x, a \cdot b \cdot c \cdot d \cdot NIL)$, es decir, qué lista x es tal, que al serle antepuesta la lista $a \cdot b \cdot NIL$, entrega la lista $a \cdot b \cdot c \cdot d \cdot NIL$. La figura A9.15 muestra una computación de esto.

El punto importante aquí es que *el mismo programa puede ser usado para computar diferentes relaciones de entrada/salida.* El programa *APPEND* puede ser usado para computar lo siguiente:

1. Dadas dos listas, determínese su concatenación.
2. Dadas dos listas, determine si hay una lista que al serle antepuesta la primera, entrega la segunda.

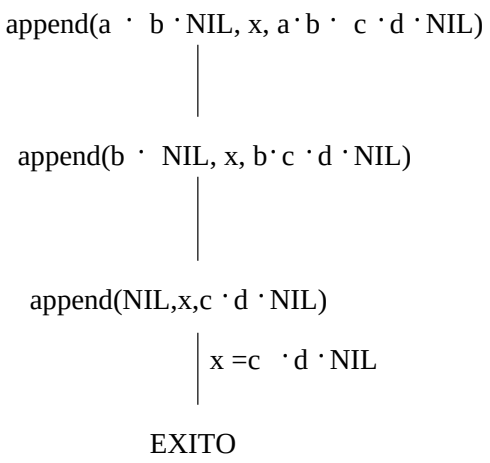


Figura A9.15:

3. Dadas dos listas, determinar si hay una lista que al ser antepuesta a la primera, da la segunda.

4. Dada una lista, determine si hay dos listas que, al ser concatenadas, entregan la lista dada. Intente esto para ver que hará PROLOG.

Esta característica de programación en lógica – que el mismo programa pueda computar múltiples relaciones de entrada/salida – no es compartida por ningún otro otro lenguaje de programación.

Volvamos al ejemplo del mundo de los bloques y consideremos una representación diferente, en la cual pensamos una escena como una sucesión de bloques, con las pilas en orden de izquierda a derecha. Por ejemplo, la escena de la figura A9.1 consiste de tres pilas de bloques, y, en consecuencia, tiene una representación natural en la cual estas pilas están colocadas en sucesión de izquierda a derecha.

$$Escena((B1 \cdot B2 \cdot NIL) \cdot (B3 \cdot B4 \cdot B5 \cdot B6 \cdot NIL) \cdot (B7 \cdot NIL) \cdot NIL)$$

Los predicados *Encima*, *Arriba*, *IzquierdaDe* y *DerechaDe* de la representación anterior pueden ser definidos ahora facilmente:

$$\begin{aligned} Encima(x, y) &\leftarrow Escena(z)\ \&\ MEMBER(p, z)\ \&\ SUBLIST(x \cdot y \cdot w, p) \\ Arriba(x, y) &\leftarrow Escena(z)\ \&\ MEMBER(p, z)\ \&\ SUBLIST(x \cdot w, p)\ \& \\ &\quad\ MEMBER(y, w) \\ IzquierdaDe(x, y) &\leftarrow Escena(z)\ \&\ MEMBER(p, z)\ \&\ MEMBER(x, p)\ \& \end{aligned}$$

$$\begin{aligned} & SUBLIST(p \cdot w, z) \ \& \ MEMBER(p', w) \ \& \\ & MEMBER(y, p') \\ DerechaDe(x, y) \ \leftarrow \ & IzquierdaDe(y, x) \end{aligned}$$

Nótese que este programa hace uso de la característica de backtracking de PROLOG. Por ejemplo, el procedimiento para *IzquierdaDe* usa $MEMBER(p, z)$ para seleccionar una pila p. Si, a conti-nuación, $MEMBER(x, p)$ falla, entonces PROLOG hará backtracking al llamado $MEMBER(p, z)$ y se seleccionará una nueva pila a la derecha de la pila antigua. De manera similar, $MEMBER(p', w)$ selecciona una pila p' a la derecha de la pila p, cuando p' es testeada por $MEMBER(y, p')$ para ver si contiene a y. Si no, entonces se hace una nueva elección de p' a la derecha de la elección anterior.

Un Ejemplo Final: Ordenando* una Lista

El problema de ordenar los elementos de una lista en orden creciente es tratado comunmente en los cursos introductorios de programación. Uno de los más eficientes de tales algoritmos es Quicksort, que funcioan como sigue:

Paso 1: Dada una lista $primero \cdot resto$, cuyo primer elemento es *primero* y cuyos elementos restantes forman la lista *resto*, sepárese la lista *resto* en dos listas l_1 y l_2 tales, que:

- todos los elementos de l_1 menores o iguales que *primero*
- todos los elementos de l_2 son mayores que *primero*.

Definimos un procedimiento $SPLIT(primero, resto, l_1, l_2)$ que significa que l_1 y l_2 son la dos versiones separadas a partir del *resto* como se describió arriba. Suponemos que tenemos un predicado $LE(x, y)$ definido sobre los elementos de la lista a ser ordenada, y que significa que x es menor o igual que y.

$$\begin{aligned} & SPLIT(primero, NIL, NIL, NIL) \\ SPLIT(primero, x \cdot resto, x \cdot l_1, l_2) \ \leftarrow \ & LE(x, primero) \ \& \\ & SPLIT(primero, resto, l_1, l_2) \\ SPLIT(primero, x \cdot resto, l_1, x \cdot l_2) \ \leftarrow \ & LE(primero, x) \ \& \\ & SPLIT(primero, resto, l_1, l_2) \end{aligned}$$

*sorting

Paso 2: Ordénese con Quicksort cada una de las listas l_1 y l_2, y péguese esas listas ordenadas, insertando el elemento *primero* entre ellas. En el siguiente programa, $QUICKSORT(l, l')$ significa que l' es una versión ordenada de la lista l.

$$
\begin{array}{lll}
 & QUICK(NIL, NIL) & \\
QUICK(primero \cdot resto, ordenada) & \leftarrow & SPLIT(primero, resto, l_1, l_2)\ \& \\
 & & QUICK(l_1, l_1')\ \& \\
 & & QUICK(l_2, l_2')\ \& \\
 & & APPEND(l_1', primero \cdot l_2', \\
 & & ordenada)
\end{array}
$$

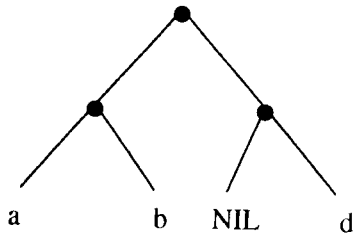
Todas las implementaciones de PROLOG proveen facilidades incorporadas para la representación de listas, de modo que no es necesario empezar de la nada con la función binaria *cons* y la constante *NIL*. Además, estas implementaciones representan listas en una manera más amistosa con el usan la notacón infija "$\cdot$" que hemos estado usando. Otras implementaciones usan una representación a estilo LISP, en la forma que sigue:

1. [] representa a *NIL*, la lista vacía.
2. $[b_1, b_2, \ldots, b_n]$ representa la lista cuyos elementos son $b_1, \ldots, b_n$.
3. $[x \mid y]$ representa la lista cuyo primer elemento es x, y cuyo resto es y.

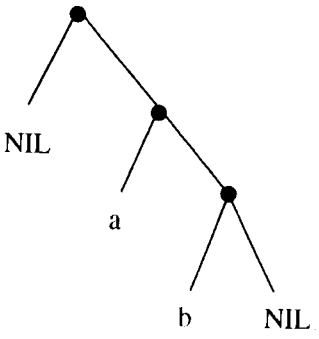
Cualquiera que sea la representación de listas que provea una implementación particular, es importante darse cuenta de que ésta es basicamente ornamento sintáctico para la notación *cons* completa, y que está ahí sólo para hacer más fácil la programación a nosotros los humanos. Visto de manera abstracta, todavía estamos computando con términos lógicos construidos a partir de una función binaria *cons* y una constante distinguida *NIL*.

9.6.1 Ejercicios

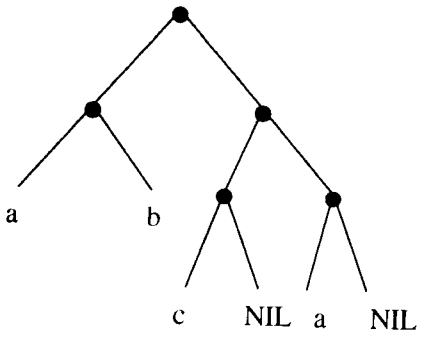
1. Represente los siguientes árboles binarios usando tanto la función binaria *cons*, como la notación infija punto. Indique cuáles de estos árboles son listas.



(i)

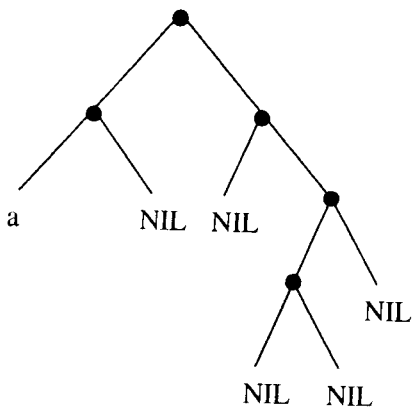


(ii)



(iii)

(v)



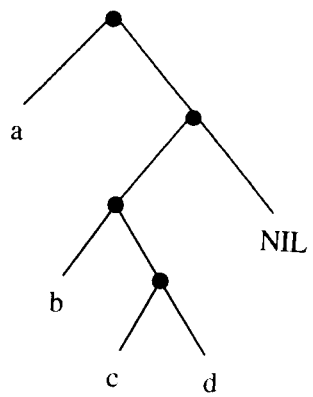
(v)

Respuestas:
(i) $cons(cons(a, b), cons(NIL, d))$
$(a \cdot b) \cdot (NIL \cdot d)$
No es lista.
(ii) $cons(NIL, cons((a, cons(b, NIL)))$
$NIL \cdot a \cdot b \cdot NIL$
Es una lista.
(iii) $cons(cons(a, b), cons(cons(c, NIL), cons(a, NIL)))$
$(a \cdot b) \cdot (c \cdot NIL) \cdot a \cdot NIL$
Es una lista.
(iv) NIL
Es una lista.
(v) $cons(cons(a, NIL), cons(NIL, cons(cons(NIL, NIL), NIL)))$
$(a \cdot NIL) \cdot NIL \cdot (NIL \cdot NIL) \cdot NIL$
Es una lista.

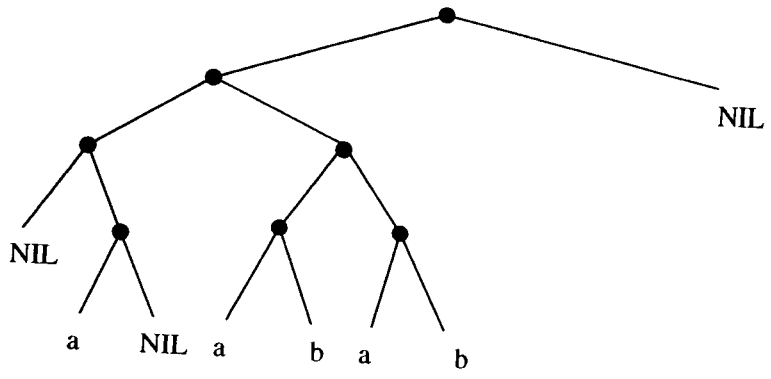
2. Represente los siguientes términos que están en notación punto, en su notación *cons* completa, y como árboles binarios.

(i) $a \cdot (b \cdot c \cdot d) \cdot NIL$
(ii) $((a \cdot b \cdot c) \cdot (d \cdot e) \cdot f) \cdot g \cdot h$
(iii) $((NIL \cdot a \cdot NIL) \cdot (a \cdot b) \cdot (a \cdot b)) \cdot NIL$

Respuestas:
(i) $cons(a, cons(cons(b, cons(c, d), NIL))$



(iii) $cons(cons(cons(NIL, , cons(a, NIL)), cons(cons(a, b), cons(a, b))), NIL)$



3. ¿Cuáles de los siguientes pares de términos unifican? Para aquéllos que unifican, dé el *umg*.

(a) $x \cdot y \cdot z \qquad a \cdot b \cdot c \cdot d \cdot NIL$

Respuesta: $\frac{x}{a}, \frac{y}{b}, \frac{z}{c \cdot d \cdot NIL}$

(b) $x \cdot (y \cdot z) \cdot x \qquad a \cdot (b \cdot c \cdot d) \cdot a$

Respuesta: $\frac{x}{a}, \frac{y}{b}, \frac{z}{c \cdot d}$

(c) $(a \cdot y \cdot NIL) \cdot z \qquad (x \cdot b \cdot NIL) \cdot (c \cdot d \cdot NIL) \cdot NIL$

Respuesta: $\frac{x}{a}, \frac{y}{b}, \frac{z}{(c \cdot d \cdot NIL) \cdot NIL}$

(d) $x \cdot y \cdot NIL \qquad (a \cdot b) \cdot c$

Respuesta: No unifican

(e) $x \cdot y \cdot NIL \qquad a \cdot b \cdot c \cdot NIL$

Respuesta: No unifican

4. Defina los siguientes predicados en PROLOG.

(a) $LAST(x, l)$ – x es el último elemento de la lista no vacía l; e.g. $LAST(a, b \cdot c \cdot a \cdot NIL)$ tiene éxito.

Respuesta:
$LAST(x, x \cdot NIL)$
$LAST(x, y \cdot z \cdot w) \leftarrow LAST(x, z \cdot w)$

(b) $REVERSE(l, l')$ – la lista l' es la lista inversa de la lista l; e.g. $REVERSE(a, \cdot b \cdot c \cdot NIL, c \cdot b \cdot a \cdot NIL)$ tiene éxito.

Respuesta:
$REVERSE(NIL, NIL)$
$REVERSE(x \cdot l, z) \leftarrow REVERSE(l, l')$ & $APPEND(l, x \cdot NIL, z)$

(c) $SUBSET(l_1, l_2)$ – todo elemento de la lista l_1 es un elemento de la lista l_2; e.g. $SUBSET(a \cdot b \cdot NIL, c \cdot b \cdot a \cdot a \cdot NIL)$ – tiene éxito.

Respuesta:
$SUBSET(NIL, l)$
$SUBSET(x \cdot l, l') \leftarrow MEMBER(x, l')$ & $SUBSET(l, l')$.

5. Usando sólo el predicado $APPEND$ como fue definido en esta sección, muestre cómo definir los siguientes predicados en PROLOG.

(a) $SUBLIST$ de esta sección.

Respuesta: $SUBLIST(x, y) \leftarrow APPEND(z, x, y)$.

(b) $LAST$ del ejercicio 4(a).

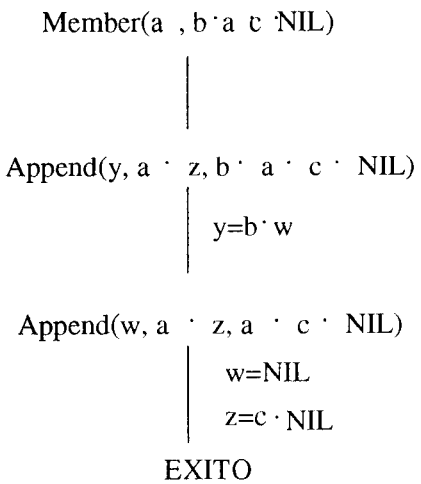
Respuesta: $LAST(x, l) \leftarrow APPEND(y, x \cdot NIL, l)$.

(c) $MEMBER$ de esta sección.

Respuesta: $MEMBER(x, l) \leftarrow APPEND(y, x \cdot z, l)$.

Dé el árbol de búsqueda primero en profundidad que conduce a una solución de la meta $MEMBER(a, b \cdot a \cdot c \cdot NIL)$.

Respuesta:



(d) $PREFIX(l_1, l_2)$ – la lista l_1 es un prefijo de la lista l_2; e.g. $PREFIX(a \cdot b \cdot NIL, a \cdot b \cdot c \cdot d \cdot NIL)$ tiene éxito.

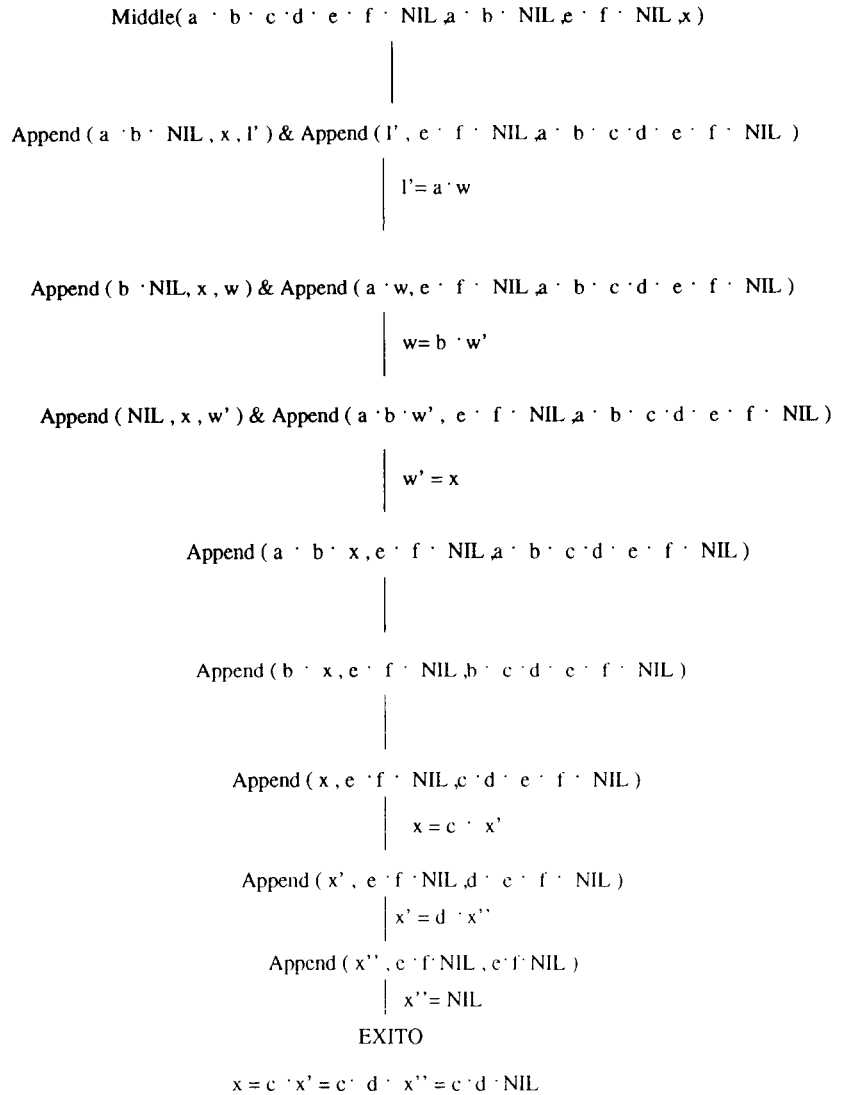
Respuesta: $PREFIX(l_1, l_2) \leftarrow APPEND(l_1, l, l_2)$.

(e) $MIDDLE(l, frontal, trasera, media)$ – Cuando las lista l, *frontal* y *trasera* son las variables de entrada, *media* es una variable de salida y denota la lista en "el medio" de l después de que su segmento inicial *frontal*, y su segmento final *trasera* son eliminados. E.g. $MIDDLE(a \cdot b \cdot c \cdot d \cdot e \cdot f \cdot NIL, a \cdot b \cdot NIL, e \cdot f \cdot NIL, c \cdot d \cdot NIL)$ tiene éxito.

Respuesta: $MIDDLE(l, frontal, trasera, media) \leftarrow APPEND(frontal, media, l') \ \& \ APPEND(l', trasera, l)$.

Muestre el árbol de búsqueda primero en profundidad que conduce a una computación del medio de la lista $a \cdot b \cdot c \cdot d \cdot e \cdot f \cdot NIL$ con parte frontal $a \cdot b \cdot NIL$ y trasera, $e \cdot f \cdot NIL$).

Respuesta:



6. (a) Defina el predicado $INSERT(x, l, l')$ en PROLOG, donde l es una lista ordenada en orden no decreciente, y l' es el resultado de insertar el elemento x en la lista l, preservando el orden de los elementos de l, es decir, l' también estará ordenada. Suponga la disponibilidad del predicado $LE(x, y)$ que significa que x es menor o igual que y.

Respuesta:
$INSERT(x, NIL, x \cdot NIL)$
$INSERT(x, y \cdot l, x \cdot y \cdot l) \leftarrow LE(x, y)$
$INSERT(x, y \cdot l, y \cdot l') \leftarrow LE(y, x)$ & $INSERT(x, l, l')$

(b) Usando el predicado $INSERT$ de la parte (a), especifique un programa en PROLOG para hacer ordenamiento de listas por inserción. En este método, cada elemento de la lista dada es considerado una vez mientras se mantiene una lista con los elementos de la lista dada ya conside-rados anteriormente. Cuando se considera un elemento de la lista dada, se le inserta en la posición apropiada de la nueva lista.

Respuesta: Si $INSERTSORT(l, l')$ significa que la lsita l' es una versión ordenada de la lista l,
$INSERTSORT(NIL, NIL)$
$INSERSORT(x \cdot l, l') \leftarrow INSERTSORT(l, l'')$ & $INSERT(x, l'', l')$.

7. Especifique un programa en PROLOG para mezclar dos listas ya ordenadas, entregando una lista ordenada.

Respuesta: $MERGE(l_1, l_2, l)$ significa que l es el resultado de mezclar las listas ordenadas l_1 y l_2.
$MERGE(NIL, l, l)$
$MERGE(l, NIL, l)$
$MERGE(x \cdot l_1, y \cdot l_2, x \cdot l) \leftarrow LE(x, y)$ & $MERGE(l_1, y \cdot l_2, l)$
$MERGE(x \cdot l_1, y \cdot l_2, y \cdot l) \leftarrow LE(y, x)$ & $MERGE(x \cdot l_1, y \cdot l_2, l)$.

9.7 Enteros No Negativos en Lógica

Hemos visto cómo la disponibilidad de símbolos de función nos permite representar en lógica estructuras de datos comunes como árboles binarios y listas. Consideremos ahora la representación de los enteros no negativos.

De manera abstracta, todo lo que necesitamos para esto es una constante 0 junto con una *función sucesor* s, donde, por $s(x)$ entendemos $x+1$. Luego, los enteros no negativos son representados por 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, ... Con esta representación a la mano, ahora podemos definir todas las com-

putaciones estándar sobre enteros no negativos por medio de programas en PROLOG. Aquí hay algunos ejemplos:

1. $+(x, y, z)$ – z es la suma de x e y.

   $+(0, x, x)$
   $+(s(x), y, s(z)) \leftarrow +(x, y, z)$

   La figura A9.16 muestra la computación para $+(s(s(0)), s(s(s(0))), w)$, es decir, para la suma de 2 y 3.

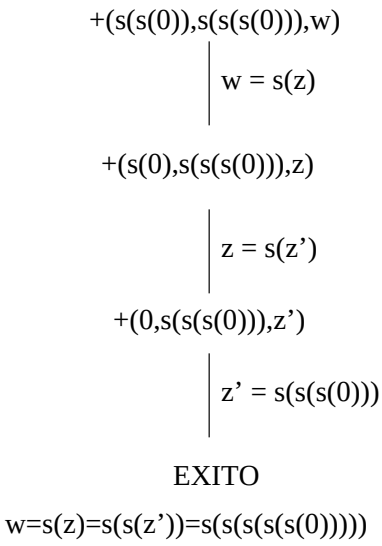


Figura A9.16:

2. $*(x, y, z)$ – z es el producto de x e y.

   $*(0, x, 0)$
   $*(s(x), y, z) \leftarrow *(x, y, w)$ & $+(w, y, z)$

   La figura A9.17 muestra la computación de $*(s(s(0)), s(s(s(0))), u)$, es decir, del producto de 2 y 3.

3. $FACT(x, y)$ – y es el factorial de x.

   $FACT(0, s(0))$
   $FACT(s(x), z) \leftarrow FACT(x, y)$ & $*(s(x), y, z)$

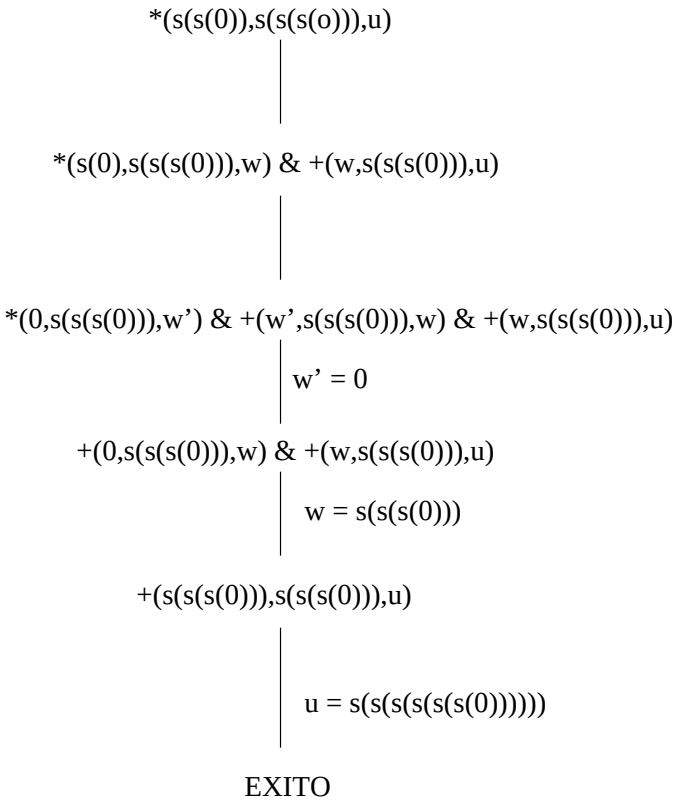

*(s(s(0)),s(s(s(o))),u)
*(s(0),s(s(s(0))),w) & +(w,s(s(s(0))),u)
*(0,s(s(s(0))),w') & +(w',s(s(s(0))),w) & +(w,s(s(s(0))),u)
w' = 0
+(0,s(s(s(0))),w) & +(w,s(s(s(0))),u)
w = s(s(s(0)))
+(s(s(s(0))),s(s(s(0))),u)
u = s(s(s(s(s(0)))))
EXITO

Figura A9.17:

4. $geq(x, y)$ – x es mayor o igual que y.

   $geq(x, 0)$
   $geq(s(x), s(y)) \leftarrow geq(x, y)$

5. $<(x, y)$ – x es menor que y.

   $<(0, s(x))$
   $<(s(x), s(y)) \leftarrow <(x, y)$

6. $\bar{\cdot}\,(x, y, z)$ – z es x menos y si x es mayor o igual que y, y 0 en otro caso.

   $\bar{\cdot}\,(x, y, 0) \leftarrow geq(y, x)$
   $\bar{\cdot}\,(x, y, z) \leftarrow <(y, x)\ \&\ +(y, z, x)$

7. $DIV(x, y, z)$ – z es el resultado de la división entera de x por y. La división entera no está definida en otro caso.

   $DIV(x, y, 0) \leftarrow <(x, y)$
   $DIV(x, y, s(z)) \leftarrow geq(x, y)\ \&\ \bar{\cdot}\,(x, y, w)\ \&\ DIV(w, y, z)$
   Notar que la división por 0 conduce a una conputación que no termina.

8. $MOD(x, y, z)$ – z es el resto de la división entera de x por y.

   $MOD(x, y, z) \leftarrow DIV(x, y, w)\ \&\ *(y, w, u)\ \&\ \bar{\cdot}\,(x, u, z)$
   O, más eficientemente,
   $MOD(x, y, z) \leftarrow <(x, y)$
   $MOD(x, y, z) \leftarrow geq(x, y)\ \&\ \bar{\cdot}\,(x, y, w)\ \&\ MOD(w, y, z)$

9. El algoritmo de Euclides para computar el máximo común divisor de dos enteros no negativos.
   $GCD(x, y, z))$ – z es el máximo común divisor de x e y.

   $GCD(x, 0, x)$
   $GCD(x, y, z) \leftarrow <(0, y)\ \&\ MOD(x, y, w)\ \&\ GCD(y, w, z)$

Hemos visto como, lógicamente, podemos representar y computar con enteros no negativos usando una sola constante 0 y una función sucesor s. De hecho, todas las implementaciones de PROLOG proveen facilidades incorporadas* para

*built–in, en inglés

enteros, tanto positivos como negativos, y también una variedad de predicados del sistema para computar con éstos. Luego, todos los sistemas de PROLOG permiten ingresar cosas como 2 y 4, en lugar de $s(s(0))$ y $s(s(s(s(0))))$, y también enteros negativos como como -2 y -1234. Algunas implementaciones también proveen el tipo de datos real. Los predicados del sistema disponibles varían con la implementación, pero normalmente se provee predicados equivalentes a $+$, $*$, $<$, $\leq$, DIV y MOD. El punto importante es que tales facilidades incorporadas, aunque importantes para la eficiencia computacional, no extienden el poder expresivo de PROLOG visto como lenguaje puramente lógico. De manera abstracta, todas esas facilidades son definibles en lógica, y, de hecho, lo hicimos arriba para los enteros no negativos.

9.7.1 Ejercicios

1. Defina un predicado $LENGHT(l, n)$ que significa que el largo de la lista l es n, por ejemplo, $LENGHT(NIL, 0)$ y $LENGHT(a \cdot b \cdot c \cdot NIL, s(s(s(0))))$ tiene éxito.

Respuesta:
$LENGHT(NIL, 0)$
$LENGHT(x \cdot l, s(n)) \leftarrow LENGHT(l, n)$

2. Defina un predicado de PROLOG $EXP(x, n, y)$ que significa que $y = x^n$ para enteros no negativos x, y y n.

Respuesta:
$EXP(x, 0, s(0))$
$EXP(x, s(n), y) \leftarrow EXP(x, n, y') \ \& \ *(x, y', y)$

9.8 Negación en PROLOG

En nuestra descripción de PROLOG hecha hasta ahora, la negación no ha jugado ningún papel. Todos los procedimientos de PROLOG han sido cláusulas de Horn positivas en las cuales no aparece la negación, y todos los teoremas han sido conjunciones de fórmulas atómicas no negadas. Pero hay computaciones de aparición natural que requieren de negación. Por ejemplo, supongamos que tenemos dos listas l_1 y l_2 y que deseamos definir el predicado $DISJOINT(l_1, l_2)$ que tiene éxito cuando l_1 y l_2 no tienen elementos en común, y que falla en otro caso. Esto definible en PROLOG a través de:

$DISJOINT(NIL, l)$

$$DISJOINT(x \cdot l_1, l_2) \leftarrow not\ MEMBER(x, l_2)\ \&\ DISJOINT(l_1, l_2)$$

Aquí hemos introducido el operador de negación de PROLOG – *not*. En PROLOG, la meta $not\ P$ tiene éxito si la meta P falla, y $not\ P$ falla si P tiene éxito. Más precisamente, al tratar de establecer la meta $not\ P$, el intérprete de PROLOG se llama a sí mismo con la meta P. Si esta llamada falla con P, entonces la meta original $not\ P$ tiene éxito. Si esta llamada tiene éxito con P, entonces la meta original $not\ P$ falla. Luego, $not\ P$ tiene éxito si y sólo si PROLOG falla en demostrar P.

Por ejemplo, considerese la meta de establecer $DISJOINT(a \cdot b \cdot NIL, c \cdot d \cdot NIL)$. Esto entrega la submeta $not\ MEMBER(a, c \cdot d \cdot NIL)\ \&\ DISJOINT(b \cdot NIL, c \cdot d \cdot NIL)$. Como es usual, PROLOG resuelve esta meta conjuntiva de izquierda a derecha, de modo que trata primero de establecer $not\ MEMBER(a, c \cdot d \cdot NIL)$. Para establecer esto, trata de establecer $MEMBER(a, c \cdot d \cdot NIL)$, lo que falla. En consecuencia, $not\ MEMBER(a, c \cdot d \cdot NIL)$ tiene éxito, y PROLOG pasa a establecer $DISJOINT(b \cdot NIL, c \cdot d \cdot NIL)$, que finalmente va a tener éxito.

El operador de negación de PROLOG también puede ser usado en teoremas. Luego, en nuestro mundo de los bloques podríamos preguntar por un ejemplo de un bloque que está a la izquierda de $B7$ y no está arriba de $B5$:

$$IzquierdaDe(x, B7)\ \&\ not\ Arriba(x, B5)$$

En general, PROLOG permite que su operador de negación sea usado en una teorema de nivel superior a ser demostrado, también en el lado derecho de "$\leftarrow$" en un procedimiento. No puede ser usado en la cabeza de un procedimiento ni en un procedimiento simple. Luego, lo siguiente es ilegal en PROLOG y no tiene significado:

$$not\ P, \quad not\ P \leftarrow A$$

Ejemplo: Considere el problema de contar el número de nodos hoja de un árbol binario. Si el árbol x es un nodo hoja, entonces el árbol tiene sólo una hoja*:

$$COUNT(x, s(0)) \leftarrow LEAF(x)$$

Si el árbol binario x tiene n_1 hojas y el árbol binario y tiene n_2 hojas, entonces el árbol $cons(x, y)$ tiene $n_1 + n_2$ hojas:

$$COUNT(x \cdot y, n) \leftarrow COUNT(x, n_1)\ \&\ COUNT(x, n_2)\ \&\ +(n_1, n_2, n)$$

Falta definir el predicado $LEAF$. Un nodo hoja es cualquier nodo que no es un nodo interno. Un nodo interno es cualquier nodo de la de la forma $cons(x, y)$.

*leaf, en inglés

Luego,

$$LEAF(x) \leftarrow not\ INTERNAL(x)$$
$$INTERNAL(x \cdot y)$$

Ejemplo: Considere el problema de eliminar la primera aparición de un elemento x de la lista l. Hagamos que $DELETE(x, l, l')$ signifique que l' es el resultado de eliminar la primera aparición de x de la lista l. Si x no aparece en l, entonces el resultado es l.

$$DELETE(x, NIL, NIL)$$
$$DELETE(x, x \cdot l, l)$$
$$DELETE(x, y \cdot l, y \cdot l') \leftarrow not\ EQ(x, y)\ \&\ DELETE(x, l, l')$$

Notar la tercera cláusula de este programa. El predicado $EQ(x, y)$ significa que x e y son idénticos, de modo que puede ser definido por:

$$EQ(z, z)$$

Esta tercera cláusula dice entonces que si x no es igual al primer elemento de la lista, entonces no elimine el primer elemento, pero en lugar de eso, siga buscandose hacia el final de la lista una aparición de x.

Consideremos más de cerca la tercera cláusula del prgrama $DELETE$. Por la forma en que trabaja el intérprete de PROLOG, uno podría pensar que podría ser reemplazada por

$$DELETE(x, y \cdot l, y \cdot l') \leftarrow DELETE(x, l, l'). \qquad (9.7)$$

Esto es porque antes de que PROLOG ejecutara la tercera cláusula, tendría que haber intentado y fallado con las dos primeras cláusulas. En particular, habría fallado en la segunda cláusula, que trata el caso de la eliminación de un elemento de una lista con un primer elemento idéntico. Luego, si PROLOG llega a ejecutar la tercera cláusula, es porque es el caso en que x e y no son iguales. Como, a esa altura, ya se sabe que no son iguales, no tiene objeto usar recursos computacionales en determinar esto, que es lo que hace la tercera cláusula. En consecuencia, es mejor usar la cláusula más simple y eficiente (9.7).

Hay una falla seria en este argumento: ignora la característica de backtracking de PROLOG. Para ver el problema, considérese una meta de la forma:

$$DELETE(a, a \cdot a \cdot NIL, l)\ \&\ TEST(l)$$

Suponga que $TEST$ es cualquier predicado tal, que $TEST(a \cdot NIL)$ falla, pero $TEST(NIL)$ tiene éxito. Claramente, dado que $DELETE(a, a \cdot a \cdot NIL, l)$ tiene sólo una solución para l (a saber, $l = a \cdot NIL$) y que $TEST(a \cdot NIL)$ falla, la meta de arriba debería fallar. Pero considérese que pasaría en la meta de arriba si usamos la cláusula (9.7) en lugar de la tercera cláusula del programa $DELETE$. La primera parte de la meta – $DELETE(a, a \cdot a \cdot NIL, l)$ – tendrá éxito con $l = a \cdot NIL$ usando la segunda cláusula, pero entonces $TEST(a \cdot NIL)$ falla, con lo que se inicia el backtracking. Entonces, PROLOG trata de resatisfacer $DELETE(a, a \cdot a \cdot NIL, l)$ usando la siguiente cláusula alternativa disponible, que es la cláusula (9.7). Esto tiene éxito con $l = NIL$, y como $TEST(NIL)$ tiene éxito, también lo hace la meta original, lo que es contario a lo que esperábamos.

La moraleja aquí es clara. Los atajos en la definición de procedimientos basados en el conocimiento de cómo trabaja el intérprete de PROLOG pueden ser peligrosos. Deberíamos haber sido alertados por la posibilidad de problemas por el simple hecho que el procedimiento (9.7) para $DELETE$ es *falso* dado nuestro significado subentendido del predicado $DELETE$. Por otro lado, la tercera cláusula del programa $DELETE$ original es verdadera: si l' es el resultado de eliminar x de l y si x e y no son iguales, entonces $y \cdot l'$ es el resultado de eliminar x de $y \cdot l$. De hecho, se puede ver que todos los procedimientos de este capítulo, con la excepción de (9.7), son verdaderos dado el significado subentendido de sus predicados. Esta es una característica poderosa y central de la programación en lógica. Uno indica una una colección de verdades sobre un dominio de aplicación – una *especificación* del dominio. Tal especificación no describe *cómo* realizar una computación; describe sólo lo que es verdadero. Depende de un intérprete demostrador de teoremas el realizar las computaciones.

El siguiente ejemplo demuestra que un compromiso rígido con la verdad de los procedimientos de PROLOG puede ser computacionalmente costoso. Supongamos que queremos un programa para $DIFF(l_1, l_2, l)$, que significa que l es una lista de aquellos elementos de la lista l_1 que no son elementos de l_2.

$$
\begin{aligned}
&DIFF(NIL, l, NIL) \\
&DIFF(x \cdot l_1, l_2, l) \leftarrow MEMBER(x, l_2) \;\&\; DIFF(l_1, l_2, l) \\
&DIFF(x \cdot l_1, l_2, x \cdot l) \leftarrow not\ MEMBER(x, l_2) \;\&\; DIFF(l_1, l_2, l)
\end{aligned}
$$

Ahora, supongamos que PROLOG trata con el segundo procedimiento, y computa $MEMBER(x, l_2)$, que falla. Entonces éste pasará a tratar con el tercer procedimiento, y computará $not\ MEMBER(x,\ l_2)$, lo que significa que

de nuevo computará $MEMBER(x, l_2)$, a pesar del hecho que sabemos que $MEMBER(x, l_2)$ falla y que, luego, $not\ MEMBER(x, l_2)$ tiene éxito. Cuando la lista l_2 es larga, tal test de pertenencia puede ser consumidor de tiempo, de modo que el hacerlo dos veces puede ser groseramente ineficiente. La situación podría ser mucho peor si, en lugar de un test de pertenencia, tuviéramos una computación muy compleja de realizar, de modo que el ejecutarla dos veces podría efectivamente doblar el tiempo de ejecución del programa. Este es un problema computacional bien conocido de programas en PROLOG. En consecuencia, una primitiva de control, llamada "cut" es provista por todas las implementaciones de PROLOG como una forma de evadir este problema. Como nuestro propósito en este capítulo es el de presentar los fundamentos lógicos de PROLOG, omitiremos los detalles del operador no lógico "cut".

9.8.1 Ejercicios

1. Escriba los siguientes programas en PROLOG:

   (a) $COUNT(t, n)$, donde n es el número total de nodos, tanto internos como hojas, del árbol binario t.

   Respuesta:
   $COUNT(x, s(0)) \leftarrow LEAF(x)$
   $COUNT(x \cdot y, n) \leftarrow COUNT(x, n_1)\ \&\ COUNT(y, n_2)\ \&$
   $+(n_1, n_2, n')\ \&\ +(n', s(0), n)$

   (b) $LABELS(t, l)$, donde l es una lista de todas las etiquetas de hojas del árbol binario t, con repeticiones permitidas si más de una hoja del árbol t tienen la misma etiqueta.

   Respuesta:
   $LABELS(t, t \cdot NIL) \leftarrow LEAF(t)$
   $LABELS(x \cdot y, l) \leftarrow (x, l_1)\ \&\ LABELS(y, l_2)\ \&\ APPEND(l_1, l_2, l)$

   (c) Lo mismo que (b), pero sin permitir repeticiones.

   Respuesta:
   $LABELS(t, t \cdot NIL) \leftarrow LEAF(t)$
   $LABELS(x \cdot y, l) \leftarrow (x, l_1)\ \&\ LABELS(y, l_2)\ \&\ U\text{-}APPEND(l_1, l_2, l)$

   $U\text{-}APPEND(l_1, l_2, l)$ significa que l es el resultado de concatenar las listas l_1 y l_2, donde l_1 ni l_2 contienen elementos repetidos, y sin elementos repetidos en l.

$U\text{-}APPEND(NIL, l, l)$
$U\text{-}APPEND(x \cdot l_1, l_2, l) \leftarrow MEMBER(x, l_2)$ & $U\text{-}APPEND(l_1, l_2, l)$
$U\text{-}APPEND(x \cdot l_1, l_2, x \cdot l) \leftarrow not\ MEMBER(x, l_2)$ & $U\text{-}APPEND(l_1, l_2, l)$

2. Escriba los siguientes programas en PROLOG:

(a) $SUBST(x, y, l, l')$ – la lista l' es el resultado de substituir por x cada aparición de y en la lista l.

Respuesta:
$SUBST(x, y, NIL, NIL)$
$SUBST(x, y, y \cdot l, x \cdot l') \leftarrow SUBST(x, y, l, l')$
$SUBST(x, y, z \cdot l, z \cdot l') \leftarrow not\ EQ(x, y)$ & $SUBST(x, y, l, l')$

(b) $BIN\text{-}SUBST(x, y, t_1, t_2)$ – t_2 es el árbol binario obtenido por substitución por x de cada aparición de la hoja y en el árbol binario t_1.

Respuesta:
$BIN\text{-}SUBST(x, y, y, x)$
$BIN\text{-}SUBST(x, y, z, z) \leftarrow not\ EQ(y, z)$ & $LEAF(z)$
$BIN\text{-}SUBST(x, y, v \cdot w, t_1 \cdot t_2) \leftarrow BIN\text{-}SUBST(x, y, v, t_1)$ &
$BIN\text{-}SUBST(x, y, w, t_2)$

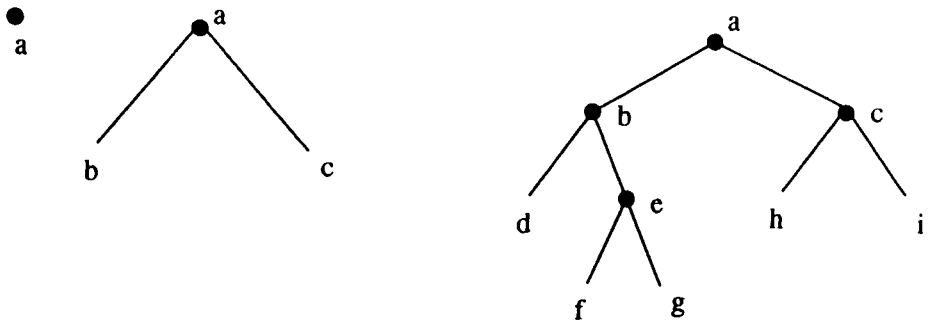
(c) $INTERSECTION(l_1, l_2, l_3)$ – l_3 es la lista de elementos en común a l_1 y l_2.

Respuesta:
$INTERSECTION(NIL, l, NIL)$
$INTERSECTION(x \cdot l_1, l_2, x \cdot l_3) \leftarrow MEMBER(x, l_2)$ &
$INTERSECTION(l_1, l_2, l_3)$
$INTERSECTION(x \cdot l_1, l_2, l_3) \leftarrow not\ MEMBER(x, l_2)$ &
$INTERSECTION(l_1, l_2, l_3)$

3. Arboles Binarios Totalmente Etiquetados

Estos son árboles binarios cuyos nodos están todos etiquetados

Ejemplos:



Podemos representar tales estructuras de datos usando el símbolo de función ternario bt, donde $bt(label, left, right)$ representa el árbol binario cuya raíz está etiquetada por *label*, y cuyos subárboles izquierdo y derecho son *left* y *right*, respectivamente. Luego, los árboles de arriba son representados así:

$$a,\ bt(a,b,c),\ bt(a,bt(b,d,bt(e,f,g)),bt(c,h,i)).$$

Escriba los siguientes programas:

(a) $LABELS(t,l)$ – l es la lista de todas las etiquetas de nodos del árbol t, con repeticiones permitidas si más de un nodo de t tiene la misma etiqueta.

Respuesta:
$LABELS(t, t \cdot NIL) \leftarrow LEAF(t)$
$LABELS(bt(x,y,z), x \cdot l) \leftarrow LABELS(y,l_1)\ \&\ LABELS(z,l_2)\ \&$
$APPEND(l_1,l_2,l)$

(b) Lo mismo que (a), pero si permitir repeticiones.

Respuesta:
$LABELS(t, t \cdot NIL) \leftarrow LEAF(t)$
$LABELS(bt(x,y,z), l) \leftarrow LABELS(y,l_1)\ \&\ LABELS(z,l_2)\ \&$
$U\text{-}APPEND(l_1,l_2,l_3)\ \&\ TEST(x,l,l_3)$

$TEST(x,l,l_3)$ significa que l es igual a l_3 si x es un miembro de l_3, y l es igual a $x \cdot l_3$ si x no es un miembro de l_3:

Respuesta:
$DELETE(t, l, t) \leftarrow LEAF(t)$
$DELETE(bt(l, x, y), l, l)$
$DELETE(bt(z, x, y), l, bt(z, x', y')) \leftarrow not\ EQ(z, l)\ \&\ DELETE(x, l, x')\ \&$
$DELETE(y, l, y')$

9.9 Otras Características de PROLOG

Para convertirlo en un lenguaje de programación genuinamente práctico, todas las implementaciones de PROLOG incorporan varias facilidades por arriba y más allá del núcleo puramente lógico de PROLOG. En consecuencia, se provee varias primitivas de entrada/salida para leer datos e imprimir resultados de manera interactiva. Otras primitivas permiten agregar y eliminar proce-dimientos dinámicamente del programa actual. Predicados para manejar archivos son provistos normalmente. También hay elaboradas facilidades de "debugging" para seguirle la pista a las ejecuciones de programas. Como observamos anteriormente, todas las implementaciones de PROLOG proporcionan representaciones incorporadas de listas y enteros, y otras proveen el tipo de datos "real", también. Facilidades para definir símbolos de función infijos también son provistos siempre, junto con precedencia de operadores. Aunque PROLOG normalmente corre de manera interpretada, también puede ser compilado, y existen compiladores para algunos dialectos de PROLOG en ciertas máquinas.

PROLOG es un programa interactivo, como LISP y APL. Típicamente, uno habla con PROLOG desde un terminal. Uno puede ingresar nuevas cláusulas, eliminar cláusulas, o editar cláusulas existentes. Cuando una meta de nivel superior es ingresada, PROLOG la ejecuta de inmediato, informando al usuario si tuvo éxito o no al ejecutar la meta. Si tiene éxito, PROLOG también informa al usuario de los valores tomados por las variables en la meta. PROLOG queda entonces listo para aceptar nuevas entradas.

Como LISP, y al contrario de APL, PROLOG es más apropiado para computaciones simbólicas, no numéricas. Ha sido ampliamente usado para aplicaciones en inteligencia artificial, bases de datos, matemática simbólica, comprensión de lenguaje natural, escritura de compiladores, etc. La experiencia disponible indica que PROLOG se compara favorablemente con LISP en rapidez de ejecución, pero proporcionando código considerablemente más compacto que lo que hace LISP.

9.10 Demostraciones Descendentes y Resolución

Ahora demostraremos que, en realidad, no hay nada nuevo en la noción de demostración descendente, que tales demostraciones son formas escondidas de una estrategia de resolución particular. Como preparación para esto, recuérdese la forma que los axiomas y teoremas tenían permitido tomar en la presentación de la teoría de demostraciones descendentes.

Axiomas:

> Cláusulas de Horn positivas, es decir, fórmulas sin cuantificadores de la forma P, o de la forma $P_1 \ \& \ \cdots \ \& \ P_n \ \rightarrow \ P$, donde P, P_1, $\ldots, P_n$ son todas fórmulas atómicas. Las variables libres en tales fórmulas se entienden universalmente cuantificadas. Luego, si $x_1, \ldots, x_n$ son todas las variables en el axioma $P_1 \ \& \ \cdots \ \& \ P_n \ \rightarrow \ P$ o el axioma P, entonces esos axiomas son abreviaturas para $\forall\, x_1 \cdots \forall\, x_n \ (P_1 \ \& \ \cdots \ \& \ P_n \ \rightarrow \ P)$ y $\forall\, x_1 \cdots \forall\, x_n \ P$, respectivamente.

Teoremas:

> Conjunciones $T_1 \ \& \ \cdots \ \& \ T_m$ de fórmuals atómicas T_i. Estos pueden contener variables que, al tomar valores en una demostración, proporcionan ejemplos de cosas para las cuales el teorema se cumple. Tales variables se entienden *existencialmente cuantificadas*. Luego, si $x_1, \ldots, x_r$ son todas las variables libres que aparecen en el teorema $T_1 \ \& \ \cdots \ \& \ T_m$, entonces $T_1 \ \& \ \cdots \ \& \ T_m$ es una abreviatura para $\exists \ x_1 \cdots \exists \ x_r \ (T_1 \ \& \ \cdots \ \& \ T_m)$. Un teorema con variables libres afirma la existencia de valores para esas variables para los cuales el teorema se cumple.

Ahora consideremos cómo los teoremas de la forma anterior pueden ser demostrados a partir de axiomas de Horn positivos usando resolución como se hace en la teoría de resolución. Negamos el teorema, convertimos esto a forma clausal, lo agregamos a la forma clausal de los axiomas, y tratamos de derivar la cláusula vacía a partir del conjunto resultante de cláusulas. La negación del teorema $\exists \ x_1 \cdots \exists \ x_r \ (T_1 \ \& \ \cdots \ \& \ T_m)$ es $\forall \ x_1 \cdots \forall \ x_r \ (\neg T_1 \vee \cdots \vee \neg T_m)$, cuya forma clausal es $\neg T_1 \vee \cdots \vee \neg T_m$. Las formas clausales de los axiomas que tienen forma de cláusula de Horn positiva $\forall\, x_1 \cdots \forall\, x_n \ (P_1 \ \& \ \cdots \ \& \ P_n \ \rightarrow \ P)$ y $\forall\, x_1 \cdots \forall\, x_n \ P$, son $\neg P_1 \vee \cdots \vee \neg P_n \vee P$ y P, respectivamente. Con

tales cláusulas podemos intentar derivar la cláusula vacía usando la siguiente estrategia de resolución lineal* (Figura A9.18):

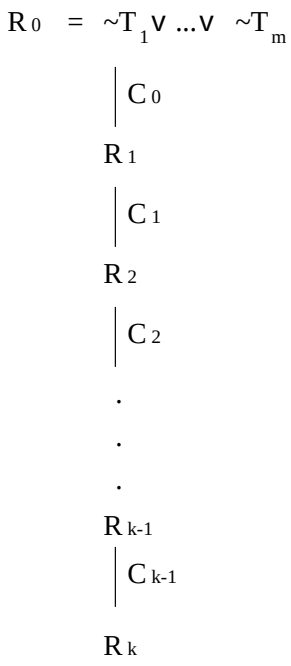


Figura A9.18: $C_0, C_1, \ldots, C_{k-1}$ son cláusulas de los axiomas

1. La cláusula superior es $\neg T_1 \vee \cdots \vee \neg T_m$, es decir, la negación del teorema a ser demostrado.

2. Cada cláusula lateral C_i es una cláusula tomada de los axiomas.

Nótese que cada C_i tiene exactamente un literal positivo. En consecuencia, como R_0 tiene sólo literales negativos, cualquier resolvente R_1 de R_0 y C_0 tiene sólo lietarles negativos. Luego, cualquier resolvente R_2 de R_1 y C_1 tiene sólo literales negativos, etc. Así, todos los R_i tienen sólo literales negativos. Ahora, además de la condiciones 1. y 2. de arriba, imponemos el requerimiento:

3. Al derivar R_{i+1} a partir de R_i y C_i, es el literal *más a la izquierda* en R_i el que se resuelve. Más encima, si R_i es $\neg G_1 \vee \cdots \neg G_n$ y C_i es

*Por definición, en un árbol de resolución lineal, en cada paso de resolución interviene una de las cláusulas originales, provenientes de los axiomas, y un predecesor de la meta superior (top–goal), viendo la demostración como un árbol con raíz $\Box$.

$\neg P_1 \vee \cdots \vee P_r \vee P$, de modo que G_1 unifica con P con $umg\,\sigma$, entonces R_{i+1} es $(\neg P_1 \vee \cdots \vee \neg P_r \vee \neg G_2 \vee \cdots \vee \neg G_n)\sigma$, es decir, *ordenamos* los literales en R_{i+1} preservando el orden de los literales negativos de R_i y C_i, colocando los literales negativos de C_i antes que los de R_i.

Ilustraremos esto con un ejemplo tomado de nuestro mundo de los bloques, cuyos axiomas eran los siguientes:

$$
\begin{array}{ll}
Encima(B1, B2) & (H1) \\
Encima(B3, B4) & (H2) \\
Encima(B4, B5) & (H3) \\
Encima(B5, B6) & (H4) \\
IzquierdaDe(B2, B6) & (H5) \\
IzquierdaDe(B6, B7) & (H6) \\
Arriba(x, y)\ \&\ IzquierdaDe(y, z) \rightarrow IzquierdaDe(x, z) & (H7) \\
Arriba(y, z)\ \&\ IzquierdaDe(x, z) \rightarrow IzquierdaDe(x, y) & (H8) \\
IzquierdaDe(x, y)\ \&\ IzquierdaDe(y, z) \rightarrow IzquierdaDe(x, z) & (H9) \\
IzquierdaDe(x, y) \rightarrow DerechaDe(y, x) & (H10) \\
Encima(x, y) \rightarrow Arriba(x, y) & (H11) \\
Encima(x, y)\ \&\ Arriba(y, z) \rightarrow Arriba(x, z) & (H12)
\end{array}
$$

Estos tienen las siguientes formas clausales:

$$
\begin{array}{ll}
Encima(B1, B2) & (C1) \\
Encima(B3, B4) & (C2) \\
Encima(B4, B5) & (C3) \\
Encima(B5, B6) & (C4) \\
IzquierdaDe(B2, B6) & (C5) \\
IzquierdaDe(B6, B7) & (C6) \\
\neg Arriba(x, y) \vee \neg IzquierdaDe(y, z) \vee IzquierdaDe(x, z) & (C7) \\
\neg Arriba(y, z) \vee \neg IzquierdaDe(x, z) \vee IzquierdaDe(x, y) & (C8) \\
\neg IzquierdaDe(x, y) \vee \neg IzquierdaDe(y, z) \vee IzquierdaDe(x, z) & (C9) \\
\neg IzquierdaDe(x, y) \vee DerechaDe(y, x) & (C10) \\
\neg Encima(x, y) \vee Arriba(x, y) & (C11) \\
\neg Encima(x, y) \vee \neg Arriba(y, z) \vee Arriba(x, z) & (C12)
\end{array}
$$

Supongamos que el teorema a ser demostrado es $\exists\ x\ (IzquierdaDe(B_1, x)\ \&\ IzquierdaDe(x, B_7)\ \&\ Arriba(B_5, x))$, que corresponde a preguntar por un ejemplo de un bloque x con las propiedades de que B_1 está a la izquierda de él, él está a la izquierda de B_7, y B_5 está arriba de él. Cuando se le niega y se le convierte a forma clausal, se obtiene:

$$\neg IzquierdaDe(b1, x) \vee \neg IzquierdaDe(x, B7) \vee \neg Arriba(B5, x).$$

La figura A9.19 muestra una derivación de la cláusula vacía para este teorema usando la estrategia de resolución particular descrita arriba.

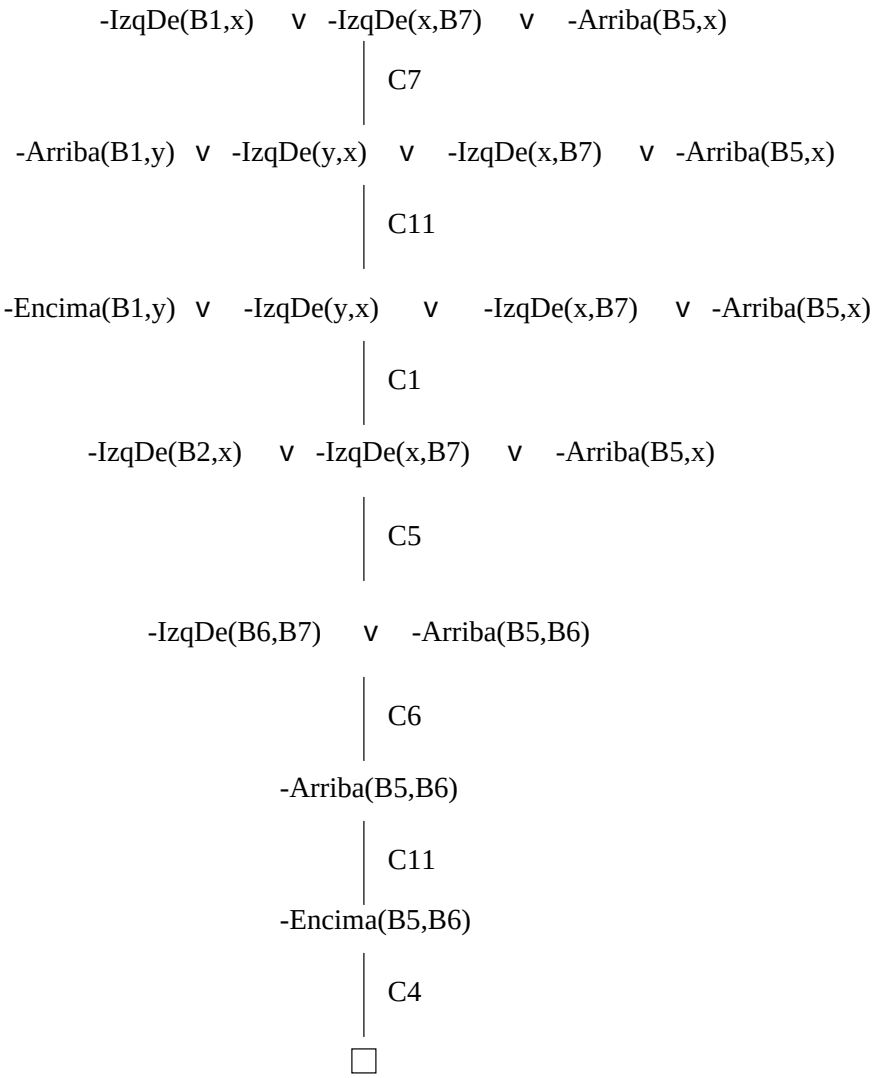


Figura A9.19:

Ahora, a partir de una demostración por resolución como la de la figura A9.19, podemos construir una demostración descendente del mismo teorema, como en la figura A9.20. En lugar de tener como nodo superior la negación del teorema a ser demostrado, como ocurre en demostraciones por resolución, se usa el teorema mismo en el nodo superior para la demostración descendente. En lugar de representar los axiomas como cláusulas como en resolución, se representan como implicaciones para la demostración descendente. Cada operación de

resolución corresponde, en el caso descendente, a una generación de submetas. La cláusula vacía de la resolución corresponde al nodo exitoso en la en la demostración descendente. Las dos demostraciones de las figuras A9.19 y A9.20 son idénticas, excepto por las etiquetas de los nodos de una son las negaciones de las etiquetas de los nodos de la otra.

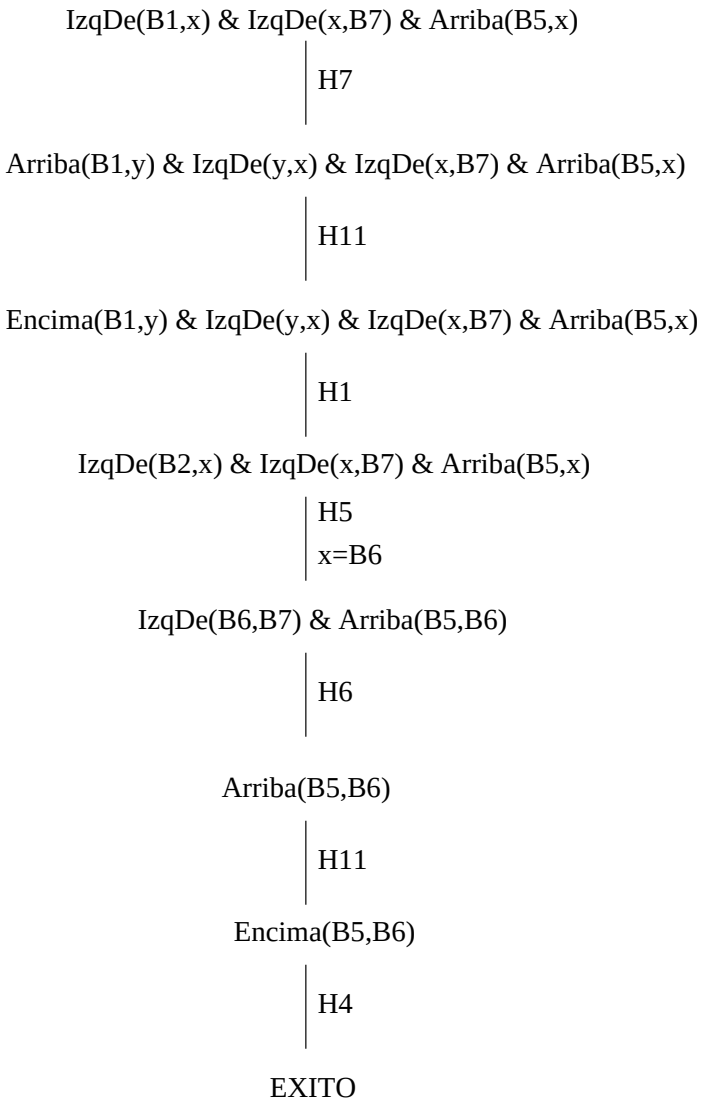


Figura A9.20:

La figura 9.21 resume la correspondencia entre la estrategia de resolución que hemos estado considerando y las demostraciones descendentes. Para cada tal demostración por resolución, hay una demostración descendente idéntica, excepto que sus nodos están etiquetados por las negaciones de las etique-

tas de los nodos de la demostración por resolución. Recíprocamente, a cada demostración descendente corresponde una tal demsotración por resolución. Luego, a partir de la demostración descendente de la figura A9.20, podemos construir trivialmente la demostración por resolución de la figura A9.19. Hay una correspondencia 1–1 entre demostraciones descendentes y demostraciones por resolución del tipo que hemos estado considerando. Concluimos que las demostraciones descendentes no son nada nuevo, sino variantes notacionales de una estrategia de resolución particular.

	Resolución	Derivación Descendente
Nodo Superior	Negación del Teorema: $\neg T_1 \vee \cdots \vee \neg T_n$ Literales ordenados de izquierda a derecha	Teorema: $T_1 \& \cdots \& T_n$ Literales ordenados de izquierda a derecha
Axiomas	1. $\neg P_1 \vee \cdots \vee \neg P_r \vee P$ Literales $\neg P_1$ ordenados de izquierda a derecha 2. P	1. $P_1 \& \cdots \& P_r \rightarrow P$ Literales $\neg P_1$ ordenados de izquierda a derecha 2. P
Nodo Típico en Derivación	$\neg G_1 \vee \cdots \vee \neg G_s$ Literales ordenados de izquierda a derecha	$G_1 \& \cdots \& G_s$ Literales ordenados de izquierda a derecha
Nodo Sucesor en Derivación	Obtenido resolviendo con G_1 1. usando axioma $\neg P_1 \vee \cdots \vee \neg P_r \vee P$ para entregar $(\neg P_1 \vee \cdots \vee \neg P_r \vee \neg G_2 \vee \cdots \vee \neg G_n)\sigma$ Literales ordenados de izquierda a derecha 2. usando axioma P para entregar $(\neg G_2 \vee \cdots \vee \neg G_n)\sigma$ Literales ordenados de izquierda a derecha	Obtenido generando una submeta de G_1 1. usando axioma $P_1 \& \cdots \& P_r \rightarrow P$ para entregar $(P_1 \& \cdots \& P_r \& G_2 \& \cdots \& G_n)\sigma$ Literales ordenados de izquierda a derecha 2. usando axioma P para entregar $(G_2 \& \cdots \& G_n)\sigma$ Literales ordenados de izquierda a derecha
Derivación Exitosa de Teorema	Nodo Inferior: $\square$	Nodo Inferior EXITO

Figura 9.21

9.11 Completidad de las Demostraciones Descendentes

Recordemos que hemos desarrollado nuestra teoría de las demostraciones descendentes para el caso de axiomas en forma clausal de Horn positiva y teoremas de la forma T_1 & $\cdot$ & T_n, donde los T_i son fórmulas atómicas. Las variables libres en los axiomas se entienden como cuantificadas universalmente, mientras que esas en los teoremas, como existencialmente cuantificadas. Una pregunta natural es la siguiente: ¿Son las demostraciones descendentes completas para esta clase de axiomas y teoremas? Más formalmente, si H es un conjunto de tales axiomas y T es un tal teorema, y si $H \models T$, ¿es cierto que hay una demsotración descendente de T a partir de los axiomas en H? Es un hecho reconfortante el que la respuesta a esta pregunta es sí. Siempre que T se concluye semánticamente de H, hay una demostración descendente de T a partir de H. Luego, las demostraciones descendentes son completas para cláusulas de Horn positivas y teoremas positivos conjuntivos. No daremos una demostración de este resultado. El Ejercicio 2 de abajo sugiere una manera de demostrarlo en el caso proposicional*.

Uno no debería concluir de este resultado de completidad que el intérprete de PROLOG es, en consecuencia, completo para la clase de arriba de axiomas y teoremas. PROLOG realiza una búsqueda primero en profundidad para las demostraciones descendentes. El resultado de completidad sólo garantiza que una demostración descendente existe, no que una búsqueda primero en profundidad no se va a perder en una rama infinita del árbol de búsqueda, antes de encontrar una demostración del teorema (Ejercicio 1, abajo). Luego, en general, PROLOG es un demostrador mecánico incompleto.

9.11.1 Ejercicios

1. Dé un ejemplo de un conjunto de cláusulas de Horn positivas, y una meta que tiene una demostración descendente a partir de esas cláusulas, pero que una búsqueda primero en profundidad falle en encontrarla.

Respuesta:
$P \rightarrow Q$ (1)
$Q \rightarrow P$ (2)

*En capítulos anteriores ya tenemos una demostración para el caso proposicional. En el ejercicio mencionado tenemos otra idea para demostrar la completidad en el caso proposicional.

P (3)
(en este orden)

Meta: Q

2. Sea H un conjunto de cláusulas de Horn positivas proposicionales.

(a) Defina una estructura Σ para H en la forma que sigue: para cada símbolo proposicional P que aparezca en una fórmula de H:

$\Sigma(P) = 1$ si hay una derivación descendente de P a partir de H
$\Sigma(P) = 0$ en caso contrario.

Demuestre que Σ es un modelo de H.

(b) En consecuencia, demuestre que si $P_1, \ldots, P_n$ son símbolos proposicionales que aparecen en fórmulas de H, entonces

$H \models P_1 \ \& \ \cdots \ \& \ P_n$ si y sólo si hay una demostración descendente de $P_1 \ \& \ \cdots \ \& \ P_n$ a partir de H.

Esta es la completidad de las derivaciones descendentes para cláusulas de Horn positivas proposicionales y teoremas positivos.

Capítulo 10

Razonamiento con Sentido Común

10.1 Bases de Conocimiento Incompletas

Para fijar ideas, definamos una base de conocimiento, BC, como un conjunto finito de proposiciones escritas en un lenguaje formal de la lógica de predicados de primer orden.

Las bases de conocimiento completas son especialmente atractivas: el valor de verdad (verdadero o falso) de toda proposición del lenguaje queda determinado con respecto a BC, es decir, tenemos conocimiento, expresable en lenguaje de la base, que es completo con respecto a BC.

En esta noción de conocimiento completo no nos preocupamos de las limitaciones fundamentales o de complejidad que tengamos para acceder computacionalmente al conocimiento con respecto a BC, es decir, de la dificultad computacional de decidir el valor de verdad de determinadas proposiciones, o de deducirlas de la BC.

Sin embargo, "la mayoría" de las teorías y bases de conocimiento son incompletas. En consecuencia, con frecuencia estamos enfrentados a conocimiento incompleto. Al respecto, surgen naturalmente los siguientes problemas, que están estrechamente ligados entre sí:

Problema 1: ¿Cómo manejamos el conocimiento incompleto?

1. ¿Como seres humanos?
2. ¿Con medios computacionales?

Problema 2: Si nuestra conocimiento en BC es incompleto,

1. ¿Cómo lo podemos completar?
2. ¿Qué conocimiento agregamos o qué suposiciones adicionales hacemos?
3. ¿Hay una manera natural de completarlo?
4. ¿Cómo lo completamos, cuando queremos determinar sólo cierto tipo restringido de proposiciones con respecto a BC, por ejemplo, oraciones atómicas?

10.2 Suposiciones Típicas y Ejemplos

Ejemplo 1.: Se nos dice *un baobab es un árbol.*

Entonces suponemos que *un baobab tiene hojas verdes* , pues no estamos enfrentados a evidencia en contra.

La suposición (o conjetura o "conclusión") *un baobab tiene hojas verdes* no es una consecuencia lógica, en el sentido clásico, de la información dada, *un baobab es un árbol.* Tampoco lo es su negación, es decir,

$\{$*un baobab es un árbol*$\} \not\models$ *un baobab tiene hojas verdes*

$\{$*un baobab es un árbol*$\} \not\models$ *un baobab no tiene las hojas verdes*

Este es un ejemplo de raciocinio con sentido común en una situación con conocimiento incompleto.

■

Ejemplo 2.:


WIZARD OF ID
FREEZE...PUT EM UP!
YOU'RE OUR PRISONERS!
DO SOMETHING!
LIKE WHAT?
TURN THEM INTO FROGS!
ALAKAZOT!
POOF
POOF
POOF
POOF
POOF
POOF
LITTLE FROGS!
by Parker & Hart

¿Qué pasó aquí? Tenemos un ser con superpoderes:

- ¿un mago?
- ¿un computador?
- ¿un sistema computacional?
- ¿un sistema computacional inteligente usual?

En todo caso, absolutamente carente de sentido común; incapaz de aplicar la regla de sentido común: "típicamente los sapos son pequeños".

■

Queremos sistemas computacionales inteligentes que exhiban sentido común (SC).

Los seres humanos sobreviven gracias al SC, permitiéndoles navegar por la vida enfrentados a cada momento al conocimiento incompleto. El SC proporciona formas de completar el conocimiento: si no nos dicen qué tipo de sapos queremos, suponemos, o conjeturamos, o concluimos, que son normales. El sentido común no sólo es necesario y aparece en la vida diaria, sino también en la práctica científica y computacional, aparte de ser un tema de investigación en inteligencia artificial.

Hay consenso en que no habrá sistemas computacionales realmente inteligentes si éstos no exhiben alguna forma de manejo de sentido común. Y éste es uno de los principales problemas de la Inteligencia Artificial: el diseño de sistemas computacionales para razonamiento automatizado que exhiban sentido común. En palabras de John McCarthy: "... el obtener un lenguaje que exprese conocimiento con sentido común general, para su inclusión en una base de datos general, es el problema clave de la inteligencia artificial".

Esta tarea requiere de:

- Conocer la fuentes del conocimiento y razonamiento por sentido común, y ésta es tarea de la psicología cognitiva, filosofía, lógica, biología, etc.
- Precisarlo y formalizarlo. Ojalá, con lógica formal.
- Mecanizar el razonamiento con sentido común.

Estos problemas son parte del tema principal de la inteligencia artificial actual, a saber, el de **representación de conocimiento**, área en la cual preguntamos cómo representar conocimiento en el computador y cómo usarlo en sistemas computacionales.

Nuestra opción es por **representación lógica de conocimiento**, donde se estudia y diseña formas de representar conocimiento usando lenguajes de la lógica formal para ser usado por sistemas computacionales que, en particular, puedan razonar con ese conocimiento simbólico.

Esta opción se basa en que la lógica simbólica, por ejemplo, en extensiones de lógica de predicados, que tienen sintaxis y semántica claras, precisas, y bien estudiadas. La lógica provee mecanismos deductivos de naturaleza formal, simbólica, que, en pricipio, la hacen apropiada para manejo computacional. Además, la lógica formal es cercana a la actividad mental humana usual, ya que (1) gran parte de ella fue creada precisamente para formalizar los procesos deductivos de los seres humanos; y (2) corresponde, en naturaleza, al razonamiento de los seres humanos basado en procesamiento simbólico de información*.

Los ejemplos 1. y 2. muestran instancias del "**problema de calificación**" (**PC**). Otro ejemplo de este tipo es el siguiente: nos gustaría incluir en una BC ornitológica con sentido común el hecho que "los pájaros vuelan", pero sin tener que explicitar los casos que no califican para la aplicación de esta regla: pingüinos, avestruces, emus, pájaros muertos, ...

Ejemplo 3.: En la figura A10.1 aparece nuestro mundo, es una instancia del mundo de los bloques. Una BC que lo describe es la siguiente:

$$Bloque(a),\ Bloque(b),\ Sobre(a, mesa),\ Sobre(b, a)$$

BC es puramente formal y puede ser usada como una base de datos corriente o como un programa en lógica, p.e. PROLOG. En ella,

- $Bloque$, $Sobre$ son predicados
- a, b $mesa$ son nombres para objetos

*Por lo menos, ésta es una hipótesis que maneja una gran parte de la comunidad de inteligencia artificial.

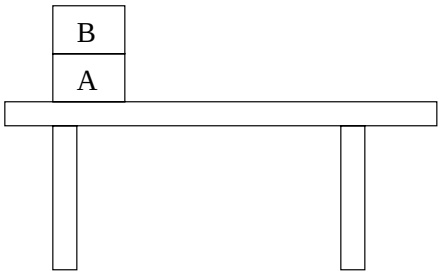


Figura A10.1: El mundo de los bloques.

¿Qué tan buena es esta descripción del mundo?

Una objeción inmediata a esta BC no dice que a, b, $mesa$ son nombres (diferentes) para objetos diferentes. Podríamos decir , ¡Pero eso es de sentido común!; y estaríamos invocando la **Suposición de los Nombres Unicos (SNU)**.

La **SNU** nos dice que nombres diferentes en una BC denotan objetos diferentes (excepto en el caso que la base de conocimiento fuerce lo contrario).

- Esta es una forma de conocimiento basado en sentido común;
- se usa corrientemente, sin explicitarla;
- y una alternativa a ella consiste en agregar explícitamente a BC las aseveraciones sobre las diferencias de nombres.

En el ejemplo, la alternativa es agregar a BC: $a \neq b$, $a \neq mesa$, $b \neq mesa$. Notemos que si hay muchos nombres distintos en BC, el número de hechos de este tipo que hay que agregar para evitar la (meta)suposición **SNU** es muy grande.

El problema de los nombres únicos no es el único problema con nuestra descripción del mundo. Para mostrar otro, quedémonos sólo con la base de conocimiento original BC.

Notemos que BC no dice que a, b, $mesa$ son los únicos objetos. Esto es porque usamos, tácitamente, una suposición de sentido común, a saber, la **Suposición de Clausura del Dominio (SCD)**.

Más precisamente, la **SCD** afirma que los únicos objetos son aquellos para los cuales tenemos nombres (,excepto si BC fuerza lo contrario).

En este ejemplo, la **SCD** (junto con la **SNU**) implica que hay sólo tres objetos (los con nombres a, b y $mesa$).

Un tercer problema con la descripción inicial es que no podemos concluir que no hay ningún objeto sobre el bloque B, es decir, que

$$BC \not\models \forall x \ \neg Sobre(x, b).$$

La **Completación de Predicados (CP)** nos permite enfrentar este problema. Veamos en qué consiste:

Las cuatro aseveraciones iniciales en la BC son lógicamente equivalentes a :

$$\forall x(x = a \vee x = b \ \rightarrow \ Bloque(x))$$

$$\forall x \forall y((x = a \wedge y = mesa) \vee (x = b \wedge y = a) \ \rightarrow \ Sobre(x, y)).$$

- Estas aseveraciones podrían ser consideradas "definiciones" de los predicados $Bloque$ y $Sobre$, respectivamente.
- Son similares a definiciones de conceptos en matemática usual (no formalizada): usamos condiciones suficientes (si), pero se subentiende que son condiciones necesarias y suficientes (si y sólo si).
- Por una convención de sentido común (**CP**), se completa estas semi–definiciones de predicados.
- Se transforma condiciones suficientes ($\rightarrow$) en la definición de un predicado en condiciones necesarias y suficientes ($\leftrightarrow$).

Ahora, si la base de conocimiento BC' es obtenida de BC por completación de los predicados $Bloque$ y $Sobre$, es decir, si se reemplaza $\rightarrow$ por $\leftrightarrow$ en la reformulación de arriba de la BC, tenemos que:

- Los bloques son exactamente (los con nombres) a y b.
- Los únicos objetos que están uno sobre otro son: a sobre $mesa$, y b sobre a.

De aquí se obtiene, en particular,

$$BC' \models \forall x \neg Sobre(x, b).$$

- La completación de predicados debería ser tarea de un mecanismo general de manejo de sentido común.
- No siempre es obvio cómo pasar de las aseveraciones en la BC original a aseveraciones que tienen la forma de una semi–definición (con $\rightarrow$) de un predicado, para después completarlo (obteniendo un $\leftrightarrow$).
- Hay un algoritmo para computar **CP** en bases de conocimiento de ocurrencia frecuente en aplicaciones.

Con la **CP** hemos materializado, a nivel objeto, la metasuposición de SC mediante un procesamiento sintáctico de la BC original*.

Por la oportunidad, su importancia y por completidad, mencionaremos nuevamente la **Suposición del Mundo Cerrado (SMC)**. Ella está estrechamente relacionada con completación de predicados; y es una suposición común en manejo de bases de datos relacionales.

La **SMC** afirma que "en una BC la información dada sobre los predicados es toda y la única información relevante sobre los predicados". En otros términos, toda la información positiva está dada en BC (o es una consecuencia lógica de BC). En consecuencia, todo hecho positivo que no está en, o no es una consecuencia lógica de BC, se supone falso.

En el ejemplo, el hecho positivo $\exists x\ Sobre(x, b)$ (o $Sobre(c, b)$, para cualquier c) no es una consecuencia de BC: $BC \not\models \exists x\ Sobre(x, b)$. Entonces, este hecho se supone falso.

■

Ejemplo 4. (El Problema del Marco†):

Consideremos nuevamente el mundo de los bloques, pero ahora también hay un robot capaz de ejecutar algunas acciones (figura A10.2), por ejemplo, mover bloques de una localidad a otra, pintarlos, ...

*Esto es interesante de observar, pues no necesariamente toda suposición a meta nivel puede ser aterrizada al mismo nivel objeto de la BC.

†Frame Problem, en inglés

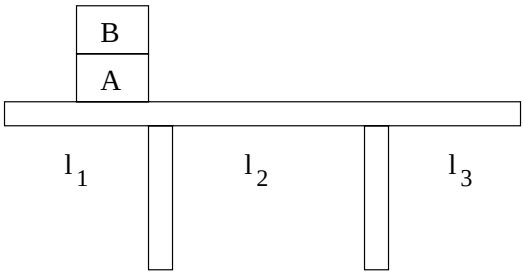


Figura A10.2: Situación inicial S_0.

Una BC (para ser usada por el robot) debe describir los objetos, las acciones, posibles estados para los objetos, situaciones que pueden surgir, ... La descripción de este mundo dinámico, cambiante, puede ser hecha en un lenguaje del cálculo de situaciones. Por ejemplo, podemos tener:

- Hechos sobre la situación inicial s_0, y otros independientes del estado, es decir, esencialmente lo que teníamos en el ejemplo anterior:

  $Manip(b1)$, $Manip(b2)$, $Manip(b3)$,
  $EsCierto(sobre(\ b2, b1), s_0)$, $EsCierto(sobre(b1, l1), s_0)$,
  $EsCierto(sobre(b3, b2), s_0)$, $EsCierto(despej(b3), s_0)$,
  $EsCierto(despej(l2), s_0)$, $EsCierto(despej(l3), s_0)$, $Pos(s_0)$. *

Esta parte de la especificación nos dice cuáles objetos son manipulables; cuáles objetos están sobre otros, cuáles están despejados, y que la situación inicial s_0 es posible.

- Precondiciones para ejecutar la acción *mover* y situaciones que son posibles:

*Hemos decidido presentar este ejemplo en una forma de aparición frecuente en la literatura de inteligencia artificial. En ella, se introduce el predicado *EsCierto* (*Holds*, en inglés) que se aplica a propiedades, que en estricto rigor formal, son términos del lenguaje, por ejemplo, $on(b2, b1)$ es un término del lenguaje. En otras palabras, se "reifica" las propiedades y la verdad de propiedades. Este enfoque tiene sus ventajas, especialmente en el contexto de programación en lógica. En todo caso, este ejemplo se puede poner en términos más cercanos a la práctica lógica clásica, donde la verdad se produce a metanivel, introduciendo fluentes, es decir, predicados dependientes de situaciones, por ejemplo, el predicado $On(x, y, s)$ que, como fórmula atómica (y no, término) expresa que x está sobre y en la situación s. De este modo se puede eliminar el predicado *EsCierto*.

$$Pos(ejecutar(mover(x,y,z),s)) \leftarrow Pos(s), Manip(x), x \neq z,$$
$$EsCierto(despej(x),s),$$
$$EsCierto(despej(z),s),$$
$$EsCierto(sobre(x,y),s).$$

Es decir, la situación que resulta de mover x desde y a z en la situación s, es posible si s es posible, x es manipulable, x es diferente de z; x y z están despejados, y x está sobre y en la situación s.

- Hechos sobre efectos de la acción *mover*:

  $EsCierto(sobre(x,z), ejecutar(mover(x,y,z),s))$,
  $EsCierto(despej(y), ejecutar(mover(x,y,z),s))$.

Es decir, en la situación que resulta de mover x desde y a z en la situación s; x, y, z están en estados: x sobre z, e y despejado.

El Problema del Marco (PM) aparece de la siguiente manera: en ninguna parte se dice lo que no pasa como resultado de una acción. Por esto, no se puede concluir que después de mover el bloque b, desde el bloque a a la localidad l_2 en la situación s_0, el bloque a permanece en la localidad l_1. Más generalmente, no se puede concluir que, pintar el bloque a, no cambia el color de los otros objetos.

No parece deseable agregar explícitamente a BC los muchos hechos de sentido común (los axiomas de marco) que explican lo que no pasa como resultado de las acciones. El problema del marco consiste en representar de manera suscinta, económica, todo este conocimiento negativo, pero de sentido común, de tal modo, que la BC no se torne enorme o inmanejable desde el punto de vista computacional. Este es un problema fundamental de la representación de conocimiento en inteligencia artificial.

Ejemplo 5. (Programación en Lógica):

Consideremos el siguiente programa en lógica ornitológico, que, de hecho, puede ser visto como una BC:

$vuela(X) \leftarrow pajaro(X)\ \&\ not\ an(x)$
$pajaro(X) \leftarrow pinguino(X)$
$pajaro(X) \leftarrow canario(X)$
$an(X) \leftarrow pinguino(X)$
$canario(piolin) \leftarrow$

El predicado *an*, por "anormal*", fue introducido por John McCarthy en su formalización de razonamiento con sentido común en relación con el problema de calificación. Entonces, la primera cláusula dice que "X vuela si X es pajaro y no es anormal".

En PROLOG podemos hacer consultas, por ejemplo: $¿vuela(piolin)?$ Obtenemos respuesta SI, al aplicar la primera cláusula del programa, ya que, primero, $piolin^{\dagger}$ es un pájaro, y, segundo, $not\ an(piolin)$ también es verdadero.

¿Por qué $not\ an(piolin)$ es verdadero para PROLOG? A partir de BC, PROLOG falla al tratar de probar el hecho positivo $an(piolin)$; y entonces, considera $not\ an(piolin)$ como verdadero.

Esto se debe a que la negación en PROLOG (not) no es exactamente negación lógica ($\neg$); es "negación como falla", una negación procedural.

Para PROLOG, "Piolín vuela'' se concluye de BC. Sin embargo, ésta no es una consecuencia lógica, clásica, de la BC. Esta "conclusión" es compatible con el razonamiento con sentido común: si todo lo que sabemos es que Piolín es un pájaro, supongamos que vuela.

10.2.1 Ejercicios

1. Reformule el ejemplo sobre el problema del marco en términos de fluentes, eliminando el predicado *EsCierto*.

2. (a) Verifique y explique por qué, si la base de conocimiento del ejemplo sobre el problema del marco se ve como un programa en lógica, la introducción de la siguiente cláusula resuelve el problema del marco:

$$EsCierto(u, ejecutar(mover(x, y, z), s)) \leftarrow EsCierto(u, s),\ u \neq despej(z),\ u \neq sobre(x, y).$$

*abnormal, en inglés
†Tweety, en inglés

(b) Explique qué papel juega en esto la reificación de propiedades.

(c) Explique qué papel juega el mecanismo de razonamiento de programación en lógica.

(d) ¿Juega algún papel la igualdad (desigualdad) de programación en lógica que aparece en la nueva cláusula? Ella se entiende como igualdad sintáctica, es decir, dos términos son iguales si coinciden como sucesiones de caracteres.

(e) ¿Cómo resolvería el problema del marco en la versión no reificada del ejercicio 1?

3. Explique cómo se puede usar un demostrador mecánico de teoremas para generar planes de acciones que lleven a ciertas configuraciones deseadas.

Ind.: ¿Qué información puede entregar la demostración, a partir de la base de conocimiento, de la siguiente oración?

$$\exists\, s\ (Pos(s) \wedge EsCierto(sobre(b3, b1), s) \wedge EsCierto(sobre(b2, l3), s))$$

4. Considere la siguiente base de conocimiento:

1. $\forall\, x\ (Canario(x) \rightarrow Vuela(x))$
2. $\forall\, x\ (Avestruz(x) \rightarrow \neg Vuela(x))$
3. $\forall\, x\ (Canario(x) \rightarrow Pájaro(x))$
4. $\forall\, x\ (Avestruz(x) \rightarrow Pájaro(x))$
5. $Vuela(Piolín)$

(a) Demuestre por razonamiento abductivo que Piolín es un pájaro.

(b) ¿En qué sentido la conclusión es no monótona?

(c) ¿Cómo se comporta programación en lógica para concluir lo mismo que en (a)?

(d) Haga la completación de Clark los predicados involucrados en la base de conocimiento.

10.3 Inferencia Monótona vs. No–Monótona

Como vimos en capítulos anteriores, el razonamiento en lógica de primer orden es monótono: si de un conjunto de hipótesis H se concluye una consecuencia c, entonces de un conjunto más grande de hipótesis, digamos $H \cup \{h\}$, se sigue concluyendo c (y posiblemente más cosas). Luego, nuevo conocimiento agregado a una base de conocimiento no invalida conclusiones previas.

Sin embargo, el razonamiento con sentido común es esencialmente **no monótono**: hipótesis o conocimiento adicionales pueden invalidar conclusiones previas. Por ejemplo, si sólo sabemos que *froggy* es un sapo, podemos concluir, con sentido común, que para llevarlo con nosotros necesitamos una caja pequeña. Esto queda invalidado si nos dicen, después, que *froggy* es un sapo mutante.

En los ejemplos de la sección anterior, saltamos a conclusiones realizando diferentes formas de inferencias no monótonas. Revisemos algunos de los ejemplos:

Ejemplo 1. (cont.):

Aquí, $BC = \{$*un baobab es un árbol*$\}$. Y la "conclusión" es: *un baobab tiene hojas verdes.*

Si $BC' = BC \cup \{$*un baobab no tiene hojas verdes*$\}$, entonces la "conclusión" queda invalidada.

Ejemplo 3. (el mundo de los bloques, cont.):

Con la BC dada, la "conclusión" es: $\forall\, x\, \neg Sobre(x, b)$. Ahora, si $BC' = BC \cup \{Sobre(c, b)\}$, entonces la "conclusión" queda invalidada.

Ejemplo 5. (programación en lógica, cont.):

Aquí, BC es el programa ornitológico, y la "conclusión" es "Piolín vuela". Sin embargo, si $BC' = BC \cup \{pinguino(piolin)\}$, entonces la "conclusión" queda invalidada.

■

Vemos que la lógica clásica no basta para modelar, dar cuenta de, estas formas de razonamiento. Hay que crear lógicas del y para el razonamiento con sentido común, ojalá, con muchas de las bondades de las clásicas con respecto a sintáxis, semántica, computabilidad, etc. Notemos sí, que en razonamiento con sentido común se puede llegar, por su naturaleza, a contradicciones o a refutar conclusiones previas, luego, en las aplicaciones, se necesita de manera

adicional al mecanismo deductivo provisto por la nueva lógica, de un sistema computacional que "mantenga la consistencia" de la base de conocimiento, detectando inconsistencia, las causas de ella, y las formas de eliminarla. Estos sistemas se llaman en la literatura de inteligencia artificial, "mantenedores de la verdad"*.

En todos los ejemplos precedentes tenemos la siguiente situación:

$$BC \ + \ \text{“}algo \quad más\text{”} \implies \varphi.$$

Denotemos esto así: $BC \ \text{“} \models \text{”} \ \varphi$, donde "$algo \quad más$" puede ser:

- un conjunto de nuevas aseveraciones o suposiciones tácitas, y agregables a la BC a nivel objeto, por ejemplo, una SNU;
- suposiciones extras, a meta nivel, no necesariamente expresables en el lenguaje de la BC (veremos ejemplos en las secciones que siguen);
- un nuevo mecanismo de inferencia (e.g. negación como falla en programación en lógica).

"$algo \quad más$" puede ser visto como una forma de enfrentar el conocimiento incompleto, como una forma de completar la BC, al menos parcialmente.

En general,

$$BC \models \varphi \implies BC \ \text{“} \models \text{”} \ \varphi,$$

pero, usualmente,

$$BC \ \text{“} \models \text{”} \ \varphi \not\implies BC \models \varphi.$$

Por esto, una posible interpretación para la relación $BC \ \text{“} \models \text{”} \ \varphi$, es la siguiente: φ se hace verdadera en todas las estructuras, de un cierto tipo distinguido, que hacen verdadera a BC.

El enfoque actual del razonamiento no monótono consiste en identificar "distinguido" con "minimal" (en algún sentido) en la clase de estructuras que hacen verdadera a BC (modelos de BC). Surge entonces el siguiente problema general:

*truth maintenance systems, en inglés

Problema: Para cada forma de razonamiento no monótono, identifíquese un correspondiente orden (parcial) entre estructuras, y, en consecuencia, una noción de minimalidad.

Un ejemplo de esto es programación en lógica (pensar en PROLOG), que puede ser vista como una forma de razonamiento no monótono. Ahí hay nociones de "preferencia entre modelos", "modelo minimo", "modelo minimal", "modelo perfecto", ..., que, de paso, proporcionan semánticas para programas en lógica.

Otro ejemplo es la noción de **circunscripción** inventada por JohnMcCarthy (1980). Esta es una forma de razonamiento con sentido común (no monótono) que está caracterizada en términos de minimalidad de predicados. Hay aplicaciones de circunscripción a: **PC**, **SCD**, **CP**, **SMC**, **PM**, semánticas para programas en lógica, **SNU** (aunque menos satisfactoria).

10.4 Circunscripción

Consideremos la BC ornitológica del ejemplo 5, pero ahora con negación como falla not reemplazada por negación lógica $\neg$:

$vuela(x) \leftarrow pajaro(x) \wedge \neg\, an(x)$

$pajaro(x) \leftarrow pinguino(x)$

$pajaro(x) \leftarrow canario(x)$

$an(x) \leftarrow pinguino(x)$

$canario(piolin)$

Vemos a BC como una teoría de la lógica de primer orden, de hecho, como una oración que habla sobre el predicado an. Esto lo escribimos así: $BC(an)$.

El significado subentendido de la primera cláusula en BC es: "los pájaros típicamente vuelan". Esto puede también ser interpretado como: el predicado de anormalidad, an, tiene una extensión minimal. Sin embargo, esto no está dicho en BC. Agreguemos, entonces, una nueva aseveración que diga precisamente eso. ¿Cómo?

Como estamos hablando sobre predicados (y no sobre objetos), usemos lógica de segundo orden. La nueva base de conocimiento es

$$BC' := BC(an) \wedge \neg\, \exists\, X\ (BC(X) \wedge X \subsetneqq an).$$

Esta nueva base de conocimiento se llama "la circunscripción del predicado *an* en BC".

Aquí, X es una variable, de segundo orden, para predicados; y BC' dice: es verdadero lo que dice BC sobre el predicado *an* y no hay ningún subconjunto propio de *an* tal, que BC diga lo mismo sobre él que sobre *ab* y la aseveración siga siendo verdadera. Es decir, *an* tiene una extensión minimal.

Esto se aprecia en la figura A10.3:

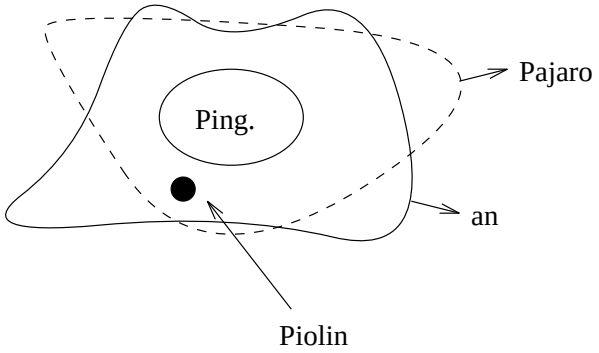


Figura A10.3: Los predicados en la circunscripción.

Ahora: $BC' \models vuela(piolin)$.

Destaquemos algunas características generales de circunscripción:

- Hay una noción bien definida de minimalidad de modelos: modelos minimales con respecto a la extensión del predicado circunscrito.
- Es posible minimizar varios predicados en paralelo; también con otros predicados variables, es decir, flexibles, en el proceso de minimización.
- Es posible asignar prioridades a los predicados en el proceso de minimización, especialmente en caso de minimizaciones en conflicto.
- Hay formulaciones semánticas y sintácticas de circunscripción que coinciden; es decir, hay un teorema de completidad.
- Hay identificadas vastas clases de fórmulas cuya circunscripción conduce a una BC consistente.
- Tal como ocurre con otras formas de razonamiento no monótono, existe la necesidad de contar con un sistema "mantenedor de la verdad", o, mejor, de la consistencia.

- Se requiere especificar cómo, cuándo y qué circunscribir.
- Falta investigación sobre "políticas de circunscripción".
- Como sabemos, la lógica de segundo orden no se comporta bien desde el punto de vista computacional, sin embargo, hay identificadas vastas clases de bases de conocimiento cuya circunscripción colapsa en una base de conocimiento de primer orden, que puede ser determinada con un algoritmo.
- Bajo ciertas condiciones, que usualmente se dan en razonamiento con sentido común, la circunscripción de una base de conocimiento puede ser compilada en un programa en PROLOG.
- La versión de segundo orden de circunscripción puede ser transformada en una base de conocimiento infinita, pero recursiva, de primer orden, introduciendo esquemas como los de inducción. Con esto hay, posiblemente, pérdida parcial de poder expresivo.

El problema principal en torno a circunscripción consiste en determinar cómo incluir un mecanismo circunscriptivo general en un sistema razonador de uso también general.

10.5 Lógica con Reglas por Defecto

La Lógica con Reglas por Defecto* fue introducida en inteligencia artificial por Raymond Reiter (1980) para hacer cierto tipo de razonamiento no monótono, con sentido común, cuando se enfrenta conocimiento incompleto.

Ejemplo: Consideremos la siguiente base de conocimiento BC:

$p\acute{a}jaro(x) \wedge \neg anormal(x) \rightarrow vuela(x)$.

$canario(x) \rightarrow p\acute{a}jaro(x)$.

$ping\ddot{u}ino(x) \rightarrow p\acute{a}jaro(x)$.

$ping\ddot{u}ino(x) \rightarrow anormal(x)$.

$p\acute{a}jaro(piol\acute{i}n)$.

*Default Logic, en inglés

Nos preguntamos: ¿Vuela *piolín*? O, $¿BC \models vuela(piolín)?$

Se tiene: $BC \not\models vuela(piolín)$, y $BC \not\models \neg\, vuela(piolín)$.

Es decir, la pregunta está indeterminada con respecto a BC, tenemos conocimiento incompleto. ¿Qué hacemos?

Notemos la intuición detrás del primer axioma: "normalmente los pájaros vuelan". Además, se nos dice que Piolín es un pájaro y no nos dicen que Piolín es anormal (ni se deduce de los otros axiomas). El "sentido común", en esta situación, nos diría que Piolín entonces vuela; o, al menos, nos permite conjeturar (concluir provisionalmente, suponer) esto, hasta que haya evidencia en contra.

¿Cómo formalizamos esta forma de razonamiento? Introduzcamos una nueva regla de deducción:

$$\frac{\neg anormal(x) \quad \text{consistente (con la } BC \text{ actual)}}{\neg anormal(x)}$$

Intuitivamente, si puede suponer consistentemente que x no es anormal, entonces hágalo (si quiere). Esta es una regla de la lógica por defecto de Reiter.

Ahora sí podemos concluir que Piolín vuela: $\neg anormal(piolín)$ es consistente con la BC, pues $BC \not\models anormal(piolín)$.

Aplicando la regla por defecto podemos concluir: $\neg anormal(piolín)$, más precisamente, que $BC \vdash_d \neg anormal(piolín)$.

Combinando esta conclusión con el primer axioma de BC, obtenemos: $BC \vdash_d vuela(piolín)$ ($\vdash_d$ es una nueva noción de consecuencia deductiva, no clásica, propia de esta lógica).

Notemos que, para esta lógica, hay un aspecto difícil desde el punto de vista computacional, a saber, la de chequear la consistencia de una fórmula con respecto a la base de conocimiento actual. Ya vimos, por el teorema de Church, que el problema general de chequear cosistencia en lógica de primer orden es indecidible, de hecho, ni siquiera es recursivamente enumerable.

Como es de esperar, se pierde la monotonía con esta lógica. Para ver esto, basta verificar que no se cumple:

$$BC \cup \{pingüino(piolín)\} \vdash_d vuela(piolín).$$

Capítulo 11

Otros Temas y Bibliografía

Hay varios temas y aplicaciones que no han sido presentados en este libro. Muchos de ellos pueden ser encontrados en las referencias que damos a continuación. Merecen especial interés los diversos Handbooks que cubren importantes y numerosos temas de lógica en su relación con computación.

Las referencias [4] y [5] contienen artículos expositivos y técnicos sobre muchos temas importantes de la lógica matemática en su forma más clásica. Por ejemplo, [4] trata la teoría axiomática de conjuntos, teoría de modelos, sistemas de deducción natural, teoría de pruebas, aplicaciones algebraicas, decidibilidad de problemas lógicos, etc. En [5] se puede encontar lenguajes de la lógica con diversas formas de expresividad obtenida a través de cuantificadores generalizados. Dentro de esta línea más clasica, recomendamos los libros [13], [19] y [20]. El libro de Kleene [29] contiene un análisis detallado de los sistemas de Hilbert que fueron introducidos en el presente libro. El artículo tutorial [38] contiene una cuidadosa demostración del teorema de incompletidad de Gödel; lo mismo [20].

Los temas tratados en [22], correspondientes a la lógica filosófica, han cobrado mayor importancia en ciencia de computación por las aplicaciones de diversas lógicas no clásicas en esta disciplina. Un ejemplo muy relevante es el de la lógica modal.

Distintas áreas de la lógica que tienen enorme importancia para ciencia de la computación, que, de hecho, ya son parte de esta última, son ilustradas en los handbooks [1], [23]. En particular, en ellos se puede encontrar temas importantes que no tratamos en este libro: lógica ecuacional, sistemas basados en reescritura de términos, sistemas deductivos naturales, lógicas modales, lógica dinámica, demostración mecánica de teoremas, lógica temporal, etc. Además, hay varias aplicaciones de lógica a programación en lógica y a representación de conocimiento en inteligencia artificial.

Los dos volúmenes de [39] contienen temas relevantes para la lógica y la ciencia de computación. El primer volumen se concentra principalmente en algoritmos, estructuras de datos, y complejidad computacional. El segundo volumen presenta diversos aspectos de semántica de lenguajes de programación, lenguajes formales, teoría de bases de datos, semántica de programación en lógica, categorías y especificaciones algebraicas, etc. También recomendamos [2], que contiene muchos temas de computación, pero gran parte de ellos se refieren a representación lógica de conocimiento, razonamiento no monótono, lógica computacional, y demostración mecánica de teoremas.

Un libro clásico de programación en lógica y lógica computacional es [30]. En [31], otro clásico, se hace un tratamiento detallado de programación en lógica y de la semántica de programas en lógica. El libro [12] trae numerosos temas interesantes en esta línea y sobre demostración mecánica de teoremas. [40] se concentra en demostración mecánica de teoremas y en sus aplicaciones.

Los libros [25] y [15] son sobre representación lógica de conocimiento en inteligencia artificial.

En los últimos años han aparecido varios libros sobre lógica para ciencia de la computación, destacamos [21], [33], [35] y [37]. Este último también trae un aspectos de computabilidad, lenguajes formales y NP–completidad. Otro libro sobre computabilidad es [14]. El libro [24] se concentra en el tema de NP–completidad y contiene todo un catálogo de problemas NP–completos.

Hay varios libros y artículos de naturaleza más general, entretenidos, y llenos de comentarios sobre la naturaleza de la computación, la matemática y la lógica matemática; historia de la lógica y la computación; y la vida de muchos de sus protagonistas. Recomendamos [3], [8], [9], [10], [17], [18], [26], [27], [28], [32], [34].

[7] complementa el presente libro con temas como: paradojas de la lógica y su tratamiento, teoría axiomática de conjuntos, construcción de sistemas numéricos, fundamentos de la geometría y el análisis matemático. En particular, se presenta un simple modelo no estándar para el análisis, en un estilo similar al empleado en el presente libro para obtener los modelos no estándar de la aritmética. En [6] se introduce detalladamente los modelos no estádar para el análisis matemático en toda su extensión y riqueza, y se presenta importantes aplicaciones a probabilidad y análisis estocástico.

Finalmente, hay que mencionar las colecciones de artículos famosos, que han hecho historia en los inicios de sus respectivas áreas: [11], [16], [36], sobre lógica matemática clásica, computabilidad e indecidibilidad, y programación

en lógica y demostración mecánica de teoremas, respectivamente.

Bibliography

[1] S. Abramsky, D. Gabbay y T. Maibaum (eds.), "Handbook of Logic in Computer Science", varios volúmenes desde el año 1992, Oxford University Press.

[2] "Annual Reviews in Computer Science", Vols. I-IV, 1986-1989, Annual Reviews Inc., Palo Alto, California.

[3] W. Aspray, "John von Neumann and the Origins of Modern Computing", MIT Press, 1990 (traducción al español, "John von Neumann y los Orígenes de la Computación Moderna", Gedisa, 1993)

[4] J. Barwise (ed.), "Handbook of Mathematical Logic", North Holland, 1977.

[5] J. Barwise y S. Feferman (eds.), "Model–Theoretical Logics", Springer, 1985.

[6] L. Bertossi, "Introducción al Análisis No Standard y a sus Applicaciones en Probabilidad", Notas Matemáticas, 14, Pontificia Universidad Católica de Chile, Fac. de Matemáticas, 1983, 69 pp.

[7] L. Bertossi, V. Marshall y I. Mikenberg, "Fundamentos de Matemática", Pontificia Universidad Católica de Chile, Fac. de Matemáticas, 1985, 138 pp.

[8] L. Bertossi, "Problemas en la Teoría de Complejidad Computacional", en 'En Chile También hay Ciencia', Ihoda, Melnick, Melnick (eds.), Universidad de Chile, Facultad de Ciencias Económicas y Administrativas, 1986, pp. 195–205.

[9] L. Bertossi, "Tendencias Actuales de la Matemática", Revista Universitaria, 23, Pontificia Universidad Católica de Chile, 1988, pp. 56–58.

[10] L. Bertossi, "Observaciones sobre la Actividad Matemática", Apuntes de Ingeniería, 47, Pontificia Universidad Católica de Chile, 1993, pp. 29–38.

[11] P. Benacerraf y H. Putnam (eds.), "Philosophy of Mathematics. Selected Readings", Prentice–Hall, 1964.

[12] A. Bundy, "The Computer Modelling of Mathematical Reasoning", Academic Press, 1983.

[13] J.N. Crossley et al. "What is Mathematical Logic", Oxford University Press, 1972.

[14] N. Cutland, "Computability", Cambridge University Press, 1980.

[15] E. Davis, "Representations of Commonsense Knowledge", Morgan Kaufmann Publ., 1990.

[16] M. Davis (ed.), "The Undecidable", Raven Press Books, 1965.

[17] Ph. Davis y R. Hersh, "The Mathematical Experience", Houghton Mifflin Company, 1982.

[18] Ph. Davis y R. Hersh, "Decartes' Dream", Harcourt Brace Jovanovich Inc., 1986.

[19] H.D. Ebbinghaus, J. Flum y W. Thomas. "Mathematical Logic", Springer, 1984, (2a edición, 1994) (Versión original: "Einfuehrung in die Mathematische Logik", Wissenschaftliche Buchgesellschaft, Darmstadt, 1978).

[20] H. Enderton. "A Mathematical Introduction to Logic", Academic Press, 1972.

[21] M. Fitting. "First–Order Logic and Automated Theorem Proving", Springer, 1990.

[22] D .Gabbay y F .Guenthner (eds.), "Handbook of Philosophical Logic", Vol. I – IV, D. Reidel Plublishing Company, 1983.

[23] D. Gabbay, C. Hogger y J.A. Robinson (eds.), "Handbook of Logic in Artificial Intelligence and Logic Programming", Oxford University Press, varios volúmenes desde 1993.

[24] M. Garey y D. Johnson. "Computers and Intractability: A Guide to the Theory of NP-completeness", W.H. Freeman and Company, New York, 1979.

[25] M. Genesereth y N. Nilsson, "Logical Foundations of Artificial Intelligence", Morgan Kaufmann, 1988.

[26] D. Harel, "Algorithmics. The Spirit of Computing", 2a. edición, Addison–Wesley, 1992.

[27] A. Hodges, "Alan Turing, The Enigma", Simon & Schuster, New York, 1983.

[28] D. Hofstadter, "Gödel, Escher and Bach: An Eternal Golden Braid", Vintage Books, 1980.

[29] S. Kleene, "Mathematical Logic", J. Wiley & Sons, 1967.

[30] R. Kowalsky, "Logic for Problem Solving", North Holland, 1979.

[31] J. Lloyd, "Foundations of Logic Programming", 2nd edition, Springer, 1987.

[32] P.H. Nidditch, "El Desarrollo de la Lógica Matemática", Ediciones Cátedra, Colección Teorema, Madrid, 1983.

[33] S. Reeves y M .Clarke, "Logic for Computer Science", Addison–Wesley, 1990.

[34] C. Reid, "Hilbert", Springer, New York, 1970.

[35] U. Schöning. "Logic for Computer Scientists", Birkhäuser, 1989.

[36] Siekmann y Wrightson (eds.), "Automation of Reasoning", 2 vols., Springer, 1983.

[37] J. Stern. "Fondements Mathematiques de L'Informatique", McGraw–Hill, Paris, 1990.

[38] V.A. Uspensky, "Tutorial: Gödel's Incompleteness Theorem", Theoretical Computer Science, 130, 1994, pp. 239–319.

[39] J. van Leeuwen (ed.), "Handbook of Theoretical Computer Science", Vol. I y II, Elsevier, 1990.

[40] L. Wos, R. Overbeek, E. Lusk y J. Boyle, "Automated Reasoning: Introduction and Applications", 2a ed., McGraw–Hill, 1992.