Libros de Cátedra

La lógica como lenguaje de programación

Aspectos declarativos y procedurales

Clara Smith

exactas

FACULTAD DE INFORMÁTICA

edulp
EDITORIAL DE LA UNLP

UNIVERSIDAD NACIONAL DE LA PLATA

LA LÓGICA COMO LENGUAJE DE PROGRAMACIÓN

ASPECTOS DECLARATIVOS Y PROCEDURALES

Clara Smith

Facultad de Informática

A mis padres,
que fueron docentes universitarios
y amaron incondicionalmente a su universidad,
mi universidad.
Clara Smith

Agradecimientos

Agradezco a mis profesores y colegas: a Marta Sagastume, por su franqueza, su apoyo en el momento de mi defensa de tesis doctoral y sus recomendaciones en el inicio de mis tareas de investigación; a Marisa Gutiérrez, por señalarme el valor y el respeto por la calidad institucional, y por compartir discusiones sobre algoritmos. A Guillermo Simari, por valorar mi trabajo como jefe de trabajos prácticos y por enseñarme la importancia de la opinión divergente en el contexto científico-institucional; a Claudia Pons, por su visión acerca del tipo de contenidos de Lógica que necesita una carrera de informática en nuestro país; y a Javier Díaz, Gabriel Baum y Armando de Giusti, por señalarme tantas veces el camino dentro de nuestra querida Universidad y del Conicet.

Finalmente, agradezco a mis queridos y excelentes alumnos: Maria Noel Adrogué Benas, Carlos Carden y Sebastian Perri, de la cohorte 2021, y Facundo Tomatis y Facundo Miglierini de la cohorte 2023, por haber leído detenidamente los borradores durante la cursada y realizar numerosas y relevantes observaciones y comentarios. Lo valoro mucho. Gracias.

Clara Smith

2023

Índice

Introducción

En el área de programación, un paradigma es una filosofía de representación y de manejo de datos en el que algunos objetos son considerados *distinguidos* o de *primera clase*: por ejemplo, en el paradigma de la programación imperativa lo son las estructuras de control; en el de la orientación a objetos lo son éstos y las jerarquías y relaciones entre ellos, en el paradigma declarativo lo son las fórmulas lógicas y las reglas de inferencia. La **Programación Lógica** pertenece a este último paradigma. El presente texto intenta proveer una vía de acceso, para los alumnos de asignaturas y carreras de Informática, a la vasta literatura técnica referida a la Programación Lógica (*Logic Programming*, en inglés).

Antes de tener alguna experiencia en programación, seguramente pensábamos que programar era una cuestión de "lógica". Al transitar asignaturas de los primeros años en nuestra Facultad, nos enfrentamos a relativamente poco contenido matemático, y hemos aprendido a programar con los esquemas procedurales clásicos; especialmente hemos manejado el concepto de *algoritmo* y escribimos y diseñamos programas que se ocupaban de los mecanismos internos de flujos de datos y de control del procesamiento de información (con estructuras *if-then-else*, *while-do*, *repeat-until*), que si bien tienen su lógica no es la lógica a la que nos referimos en este texto. El diseño de planes de estudio actuales para carreras de informática en nuestro país ha conspirado contra el estudio de asignaturas con intenso contenido formal y contra el estudio pormenorizado de la Programación Lógica. En la actualidad se prioriza una veloz salida laboral de los estudiantes, a quienes se los capacita, legítimamente, con aspectos más pragmáticos de técnicas de programación del momento, requeridas por un mercado ágil. Es tal vez por esta realidad que pocos alumnos y programadores aspiran a un estudio amplio y sistematizado de todos los paradigmas de programación.

Este libro de cátedra es, así, el resultado de múltiples interacciones con quienes, especialmente en el ámbito de nuestra universidad, se han formado técnicamente y han desarrollado sus habilidades de programación casi con exclusividad con lenguajes imperativos y esquemas procedurales. La Programación Lógica aparece en dicho contexto de formación profesional como un universo distinto, en el que concebir a una *fórmula lógica* como una especificación de requerimientos y como un *programa* que puede "ejecutarse" y en el que hay estructuras de datos, resulta, lo menos, sorprendente.

Un **programa lógico** es una descripción hecha en un lenguaje lógico y que expresa conocimiento relevante de una cuestión a resolver.

Como programadores, la Programación Lógica requiere que -dicho a grandes rasgos- describamos la *esencia* de los problemas en lugar de decirle a la computadora cómo debe proceder para resolver dichos problemas, o cómo hacer algunos cálculos, o qué camino seguir para llegar a una solución. Metafóricamente, programar dentro del paradigma declarativo es como trasladarse a un lugar en taxi: vamos en un automóvil que conduce *otra persona*, el conductor, a quien nosotros le especificamos el destino. El conductor es el intérprete lógico, un programa en el que confiamos y gracias al que nos desligamos de detalles del control de flujo de

datos. Este intérprete puede y sabe explotar las capacidades computacionales para sacar conclusiones razonables. La responsabilidad del programador lógico es la de asegurarse de darle *al conductor del taxi* el conocimiento suficiente y correcto para que saque conclusiones.

Pretendemos entusiasmar tanto a alumnos de distintas carreras de informática como a los amantes de la programación en general, aunque provengan de otras áreas de formación básica, en la visión de la lógica tanto como un lenguaje de especificación de conocimiento como de un lenguaje de programación. Aspiramos a que los lectores puedan hacer confluir, a medida que pasan las páginas, sus conocimientos previos sobre matemáticas, lógica, algoritmos básicos, bases de datos, computabilidad y complejidad, y paradigmas de programación. La lógica que usamos es clásica: el Cálculo de Enunciados y un *fragmento decidible* del Cálculo de Predicados. Este es un punto crucial: los programas que escribiremos, bajo ciertas condiciones, normalmente *terminan*. Es deseable aunque no imprescindible que el lector tenga conocimientos básicos de teoría de conjuntos, estructuras de datos, y teoría de la computación. Cuando resulta necesario, reproducimos algunas definiciones para crear el contexto adecuado alrededor de algún concepto que consideramos importante y queremos enfatizar.

Deseamos que este libro sirva como un puente para acceder a literatura técnica de gran nivel como lo son los textos *Foundations of Logic Programming*, de J. W. Lloyd (segunda edición, Springer, 1993), *Modal Logic*, de P. Blackburn, M. de Rijke y Y. Venema (Cambridge University Press, 2001), y *Programming in Prolog Using the ISO Standards*, de William F. Clocksin y Christopher S. Mellish (quinta edición, Springer, 2003). Prevemos que el lector adquiera, al avanzar en la lectura de este texto, un fortalecimiento en su base innovativa ingenieril: en el pensar y utilizar a la lógica como un lenguaje de representación de conocimiento (KRL, por *Knowledge Representation Language*) y así poder planificar y analizar formalmente, desde el punto de vista declarativo, situaciones conflictivas que ameritan una solución automática. También que puedan usar las principales características de un sistema lógico proposicional (especialmente *consistencia* y *decidibilidad*) para poder diseñar, especificar, implementar y validar sistemas computacionales, describirlos con precisión, prever y plantear sus propiedades, potencialidades, y eventuales problemas.

En un orden más complejo de cosas, es bien sabido que las teorías de agentes, y más ampliamente las de toda el área de Inteligencia Artificial, tienen sus fundamentos en la lógica. Esperamos que este libro sirva de base para abordar variados aspectos técnicos de diseño y de programación de sistemas de *agentes inteligentes*. Los sistemas de agentes inteligentes modelan entidades computacionales cognitivas y reactivas que interactúan, coexisten, y dependen unas de otras para alcanzar sus objetivos. Los agentes artificiales copian atributos y capacidades de los humanos tal como aparecen descriptos en la psicosociología y más ampliamente en las ciencias cognitivas: los agentes artificiales pueden “adaptarse”, “razonar”, “aprender”, etc. La lógica que abordamos en este texto seguramente resultará inadecuada para grandes proyectos de Inteligencia Artificial. No obstante, a decir de A. Ramsay en su libro *Formal Methods in Artificial Intelligence* (Cambridge Tracts in Theoretical Computer Science, Series Number 6, revised edition, 1991) existen razones para estudiar seriamente lenguajes formales

que finalmente dejaremos de lado por otros más poderosos. El primer motivo es que estudiando el Cálculo de Enunciados y el Cálculo de Predicados internalizamos sus propiedades (y al internalizarlas las volvemos básicas o *naturales* dentro nuestro). El segundo motivo es que estas propiedades proveen una base sólida para el estudio de otros lenguajes formales (o lógicos) de propósitos más específicos.

Finalmente, es otro de nuestros objetivos que este texto favorezca e incremente en el lector las capacidades de razonamiento abstracto a la hora de pensar un programa. Quizás el concepto que más ayuda a este objetivo es el concepto de *modelo canónico*. Al abordar en el Capítulo 3 la semántica orientada a modelos matemáticos de los programas lógicos, nos enfrentamos a la idea de que algunas estructuras matemáticas podrían corresponderse con una configuración del mundo que constituya una solución del problema a resolver. Sin embargo, conseguir o identificar tal estructura puede ser una tarea difícil, pues a priori podemos no intuir cuál es, o puede ser suficientemente compleja, o puede ser incluso desconocida. Sin embargo, aprenderemos una técnica que permite determinar *qué forma esencial* tienen esas estructuras, si es que existen. Esa técnica permite encontrar una *forma canónica* subyacente a todas las configuraciones del mundo sobre las que podría apoyarse una solución del problema originalmente planteado, más allá de las cualidades de los objetos que luego intervengan, más allá de las funciones definidas entre esos objetos, más allá de las relaciones existentes entre esos objetos. Dominar la técnica de descubrimiento de esas estructuras canónicas estimula intensamente en el lector su capacidad de abstracción.

Para comenzar con la lectura de este libro de cátedra es deseable tener presente los rudimentos de la Lógica de Enunciados: su semántica y su sintaxis, al menos tal como está presentada en el libro *Lógica para Matemáticos*, de A. G. Hamilton (Paraninfo, 1981). Recomendamos repasar los conceptos de: funciones de verdad, tablas de verdad, conectivos lógicos, fórmulas bien formadas, variables proposicionales, formas normales, equivalencia lógica, consecuencia lógica. También es útil manejar los conceptos de: algoritmo, sustitución e instanciación de variables, lenguajes de primer orden, e interpretaciones para lenguajes de primer orden.

Es importante tener presente que en este libro no nos enfocamos en técnicas de programación en Prolog sino en estudiar algunos de los fundamentos de la implementación computacional de Prolog y así comprender cabalmente su funcionamiento para posteriormente poder enfrentar el estudio de técnicas de programación en dicho lenguaje.

En el Capítulo 1 damos una visión de la Lógica de Enunciados como KRL. Para favorecer la intuición, guiamos la presentación con un ejemplo: el problema de 3-colorabilidad de un grafo. Veremos las ventajas y las limitaciones de esta lógica para erigirse como un KRL destacado. Como principal aspecto favorable, tendremos decidibilidad, y simples y efectivas *estructuras de datos* para almacenar y representar estados de cosas del mundo. No obstante, la limitada capacidad para modelar detalles puntuales de universos complejos constituye, en principio, una desventaja de la Lógica de Enunciados desde el punto de vista de los KRL. En el Capítulo 2 tratamos de dejar atrás este obstáculo usando como KRL un fragmento computable de la Lógica

de Predicados, más rico en símbolos y más complejo en su semántica que la Lógica de Enunciados. Abordamos dicho fragmento desde un enfoque semántico u *orientado a modelos* y estudiamos algunos fundamentos lógicos (especialmente el concepto de modelo canónico) del lenguaje Prolog, un clásico en la disciplina. En el Capítulo 3 nos aproximamos a los programas lógicos desde una perspectiva sintáctica u orientada a la demostración automática de teoremas, que llamamos también *orientada a pruebas*. Finalmente, en el Capítulo 4 estudiamos la definición de la regla de inferencia de Prolog llamada *Resolución*, junto con algunos aspectos de la escritura de programas lógicos. Comentamos algunos aspectos procedurales de los programas lógicos, y también la relación entre dichos aspectos procedurales y la semántica declarativa de los programas. En el Epílogo presentamos una breve síntesis de lo que este libro aporta al lector, y remitimos a literatura técnica de primer nivel para continuar el estudio pormenorizado de aspectos formales, computacionales y de ingeniería de la programación declarativa.

CAPÍTULO 1
Determinación de satisfactibilidad en la lógica de enunciados

Clara Smith

En términos generales, la Lógica estudia las formas del pensamiento. En este Capítulo abordamos algunos aspectos de la Lógica como una herramienta de representación formal del conocimiento, representación que luego usamos para obtener conclusiones automáticamente.

La Lógica de Enunciados como Lenguaje de Representación de Conocimiento (KRL)

En las áreas de Ingeniería de Software y de Inteligencia Artificial un lenguaje de representación de conocimiento (abreviamos KRL por *Knowledge Representation Language*) es un conjunto de construcciones sintácticas y semánticas para representar conceptualmente modelos de problemas o situaciones del mundo.

La Lógica de Enunciados[1] puede usarse como KRL porque su lenguaje sirve para escribir con precisión frases sencillas para comunicarnos ya sea con otros programadores, con otros lógicos o con otros matemáticos, o directamente con una computadora. Dichas frases se llaman comúnmente *fórmulas bien formadas (fbfs)* y permiten representar estados de cosas o situaciones del mundo que nos rodea y sobre los que podemos afirmar si ocurren o no ocurren. Por ejemplo: "*llueve"* representa un estado del mundo que nos rodea, y como afirmación puede ser verdadera o falsa.

En este libro, uno de nuestros objetivos como informáticos y como ingenieros de sistemas es el de usar a la Lógica de Enunciados para representar una porción de *conocimiento* asociado a un escenario o porción del mundo a partir del cual pretendemos extraer algunas conclusiones. Esto es muy natural en nosotros los seres humanos; tenemos conocimiento sobre ciertas cosas y ciertas situaciones, conocimiento representado en, por ejemplo, nuestra mente, y a partir de dicho conocimiento sacamos conclusiones todo el tiempo no sólo para saber otras cosas sino también para decidir cómo actuar. El pensamiento y el comportamiento *más* o *menos* inteligente

de una entidad (natural o artificial) dependen del conocimiento del que aquella disponga, y de su capacidad para manejarlo.

Conocimiento. Aproximación general

Llamamos "conocimiento" a un conjunto de afirmaciones justificadamente ciertas que tenemos sobre algún tema o situación: "lo que sabemos" de ese tema o situación. Más modernamente, en contextos de sistemas multiagente, se llama "creencia" al conocimiento que un agente tiene. Así, "lo que el agente sabe" es equivalente a "lo que el agente cree"[2].

Tal como hemos señalado en la Introducción, es conveniente que en este punto tengamos presente los rudimentos de la Lógica de Enunciados: su semántica y su sintaxis, al menos como está presentada en los dos primeros capítulos del libro *Lógica para Matemáticos*, de A. G. Hamilton, texto que usaremos como base para lo que sigue.

El lenguaje de la Lógica de Enunciados

Consiste en un conjunto de símbolos más un conjunto de reglas para organizar esos símbolos y poder escribir frases de un modo tal que éstas puedan ser correctamente reconocidas y leídas por nuestro interlocutor. El lenguaje de la Lógica de Enunciados[3] es el siguiente:

1) un alfabeto (potencialmente infinito) de símbolos: v, ^, →, ¬, respectivamente llamados conectivos para la disyunción, la conjunción y el condicional (binarios), más el símbolo (unario) para la negación; las letras p, q, r,…, x_1, x_2,…, a las que llamamos letras de enunciados, o letras de proposición, o variables proposicionales, o letras proposicionales; y los paréntesis ´ () ´.
2) un conjunto de cadenas finitas de estos símbolos, las fórmulas bien formadas (abreviamos *fbfs*), que se construyen inductivamente de la siguiente manera:

i) cada variable proposicional es una *fbf*, y

ii) si A y B son fbfs entonces también lo son: ¬A, A v B, A ^ B, A → B.

Ejemplos

Podemos simbolizar "llueve" con la letra p. "Si llueve entonces voy al cine" puede simbolizarse como p → q; "Hay sol y haré un paseo" puede escribirse como r ^ s. Cuando la variable proposicional p representa "llueve", a la expresión "no llueve" se la representa: ¬p. En otro momento y en otro contexto la variable p puede representar otro estado de cosas cualquiera.

Observación

A y B son *metavariables*, es decir, variables en las que podemos "almacenar" o "guardar" *fbfs* (que a su vez están compuestas por variables proposicionales). Por ejemplo, la metavariable A puede almacenar ahora a la *fbf* p, y en otro momento o contexto almacenar a la fbf: (q ^ r) → s. Notemos que las metavariables *no son parte del lenguaje*. Son variables que usaremos para referirnos a algunos aspectos de la lógica.

Avancemos con la visión de la Lógica de Enunciados como KRL. Lo que pretendemos de un KRL es que sea sencillo de usar, que nos permita representar conocimiento con cierto nivel de detalle y que dicha representación sea comprensible por nosotros y por otros, que sea legible y transmisible. La Lógica de Enunciados cumple con estos requisitos generales pues es precisa y sencilla en escritura y en significado: los bloques básicos de representación son las fbfs, a las que sumamos a la particular habilidad de modelización que cada uno de nosotros tiene. Ambos elementos, las fbfs y nuestra capacidad de modelizar, nos permitirán construir representaciones en mayor o menor medida fieles a los escenarios con los que hemos de trabajar[4].

Variables proposicionales como estructuras de datos

Desde el punto de vista de los KRL, las variables proposicionales son las mínimas "estructuras de datos" que tenemos disponibles. Cada variable representa abreviadamente una afirmación respecto del mundo o del problema, variable a la que luego le corresponderá o el valor verdadero (V, o *true*, o 1) o el valor falso (F, o *false*, o 0). De este modo, las fbfs (construidas inductivamente como vimos más arriba) son descripciones más o menos sofisticadas de estados del mundo.

Ejemplo de uso de la Lógica de Enunciados como KRL

El problema de la 3-colorabilidad de un grafo. El ejemplo que sigue fue provisto por Daniele Mundici (ex profesor del Departamento de Matemáticas de la Universidad de Milán, Italia) en una serie de clases que dictó en el Departamento de Matemáticas, a principios de la década del dos mil, para oyentes alumnos y profesores de carreras de Matemática, Informática e Ingeniería de la Universidad Nacional de La Plata, Argentina. Con variantes, presentamos a continuación el ejemplo, que ha constituido durante más de quince años el ejemplo motivacional de la cátedra para presentar el uso de la lógica como KRL.

Colorabilidad de un grafo

Supongamos que nos proveen con un grafo[5] particular y nos dan la tarea de tricolorearlo (3-colorearlo). Recibimos el grafo, que identificamos como un conjunto finito de vértices V y un conjunto E de aristas que conectan los vértices, G = (V,E). El grafo dado G es el siguiente:

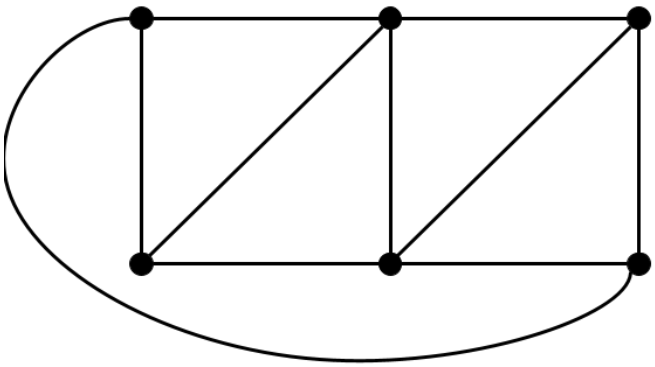


Figura 1. El grafo que intentaremos 3-colorear

Contemplemos el grafo G dado: tiene seis vértices.

Disponemos también de una paleta de tres colores, digamos los colores 1, 2, y 3. Tenemos tres colores disponibles para pintar los vértices.

Colorear el grafo consiste en colorear sus vértices de modo tal que los vértices adyacentes no tengan el mismo color. Posiblemente tengamos ya una primera idea de *qué* es lo que debemos hacer, aunque no sabemos *cómo* va a quedar finalmente coloreado el grafo G, si es que efectivamente podemos 3-colorearlo.

Imaginemos que para resolver el problema trabajamos en colaboración con, digamos, *un marciano*[6], y que es el único que nos asistirá en la tarea de 3-colorear. Nos comunicaremos con el marciano mediante el lenguaje de la Lógica de Enunciados, pero no con todo el lenguaje sino con una porción[7] de éste. Esto es similar a lo que hacemos a diario para comunicarnos con nuestros interlocutores, usamos *parte* del lenguaje que conocemos, no lo usamos completo todo el tiempo.

Para intercambiar información con el marciano, pensemos en alguna pregunta que sea relevante a la tarea de colorear el grafo y que el marciano pueda contestar fácilmente. Por ejemplo: “¿coloreamos el vértice 1 con el color 1?”. Esta es una pregunta sencilla y sumamente

pertinente al problema porque pregunta por el resultado de una acción positiva y efectiva que puede cambiar el estado de cosas en el grafo y así avanzar hacia la terminación del problema (que es comprobar si podemos 3-colorearlo). La pregunta además está formulada con suficiente detalle ya que, vinculada a la acción de colorear, identifica dos elementos esenciales al problema: el vértice a colorear y el color a usar para colorear.

Definimos entonces un conjunto de dieciocho variables proposicionales: $\{x_{ij}\}$. Interpretamos a cada variable como la representación de un vértice pintado de un color[8]. La lectura intuitiva de cada variable x_{ij} es: "¿coloreamos el vértice i con el color j?". A cada una de estas preguntas podemos responder "sí", o "no". Estas variables son todas las que necesitamos, es decir, estas son todas las estructuras de datos que nos son útiles para modelar el problema.

Especificación proposicional (o booleana) del problema

Dijimos que el marciano colabora con nosotros y podrá ayudarnos a conseguir la respuesta acerca de si *es posible 3-colorear el grafo G dado*. De lo primero que debemos estar seguros es de que el marciano conoce el significado de 3-colorear, esto es, que el marciano sabe tres cosas:

que i) cada vértice tiene que estar pintado de *al menos un color*,

que ii) cada vértice tiene que estar pintado *de un único color*,

y que iii) los vértices que son adyacentes deben estar pintados con colores distintos.

Explicarle estas tres restricciones al marciano es *ponernos de acuerdo* con él en lo que significa 3-colorear el grafo. Y cuando nos ponemos de acuerdo con un interlocutor, establecemos o *especificamos* ciertas pautas o *cláusulas* de un pacto o contrato (verbal o escrito), para estar seguras -ambas partes- de lo que estamos hablando, porque vamos a trabajar juntos. En algunas áreas de automatización de procesos, en las que interactúan diferentes agentes naturales o artificiales, estos hitos suelen llamarse *agreement points* (puntos de acuerdo).

Formalicemos ahora cada una de las tres restricciones constitutivas del problema de 3-colorear. Notaremos, al finalizar la tarea, que la Lógica de Enunciados resulta útil y apropiada para escribir sistemas de restricciones como el de la tri-coloración del grafo.

Primera restricción

Cada vértice tiene que estar pintado de al menos un color.

Debemos acordar con el marciano que cada uno de los vértices del 1 al 6 (ver Figura 1), puede ser pintado o con el color 1, o con el color 2, o con el color 3. Expresamos así un aspecto de la tarea de colorear el grafo aunque sin decirle al marciano *de qué color hay que pintar cada vértice*. Descripta exhaustivamente, la restricción queda: "pintamos el vértice 1 del color 1, o pintamos el vértice 1 del color 2, o pintamos el vértice 1 del color 3; y pintamos el vértice 2 del color 1, o pintamos el vértice 2 del color 2, o pintamos el vértice 2 del color 3,..." y así siguiendo hasta "... y pintamos el vértice 6 del color 1, o pintamos el vértice 6 del color 2, o pintamos el vértice 6 del color 3". Formalmente simbolizamos:

$$(x_{11} \vee x_{12} \vee x_{13}) \wedge$$

$$(x_{21} \vee x_{22} \vee x_{13}) \wedge$$

$$(x_{31} \vee x_{32} \vee x_{33}) \wedge$$

$$(x_{41} \vee x_{42} \vee x_{43}) \wedge$$

$$(x_{51} \vee x_{52} \vee x_{53}) \wedge$$

$$(x_{61} \vee x_{62} \vee x_{63}).$$

Esta fbf está escrita en *forma normal conjuntiva* (abreviamos FNC). La forma normal conjuntiva es una manera particular[9] de escribir fbfs.

Notemos que cada uno de los seis paréntesis en la fbf de arriba representa la indicación de pintar el vértice en cuestión con alguno de los tres colores disponibles, sin imponer cuál color. Por ejemplo, el primer paréntesis de la fbf podemos leerlo coloquialmente como: "pintamos el vértice 1 del color 1, o pintamos el vértice 1 del color 2, o pintamos el vértice del color 3". Formalmente, sabemos que (conforme a la función de verdad de la disyunción) dicha disyunción es cierta cuando ocurre que (al menos) uno de los *disyuntores* (esto es, al menos una de las letras del paréntesis) es verdadero, sin importarnos cuál.

Segunda restricción

Debemos acordar con el marciano en que, si bien cumpliendo con la primera restricción cada uno de los seis vértices estará pintado de algún color, *ese color debe ser único para cada vértice*. Esto para evitar que el grafo tenga algún vértice pintado a la vez con dos colores diferentes, que sería una configuración inaceptable. Intuitivamente decimos: "no pintamos el vértice 1 del color

1 y del color 2 a la vez, y tampoco pintamos el vértice 1 del color 1 y del color 3 a la vez, y tampoco pintamos el vértice 1 del color 2 y del color 3 a la vez". Y así para los vértices restantes.

Formalizamos:

$$(\neg(x_{11} \wedge x_{12}) \wedge \neg(x_{11} \wedge x_{13}) \wedge \neg(x_{12} \wedge x_{13})) \wedge$$

$$(\neg(x_{21} \wedge x_{22}) \wedge \neg(x_{21} \wedge x_{23}) \wedge \neg(x_{22} \wedge x_{23})) \wedge$$

$$(\neg(x_{31} \wedge x_{32}) \wedge \neg(x_{31} \wedge x_{33}) \wedge \neg(x_{32} \wedge x_{33})) \wedge$$

$$(\neg(x_{41} \wedge x_{42}) \wedge \neg(x_{41} \wedge x_{43}) \wedge \neg(x_{42} \wedge x_{43})) \wedge$$

$$(\neg(x_{51} \wedge x_{52}) \wedge \neg(x_{51} \wedge x_{53}) \wedge \neg(x_{52} \wedge x_{53})) \wedge$$

$$(\neg(x_{61} \wedge x_{62}) \wedge \neg(x_{61} \wedge x_{63}) \wedge \neg(x_{62} \wedge x_{63})).$$

Es fácil ver que esta fbf no está escrita en FNC[10]: los símbolos de negación no están ajustados a las letras proposicionales, y los paréntesis internos no contienen disyunciones. (Estas dos cualidades sí debe cumplir una fbf que está en FNC.)

Como pretendemos lograr una notación uniforme para ganar en prolijidad y legibilidad, aplicamos una equivalencia de De Morgan[11] a la fbf previa para conseguirnos una fbf en FNC lógicamente equivalente:

$$((\neg x_{11} \vee \neg x_{12}) \wedge (\neg x_{11} \vee \neg x_{13}) \wedge (\neg x_{12} \vee \neg x_{13}) \wedge$$

$$(\neg x_{21} \vee \neg x_{22}) \wedge (\neg x_{21} \vee \neg x_{23}) \wedge (\neg x_{22} \vee \neg x_{23}) \wedge$$

$$(\neg x_{31} \vee \neg x_{32}) \wedge (\neg x_{31} \vee \neg x_{33}) \wedge (\neg x_{32} \vee \neg x_{33}) \wedge$$

$$(\neg x_{41} \vee \neg x_{42}) \wedge (\neg x_{41} \vee \neg x_{43}) \wedge (\neg x_{42} \vee \neg x_{43}) \wedge$$

$$(\neg x_{51} \vee \neg x_{52}) \wedge (\neg x_{51} \vee \neg x_{53}) \wedge (\neg x_{52} \vee \neg x_{53}) \wedge$$

$$(\neg x_{61} \vee \neg x_{62}) \wedge (\neg x_{61} \vee \neg x_{63}) \wedge (\neg x_{62} \vee \neg x_{63})).$$

Seguidamente, conectamos esta fbf con un "^" a la fbf que obtuvimos para la primera restricción para mantener unidas -en una única fbf- tanto la primera como la segunda restricción del problema. Ello porque la conjunción de dos fbf en FNC nos da una fbf también en FNC. (Algunos paréntesis pueden sobrar, y se descartan).

Nota. Cotemporalidad de la conjunción

Prestemos atención a la intuición de cotemporalidad que subyace a la conjunción. Para que una conjunción pueda interpretarse como verdadera, las subfórmulas que integran dicha conjunción deben ser ciertas *todas a la vez*. Esta intuición surge patente de contemplar la tabla de verdad para la función de verdad del conectivo de la conjunción[12]. Recordemos que el único renglón de la tabla de verdad en el que una conjunción es verdadera es aquel en el que todas las subfórmulas son verdaderas.

Tercera restricción

Los vértices que son adyacentes han de tener colores distintos[13].

Formulamos esta restricción intuitivamente, contemplando nuestro grafo: “no podemos pintar el vértice 1 y el vértice 2 a la vez del color 1, y tampoco podemos pintar el vértice 1 y el vértice 4 a la vez del color 1, y tampoco podemos pintar el vértice 1 y el vértice 5 a la vez del color 1” y “no podemos pintar el vértice 1 y el vértice 2 a la vez del color 2, y tampoco podemos pintar el vértice 1 y el vértice 4 a la vez del color 2, y tampoco podemos pintar el vértice 1 y el vértice 5 a la vez del color 2” y “no podemos pintar el vértice 1 y el vértice 2 a la vez del color 3, y tampoco podemos pintar el vértice 1 y el vértice 4 a la vez del color 3, y tampoco podemos pintar el vértice 1 y el vértice 5 a la vez del color 3”. Nos queda:

$$(\neg(x_{11} \wedge x_{21}) \wedge \neg(x_{11} \wedge x_{41}) \wedge \neg(x_{11} \wedge x_{51})) \wedge$$

$$(\neg(x_{12} \wedge x_{22}) \wedge \neg(x_{12} \wedge x_{42}) \wedge \neg(x_{12} \wedge x_{52})) \wedge$$

$$(\neg(x_{13} \wedge x_{23}) \wedge \neg(x_{13} \wedge x_{43}) \wedge \neg(x_{13} \wedge x_{53})).$$

Seguidamente, aplicamos a esta fórmula la equivalencia de De Morgan para obtener una fbf lógicamente equivalente en FNC:

$$((\neg x_{11} \vee \neg x_{21}) \wedge (\neg x_{11} \vee \neg x_{41}) \wedge (\neg x_{11} \vee \neg x_{51})) \wedge$$

$$((\neg x_{12} \vee \neg x_{22}) \vee (\neg x_{12} \vee \neg x_{42}) \wedge (\neg x_{12} \vee \neg x_{52})) \wedge$$

$$((\neg x_{13} \vee \neg x_{23}) \wedge (\neg x_{13} \vee \neg x_{43}) \wedge (\neg x_{13} \vee \neg x_{53})).$$

Esta fbf está en FNC y se corresponde con lo que establece la tercera restricción para el vértice 1 y sus adyacentes.

A esta FNC recién obtenida la unimos con el conectivo "^" a las FNC que construimos previamente para la primera y para la segunda restricción.

Ejercicio

No hemos concluido con la formalización de la tercera restricción. También debemos especificar, para cada vértice del 2 al 6 (del grafo dado G) con sus respectivos adyacentes, la restricción de no pintarlos con el mismo color a la vez. Dejamos esta tarea al lector.

Al completar este ejercicio, unimos con "^" todas las fbfs en una gran FNC. Tenemos entonces formalizado todo el conocimiento referido a lo que es 3-colorear el grafo G en una fbf escrita en FNC.

Ahora: ¿existe una 3-coloración del grafo de la Figura 1? Estamos en condiciones de dar una respuesta a esta pregunta. Ejercitemos nuestra intuición del asunto, desde el punto de vista semántico:

Primero. Armamos la tabla de verdad[14] de la fbf en FNC que tenemos escrita.

Sabemos que podemos interpretar a cada *renglón* de la tabla de verdad como una *configuración posible del mundo*[15], esto es, un estado posible de cosas. Entonces:

Segundo. Controlamos si existe un renglón de la tabla que dé resultado verdadero. Si existe dicho renglón, entonces el grafo sí es 3-coloreable, y el color de cada vértice i es el color j, conforme x_{ij} tenga valor verdadero en ese renglón.

Notemos que de algún modo *transformamos el problema* de determinar 3-colorabilidad en *otro problema*: el de encontrar un "renglón verdadero" en la tabla de verdad de la FNC correspondiente[16]. Entonces, en rigor, ya conocíamos un método para determinar si tenemos una solución al problema; a este método ya lo habíamos aprendido cuando estudiamos lógica básica.

Organicemos a continuación, más prolijamente, cómo trabajar con el marciano.

Método 1 (Semántico). Construcción de la tabla de verdad. Detección de satisfactibilidad

Sabemos que construir la tabla de verdad para la FNC nos llevará 2^n "renglones" o "combinaciones de valores de verdad" (cf. Hamilton), siendo *n* el número de variables proposicionales de la FNC. Para nuestro problema de 3-colorabilidad tendremos entonces una tabla de verdad de 2^{18} renglones, porque tenemos 18 variables proposicionales. Por abuso de lenguaje y notación, leemos entonces a cada renglón de la tabla como la asignación de valores de verdad dada a las variables en ese renglón[17].

Dada una asignación de valores de verdad α, recordemos que[18]:

- α satisface x_{ij} sii $\alpha(x_{ij})$=True,
- α satisface $\neg x_{ij}$ sii $\alpha(x_{ij})$= False,
- α satisface una *cláusula* $C = (L_1 \vee \ldots \vee L_k)$ si y sólo si existe $i \in \{1...k\}$ / $\alpha(L_i)$=True, con L_i literal[19].
- α satisface a la FNC si y sólo si para toda $C_j = (L_1 \vee \ldots \vee L_k) \in$ FNC, $\alpha(C_j)$=True.

La tabla de verdad nos asegura que si la asignación α existe, entonces la encontraremos, aunque α esté en el último renglón de la tabla. El método de construcción de las tablas de verdad es correcto, y es finito; esto último nos importa porque somos informáticos y pretendemos que nuestros métodos de procesar la información *terminen*.

Ejercicio

Dejamos al lector la construcción de la tabla de verdad de la FNC para el problema de colorabilidad del grafo dado, y el hallazgo de α, si existe.

Observación

En una suerte de *dualidad* con la idea de satisfactibilidad, notemos que una fbf A es insatisfactible si y solo si no existe ninguna asignación α de valores de verdad para las variables proposicionales en A que la hagan verdadera. Por ejemplo, sabemos que la fbf: $(p \wedge \neg p)$ es insatisfactible porque no es posible asignar valores de verdad a la única variable proposicional de dicha fbf (p) de modo que la fbf tenga un renglón verdadero en su tabla de verdad, pues si p es verdadero entonces ¬p es falso, y viceversa. Podemos plantearnos entonces, naturalmente,

un segundo método, dual del Método 1: en lugar de intentar detectar satisfactibilidad de la FNC, intentar detectar su insatisfactibilidad. Avancemos con esta idea.

Definición

Una fbf A es insatisfactible cuando no existe una asignación α de valores de verdad para las letras de proposición[20] en A tal que la fbf es verdadera. Esto sucede cuando la fbf contiene una contradicción. Intentemos, entonces, *detectar contradicciones* dentro de una fbf. Veamos un método para esto.

Método 2. (Sintáctico u orientado a la demostración automática de teoremas). Detección de insatisfactibilidad. Regla de Resolución

En nuestra iniciación a la Lógica seguramente hemos estudiado la regla de inferencia llamada *Modus Ponens*[21]. El Modus Ponens (abreviamos MP) es usualmente considerado como la regla de inferencia esencial en cualquier sistema formal proposicional; una regla que se usa para obtener deducciones.

Modus Ponens

Podemos describir intuitivamente cómo trabaja el MP como sigue: tenemos dos fbfs dadas como verdaderas: un condicional y una fbf que coincide exactamente con el antecedente de ese condicional. A partir de estas dos fbfs podemos concluir el consecuente del condicional, al que asumimos también como verdadero. Formalmente escribimos:

$$\frac{A \rightarrow B \qquad A}{B}$$

con A y B fbfs cualesquiera del Cálculo de Enunciados.

Observación desde el punto de vista sintáctico

MP es una "regla sintáctica" que nos permite manipular fbfs gracias a *la forma* de éstas, sin prestar demasiada atención a cuáles son las subfórmulas integrantes. MP es una *Rule of Detachment*[22] porque podemos "desprender" o "despegar" el consecuente de un modo seguro[23]. Esto significa, dicho coloquialmente, que en lugar de trabajar con dos fbfs, $A \rightarrow B$ y A, podemos trabajar con una sola fbf, B. MP nos da así una idea de *dinámica*, de movimiento, al *reducir la cantidad de elementos* con la que trabajamos; nos quedamos con menos fbfs, y posible pero no necesariamente, con menos variables proposicionales: pasamos de trabajar con dos fbfs (las premisas) a trabajar con una (la conclusión). Trabajar con menos elementos siempre es algo importante para nosotros como informáticos.

Observación desde el punto de vista semántico

MP es un *detector de contradicciones*. Para comprobar esto, reescribamos el MP usando la equivalencia lógica que establece que un condicional es lógicamente equivalente a la disyunción del consecuente con la negación del antecedente (ya que: $A \rightarrow B \equiv \neg A \vee B$). Tenemos así las siguientes fbfs escritas en FNC:

$$\neg A \vee B$$

$$\underline{\quad A}$$

$$B$$

Notemos que en la primera fbf aparece el literal $\neg A$ y en la segunda fbf aparece el literal A. MP "elimina" dichos *opuestos*, hace que ambas fbfs $\neg A$ y A "desaparezcan" de la conclusión.

Con esta nueva perspectiva del MP como "detector de contradicciones", retomemos la definición del Método 2.

(Método 2, cont.) Resolución

Usamos *Resolución*, una regla de inferencia de la que MP es un caso particular. Intuitivamente, podemos describir Resolución como sigue: dadas dos fbfs C y D, identificamos un literal p_i en una de ellas cuyo literal opuesto, llamemoslo q_j, aparece en la otra; seguidamente generamos una nueva cláusula R como la unión entre C y D -unión de la que eliminamos los literales p_i y q_j- y que llamamos resolvente entre C y D.

Definición formal de la regla de Resolución

C: p_1 v ……. v p_i v …… v p_n

D: q_1 v ……. v q_j v …… v q_m

R: p_1 v… v p_{i-1} v p_{i+1} v ….v p_n v q_1 v …. v q_{j-1} v q_{j+1} v ….v q_m ,

con p_i literal opuesto al literal q_j, y con R resolvente entre C y D.

Notemos que R es una *nueva* cláusula, que una vez generada pasa a integrar el conjunto de cláusulas con el que trabajamos. Esto sucede del mismo modo que nos pasa a los humanos: las conclusiones que obtenemos pasan a integrar nuestro conocimiento, y, por supuesto, las podemos usar para seguir sacando conclusiones.

Ejemplo

Aplicar Resolución entre las cláusulas C: ¬a v b, y D: a v ¬c nos da la resolvente R: b v ¬c.

Seguidamente le explicamos a nuestro colega el marciano cómo aplicar Resolución para probar insatisfactibilidad de una fbf escrita en FNC. Esto redondeará la idea detrás del Método 2.

(Método 2, cont.) Testeo de insatisfactibilidad. Algoritmo

Paso 1. **Reescritura.** Le pondremos una "máscara sintáctica" a las fbfs en FNC que recibimos como input. Esto lo hacemos simplemente por una cuestión de uso y costumbre entre los programadores lógicos: por una convención de escritura, la FNC se vuelve fácilmente legible y reconocible. Procedemos como sigue: tomamos cada disyunción de literales (cada paréntesis) de la FNC. A cada una de estas disyunciones la escribimos entre llaves (los símbolos "{" y "}"). Seguidamente, separamos los literales dentro del paréntesis con comas (","). Luego reemplazamos las conjunciones (que unen a las disyunciones entre sí en la FNC) también con comas (","). Por ejemplo, la FNC: (¬a v b) ^ (a v ¬b) queda reescrita como: {¬a, b}, {a,¬b}. Es decir, las comas que aparecen por fuera de las llaves se leen como conjunciones, mientras que las comas que aparecen dentro de las llaves se leen como disyunciones. A cada disyunción entre llaves la llamamos, a partir de ahora, cláusula.

Al conjunto de cláusulas lo denotaremos usualmente de aquí en más, Γ. La notación que vimos en este paso es propia de la Programación Lógica[24].

Paso 2. **Eliminación de contradicciones.** Aplicamos la regla de Resolución sobre Γ de la siguiente manera: tomamos dos cláusulas cualesquiera en Γ, identificamos un literal en una de ellas, identificamos el correspondiente literal opuesto en la otra cláusula, y armamos la unión de ambas cláusulas eliminando esos literales opuestos[25]. Generamos así la resolvente entre ambas. Repetiremos las veces que sea necesario este paso 2.

Paso 3. Detección de insatisfactibilidad. Puede suceder que estemos aplicando el Paso 2 y las dos cláusulas que tomemos tengan, cada una, un único literal. A la resolvente entre ambas la llamamos *cláusula vacía* y la simbolizamos "□". Ejemplo: la resolvente entre las dos cláusulas {p} y {¬p} nos da □:

{p} {¬p}

\ /

□

Nota

Dadas dos cláusulas C y D y dada una resolvente R entre ellas, R es una consecuencia lógica de C y D[26]. Por ello, cada asignación α que satisface a C y a D satisface también a R. La demostración de esta propiedad queda para el lector.

Estamos ya en condiciones de enunciar el Teorema de Robinson.[27]

Teorema de Robinson

Un conjunto $\Gamma = \{C_1, \ldots, C_n\}$ de cláusulas no es satisfactible (es insatisfactible) si y sólo si a partir de Γ puede obtenerse □ aplicando la regla de Resolución.

Es momento de darle forma al Método 2. Como somos informáticos, seguramente ya hemos notado que la tarea de detectar insatisfactibilidad de una FNC es perfectamente automatizable. Los programas que a partir de un conjunto de cláusulas tratan de determinar si dicho conjunto es insatisfactible se conocen como *sistemas refutacionales*[28].

Intuición sobre la implementación computacional del Método 2

Analicemos el siguiente pseudocódigo:

```
(1) Procedure Refutación (Γ)
(2)  clauses := Γ
(3)  until ((□ ∈ clauses) or existe punto de saturación(Γ)) do
(4)          elegir 2 cláusulas cualesquiera C1 y C2 ∈ clauses
(5)          R:= Resolución(C1, C2)
(6)          clauses := clauses ∪ {R}
(7)  end
(8) end Refutación
```

Comentarios

El programa recibe como input un conjunto finito de cláusulas, Γ, sobre el que aplicará Resolución. La estrategia del procedimiento es "*ciega"* en el sentido de que el algoritmo, tal como está planteado, elegirá pares de cláusulas C_1 y C_2 en Γ y tal elección se hará sin usar ningún criterio específico[29] (línea (4)). La condición de finalización del bucle `until` (línea (3)) es el hallazgo de $\square$, que hace innecesario continuar con el proceso ya que encontrar $\square$ nos asegura que Γ es insatisfactible[30]. También el algoritmo debe finalizar ante la imposibilidad de encontrar $\square$ habiendo agotado todas las aplicaciones útiles de Resolución (segunda parte de la condición del `until`). Esta situación se conoce como **punto de saturación**. El punto de saturación de un conjunto Γ es el conjunto al que se llega cuando no se pueden generar más nuevas resolventes entre las cláusulas de Γ.

Nota

Si alcanzamos el punto de saturación entonces sabemos que podemos armar la tabla de verdad de la FNC (que entró como input) y encontrar un renglón de dicha tabla que satisface a la FNC.

Ejemplo 1. Γ = {{¬a,b}, {¬a,¬b}, {a}} es insatisfactible. Generamos la resolvente: {¬a} entre las cláusulas primera y segunda al eliminar los literales b y ¬b mediante Resolución. Luego generamos la resolvente: $\square$ al aplicar Resolución entre la cláusula recién generada {¬a} y la tercera cláusula de Γ, {a}.

Ejemplo 2. Γ = {{a,c}, {¬a}, {b}, {¬a,¬b}} es satisfactible y su punto de saturación es: $\Gamma \cup$ {{c}, {c,¬b}, {¬a}}. Dejamos la comprobación al lector.

Nota

El punto de saturación puede definirse recursivamente por "niveles". El Nivel 0 es el conjunto Γ, el Nivel i lo constituye el conjunto de resolventes que se obtienen de aplicar Resolución entre todas las cláusulas del Nivel i-1 y todas las cláusulas de todos los niveles j $\leq$ i-1[31].

Para el ejemplo 2, el Nivel 0 es: {{a,c}, {¬a}, {b}, {¬a,¬b}}, el Nivel 1 es: {{c}, {c,¬b}, {¬a}}, el Nivel 2 es: {{c}}, cláusula que ya existía en el nivel previo, y a partir de ahora no se pueden generar más nuevas resolventes. Dejamos esta comprobación al lector.

Formalicemos ahora algunos aspectos de los métodos Método 1 y Método 2 en relación con la clase de los problemas NP.

Acerca de las clases de problemas NP y NP-completos

Una manera de clasificar problemas en el área de Informática es en función de la dificultad computacional que tienen los algoritmos diseñados para resolverlos. Por ejemplo, existe una clase de problemas que se llama NP (por las siglas de *Non-deterministic Polynomial*)[32]. NP constituye una clase de problemas de decisión resolubles en un número de pasos de computación que es función polinomial del tamaño del input. El modo de comprobar intuitivamente si un problema pertenece a la clase de los problemas NP es parafrasearlo como una pregunta cuya respuesta es "sí" o "no". Luego, si en tiempo polinomial puede encontrarse una respuesta por "sí", entonces el problema pertenece a la clase de los problemas NP (no tenemos que preocuparnos por las respuestas por "no" pues la máquina no determinística no nos penaliza por los caminos incorrectos que tomemos en la búsqueda de una posible solución).

La importancia de la clase de problemas NP es que contiene muchos ejemplos de problemas para los cuales se desearía tener algoritmos muy eficientes[33]. Por ejemplo: el problema del viajante, la búsqueda de ciclo hamiltoniano de un grafo, la búsqueda de un subgrafo clique de un grafo, etcétera.

NP-completitud

Existe un subconjunto de problemas NP que contiene los problemas más complejos, conocidos como los problemas NP-completos. Un problema, digamos *x*, está en el conjunto de los problemas NP-completos si cualquier otro problema que está en la clase NP puede reescribirse o traducirse, en tiempo polinomial, al problema *x*. Así, un problema NP-completo *x* puede usarse como una *subrutina* para cualquier problema (digamos *y*) en NP[34] del siguiente modo: se toma cualquier instancia del problema *y* y se define una transformación de aquella instancia a una instancia del problema *x*.

SAT

El problema denominado SAT se define como la cuestión de determinar si una fbf escrita en FNC[35] es o no satisfactible. El input de SAT es una FNC y su output es una asignación α de valores de verdad para las letras proposicionales que aparecen en la FNC tal que α satisface la FNC. En 1971 S. Cook[36] demostró que SAT pertenece a la clase de los problemas NP-completos.

SAT es un caso especial e importante de problema NP-completo, como veremos seguidamente.

Importancia de SAT

Entre todos los problemas que pertenecen a la clase NP-completo, SAT es, para nosotros los informáticos, el de mayor popularidad y trascendencia ya que nos resulta casi directo tomar cualquier instancia de un problema y *parafrasearla*, *reescribirla como, traducirla a* una instancia de SAT. Esto lo hacemos inmediata e intuitivamente cuando, en un primer momento, repensamos al problema dado como un problema cuya solución se vincula a un *output* "sí/no" de algún algoritmo. En un segundo momento, formalmente, construimos el sistema de restricciones del problema, por ejemplo, escribiendo una FNC que establece las restricciones o características de la instancia del problema (como hicimos con 3-colorabilidad). Y luego determinamos si dicha FNC es satisfactible o no. Así, la correspondencia entre el problema original y su reformulación como un problema SAT aparece patente al vincular la existencia de una asignación α de valores de verdad para las letras proposicionales en la FNC con la existencia de una configuración del mundo que constituye la solución efectiva para el problema dado.

Existen, en el Área de la Computación, otras maneras conocidas y útiles de resolver problemas traduciéndolos a otros problemas. Por ejemplo, el problema de colorabilidad es NP-completo y constituye *per se* una estrategia de solución de muchos otros problemas. Esto es, hay problemas que se modelan y se resuelven bien cuando los traducimos a un problema de colorabilidad. Las estrategias de traducción de problemas a otro tipo de problemas es naturalmente exitosa cuando conocemos bien el problema al cual hemos de traducir el problema original; normalmente los humanos intentamos resolver problemas nuevos o desconocidos tratando de verlos con la forma de otros problemas que ya sabemos que tienen soluciones conocidas o fáciles de encontrar.

Síntesis para nuestro grafo ejemplo

Hemos trabajado en este Capítulo 1 traduciendo una instancia del problema de colorabilidad a una instancia de SAT. Usamos al Cálculo de Enunciados como KRL para escribir una FNC que representa las restricciones de 3-colorabilidad del grafo. El grafo de la Figura 1 es 3-coloreable. Tenemos dos métodos para resolverlo. Desde el punto de vista semántico, aplicando el Método 1 pintamos el vértice 1 con el color 1, el vértice 2 con el color 3, el vértice 3 con el color 1, el vértice 4 con el color 2, el vértice 5 con el color 2 y el vértice 6 con el color 3. Es decir, existe una asignación α de valores de verdad para las dieciocho letras de proposición, asignación que satisface a la FNC. Dejamos esta verificación al lector.

Desde el punto de vista de la demostración automática de teoremas, podemos usar el Método 2: aplicamos Resolución y alcanzaremos el punto de saturación (pues la FNC es satisfactible). Dejamos esta verificación al lector.

Limitaciones del Cálculo de Enunciados como KRL

Desde el punto de vista de la representación del conocimiento, el Cálculo de Enunciados es fácil de usar. Tiene una semántica bien definida, tiene una estructura de datos simple -la variable proposicional- para describir estados de cosas sobre los que después haremos suposiciones. Para escenarios relativamente sencillos, el Cálculo de Enunciados funciona suficientemente bien: podemos construir una entidad inteligente proposicional -un *agente*- como un conjunto de fbfs en FNC cuyo mecanismo de pensamiento es Resolución. Sin embargo, desde el punto de vista de los KRL, el Cálculo de Enunciados muestra limitaciones a medida que los problemas se tornan más complejos. Al querer hacer descripciones más sofisticadas de las cosas y de las situaciones, posiblemente intentemos buscar un lenguaje más elaborado que nos permita describir más detalles del universo que queremos representar.

La Lógica de Predicados, o Lógica de Primer Orden, o Cálculo de Predicados, aparece como más expresiva que el Cálculo de Enunciados (ver Capítulo 3 de *Lógica para Matemáticos*, de A. G. Hamilton). La Lógica de Predicados nos brinda herramientas para realizar descripciones detalladas de los objetos, de las relaciones entre ellos, y además nos provee con una teoría de cuantificación: podemos referirnos a cualidades de algunos objetos, de todos los objetos, de ningún objeto.

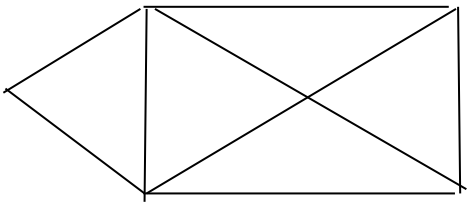
Otra posible variante que permite riqueza en la modelización es trabajar con extensiones de la Lógica de Enunciados como la Lógica Modal[37].

Habiendo comprendido la importancia de SAT y las limitaciones del Cálculo de Enunciados como KRL, abordamos a continuación, en el Capítulo 2 un KRL más rico en capacidad de modelización: La Lógica de Predicados, en un fragmento decidible.

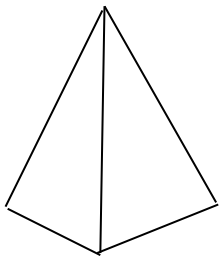
Ejercicios para el Capítulo 1

Modelizar utilizando el Cálculo de Enunciados como KRL los siguientes problemas, indicando qué porción del mundo representan las variables proposicionales y logrando una FNC que modelice las características y restricciones del problema.

1. **Scheduling.** Input: alguna secuencia de números $x_1,...x_n$, y un valor dado j. El problema consiste en tomar k números de la secuencia de modo tal que la suma de ellos sea j. Output: respuesta por sí o por no.
2. **Viajante.** Input: matriz de costos de dimensión NxN, en la que el contenido de cada celda x_{ij} es el costo de viajar desde la ciudad i hasta la ciudad j; más un valor n. Problema: si pueden visitarse todas las ciudades gastando un monto menor al n dado. Output: respuesta por sí o no.
3. **Ciclo Hamiltoniano**. Consiste en, dado un grafo G = (V, E), con card(V)=n, determinar si existen ciclos de n nodos que contengan a cada nodo una vez. Dado el grafo G = (V, E) a continuación, con card(V) = n = 5, o**btener** una fbf del Cálculo de Enunciados escrita en FNC que se corresponda con la cuestión de determinar la existencia de un ciclo hamiltoniano.



4. **Clique**. Una variante del problema clique se define como sigue: dado un grafo no dirigido G= (V, E) y un entero k, ¿G contiene un subgrafo completo de k vértices? (un grafo completo es aquel donde hay una arista entre cada par de vértices). Para el siguiente grafo:



y dado k=3, obtener una reducción del problema clique a una instancia de SAT.

Ejercicios extra

i) Escribir un programa en algún lenguaje de programación tal que reciba como input un grafo G de tres vértices {A, B, C} y retorne como output una fbf F_G en términos de variables proposicionales a_{xy} , x,y $\in$ {A, B, C}, con x distinto de y tal que F_G es satisfactible sii G es completo.

ii) Dado un grafo construir una fbf F_G tal que, dado cualquier átomo a_{mn}, la fbf ($F_G \wedge a_{mn}$) es satisfactible si y sólo si existe una arista entre los vértices m y n en G. Recordar que $a_{mn} = a_{nm}$ por tratarse de grafos no dirigidos.

---

[1] O Cálculo de Enunciados, o Lógica Proposicional, o Cálculo Proposicional, según diferentes textos de la materia. A lo largo del libro utilizamos esta terminología indistintamente.

[2] Ver ejemplo 1.21 (Ejercicios para la Sección 1.2) del libro *Modal Logic*. P. Blackburn, M. de Rijke, Y. Venema. Cambridge, 2001.

[3] Ver Definición 2.1 de libro *Lógica para Matemáticos*, de A. G. Hamilton. Ed Paraninfo, Madrid, 1981.

[4] No obstante, más adelante veremos que hay problemas para los cuales la Lógica de Enunciados resulta insuficiente como KRL.

[5] Asumimos que un grafo G = (V,E) es una dupla formada por un conjunto finito V de vértices, también llamados puntos, o estados; y un conjunto finito E de conexiones entre dichos vértices. Estas conexiones pueden representarse citando los pares de vértices (v,w) que cada conexión une.

[6] El marciano en este contexto es un *oráculo*, en el sentido que se le da a esta expresión en Ciencias de la Computación, esto es: una "caja negra" capaz de resolver un problema cualquiera (o responder a una pregunta) en un solo paso.

[7] Decimos "una porción" del lenguaje pues debemos tener claro que no necesitamos de todos los símbolos de la Lógica de Enunciados para representar el problema. Unas pocas variables y tres conectivos serán suficientes.

[8] Son dieciocho variables porque tenemos seis vértices y tres posibles colores para cada vértice.

[9] Ver Corolario 1.21 de *Lógica para Matemáticos*, de A. G. Hamilton. Las formas normales sirven para organizar y unificar la escritura, lo que hace más fácil entenderse entre matemáticos, lógicos e informáticos ya que las fbfs escritas en formas normales respetan una forma bien conocida y estandarizada. Una fbf está escrita en FNC si es de la forma: $\bigwedge(\bigvee Q_{ij})$ con cada Q_{ij} una variable proposicional o la negación de una variable proposicional. Dicho más coloquialmente: una fbf está escrita en FNC cuando está escrita como una *conjunción de disyunciones*.

[10] Ver Sección 1.4 de *Lógica para Matemáticos*, de A. G. Hamilton.

[11] Ver Proposición 1.24 de *Lógica para Matemáticos*, de A. G. Hamilton. Recordemos que una de las leyes de De Morgan establece que: $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$, para fbfs A y B cualesquiera.

[12] Ver Sección 1.2 de *Lógica para Matemáticos*, de A. G. Hamilton.

[13] Pensemos en un mapa planisferio con los países coloreados. Los países limítrofes tienen colores diferentes entre sí de modo de poder distinguirlos visualmente de manera rápida.

[14] Sabemos construirla usando nuestros conocimientos básicos de lógica (ver Sección 1.2 de *Lógica para Matemáticos*, de A. G. Hamilton).

[15] Puede decirse que interpretar un renglón de la tabla de verdad como una "posible configuración del mundo" es interpretarlo desde una perspectiva *orientada a modelos*.

[16] Intentar resolver problemas transformándolos en otros que ya conocemos (y que tal vez ya sepamos resolver) es una estrategia usual y práctica de resolución de problemas que aplicamos los humanos. Más ejemplos y ejercicios de este estilo aparecen al final de este capítulo.

[17] Así, una asignación α "es" un renglón de la tabla de verdad, por abuso del lenguaje.

[18] Ver Definición 1.5 y Ejemplo 1.6 del libro de Hamilton. Allí se definen y ejemplifican fbfs que algunos lógicos denominan *contingencias*. Son fbfs que no son ni tautologías ni contradicciones. Ver también Definición 3.20 de *Lógica para Matemáticos*, de A. G. Hamilton; allí se introduce la noción de *satisfactibilidad* de una fórmula. Ver asimismo Definición 1.20 (satisfactibilidad) del libro *Modal Logic*, P. Blackburn, M. De Rijke, Y. Venema.

[19] L_i es un literal, es decir, una letra de proposición o la negación de una letra de proposición ($L_i = x_{ij}$ ó $L_i = \neg x_{ij}$.) Ver apartado 2, "First Order Theories" del Capítulo 1 del libro *Foundations of Logic Programming*, de W. L. Lloyd.

[20] Ver el concepto de valuación en la Definición 2.12 de *Lógica para Matemáticos*, de A. G. Hamilton.

[21] Ver Definición 2.1 de *Lógica para Matemáticos*, de A. G. Hamilton.

[22] Una rule of detachment o *regla de desprendimiento* es, en líneas generales, una ley o principio que permite *desprender* una conclusión de un conjunto de premisas. MP como rule of detachment permite aceptar una conclusión

como válida bajo la suposiciòn de que las premisas lo son, usando un mecanismo que garantiza dicha validez. Decimos que la conclusión *se desprende* de las premisas porque inicialmente aparece inserta en una de las premisas, y luego de aplicar MP pasa de ser una subfórmula a ser una fbf con la que se puede trabajar como una fbf más.

[23] MP *preserva verdad* pues de premisas verdaderas nos permite sacar conclusiones verdaderas. Ver Proposición 1.9 de *Lógica para Matemáticos*, de A. G. Hamilton.

[24] Ver *Foundations of Logic Programming*, de J. W. Lloyd.

[25] Tomamos un solo par de literales opuestos en cada aplicación de Resolución.

[26] Pues R intuitivamente "está en sus cláusulas padres", ver Definición 2.5 de *Lógica para Matemáticos*, de A. G. Hamilton. Aunque en esa definición el sistema formal usa MP, podemos extender naturalmente la validez de la proposición para un sistema que usa Resolución.

[27] Ver *A review of automated theorem proving*. J. A. Robinson, Proc. Symp. Appl. Math, Amer. Math. Society, Providence, R.I. 19:1-18, 1967; y *The Generalized Resolution Principle*. J. A. Robinson, Machine Intelligence, 3:77-93, 1968.

[28] Ver Capítulo 5 de *Principles of Artificial Intelligence*. Nils J. Nilsson, Springer-Verlag, 1982.

[29] Podemos suponer, para el caso, que las cláusulas son elegidas al azar.

[30] Nos lo asegura el Teorema de Robinson.

[31] La manera de tomar las cláusulas para aplicar Resolución no tiene importancia, al menos por ahora, para nosotros. Si lo pensamos desde el punto de vista lógico, *no hay determinismo*, no existe un orden que debamos seguir para elegirlas. Luego, cuando llevemos a la regla de Resolución a ser el motor de razonamiento de un lenguaje de programación, sí deberemos tomar decisiones respecto al orden de selección de las cláusulas. Y también respecto del orden de selección de los átomos dentro de una cláusula. Ver Sección II.2: "The Standard Control Strategy", Capítulo 2: "Logic Programs" de *Introduction to Logic Programming*, de J. C. Hogger. AP, US Edition, 1984.

[32] Una máquina de computación no determinística es una construcción teórica conocida como Máquina de Turing. En ella el *backtracking* de un algoritmo está permitido sin costo de computación: si el algoritmo que está corriendo llega a una instrucción en el que varios caminos son posibles de ser elegidos, podemos tomar cualquiera de ellos, y si tomamos un camino fallido (que no nos condujo a ninguna solución) el algoritmo puede volver al punto de la elección e intentar otro camino sin que se contabilice el tiempo de computación perdido. Ver Sección 9.7.2 de *Data Structures and Algorithm Analysis*, M. A. Weiss. Benjamin Cummings, CA, 1992.

[33] La clase NP también incluye a todos los problemas que tienen soluciones que pueden computarse en tiempo polinomial. Podríamos pensar que hay problemas que están en la clase NP pero que no tienen solución en tiempo polinomial. Hasta la fecha no se ha encontrado un problema tal. Ver Sección 9.7.2 "The Class NP" de *Data Structures and Algorithm Analysis*, M. A. Weiss. Benjamin Cummings, CA, 1992.

[34] Ver Sección 9.7.3 "NP-Complete Problems" de *Data Structures and Algorithm Analysis*, M. A. Weiss. Benjamin Cummings, CA, 1992.

[35] Nos hemos estado refiriendo a "una fbf escrita en FNC" como "una FNC" para abreviar.

[36] Ver *The Complexity of Theorem Proving Procedures*. S. Cook. Proceedings of the Third Annual ACM Symposium on Theory of Computing (1971), 151-158.

[37] Ver *Modal Logic*, de P. Blackburn, M. De Rijke, Y. Venema. Cambridge Press.

CAPÍTULO 2
Determinación de satisfactibilidad en la lógica de primer orden

Clara Smith

Desde el punto de vista de la representación del conocimiento, la Lógica de Primer Orden (abreviamos LO1 por *lógica de orden uno*) es un lenguaje muy expresivo. Con la LO1 podemos describir objetos de un *universo* o *dominio* de objetos[1], y funciones y relaciones entre ellos de un modo más directo y detallado a como ya lo sabemos hacer con el Cálculo de Enunciados.

Asumimos que el lector tiene algún conocimiento básico de la LO1. Lo que sigue será especialmente familiar si se ha dado una lectura al Capítulo 3 y al Capítulo 4 de *Lógica para Matemáticos*, de A. G. Hamilton (Ed. Paraninfo), y/o al Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd (Ed. Springer).

Lógica de Primer Orden

Comencemos con la parte sintáctica de la LO1.

Sintaxis

Los símbolos que se usan en la LO1 son[2]:

· Un conjunto X de variables: x, y, z, …

· Un conjunto C de constantes: a, b, c, ...

· Un conjunto F de símbolos de función: f_i, g_j, h_k, ...

· Un conjunto P de símbolos de predicado: p_n, q_m, r_n, …

[1] Usaremos los términos *universo* o *dominio* (de objetos) indistintamente.

[2] Ver Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd. 2nd Edition, Springer, 1993.

· Conectivos: ¬, ^, ∨, →.

· Cuantificadores: ∀, ∃.

Los conjuntos C, F y P de constantes, de símbolos de función y de símbolos de predicado respectivamente, pueden eventualmente ser vacíos. El conjunto X de variables no puede ser vacío; debemos tener al menos una variable para poder referirnos a los objetos del dominio, esto es, tenemos que tener al menos una *herramienta* (al menos una variable) con la que denotar o "capturar" a los objetos. Con una variable sería, en principio, suficiente. Los conjuntos X, C, V, y F podrían ser infinitos. Los símbolos de función y los símbolos de predicado tienen diferentes aridades[3] que por el momento serán expresadas con un subíndice para cada letra. Los símbolos ∀ y ∃ son los símbolos de la cuantificación universal y existencial respectivamente, y nos permiten referirnos ya sea a *todos los objetos* del universo de objetos el primero, ya sea a *algunos objetos* del universo de objetos, el segundo.

Lenguajes de Primer orden

Un lenguaje de primer orden queda identificado por el *subconjunto de símbolos* que usa. *Hay, así, infinitos lenguajes de primer orden*. Este aspecto es relevante para nosotros como informáticos, pues siempre intentamos manejar pocos símbolos cuando programamos: a veces preferimos la menor cantidad posible de variables, a veces preferimos manejar la menor cantidad posible de estructuras de datos, etcétera.

Típicamente, para referirnos a algunos dominios o universos de trabajo usamos algunos lenguajes, y para referirnos a otros universos, otros lenguajes. Es por esto que, posible pero no necesariamente, los lenguajes que definamos o usemos difieran entre sí en sus símbolos. De todos modos, los símbolos pueden repetirse de un lenguaje a otro y también puede suceder que las lecturas intuitivas de los mismos símbolos sean iguales, o sean diferentes. Tenemos múltiples posibilidades.

Ejemplo de lenguaje de O1

Un lenguaje de primer orden (abreviamos también: LO1)[4] sencillo para trabajar con la aritmética de los números naturales, llamémoslo L, contiene:

· una constante: a,

[3] Esto es, pueden recibir más de un objeto como parámetro. En algunos textos de la materia se usa la palabra "cardinalidad" como sinónimo de aridad. Usaremos, comúnmente, aridad.

[4] Por abuso de notación usamos "LO1" para "lógica de primer orden" y también para "lenguaje de primer orden", cuando el contexto lo permite y no hay confusión.

· tres símbolos de función: suc (unario), "+" (binario), y "×" (binario)[5], y

· un símbolo de predicado (binario): "=".

Usando la notación sintáctica del comienzo de este Capítulo[6], tenemos:

L = X= {x, y, z,...}, C= {a}, F= {suc_1, $+_2$, $\times_2$}, P= {$=_2$}.

La lectura intuitiva de los símbolos de este lenguaje es como sigue: la constante "a" denota al número 1; suc se interpreta como la función sucesor (recibe como parámetro un número y retorna el siguiente número), "+" se interpreta como la suma (recibe dos números y retorna el número que es la suma de aquellos dos), y "×" se interpreta como la multiplicación (recibe dos números y retorna el producto de ambos). El predicado "=" retorna el valor de verdad True si dados dos números ambos son el mismo, y retorna False en caso contrario.

Ejemplo

La fbf =(×(z,y), +(z,y)) intuitivamente se lee como: "la multiplicación de dos números z e y es igual a la suma de esos dos números z e y".

Definición. Términos

Los términos de un LO1 son *todas las expresiones sintácticas* que podemos escribir para *nombrar* a los objetos del universo con el que trabajamos[7]. Los términos de un LO1 L cualquiera se generan recursivamente del siguiente modo:

· una constante de L es un término de L,

· una variable de L es un término de L,

· un símbolo de función f_n de L aplicado a términos t_1, .., t_n de L, $f_n(t_1, .., t_n)$, es un término de L (siendo n un número natural que representa la aridad de la función).

Ejemplos

x, a, f(x), f(a), g(x,y), g(a,b), h(f(x),a) son todos términos escritos en algún LO1 que tiene entre sus símbolos la constante a, el símbolo de función (unario) f, y los símbolos de función (binarios)

[5] Ver Ejemplo 3.5 (a) de *Lógica para Matemáticos*, de A. G. Hamilton.

[6] Cuando la anotación de la aridad de un símbolo complica la lectura y además es de fácil deducción de la lectura de la expresión, evitamos escribir el subíndice.

[7] Ver Definición 3.6 de *Lógica para Matemáticos*, de A. G. Hamilton, y ver también la Definición de término en el Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

g y h. Los términos son, dicho informalmente, los *nombres* con los que denotamos a los objetos del universo, son las expresiones con las que nos referimos a los objetos[8].

Definición. Fórmulas bien formadas

Podemos escribir fórmulas simples aplicando los símbolos de predicado de un LO1 a términos escritos en dicho lenguaje. Combinando estas fórmulas simples con los conectivos lógicos que ya conocemos (y cuyos comportamientos se mantienen igual a como ya los estudiamos), sumado a la aplicación de los cuantificadores, podemos escribir fórmulas bien formadas (fbfs) en el LO1, de la siguiente manera:

Sea L un lenguaje de O1:

· Un símbolo de predicado de L aplicado a un término de L (o a más términos, según la aridad del predicado) es una fbf, usualmente llamada *fórmula atómica* o *átomo*[9] de L.

· Si A y B son fbfs de L, entonces también lo son: ¬A , A ^ B, A v B, A → B, y (∀x)A , (∃x)A, para alguna variable x en A.

Ejemplos

Si p es el símbolo de predicado (de un lenguaje L) que se interpreta como la cualidad de *ser número par*, y a es la constante de L que denota el número 1, entonces el átomo p(a) de L simplemente simboliza: “el número 1 es par”.

A la fbf: (∀x)p(x) la leemos como: “para todo objeto del dominio, ese objeto cumple la cualidad p”.

La fbf: (∃x)(p(x) v q(x)) informalmente se lee: “existe un objeto del dominio tal que sucede que ese objeto o bien cumple la propiedad p, o bien cumple la propiedad q”.

Comentarios

Notemos que en este Capítulo la sintaxis que usamos introdujo algunos cambios respecto de cómo escribíamos a las fbfs en el Cálculo de Enunciados. Lo más evidente que notamos es que en un LO1 aparecen más símbolos que en el lenguaje del Cálculo de Enunciados. El cambio fundamental, no obstante, como veremos más adelante, se da en el plano de la semántica: no nos servirán más las tablas de verdad porque ahora, en esta nueva lógica, no solo tenemos variables sino que tenemos, además, términos que denotan objetos, y también podemos aplicar funciones a objetos para conseguiremos o nombrar a otros objetos. Asimismo, podemos predicar

[8] Al estudiar el curso inicial de Programación, el término usado típicamente es el de *puntero a un objeto*.

[9] Los átomos constituyen la *prédica mínima* que podemos hacer sobre los objetos de un universo. Expresan una cualidad simple de un objeto.

sobre los objetos, y podemos cuantificarlos. Entonces, como ahora podemos escribir fbfs más detalladas y complicadas que las que escribíamos antes usando el Cálculo de Enunciados, necesitamos, correspondientemente, tener una estructura más detallada sobre la cual verificar la validez de esas fbfs[10]. A dicha estructura la llamamos *interpretación*; volveremos sobre ella más adelante.

SAT y decidibilidad en la LO1

También hay cambios en el planteo del problema SAT. Ocurre que la LO1, caracterizada como el conjunto de sus fbfs lógicamente válidas[11], no es *decidible*; esto es, no existe un algoritmo que reciba como input una fbf A *cualquiera* escrita en un lenguaje de primer orden y pueda determinar si A es verdadera o no. En otras palabras: no existe un procedimiento mecánico que permita decidir si existe o no una prueba sintáctica para demostrar una fbf A de la LO1, sea cual sea A[12].

Fragmento especial de la LO1

Sin embargo, existe -afortunadamente para nosotros como informáticos- una clase de fbfs que tienen ciertas características particulares. Son fbfs que tienen una forma *especial*, forma que se consigue a través de transformaciones lógicas garantizadas. Usaremos esta clase de fbfs especiales para representar los problemas y poder sacar conclusiones, aplicando Resolución sobre ellas, de un modo seguro. Esto gracias a que existen dos teoremas que nos garantizan que: si una fbf es consecuencia lógica de un conjunto de estas fbfs especiales entonces podremos demostrar aquella usando Resolución, y que: el último eslabón de una demostración hecha con la regla de Resolución se corresponde con una consecuencia lógica del conjunto de fbfs sobre el que se aplica aquella regla de inferencia. Veremos las enunciaciones de dichos teoremas más adelante.

---

10 Ver Definición 3.14 de *Lógica para Matemáticos*, A. G. Hamilton.

11 Hay distintas maneras de caracterizar (identificar, describir) a una lógica. Una manera es caracterizarla como el conjunto de los teoremas demostrables en dicha lógica.

12 La demostración de esta proposición requiere de conceptos de computación teórica. Asumimos que el lector conoce la definición de Máquina de Turing. Este tipo de máquina tiene una cinta de longitud infinita que puede moverse hacia la izquierda o hacia la derecha. La máquina puede escribir un símbolo cualquiera de entre los símbolos del conjunto finito $s_1 ... s_k$ en cada porción de cinta. La máquina puede estar, en cualquier momento, en cualquier estado del conjunto finito de estados $Q_1 \dots Q_n$.
Es ampliamente conocido que una máquina de este tipo es lo suficientemente potente como para computar cualquier cálculo que sepamos cómo computar. Asumimos que el lector, además, sabe que *no es posible construir una Máquina de Turing que pueda decirnos si una Máquina de Turing cualquiera se detendrá*, comenzando desde algún estado arbitrario. El centro de la demostración de la indecidibilidad del Cálculo de Predicados radica en que podemos describir Máquinas de Turing y los estados de estas máquinas usando el Cálculo de Predicados. En particular, dada una máquina específica, podemos escribir un conjunto de fbfs que será válido si y sólo si la Máquina de Turing se detiene. Entonces, de esto podemos deducir que, si tuviéramos un procedimiento mecánico que pudiera decirnos si ese conjunto de fbfs es o no válido, entonces tendríamos un procedimiento para decidir si una Máquina de Turing arbitraria se detiene. Ver *Prueba de Indecidibilidad del Cálculo de Primer Orden*, Sección 3.15, Capítulo 3 (Cálculo de Predicados) de *Formal Methods in Artificial Intelligence* (Cambridge Tracts in Theoretical Computer Science), de A. Ramsay. Cambridge University Press, 1991.

Para conseguir las fbfs especiales tenemos que procesar a las fbfs "originales" que describen el conocimiento relacionado con el problema que pretendemos resolver. Para esto usamos un algoritmo de "transformación" de fbfs, muy específico, llamado *Algoritmo de Representación Clausal*. Este algoritmo realiza ciertas operaciones sintácticas sobre las fbfs "originales" que recibimos como input.

Es importante anticipar que este algoritmo no convierte a las fbfs que ingresan en otras fbfs lógicamente equivalentes a aquellas: **las fbfs resultantes pueden no ser lógicamente equivalentes a las que ingresan**.

A continuación estudiamos el Algoritmo de Representación Clausal.

Representación Clausal

La representación clausal es una forma de escribir fbfs ampliamente aceptada entre los programadores lógicos, los lógicos, los informáticos e ingenieros de sistemas. Los programas lógicos se escriben con fbfs en representación clausal[13].

Algoritmo de Representación Clausal

Input: un conjunto (finito) de fbfs escritas en algún LO1.

Output: un conjunto de fbfs, llamémoslo Γ, que constituye la *representación clausal* de las fbfs que ingresaron como input. Γ *se corresponde* con el conjunto de fbfs que ingresó como input, aunque ambos conjuntos *podrían no ser lógicamente equivalentes*. El conjunto resultante Γ cumple determinadas características que hacen posible que sobre él podamos aplicar la regla de Resolución sin inconvenientes.

A continuación presentamos el algoritmo.

Para cada fbf del conjunto de fbfs que ingresa como input, aplicamos la siguiente secuencia de pasos[14]:

Paso 1. Clausura de variables libres. Si la fbf que ingresa como input tiene una variable libre[15] (por ejemplo, ingresa la fbf: p(x)) entonces a esa variable libre debemos *ligarla* a un

[13] En, por ejemplo, el lenguaje Prolog. Ver Capítulo 1, Sección 1, *Preliminaries*, del libro *Foundations of Logic Programming*, de W. L. Lloyd.

[14] En diferentes textos referidos a Programación Lógica algunos de los pasos del Algoritmo de Representación Clausal pueden aparecer permutados.

[15] En una fbf se dice que una variable está libre cuando no aparece dentro del alcance (o radio de acción) de un cuantificador en la fbf. Si una variable no está libre se dice que está ligada. Ver Definición 3.8 de *Lógica para Matemáticos*, de A. G. Hamilton.

cuantificador[16]. Sabemos que en la LO1 a una variable libre se la entiende *como si* estuviera clausurada universalmente[17]. Pero desde el punto de vista de la Ingeniería de Software, ¿es *prudente* esa interpretación? Parece *osado* interpretar como clausurada universalmente a una variable libre porque ello implica asumir que la cualidad predicada se cumple *para todos* los objetos del dominio. Y en este punto de análisis no nos consta que esa haya sido la percepción del analista o del diseñador que capturó la cualidad en cuestión. Como desconocemos dicha percepción, ligamos la variable libre con un cuantificador existencial (símbolo "∃"). Esta imposición es algo más *distendida* porque es menos restrictivo ligar una variable libre con un cuantificador existencial. Ligar una variable con un cuantificador universal significa imponer que una cualidad se satisfaga *para todos* los objetos, esta es una exigencia más alta que pedir que la cualidad se cumpla *para al menos un* objeto.

Ejemplo: si recibimos la fbf abierta: p(x), la clausuramos existencialmente: (∃x)p(x).

Las fbfs con variables libres no son inusuales. Puede suceder que la fbf que ingresa tenga una variable libre por descuido de un ingeniero de software que omitió transcribir un cuantificador.

Paso 2. Renombre de variables. Verificamos que las variables de la fbf no aparezcan más de una vez y cuantificadas con distinto alcance[18] dentro de la misma fbf. Las renombramos para evitar posibles confusiones posteriores.

Ejemplo: ingresa la fbf: (∀x)p(x) → (∀x)q(x), la reescribimos como: (∀x)p(x) → (∀y)q(y), renombrando la x del consecuente por y.

Paso 3. Reescritura de los condicionales. Aplicamos las equivalencias lógicas[19] -que ya conocemos- sobre la fbf que ingresa a este Paso 3 para conseguirnos una fbf lógicamente equivalente que solo use negación, conjunción y disyunción. Recordemos que: A → B ≡ (¬A) v B, que: (¬A) → B ≡ A v B, y que: ¬(A → (¬B)) ≡ A ^ B, con A y B fbfs escritas en algún LO1.

Paso 4. Ajuste de las negaciones a *los átomos*. Los símbolos de negación deben estar aplicados sobre los átomos de la fbf. Solo deben afectar a éstos y no a subfórmulas con estructura más complicada que un átomo. Este ajuste forma parte de una búsqueda de estandarización en la escritura (lo hemos aprendido cuando aprendimos a escribir fbfs en FNC en el Cálculo de Enunciados). Para realizar dicho ajuste usamos las equivalencias de De

[16] Esto porque el Algoritmo de Representación Clausal produce como output cláusulas cuyas variables (todas) están ligadas. Dichas cláusulas eventualmente integrarán un programa lógico.

[17] Ver Definición 3.27 y Corolario 3.28 de *Lógica para Matemáticos*, de A. G. Hamilton.

[18] En la fbf (∀x)A decimos que A es el radio de acción o alcance del cuantificador. Ver Definición 3.8 de *Lógica para Matemáticos*, de A. G. Hamilton.

[19] Ver Proposición 1.24 de *Lógica para Matemáticos*, de A. G. Hamilton. Si bien allí las equivalencias lógicas están explicadas para fbfs del Cálculo de Enunciados, dichas equivalencias valen también en la LO1 porque el comportamiento de los conectivos permanece inalterable.

Morgan[20] de modo de mover las negaciones hacia adentro de los paréntesis (*distribución de la negación sobre la conjunción y sobre la disyunción*) y "pegar" las negaciones a los átomos. También podemos usar la equivalencia llamada Doble Negación ($\neg\neg A \equiv A$). Debemos asimismo prestar atención al ajuste de las negaciones respecto de los cuantificadores; para ello usamos las equivalencias entre cuantificadores, que definen, a su vez, la dualidad entre ambos: $(\forall x)A(x) \equiv \neg(\exists x)\neg A(x)$, y, correspondientemente, $(\exists x)A(x) \equiv \neg(\forall x)\neg A(x)$[21].

Luego de aplicar estos cuatro primeros pasos del algoritmo, tenemos (temporariamente) a la fbf que ingresó como input cumpliendo que: no contiene variables libres, tiene sus variables renombradas convenientemente, está escrita con el conjunto adecuado de conectivos {^, v, ¬}, y con las negaciones ajustadas a los átomos. Notemos que paso a paso transitamos hacia una escritura en FNC. Recordemos que nuestro objetivo es lograr una notación estandarizada y ya ampliamente reconocida.

Debido a la posible alternancia entre cuantificadores universales y existenciales, nuestra fbf en tratamiento podría ser difícil de interpretar intuitivamente[22]. Si sucede que hay muchos cuantificadores alternados puede ser complicado identificar dentro de la fbf el alcance de cada uno de ellos. Es por este motivo que -una vez más- conviene reescribir la fbf no solo para visualizarla de un modo prolijo sino para leerla y comprenderla mejor.

En el próximo paso (Paso 5) transformaremos la fbf que acabamos de procesar en el Paso 4 en otra fbf lógicamente equivalente que tiene todos los cuantificadores *al inicio, al principio* (*a la izquierda*). Esta reorganización de los cuantificadores permite una buena lectura y manipulación de la fbf.

Paso 5. Obtención de la FNP. Definición. Una fbf $A(x_1, x_2, ..., x_k)$ está en *forma normal prenexa*[23] (abreviamos FNP) si tiene la forma: $(Q_1(x_1))(Q_2(x_2))$ ….$(Q_k(x_k))$ $B(x_1, x_2, ..., x_k)$ con Q_1, Q_2,…., Q_k cuantificadores para las variables $x_1, x_2, ..., x_k$ en $B(x_1, x_2, ..., x_k)$ respectivamente, siendo $B(x_1, x_2, ..., x_k)$ una subfórmula abierta (sin cuantificadores).

Ejemplo. La fbf: $((\forall x)p(x)) \rightarrow ((\forall y)(\exists z)q(y,z))$ se escribe en FNP como: $(\forall x)(\forall y)(\exists z)(p(x) \rightarrow q(y,z))$.

La obtención de una FNP de una fbf se logra con la aplicación oportuna de equivalencias lógicas ya conocidas entre cuantificadores y aplicando también las equivalencias lógicas que

[20] Ver Proposición 1.11 de *Lógica para Matemáticos*, de A. G. Hamilton; dicha Proposición presenta las equivalencias de De Morgan: $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ y $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$. Tal como están enunciadas son aplicables a fbfs de un LO1 reemplazando convenientemente las metavariables por subfórmulas del lenguaje.

[21] La notación $A(x)$ debe entenderse como referencia a una fbf de la LO1 en la que aparece la variable x. Para las equivalencias entre cuantificador universal y existencial consultar los Ejemplos 3.1, 3.2 y 3.3 de *Lógica para Matemáticos*, de A. G. Hamilton.

[22] Por ejemplo, tanto la lectura en lenguaje natural como la comprensión desde el punto de vista técnico de la fbf: $(\exists x)(\forall x)(\exists z)(p(x,y) \vee r(y,z)) \rightarrow ((\forall t)(\exists w)q(t,w))$ no es inmediata.

[23] Ver Definicion 4.27 de de *Lógica para Matemáticos*, de A. G. Hamilton. La FNP proporciona una manera de medir cuán complicada es una fbf.

definen la dualidad de los operadores[24]. En este punto es importante saber que para toda fbf A existe una fbf B que está en FNP y es demostrablemente equivalente a A[25]. En resumen, para cualquier fbf de la LO1 podemos conseguir otra fbf lógicamente equivalente *con todos sus cuantificadores adelante*.

Tengamos presente que, a pesar de que conseguir una FNP puede involucrar manipulaciones sintácticas a simple vista no triviales, ganamos en legibilidad, que es la idea detrás de las *formas normales*.

Continuando con el Algoritmo de Representación Causal, en el siguiente Paso 6 procedemos a "eliminar" los cuantificadores existenciales que pudiera contener la FNP. Luego del Paso 6 la fbf solo contendrá cuantificadores universales. Desde el punto de vista lógico trabajar con solo el cuantificador universal es posible pues conocemos su dualidad con el cuantificador existencial (conocemos las equivalencias entre ellos). El cuantificador universal constituye, él solo, un "conjunto de cuantificadores adecuado" para trabajar. Además, mejorará nuestra lectura intuitiva de la fbf ya que los predicados en la fbf, por estar todos cuantificados universalmente, se "aplican a todos los objetos" y no tenemos que prestar atención a posibles apariciones de cuantificadores existenciales intercalados[26] en la fbf.

Paso 6. Eliminación de existenciales. Skolemnización. El procedimiento de "eliminación de existenciales" se realiza a través de la técnica llamada *skolemnización*[27]. *Skolemnizar* es reemplazar el símbolo "∃" y la variable que este símbolo esté ligando por un término llamado *función de Skolem*. Esta función de Skolem es un símbolo de función introducido a propósito y que no aparece originalmente en el LO1 (con el que se escriben las fbfs originales). La función de Skolem recibe como parámetro las variables que, en la fbf, están cuantificadas universalmente fuera del alcance del cuantificador existencial que se está eliminando, esto es, la función de Skolem recibe como parámetros todas las variables cuantificadas universalmente que aparecen *a la izquierda* del cuantificador existencial que se está skolemnizando.

Ejemplos. i) Sea la fbf en FNP: (∀x)(∃y)p(x,y). La variable y aparece cuantificada existencialmente dentro del rango de un cuantificador universal que liga a la variable x. Notemos que hay un cuantificador universal a la izquierda del cuantificador existencial en la fbf que se está skolemnizando. Entonces la skolemnización correspondiente es: (∀x)p(x,f(x)), con f nuevo símbolo de función y con f(x) *término de Skolem*.

ii) Sea la fbf en FNP: (∃y)(∀x)p(x,y). Skolemnizamos como: (∀x)p(x,a) con "a" término o *(función) constante de Skolem*. Notemos que no hay ningún cuantificador universal a la izquierda

[24] Como ya las recordamos en el Paso 4 de este Algoritmo.

[25] Ver Proposicion 4.28 de *Lógica para Matemáticos*, de A. G. Hamilton.

[26] En el Paso 5 los cuantificadores solamente se mueven hacia adelante de la fbf, pero la alternancia entre cuantificadores universales y existenciales pudo haber permanecido.

[27] Ver Capítulo 3 de la versión original inglesa, *Logics for Mathematicians*, A. G. Hamilton. CUP, 1988.

del cuantificador existencial en la fbf que se está skolemnizando, hay un mismo "objeto testigo" (a) para todos los x tal que verifican p(a,x).

iii) Sea la fbf en FNP: (∀x)(∃y)(∃z)(p(x) ^ q(y) ^ r(z,y)). Skolemnizamos: (∀x)(p(x) ^ q(f(x)) ^ r(g(x),f(x))) con g(x), f(x) términos de Skolem.

Paso 7. "Olvido" de los cuantificadores universales. En este paso *omitimos la escritura* de los cuantificadores universales a sabiendas de que efectivamente aparecen en la FNP skolemnizada. De aquí en más asumiremos, como ciertamente ocurre, que todas las variables que aparecen en la fbf están cuantificadas universalmente. Como programadores lógicos a partir de ahora no escribiremos esos cuantificadores por una simple cuestión de economía de escritura, pero sabemos que están.

Ejemplo. La fbf: (∀x)(∀y)(¬p(x,y) v r(f(x,y),a)) a partir de este Paso 7 se escribe: (¬p(x,y) v r(f(x,y),a)).

Comentario. Parecería existir alguna contradicción entre la clausura hecha en el Paso 1 y el "olvido" de los cuantificadores universales que hacemos en este paso 7. Pero no. La aparente contradicción surgiría de la introducción de cuantificadores en el Paso 1 y luego la posterior eliminación de cuantificadores en este paso 7. Notemos que en el Paso 1 los cuantificadores con los que clausuramos son *existenciales*, luego posiblemente sean *skolemnizados* en el Paso 6 (se "eliminarán" porque no los vamos a escribir pero de algún modo permanecen en el término de Skolem que ingresa como testigo). En este paso 7 simplemente "retiramos de la vista" (la expresión en inglés es "drop") los cuantificadores *universales* solo para no escribirlos (para no tener tantos símbolos dando vueltas) pero sabemos que están.

Paso 8. Obtención de la FNC. Este paso es sencillo y podemos intuir en qué consiste. A la fbf que proviene del Paso 7, si hiciera falta, simplemente le aplicamos las equivalencias lógicas conocidas y las también conocidas propiedades distributivas de la negación sobre la conjunción y sobre la disyunción para conseguirnos una fbf lógicamente equivalente en FNC.

Paso 9. Obtención de la Notación Clausal. En este paso escribimos la fbf en la notación clásica conocida como *notación de flechas*. Lo hacemos como sigue. Para cada paréntesis de disyunciones de la FNC[28], por aplicación repetida de la propiedad conmutativa, reunimos los literales positivos por un lado y los negativos por otro[29] como visualmente nos ilustra J. W. Lloyd[30]:

$$(A_1 \text{ v } \dots \text{ v } \neg B_1 \text{v } \dots \text{ v } A_k \text{ v } \dots \text{ v } \neg B_n) \equiv (A_1 \text{ v } \dots \text{v } A_k \text{ v } \neg B_1 \text{ v } \dots \text{ v } \neg B_n)$$

[28] Recordemos que la FNC es una conjunción de disyunciones que contiene átomos y átomos negados que llamamos respectivamente literales positivos y literales negativos.

[29] Esto podemos hacerlo porque la disyunción cumple con la propiedad de conmutatividad. Ver Ejemplo 1.13 de *Lógica para Matemáticos*, de A. G. Hamilton.

[30] Ver Capítulo 1, Sección 2, *First Order Theories*, del libro *Foundations of Logic Programming*, de W. L. Lloyd.

siendo los A_i y los B_j átomos. Seguidamente, por aplicación de la equivalencia de De Morgan, conseguimos la siguiente equivalencia:

$$(A_1 \text{ v } ...\text{v } A_k \text{ v } \neg B_1 \text{ v } \dots \text{ v } \neg B_n) \equiv (A_1 \text{ v } ...\text{v } A_k) \text{ v } \neg(B_1 \text{ ^ } \dots \text{ ^ } B_n).$$

A su vez, tenemos la siguiente equivalencia lógica:

$$(A_1 \text{ v } ...\text{v } A_k) \text{ v } \neg(B_1 \text{ ^ } \dots \text{ ^ } B_n) \equiv (A_1 \text{ v } ...\text{v } A_k) \leftarrow (B_1 \text{ ^ } \dots \text{ ^ } B_n).$$

Es fácil ver que tenemos un condicional cuyo antecedente es la conjunción de los literales negativos y el consecuente es la disyunción de los literales positivos.

Por convención, escribimos a esta fbf como la expresión:

$$A_1, \dots, A_k \leftarrow B_1, \dots, B_n$$

y decimos que está escrita en *notación de flechas*. Aquí finaliza el Paso 9.

Comentarios

La notación de flechas es la notación estándar en la que se escriben los programas lógicos[31] y la que adoptaremos de aquí en más para escribirlos. Notemos que en: $A_1, \dots, A_k \leftarrow B_1, \dots, B_n$ las comas (“,”) entre los B_j se entienden como conjunciones mientras que las comas entre los A_i se entienden como disyunciones. Estos detalles de escritura son muy significativos en lo que sigue de nuestro estudio porque, si bien las comas son símbolos ortográficos, *se interpretan en la jerga técnica como conectivos lógicos*, y el mismo símbolo ortográfico representa un conectivo u otro según el lado de la flecha en el que aparece. La notación de flechas es más usada para cuestiones sintácticas y procedurales de los programas lógicos.

La llamada *notación clásica* escribe cada cláusula como un conjunto de literales separados por comas que representan disyunciones: $\{A_1 , \dots, \neg B_1, \dots, A_k, \dots, \neg B_n\}$. Usamos también esta notación, mayormente al tratar cuestiones referidas a la semántica de los programas lógicos.

[31] Ver Capítulo 1, Sección 2, *First Order Theories*, del libro *Foundations of Logic Programming*, de W. L. Lloyd.

Cláusulas de programas lógicos

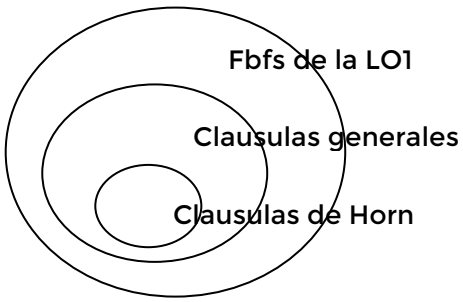
Notemos que cuando comenzamos el estudio del Algoritmo de Notación Clausal restringimos el uso de fbfs a únicamente cláusulas. Las cláusulas que se obtienen con el Algoritmo de Representación Clausal se llaman *cláusulas generales*.

Por motivos de computabilidad restringimos nuevamente la selección de cláusulas. Usaremos un subconjunto particular dentro de las cláusulas generales: las *cláusulas de Horn*. Solo con este subconjunto de cláusulas escribiremos programas lógicos.

La Figura 2 ilustra la relación entre las fbfs de la LO1, las cláusulas generales y las cláusulas de Horn:

Figura 2

Relación entre fbfs de la LO1, cláusulas generales y cláusulas de Horn



Definición. Cláusulas de Horn

Las cláusulas de Horn son cláusulas que tienen *a lo sumo un literal positivo*. Esto es, tienen alguna de las siguientes formas:

A	cláusula unitaria de programa definido
$A \leftarrow B_1, \ldots, B_k$	cláusula de programa definido
$\leftarrow B_1, \ldots, B_k$	cláusula objetivo.

Comentarios

Como una cláusula de Horn tiene a lo sumo un literal positivo entonces o bien puede tener un literal negativo o bien ningún literal negativo. Ese literal negativo, si aparece, se llama *cabeza de cláusula* (lado izquierdo de la punta de flecha, en la notación de flechas). Por su parte, la cláusula objetivo no tiene cabeza. Los literales que aparecen en el cuerpo de la una cláusula de Horn (ya sean los B_i, ya sea ninguno como en la cláusula unitaria, también llamada *hecho* o *incondicional*) componen lo que se llama *cuerpo de la cláusula*.

Punto de vista declarativo

Una cláusula de Horn es un condicional en el sentido intuitivo, tiene un significado lógico esperado, esto es: cuando el antecedente (una conjunción de literales) es verdadero ocurre que el consecuente también debe ser verdadero para que toda la cláusula sea verdadera. Una cláusula unitaria es un "incondicional" pues carece de antecedente y siempre se cumple (siempre "sucede"). Por ello a la cláusula unitaria la llamamos, también, *hecho* (*fact*, en inglés).

Punto de vista procedural

Una cláusula de Horn puede verse como una "llamada a procedimientos", esto es: para que se ejecute la *subrutina* que aparece en la cabeza de la cláusula deben ejecutarse primero, en algún orden, *todas las subrutinas* que están en el cuerpo de la cláusula (pues son sus subrutinas "antecedentes"). Informalmente decimos: "para ejecutar el átomo A, deben ejecutarse (en algún orden) todos los átomos B_1 y B_2, …, y B_k".

Fundamento computacional de la restricción de uso a solo cláusulas de Horn

Continuemos con el análisis de las cláusulas desde un enfoque procedural. Existen buenos motivos desde el punto de vista computacional para restringir la elección de nuestras cláusulas a solo cláusulas de Horn y descartar el uso de las cláusulas generales. Las cláusulas de Horn, al tener por definición a lo sumo un literal positivo, especifican de modo determinado y exacto cuál es la subrutina que debe ejecutarse inmediatamente después de la ejecución de las subrutinas del cuerpo de la cláusula: se debe ejecutar la subrutina de la cabeza de cláusula. Por ello los programas lógicos que estudiamos se llaman programas *definidos*: intuitivamente, porque está perfectamente determinado el orden de ejecución de las subrutinas: primero el cuerpo de cláusula (tomando los átomos en algún orden), luego la cabeza de cláusula. El término *definido*, además, aplica perfectamente en un sentido técnico, por lo que sigue. Observemos que si permitimos una cláusula general en un programa lógico podría darse el caso de tener que procesar una cláusula con, supongamos, dos literales positivos en la cabeza de cláusula, por ejemplo:

$$A_1, A_2 \leftarrow B_1, \ldots, B_k.$$

En este caso, luego de ejecutar el cuerpo de la cláusula (es decir, ejecutar todos los átomos B_i, en algún orden) computacionalmente se debe "decidir" o "seleccionar" cuál de los dos átomos en la cabeza de cláusula, si A_1 o A_2, se ejecutará primero. Supongamos que el sistema elige (o nosotros elegimos) ejecutar primero el átomo A_2. Supongamos también, seguidamente, que A_2

no conduce a una solución. Tenemos todavía al átomo A_1 para elegir: (recordemos que en la cabeza de una cláusula general las comas representan disyunción) entonces regresamos con backtracking a la cabeza de la cláusula para seguir la ejecución con el otro átomo, A_1. Vemos, de este modo, que con más de un átomo en la cabeza de la cláusula el resultado de la ejecución final puede ser incierto: una incorrecta selección del átomo de la cabeza con el cual seguir podría consumir tiempo computacional extra y, además, no garantizarnos el hallazgo de una solución[32] (por ejemplo, supongamos que el átomo seleccionado nos conduce a un *loop* infinito haciendo que el programa nunca termine).

Definición. Programa Lógico Definido

Un programa lógico definido es un conjunto de cláusulas unitarias (definidas) y cláusulas de programa (definido). Esto es, un conjunto de condicionales (que tienen cada uno un solo consecuente) y de "incondicionales" (hechos). Notemos que hemos dejado a las cláusulas objetivo fuera de esta definición.

En lo que sigue, omitiremos la descripción "definido" para los programas y para las cláusulas pues asumimos que trabajamos con programas y cláusulas definidas.

Ejemplo

Supongamos que escribimos los dos siguientes programas lógicos. Ambos programas capturan el conocimiento asociado a si una persona es ancestro de otra persona:

```
ancestro(x,y) :- progenitor(x,y).
ancestro(x,y) :- ancestro(x,z), progenitor(z,y).
```

y:

```
ancestro(x,y) :- ancestro(x,z), progenitor(z,y).
ancestro(x,y) :- progenitor(x,y).
```

Llamaremos a estos programas `Ancestro1` y `Ancestro2` respectivamente.

Comentarios desde el punto de vista lógico

Notemos que, desde el punto de vista de lo que *declaran*, el significado intuitivo de ambos programas es el mismo. Esto surge de la simple lectura de las líneas de código; tanto `Ancestro1`

[32] Recordar la noción de *no determinismo* vista en el Capítulo 1 de este texto.

como `Ancestro2` declaran qué significa que entre un *x* y un *y* dados haya una relación de ancestro/descendiente.

Comentarios desde el punto de vista procedural, o de la programación lógica

Las cláusulas cuyas cabezas contienen el mismo símbolo de predicado constituyen lo que llamamos la *definición* de ese predicado.

El símbolo ":-" representa históricamente en la disciplina una flecha[33] apuntando hacia la izquierda, como lo establece la notación de flechas. En un programa lógico las cláusulas típicamente se escriben una por renglón, es decir, cada línea de código es una cláusula, y éstas *se leen* sabiendo que están cuantificadas universalmente, y conectadas entre sí con conjunciones (en FNC).

Notemos la diferencia entre `Ancestro1` y `Ancestro2`: varía el orden en el que están escritas sus cláusulas. La primera cláusula en `Ancestro1` aparece en el segundo renglón de `Ancestro2`, y la segunda cláusula de `Ancestro1` aparece primera en `Ancestro2`. Si bien este orden no tiene relevancia desde el punto de vista lógico[34], podría tener alguna repercusión en el plano procedural, durante la ejecución de los programas, como veremos en la siguiente Sección.

Input de los programas lógicos

Las cláusulas objetivo (con cuerpo pero sin cabeza) *no integran un programa lógico*. Como su nombre lo indica, son objetivos *a dilucidar*. Como tales, se les presentan a un programa para que éste los "alcance" es decir, los "resuelva". Volveremos sobre las cláusulas objetivo más adelante. (Notar que tanto `Ancestro1` como `Ancestro2` carecen de cláusulas objetivo.)

SAT

En esta Subsección veremos cómo determinar satisfactibilidad en el contexto de los programas lógicos. Usamos como ejemplo motivador el siguiente escenario. Dada cierta información genealógica de unas personas, tenemos, asociado a este conocimiento, el siguiente programa lógico, que llamamos `Ancestro3`:

```
progenitor(Juan,Pedro).
```

[33] Esto es, un condicional.

[34] Porque las cláusulas están unidas entre sí por conjunciones, y la conjunción cumple con la propiedad conmutativa. La lógica, aquí, es "no determinística", a la LO1 le es irrelevante el orden de escritura de las cláusulas. Sin embargo el orden de escritura de las cláusulas tiene relevancia en programación lógica.

```
progenitor(Pedro,Luis).
ancestro(x,y) :- padre(x,y).
ancestro(x,y) :- ancestro(x,z), progenitor(z,y).
```

Supongamos que queremos saber si Juan es ancestro de Luis. Con todo lo que sabemos y hemos estudiado hasta el momento, nuestra *intuición lógica* nos indica que debemos comprobar si el átomo `ancestro(Juan, Luis)` "se desprende" (es consecuencia) del conjunto de cláusulas que componen `Ancestro3`. También nuestra *intuición informática* nos indica que debemos verificar si el átomo `ancestro(Juan, Luis)` es o no un output de `Ancestro3`. Ambas intuiciones son correctas.

Para enfrentarnos al funcionamiento computacional de `Ancestro3` y de programas lógicos en general (funcionamiento que, si bien intuimos, hasta ahora desconocemos en sus detalles técnicos) intentaremos averiguar primero si Juan es el ancestro de Luis desde el punto de vista lógico (también le decimos punto de vista *declarativo*).

Observación

Abordar primero el aspecto lógico nos permitirá comprender mejor el aspecto procedural de los programas lógicos. Sencillamente porque en este punto estamos más familiarizados con la lógica que con la programación lógica. Muchas de las buenas propiedades del Cálculo de Enunciados valen también para la LO1 y entonces las aprovecharemos. En lo que sigue usamos algunos conceptos y métodos que ya aprendimos al estudiar el Cálculo de Enunciados.

Desde el punto de vista lógico, está claro para nosotros que `Ancestro3` es un conjunto de cláusulas. Entonces podemos averiguar si el átomo `ancestro(Juan, Luis)` es consecuencia lógica[35] de `Ancestro3`. Ya tenemos un modo de resolver la consulta.

Para resolverla, y como en este punto es útil ganar un poco de generalidad para lo que sigue, llamaremos Γ a `Ancestro3` (hacemos un simple renombre) y llamaremos A al átomo `ancestro(Juan, Luis)`.

Evocamos ahora algunos conceptos que ya dominamos. Sabemos que si una fbf A es consecuencia lógica de un conjunto Γ entonces el *escenario* en el que las cláusulas de Γ son verdaderas también hace verdadera a la conclusión[36]. En la LO1 los escenarios (del mundo *real* o *ideal*) se describen con estructuras llamadas interpretaciones[37]. Cuando ocurre que un conjunto Γ es verdadero en un escenario decimos que ese escenario o interpretación es un *modelo* para Γ[38], y lo escribimos típicamente como M.

[35] Ver Proposición 2.5 de *Lógica para Matemáticos*, de A. G. Hamilton.
[36] Esto lo sabemos por la Proposición 2.5 de *Lógica para Matemáticos*, de A. G. Hamilton.
[37] Ver Definición 3.14 de *Lógica para Matemáticos*, de A. G. Hamilton.
[38] Ver Definición 4.40 de *Lógica para Matemáticos*, de A. G. Hamilton.

También sabemos que si un conjunto Γ tiene algún modelo entonces Γ es consistente[39]. Seguidamente, podemos formularnos la siguiente pregunta: si sucede que el átomo A es consecuencia lógica de un conjunto Γ de fbfs, ¿podría suceder que el átomo ¬A sea consecuencia lógica de ese Γ?. La respuesta es negativa, porque si A es consecuencia lógica de Γ entonces el conjunto $\Gamma \cup \{A\}$ tiene un modelo, digamos M; entonces M es modelo de Γ y también es modelo de A , entonces M es modelo de es modelo de $\Gamma \cup \{A\}$. Todo esto porque los modelos hacen verdaderos tanto al conjunto de fbfs que son premisas como a las consecuencias lógicas[40] de esas premisas. Por lo tanto, si el átomo ¬A fuese consecuencia lógica de Γ, M necesariamente sería modelo de ¬A y también lo sería de $\Gamma \cup \{\neg A\}$. Pero esto no puede ocurrir porque M es modelo de $\Gamma \cup \{A\}$ y M no puede ser modelo a la vez de $\Gamma \cup \{A\}$ y de $\Gamma \cup \{\neg A\}$. Por lo tanto $\Gamma \cup \{\neg A\}$ es inconsistente, es decir, no tiene modelo, tiene una contradicción dentro, es insatisfactible.

Enunciamos la siguiente proposición[41]:

Si A es consecuencia lógica de Γ,entonces $\Gamma \cup \{\neg A\}$ es insatisfactible.

Observación

Es crucial en este punto de nuestro estudio advertir la correspondencia entre averiguar si un átomo A es *output* de un programa lógico y averiguar si A es consecuencia lógica de Γ (las cláusulas que conforman ese programa lógico). Por lo analizado más arriba, ello es equivalente a preguntarnos si *la negación de lo que queremos saber*, es decir: ¬A, entra en contradicción con lo que se sabe, es decir, Γ[42]. ¿Cómo lo averiguamos?

Nos hemos planteado un dilema semejante en el Capítulo 1 cuando vimos que, o conseguimos un renglón de la tabla de verdad (un α) que satisface a las cláusulas de la FNC, o comprobamos que no existe tal renglón. En su momento, también visualizamos que cada renglón de la tabla de verdad puede entenderse como una configuración posible del mundo.

Si bien las estructuras sobre las que se define la semántica de los lenguajes de O1 son diferentes a las del Cálculo de Enunciados, a grandes rasgos los dos métodos que tenemos para enfrentarnos a SAT se asemejan a los ya estudiados en el Capítulo 1 en el sentido de que:

· o bien nos conseguimos un escenario en el cual A es una consecuencia lógica de Γ, esto es, nos conseguimos un M para $\Gamma \cup \{A\}$,

[39] Un conjunto Γde fbfs es consistente si a partir de él no pueden deducirse a la vez las fbfs A y ¬A. Ver Proposiciones 2.16 y 4.42 de *Lógica para Matemáticos*, de A. G. Hamilton.

[40] Ver Proposición 2.5 de *Lógica para Matemáticos*, de A. G. Hamilton.

[41] Ver Proposición 3.1 de *Foundations of Logic Programming*, de W. L. Lloyd. Notemos que la proposición cobra real interés para nosotros como informáticos cuando Γ es consistente.

[42] Si esto sucede, entonces A es consecuencia lógica de Γ.

· o bien probamos que tal escenario no existe, esto es, no existe M por ser $\Gamma \cup \{\neg A\}$ insatisfactible.

Entonces debemos concentrarnos en encontrar ese M para Γ, o probar que no existe.

Recordemos que en la LO1 no tenemos tablas de verdad. Trabajamos con unas estructuras llamadas interpretaciones[43]. Hemos dicho que un modelo M es una interpretación para el lenguaje de O1 con el que están escritas las fbfs en Γ de modo tal que todas las fbfs en Γ son verdaderas en esa interpretación.

Definición. Interpretación

Una interpretación I para un lenguaje de O1 es una cuaterna[44]:

I = <D, C, F, R>

en la que:

· D es un conjunto que constituye el dominio o universo de objetos. D es por definición no vacío, y eventualmente puede ser infinito. Sobre D varían (o *corren*) las variables del lenguaje, esto es, las variables son interpretables como objetos de D.

· C es un conjunto de objetos distinguidos, *objetos especiales* a los que llamamos por sus nombres particulares: $c_1 \ldots c_n$.

· A las funciones en F y a las relaciones en R las consideramos efectivas concreciones de las letras de función y de predicado del lenguaje. F es, precisamente, el conjunto de funciones de diferentes aridades sobre D a las que se refieren las letras de función. Cada símbolo de función f_i se corresponde con una función de cierta aridad i que recibe una cantidad i de objetos de D como argumento y retorna como resultado una cantidad j de objetos de D.

· R es un *conjunto de colecciones de n-tuplas* de objetos de D. Por cada símbolo de predicado p_n hay una colección de n-tuplas de objetos de D[45].

Cómo encontrar un modelo para Γ

Dijimos que un modelo es una interpretación. Encontrar un modelo puede ser una tarea no trivial. Esto porque si bien puede existir nosotros podríamos no darnos cuenta de cuál es porque, por ejemplo, los objetos de su dominio son suficientemente complicados, las funciones pueden ser difíciles de definir, etcétera. El modelo que buscamos, incluso, puede no existir, y nosotros podríamos continuar intentando descubrirlo o construirlo sin saber si efectivamente existe o no.

[43] Ver Proposición 3.14 de *Lógica para Matemáticos*, de A. G. Hamilton.

[44] Ver Definición 3.14 de *Lógica para Matemáticos*, de A. G. Hamilton.

[45] Haciendo una analogía con nuestros conocimientos referidos a bases de datos relacionales, para cada predicado n-ario p_n hay una tabla (potencialmente infinita) de n columnas, cada fila de la tabla es una n-tupla de objetos de D. La colección de estas "tablas" es R.

Afortunadamente, en el contexto de la programación lógica existe una clase especial de interpretaciones que sirven para enfrentar este problema. Son interpretaciones que nos garantizan que si existe un modelo M para un Γ entonces ese M tiene una *forma* garantizada. Y viceversa, si no podemos hallar esa forma garantizada que tendrá M, entonces Γ no tiene modelo.

Abordamos el estudio y tratamiento de estas interpretaciones especiales en el Capítulo 3.

Ejercicios para el Capítulo 2

1. Representar en algún lenguaje de O1:

a) Alguien es la madre de un individuo si es mujer y es progenitor de ese individuo. Alguien es el padre de un individuo si es varón y es progenitor de ese individuo. Cualquiera que tenga un progenitor que es humano, es humano. Un individuo es humano si su madre y su padre lo son. Nadie es progenitor de sí mismo.

b) Todo peluquero afeita a todo aquel que no se afeita a sí mismo. Ningún peluquero afeita a alguien que se afeite a sí mismo.

c) Ningún dragón que viva en un zoológico es feliz. Cualquier animal que encuentre gente amable es feliz. Las personas que visitan los zoológicos son amables. Los animales que viven en zoológicos encuentran personas que visitan zoológicos.

d) Si alguien hace algo bueno, ese alguien es bueno. Del mismo modo, si alguien hace algo malo, es malo. Sebastián ayuda a su madre y también miente algunas veces. Mentir es malo y ayudar es bueno.

e) El Capitán Wine era responsable de la seguridad de sus pasajeros y su carga. Pero en su último viaje, se emborrachaba todas las noches y fue responsable de la pérdida del barco, con todo lo que llevaba. Se rumoreaba que estaba loco, pero los médicos lo encontraron responsable de sus actos. Usualmente, el capitán Wine no actuaba borracho. Durante aquel viaje, el capitán Wine se comportó muy irresponsablemente. El capitán Wine sostuvo que las tormentas fueron las responsables de la pérdida del barco, pero en el proceso que se le siguió fue encontrado responsable de la pérdida de vidas y bienes. Todavía vive, y es responsable de la muerte de muchas mujeres y niños.

Representar este conocimiento dado esta vez teniendo en cuenta 4 tipos de definiciones distintas de responsabilidad: como obligación o función derivada del cargo, en el sentido de factor causal, como estado mental, como punible o moralmente reprochable.

2. Traducir a notación clausal las siguientes fbfs escritas en un lenguaje de O1 que contiene al símbolo “a” como constante; las variables: x, y, z; los símbolos de función: f (binario) y g (unario), y los símbolos de predicado: p, q, s (unarios), r, t (binarios).

i. $(\forall x)(t(x,y) \rightarrow \neg r(x))$.

ii. $(\forall x)(p(x) \rightarrow p(x))$.

iii. $((\forall x)p(x)) \rightarrow ((\forall x)p(x))$.

iv. $\neg((\forall x)r(x)) \rightarrow (\neg(\forall x)(\exists z)s(x,z)))$.

v. $(\forall x)(\neg r(x,a) \rightarrow ((\exists y)(r(y,g(x)) \wedge (\exists z)(r(z,g(x)) \rightarrow r(y,z)))))$.

vi. $(\forall x)(\forall y)((s(x) \wedge t(x,y)) \rightarrow ((\exists z)(q(z) \wedge r(x,z)))$.

vii. $(\forall x)(\forall y)((s(x) \wedge t(x,y)) \rightarrow (\exists z)(q(z) \wedge r(x,z)))$.

viii. $(\neg(\exists y)p(x)) \rightarrow ((\forall y)(\forall x)((\neg r(f(x),y) \wedge s(y)) \rightarrow \neg r(x,y)))$.

CAPÍTULO 3
El enfoque orientado a modelos

Clara Smith

En este Capítulo abordamos el aspecto *orientado a modelos* del problema SAT para cláusulas de Horn.

Existe, de algún modo, una *razón histórica* para primero estudiar el enfoque orientado a modelos (o enfoque semántico), y a continuación estudiar el enfoque orientado a la demostración automática de teoremas (o enfoque sintáctico). Esta razón tiene su base en que la teoría matemática que sustenta a la Programación Lógica -del estilo clásico de los programas Prolog que tratamos en este libro de cátedra- fue desarrollada cerca de 1930, y solo posteriormente, entre los años 1965 y 1967 se la usó como fundamento formal para implementar computacionalmente la regla de Resolución.

Interpretaciones de Herbrand

En el Capítulo 2 estudiamos la siguiente Proposición[46]:

> Sea Γ un conjunto de fbfs cerradas y sea A una fbf cerrada escrita en un LO1.
> Entonces A es consecuencia lógica de Γ si y solo si $\Gamma \cup \{\neg A\}$ es insatisfactible.

Hemos prestado especial atención a la correspondencia existente entre averiguar si A es consecuencia lógica de Γ y averiguar si A es output del programa lógico (cuyas cláusulas son las de Γ).

Vimos que los dos métodos disponibles para aplicar la proposición enunciada más arriba se asemejan a los que estudiamos en el Capítulo 1 en el sentido de que, o bien conseguimos un escenario en el cual A es una consecuencia lógica de Γ, esto es, conseguimos algún modelo M

[46] Ver Proposición 3.1 de *Foundations of Logic Programming*, de W. L. Lloyd.

para $\Gamma \cup \{A\}$, o bien probamos que tal M no existe. Luego hacia el final del Capítulo 2 mencionamos una "clase especial" de interpretaciones que nos garantizan que si existe ese M entonces es posible determinar su *forma canónica*[47] (aunque M sea difícil de describir con precisión). Y también enunciamos un contrarrecíproco de esta afirmación: dijimos que si no podemos identificar esa forma de M, entonces M no existe, y Γ no tiene modelo[48].

Aquellas interpretaciones especiales se llaman **Interpretaciones de Herbrand[49]**. En este Capítulo nos dedicamos a estudiarlas.

Intuición Principal

Primero adoptaremos un modo lo suficientemente amplio para referirnos a los objetos del universo usando una manera de nombrar (de denotar) a esos objetos con todos los nombres posibles que pudieran tener y con los que pudiéramos llamarlos, siempre de acuerdo a los símbolos del LO1 con el que está escrito Γ[50].

¿Cómo lo haremos? Combinando de todas las maneras posibles las constantes y los símbolos de función del lenguaje en uso, generando así *todas las maneras posibles de escribir términos*.

Notemos que esta generación de nombres es "a propósito", *intencional*. Lo hacemos para no perdernos ningún nombre posible con los que denotar a los objetos.

A continuación describimos los pasos para, dado un Γ, conseguir ese gran conjunto de nombres de objetos, que llamamos *Universo de Herbrand*.

Construcción del Universo de Herbrand[51]

Input: un conjunto finito Γ de cláusulas escritas en algún LO1.

Output: el Universo de Herbrand para Γ. Escribimos $U_H(\Gamma)$.

Paso 1. Controlamos si en Γ aparece alguna constante. Si no la hay, introducimos alguna a propósito, para poder construir términos[52] (como vimos en el Capítulo 2).

---

[47] Sea Γ un conjunto de cláusulas. Si Γ tiene un modelo entonces tiene un modelo de Herbrand. Ver Proposición 3.2 de *Foundations of Logic Programming*, de W. L. Lloyd.

[48] Sea Γ un conjunto de cláusulas. Entonces Γ es insatisfactible si y sólo si no tiene modelo de Herbrand. Ver Proposición 3.2 de *Foundations of Logic Programming*, de W. L. Lloyd.

[49] Ver Sección 3 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

[50] Esta restricción de trabajo se llama DCA, *Domain Closed Assumption* (Suposición de Dominio Cerrado). Se relaciona con el interés nuestro como informáticos de intentar manejar siempre la menor cantidad posible de símbolos.

[51] Ver Sección 3 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

[52] Vamos a denotar a todos los objetos con todos sus posibles "nombres concretos". Como ocurre entre nosotros los humanos, que nos llamamos por nuestros nombres, por nuestros diminutivos, por alias, o por sobrenombres. Es por ello entonces que no tomamos las variables para trabajar, queremos referirnos a los objetos por sus nombres y no de manera genérica (o "anónima") como lo hacemos cuando usamos variables.

Paso 2. Controlamos si en Γ aparecen símbolos de función. Si hay, los tomamos para trabajar. Si no hay, no introducimos ninguno nuevo.

Paso 3. Con los símbolos del Paso 1 y del Paso 2 armamos un conjunto especial de términos que llamamos *Universo de Herbrand*. Es un conjunto de *términos ground* (sin variables). Lo construimos inductivamente como sigue:

$U_{H0}(\Gamma)$ = if *card*[53]$(C(\Gamma)) \neq 0$ then $C(\Gamma)$ else {a}[54].

$U_{Hi+1}(\Gamma) = U_{Hi}(\Gamma) \cup \{f_k(t_1, \ldots, t_k)\}$ con $t_j \in U_{Hi}(\Gamma)$, $f_k \in F(\Gamma)$[55].

Esta construcción se entiende así: el Universo de Herbrand puede escribirse como una sucesión de *cortes analíticos*, donde el corte (i+1)-ésimo es igual a el corte i-ésimo más los términos ground que se obtienen al aplicar todos los símbolos de función a los términos ground que aparecen en el corte i-ésimo. Notemos que i es el subíndice para los cortes del Universo, j es el subíndice para identificar a los términos, y el índice k señala la aridad de la función.

Definición. Universo de Herbrand

$U_H(\Gamma) = \cup_i U_{Hi}(\Gamma)$. El Universo de Herbrand para Γ es la unión de todos los cortes U_{Hi}. Es decir, el Universo de Herbrand para Γ es el conjunto de todos los posibles *términos ground*[56] armados a partir de las constantes y de los símbolos de función que aparecen en Γ.

Ejemplo

Sea un LO1 con los siguientes símbolos: constante: a, símbolos de función: f (unario), y símbolos de predicado p, q, r (dos unarios y uno binario). Sea Γ el siguiente conjunto de cláusulas, escrito en notación clausal clásica: Γ = {{p(a)}, {r(x,f(x)), ¬q(x)}}. Entonces $U_{H0}(\Gamma)$ = {a}, $U_{H1}(\Gamma)$ = {a, f(a)}, $U_{H2}(\Gamma)$ = {a, f(a), f(f(a))}, $U_{H3}(\Gamma)$ = {a, f(a), f(f(a)), f(f(f(a)))}, y así siguiendo. Esta definición del $U_H(\Gamma)$ está escrita *por extensión*, aunque en rigor nunca podremos terminar de escribirla porque es infinita. Podemos, sin embargo, escribir: $U_H(\Gamma) = \{f^n(a), n \geq 0$[57]$\}$. Esta definición del $U_H(\Gamma)$

[53] La función *card* retorna la cantidad de elementos de un conjunto, en este caso, la cantidad de constantes que aparecen en las cláusulas en Γ.

[54] Introducimos la constante "a" a propósito. Puede ser cualquier otra constante; "a" es la comúnmente usada en este caso.

[55] Recordemos que en el Capítulo 2 hemos definido a C y a F, respectivamente, como los conjuntos de constantes y de símbolos de función del lenguaje de O1 con el que está escrito Γ. Es conveniente ahora escribirlos como $C(\Gamma)$ y $F(\Gamma)$ para indicar claramente que nos referimos a los respectivos conjuntos de símbolos del lenguaje pero restringidos a los que aparecen en Γ.

[56] Son términos sin variables. Ver Sección 3 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

[57] Por convención, $f^0(a) = a$.

está escrita *por comprensión*, según la terminología de Teoría de Conjuntos clásica. Incluye a todos los términos del $U_H(\Gamma)$.

Nota

En algunos casos, como en el ejemplo previo, el $U_H(\Gamma)$ puede ser un conjunto infinito. Esto sucede cuando tenemos al menos un símbolo de función en el lenguaje con el que está escrito Γ. La infinitud es consecuencia del mecanismo de generación del universo (saturación de símbolos de función sobre elementos del corte previo).

Siguiendo con la intuición principal, ahora que tenemos todas las maneras de denotar objetos, conseguiremos todas las maneras de *predicar* sobre ellos.

Definición. Base de Herbrand

$B_H(\Gamma) = \{p(t_k, \ldots, t_j)$ con $t_i \in U_H(\Gamma)$, $p \in P(\Gamma)$[58]$\}$. La aplicación por saturación sobre el $U_H(\Gamma)$ de todos los símbolos de predicado que aparecen en Γ forma la Base de Herbrand, escribimos $B_H(\Gamma)$. Es el conjunto de todos los posibles átomos ground[59] que podemos escribir (restringido a los símbolos que aparecen en Γ).

Nota

En la $B_H(\Gamma)$ tenemos todas las maneras posibles de *predicar propiedades* -expresarlas, manifestarlas- sobre todas las maneras posibles de denotar a los objetos en el $U_H(\Gamma)$.

Ejemplo

Tomemos el ejemplo previo. Sea $\Gamma = \{\{p(a)\}, \{r(x,f(x)), \neg q(x)\}\}$. Tenemos que $U_H(\Gamma) = \{f^n(a), n \geq 0\}$. Entonces podemos intuitivamente ver que la $B_H(\Gamma)$ luce como sigue: $B_H(\Gamma) = \{p(a), p(f(a)), p(f(f(a))), ..., q(a), q(f(a)), q(f(f(a))), ..., r(a,a), r(a,f(a)), r(f(a),a)\ r(f(a),f(a)), r(f(f(a)),a), r(a,f(f(a))), \ldots\}$ y así siguiendo. Esta es una descripción *por extensión* de la $B_H(\Gamma)$, que no podemos terminar de escribir porque es infinita. Escrita más prolijamente, por comprensión, nos queda: $B_H(\Gamma) = \{p(f^n(a)), n \geq 0\} \cup \{q(f^n(a)), n \geq 0\} \cup \{r(f^n(a),f^m(a)), n \geq 0, m \geq 0\}$, con $f^n(a), f^m(a) \in U_H(\Gamma)$.

[58] $P(\Gamma)$ es el conjunto de símbolos de predicado del LO1 restringido a Γ.
[59] Son átomos sin variables.

Ahora que conocemos cómo construir el $U_H(\Gamma)$ y la $B_H(\Gamma)$, estudiamos algunas definiciones técnicas útiles.

Definición. Interpretación de Herbrand para Γ

Cualquier subconjunto de átomos de la $B_H(\Gamma)$ constituye una interpretación de Herbrand para Γ, escribimos: $I_H(\Gamma)$. Si tenemos que trabajar con varias interpretaciones para un mismo Γ, agregamos subíndices.

Ejemplo. Para el conjunto de cláusulas Γ del ejemplo previo, {p(a)} es una interpretación de Herbrand, la llamamos $I_{H1}(\Gamma)$. Otra posible interpretación es $I_{H2}(\Gamma)$: $\{q(f^n(a)), n \geq 0\}$. Notemos que $I_{H1}(\Gamma)$ es finita y que $I_{H2}(\Gamma)$ es infinita.

Hay claramente otras interpretaciones de Herbrand para esa base, dejamos como ejercicio al lector el encontrarlas.

Definición. Modelo de Herbrand para Γ

Una $I_H(\Gamma)$ que es modelo[60] de Γ se dice modelo de Herbrand para Γ. Escribimos $M_H(\Gamma)$; sub-indicamos $M_{Hi}(\Gamma)$ si trabajamos con varios modelos para un mismo Γ.

Por ejemplo, para el conjunto de cláusulas Γ del ejemplo previo, la $I_{H1}(\Gamma)$ = {p(a)} es un $M_H(\Gamma)$. Si bien ello surge de la propia definición de modelo, quedará más claro a continuación y el lector podrá comprobarlo fácilmente.

Modelos de Herbrand y SAT.

Estamos en condiciones de enunciar la siguiente **Proposición**[61]:

Sea Γ un conjunto de **cláusulas**[62];
Si Γ tiene un modelo, entonces Γ tiene un Modelo de Herbrand.

[60] Recordemos que un modelo de Γ es alguna interpretación que hace verdaderas a las fbfs en Γ. Ver Definición 4.40 de *Lógica para Matemáticos*, de A. G. Hamilton.
[61] Ver Proposicion 3.2 de *Foundations of Logic Programming*, de W. L. Lloyd.
[62] Esta proposición es falsa si estamos trabajando con fbfs de la LO1 que no son cláusulas.

Comentario

La proposición establece que, si sucede que Γ tiene un modelo, llamémoslo M, no importa cuán complicado sea M de describir o de identificar, podemos asegurar que hay un $M_H(\Gamma)$. A esta proposición se la ha llamado también *Propiedad de existencia de una interpretación de Herbrand asociada* a una interpretación I para Γ.

Ahora bien: dada una interpretación arbitraria I para el lenguaje en el que está escrito un Γ, ¿cómo conseguimos una $I_H(\Gamma)$ *asociada* a I para Γ? ¿Cómo aseguramos una correspondencia entre ambas I y $I_H(\Gamma)$?. Una vez que asociemos a I con una $I_H(\Gamma)$, comprobar si ambas son modelos de Γ será tarea directa.

Interpretación asociada o canónica. Intuición

Dada una I cualquiera para el lenguaje en el que está escrito un conjunto de fbfs Γ, conseguir una interpretación de Herbrand $I_H(\Gamma)$ asociada a I para Γ significa conseguir una interpretación de Herbrand tal que ambas interpretaciones I e $I_H(\Gamma)$ *se corresponden*. Es decir, "conectaremos" -en una y otra interpretación- las maneras de denotar a los mismos objetos, y también a las maneras de predicar sobre ellos (conectaremos términos en una y otra interpretación, y predicados en una y otra interpretación, para que se refieran a lo mismo). Y ambas interpretaciones deberán entonces, razonablemente, comportarse del mismo modo respecto de Γ: hacer verdaderas a las mismas fbfs de Γ, y hacer falsas a las mismas de Γ.

¿Cómo hacemos esta conexión? A los átomos de Γ que (en I) podemos verificar que hacen verdaderas a las fbfs en Γ, los "vinculamos" con aquellos átomos de la $B_H(\Gamma)$ que predican sobre los mismos objetos que predican aquellos átomos. Esto lo hacemos así: tomamos los átomos en I que hacen verdaderas a las fbfs de Γ, y para cada uno de ellos tomamos un correspondiente átomo en la $B_H(\Gamma)$. Mas formalmente:

Interpretación asociada o canónica. Definición.

Dada una I cualquiera para el lenguaje en el que está escrito un conjunto Γ de fbfs, una interpretación de Herbrand asociada a I para Γ, llamémosla $I_H(\Gamma)^*$, se define como:

$$I_H(\Gamma)^* = \{p(t_1, \ldots, t_n) \in B_H(\Gamma) \,/\, p(t_1, \ldots, t_n) \text{ es verdadero con respecto a I en } \Gamma\}.$$

con $p \in P(\Gamma)$[63], y t término del lenguaje de O1 con el que está escrito Γ.

[63] Esto es, p es un símbolo de predicado del LO1 con el que está escrito Γ.

Una $I_H(\Gamma)^*$ es entonces un subconjunto de átomos de la $B_H(\Gamma)$ tal que, **instanciados en Γ,** hacen verdaderas a las fbfs de Γ[64]. Al conocer nosotros cuál es la I (pues nos es dada) ya sabemos cómo se interpretan las constantes, los símbolos de función y los símbolos de predicado, y entonces sabemos evaluar las cláusulas de Γ en esa interpretación I[65]. Por lo tanto, para construir $I_{Hi}(\Gamma)^*$ simplemente seleccionamos átomos de la $B_H(\Gamma)$ que se corresponden con aquellos átomos de Γ tal que, instanciados hacen verdadero al conjunto Γ de fbfs.

Ejemplo

Supongamos el siguiente conjunto de fbfs Γ, y supongamos la siguiente interpretación arbitraria I para Γ.

Sea Γ = {{s(g(x))}, {r(x), ¬q(x)}}, con s, r y q símbolos de predicado de un LO1, y con g símbolo de función de ese lenguaje (todos los símbolos son unarios). Sea I una interpretación arbitraria para Γ tal que el dominio de I es el conjunto finito D: {1,2}, la función g está definida de modo tal que g(1) = 1 y g(2) = 1, el predicado unario s es verdadero sólo cuando su argumento es 1 (escribimos: s → 1), el predicado unario r es verdadero para 1 y para 2 (escribimos: r → {1,2}), y el predicado q nunca es verdadero, es siempre falso (escribimos: q → {}). Para conseguir una $I_H(\Gamma)$ primero armemos el $U_H(\Gamma)$ y la $B_H(\Gamma)$. Como en Γ no hay constantes, introducimos una a propósito, por ejemplo: a. Seguidamente, tenemos entonces que: $U_H(\Gamma) = \{g^n(a), n \geq 0\}$ y que: $B_H(\Gamma) = \{s(g^n(a)), n \geq 0\} \cup \{r(g^n(a)), n \geq 0\} \cup \{q(g^n(a)), n \geq 0\}$. Fijemos ahora una interpretación posible para la constante a; ya que tenemos dos posibilidades porque los objetos de D son dos. Elegimos un objeto cualquiera de D, por ejemplo el 1. Entonces hacemos la asignación: a → 1 (de este modo la constante apunta a un objeto del Dominio, como se espera). Notemos que al hacer esta asignación los objetos apuntados por los términos del $U_H(\Gamma)$ quedan completamente determinados, y también quedan determinados cuáles átomos de la $B_H(\Gamma)$ se corresponden con los evaluados como verdaderos en I para Γ (y cuáles no). Sólo queda entonces que tomemos aquellos átomos de la $B_H(c)$ que están en correspondencia con los átomos verdaderos en I para Γ. Armamos una $I_{H1}(\Gamma)^* = \{s(g^n(a)), n \geq 0\} \cup \{r(g^n(a)), n \geq 0\}$. Con esta selección de átomos, la primera cláusula de Γ: {s(g(x))} es verdadera pues el símbolo de predicado s es verdadero en I cuando es instanciado con objetos que apuntan al 1 (todos los $g^n(a)$ darán 1), y todos los términos $g^n(a)$ apuntan al 1, por la interpretación que hemos hecho de

[64] Esto independientemente de que I sea un modelo o no para Γ.

[65] Ver Definición 3.20 de *Lógica para Matemáticos*, de A. G. Hamilton.

a y por definición de la función g en I, que retorna siempre el valor 1. La segunda cláusula de Γ es verdadera porque el símbolo de predicado r es verdadero en I cuando tomamos todos los términos $g^n(a)$, y sabemos que en I la función g retorna siempre el 1 y en I el símbolo de predicado es verdadero cuando su argumento es 1. Al no haber tomado ningún átomo para $I_{H1}(\Gamma)^*$ que contenga a q se verifica la correspondencia ya que el predicado q nunca es verdadero $I_{H1}(\Gamma)^*$ (es siempre falso en $I_{H1}(\Gamma)^*$ tal como en I).

Ejercicio

Armar una $I_{H2}(\Gamma)^*$ diferente, con a → 2.

Pasemos ahora al análisis de la proposición enunciada más arriba: **Si Γ tiene un Modelo entonces tiene un Modelo de Herbrand.**

Sabemos que una interpretación que es modelo -llamémoslo M- puede identificarse con un conjunto de átomos tales que, evaluados en M, hacen *verdaderas* a las cláusulas en Γ. Sabemos también que, para que todas las cláusulas en Γ sean verdaderas, cada una de ellas debe ser verdadera.

Sucede además que las cláusulas en Γ pueden no ser todas ground, esto es, tener algunas variables cuantificadas universalmente[66]. Entonces, para que cada una de dichas cláusulas sea verdadera, *todos* los elementos del dominio deben hacer verdadera a dichas cláusulas, respecto de esa variable cuantificada universalmente[67].

Sabemos cuáles son los átomos que hacen verdadero a Γ en M (porque sabemos que M es modelo). Así, hemos relacionado estos átomos con algún subconjunto de átomos de la $B_H(\Gamma)$; esto lo podemos hacer porque sabemos cómo conseguir interpretaciones de Herbrand asociadas. Como nos hemos ocupado de construir a la $B_H(\Gamma)$ exhaustivamente, aquellos átomos que son verdaderos en M son algún subconjunto de la $B_H(\Gamma)$. De este modo, para conseguirnos el $M_H(\Gamma)$ que la Proposición asegura que existe -sabiendo que existe M- es suficiente con seleccionar de la $B_H(\Gamma)$ aquellos átomos que apunten a los mismos objetos que en M hacen verdaderas a las cláusulas en Γ. Para los átomos ground en M, debe quedar claro que tomarlos de $B_H(\Gamma)$ es inmediato.

[66] Por la definición de cláusula que estamos manejando en este texto.

[67] Por ejemplo, si la cláusula es p(x) todos los elementos del dominio tienen que satisfacer el predicado p y entonces el predicado p es verdadero.

Ejercicio

Verificar si la $I_{H1}(\Gamma)$* identificada más arriba es un $M_H(\Gamma)$. Idem para la $I_{H2}(\Gamma)$* del ejercicio previamente propuesto.

El método que sigue continuación es útil para detectar si un conjunto Γ de cláusulas tiene o no modelo, porque identifica de modo directo la forma canónica de tal modelo sin pasar previamente por el análisis de interpretaciones I cualesquiera dadas para Γ.

Técnica de *búsqueda directa* de un $M_H(\Gamma)$ (sin que se nos provea de una interpretación cualquiera o de una interpretación que sea modelo para Γ)**.** Dado Γ, armamos el $U_H(\Gamma)$ y la $B_H(\Gamma)$. Intentamos encontrar directamente un $M_H(\Gamma)$ seleccionando de la $B_H(\Gamma)$ alguna $I_H(\Gamma)$ tal que haga verdadera a las cláusulas en Γ. Para esto, para cada fbf A[68] en Γ, elegimos átomos de la $B_H(\Gamma)$ tal que al "instanciarlos" en A hacen verdadera a A(cuando A no tiene variables, tomamos directamente de la $B_H(\Gamma)$ el átomo ground que hace verdadera a A).

Si podemos conseguir esa $I_H(\Gamma)$, entonces tenemos un $M_H(\Gamma)$.

Ejemplo

Trabajamos con el mismo conjunto Γ con el que venimos trabajando. Sea Γ = {{p(a)}, {r(x,f(x)), ¬q(x)}}. Tenemos que $U_H(\Gamma)$ = {$f^n(a)$, $n\geq 0$} y tenemos que $B_H(\Gamma)$ = {$p(f^n(a))$, $n\geq 0$} ∪ {$q(f^n(a))$, $n\geq 0$} ∪ {$r(f^n(a),f^m(a))$, $n\geq 0$, $m\geq 0$}; con $f^n(a)$ y $f^m(a) \in U_H(\Gamma)$. Sabemos que en la $B_H(\Gamma)$ tenemos la aplicacion exhaustiva (todas las posibles aplicaciones) de predicados sobre todos los términos ground. Seguidamente aplicamos la técnica de búsqueda directa de un $M_H(\Gamma)$ que describimos más arriba. Al saber que podemos identificar interpretaciones de Herbrand como subconjuntos de la $B_H(\Gamma)$, seleccionamos algunos átomos de ésta, armamos algunas interpretaciones de Herbrand, y controlamos si son o no un $M_H(\Gamma)$.

Sean: $I_{H1}(\Gamma)$ = {p(a)}, $I_{H2}(\Gamma)$ = {p(a), r(a,f(a))}, $I_{H3}(\Gamma) = B_H(\Gamma)$, $I_{H4}(\Gamma)$ = {$r(f^n(a),f^m(a))$}. Ahora analizamos si estas interpretaciones son o no modelos de Herbrand.

$I_{H1}(\Gamma)$ es modelo de Herbrand, llamémoslo $M_{H1}(\Gamma)$, porque: i) hace verdadera a la primera cláusula de Γ (ground), y la segunda cláusula de Γ (que no es ground) también es verdadera pues al no tomar ningún átomo que tenga la *forma sintáctica* $q(f^n(a))$ ocurre que la fbf (∀x)(r(x,f(x)) v ¬q(x)) es verdadera. Siguiendo, $I_{H2}(\Gamma)$ es modelo de Herbrand para Γ por los mismos motivos que lo es $I_{H1}(\Gamma)$ (llamémoslo $M_{H2}(\Gamma)$); esto porque el hecho de seleccionar solo el átomo r(a,f(a)) no invalida que la segunda cláusula de Γ sea verdadera: lo es porque no seleccionamos ningún átomo con la *forma sintáctica* $q(f^n(a))$[69]. Continuando, tenemos $I_{H3}(\Gamma) = B_H(\Gamma)$, es decir, toda la

[68] Recordemos que A puede no ser ground.

[69] Vale el (∀x) (¬q(x)).

base de Herbrand. Vemos que $I_{H3}(\Gamma)$ es modelo de Γ (llamémoslo $M_{H3}(\Gamma)$) pues: i) hace verdadera a la primera cláusula, ya que el átomo p(a) $\in B_H(\Gamma)$; y ii) al tomar todos los átomos de la *forma sintáctica* $r(f^n(a),f^{n+1}(a))$, $n\geq 0$, es verdadera la segunda cláusula por ser verdadera la primera componente de la disyunción. Finalmente, $I_{H4}(\Gamma)$ no es modelo de Γ porque si bien hace verdadera a la segunda cláusula de Γ por los mismos motivos que la hace verdadera $I_{H3}(\Gamma)$, falta tomar el átomo p(a) para que la primera cláusula de Γ sea verdadera.

Insatisfactibilidad e inexistencia de Modelos de Herbrand

Enunciamos a continuación la proposición que completa la manera de enfrentarnos a SAT desde el punto de vista orientado a modelos.

Proposición[70]: Un conjunto Γ de cláusulas escrita en un LO1 es insatisfacible si y solo si no tiene Modelo de Herbrand.

Para comprobar cómo funciona esta proposición intentemos favorecer nuestra intuición.

Sabemos identificar un modelo canónico para Γ, esto es, dado Γ sabemos calcular su $U_H(\Gamma)$ y su $B_H(\Gamma)$, sabemos identificar alguna $I_H(\Gamma) \subseteq B_H(\Gamma)$ y verificar si es o no es un $M_H(\Gamma)$. Si dentro de la $B_H(\Gamma)$ no hay un subconjunto que sea un $M_H(\Gamma)$ quiere decir que es *imposible* que exista algún modelo arbitrario (digamos, "no de Herbrand"), pues es imposible encontrar una *estructura canónica* para él. Comprobémoslo con un ejemplo.

Ejemplo

Sea Γ = {{p(a)}, {q(a),¬p(a)}, {¬q(x)}}. Tenemos que $U_H(\Gamma)$ = {a} es finito, porque no hay símbolos de función en el LO1 con el que está escrito Γ. Tenemos $B_H(\Gamma)$ = {p((a),q(a)}, que es finita porque saturamos los símbolos de predicado de Γ sobre el $U_H(\Gamma)$, que es finito. Analicemos ahora si Γ tiene modelo. Si Γ tiene modelo, sabemos por la proposición enunciada previamente[71] que existe un $M_H(\Gamma)$ y que debemos poder armarlo a partir de algún subconjunto de la $B_H(\Gamma)$. Hemos estudiado en Teoría de Conjuntos que el conjunto de los subconjuntos de un conjunto se llama *conjunto de partes de un conjunto*. Para la $B_H(\Gamma)$ el conjunto de todos sus subconjuntos es: $Pow(B_H(\Gamma))$[72] = {{}, {p(a)}, {q(a)}, {q(a),p(a)}}. Esto quiere decir que si Γ tiene algún $M_H(\Gamma)$ entonces

[70] Ver Proposición 3.3 de *Foundations of Logic Programming*, de W. L. Lloyd.

[71] Proposición 3.2 de *Foundations of Logic Programming*, de W. L. Loyd.

[72] Pow, por *Powerset* en inglés, es por definición el conjunto de partes de un conjunto (todos los subconjuntos de un conjunto). Ver Definición 1.2 y Ejemplo (1) de *A Course in Universal Algebra*. S. Burris, H. P. Sankappanavar. Springer, 1981.

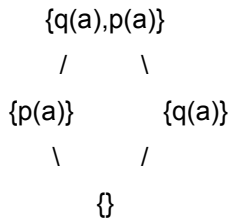
es alguno de estos cuatro subconjuntos (recordemos que, por definición, cualquier subconjunto de la $B_H(\Gamma)$ es una $I_H(\Gamma)$.

Analicemos: tomemos $I_{H1}(\Gamma)$ = {}; el conjunto vacío no es modelo de Γ porque no hace verdadera a la primera cláusula, pues necesitamos seleccionar p(a) para que la primera cláusula sea verdadera (y no hemos tomado ningún átomo en $I_{H1}(\Gamma)$). Siguiendo, $I_{H2}(\Gamma)$ = {p(a)} hace verdadera a la primera cláusula de Γ ({p(a)})y también a la tercera cláusula de Γ ({¬q(x)}), pero no hace verdadera a la segunda cláusula de Γ ({q(a), ¬p(a)}) porque para eso necesitamos seleccionar el átomo q(a) y no seleccionar el átomo p(a). La $I_{H3}(\Gamma)$ = {q(a)} tampoco es modelo por los mismos motivos que no lo es $I_{H1}(\Gamma)$, la primera cláusula no es verdadera. Finalmente, $I_{H4}(\Gamma) = B_H(\Gamma)$ = {p(a),q(a)} tampoco es modelo de Γ, porque si bien p(a) hace verdadera a la primera cláusula de Γ y q(a) hace verdadera a la segunda cláusula, la tercera cláusula es falsa porque seleccionar el átomo q(a) vuelve falsa a la tercera cláusula (que es $\forall(x)\neg q(x)$). Ninguno de los cuatro subconjuntos posibles de $B_H(\Gamma)$ es modelo. Esto quiere decir que no es posible dar con la forma de ningún modelo que pudiera tener Γ. Por lo tanto, podemos afirmar que Γ no tiene modelo y entonces es insatisfactible.

Nota

Sabemos que el conjunto de partes del cualquier conjunto está equipado naturalmente con una estructura de Álgebra de Boole[73]. Como todas las interpretaciones de Herbrand para Γ son el conjunto de partes de $B_H(\Gamma)$, podemos organizarlas en una estructura algebraica llamada reticulado[74], esto es, un orden parcial bajo la relación de inclusión de conjuntos, que cumple ciertas propiedades.

Para el ejemplo que resolvimos arriba, gráficamente tenemos:



Este reticulado es acotado y finito. Pero cuando tenemos infinitas interpretaciones para un Γ entonces el reticulado quedará infinito. Esta estructura de reticulado de todas las interpretaciones de Herbrand para Γ es útil para comprender un aspecto fundamental de la semántica procedural de los programas lógicos[75].

[73] Ver Capítulo 1, *Lattices*, *A Course in Universal Algebra*. S. Burris, H. P. Sankappanavar. Springer, 1981.

[74] Ver Capítulo 1, *Lattices*, especialmente Definiciones 1.1, 1.2, y Ejemplo (1) de *A Course in Universal Algebra*. S. Burris, H. P. Sankappanavar. Springer, 1981.

[75] Ver Semántica de Punto Fijo. *Foundations of Logic Programming*, de W. L. Lloyd.

A continuación, algunas definiciones útiles, y terminología técnica.

Definición. Instancia básica de una cláusula de Γ. Una instancia básica de una cláusula C de Γ es una cláusula que se obtiene de reemplazar uniformemente variables de C por términos del $U_H(\Gamma)$.

Ejemplo. Sea C $\in \Gamma$, C: {p(x),¬q(x)}. Una instancia básica (abreviamos: i.b.) de C es: {p(a),¬q(a)}, con a $\in U_H(\Gamma)$.

Proposición. Un modelo para C satisface todas las instancias básicas de C.

Esto ya lo sabíamos intuitivamente. La proposición se demuestra por aplicación directa de la definición de semántica del cuantificador universal[76]. Recordemos que una cláusula, si tiene variables, las tiene todas cuantificadas universalmente (esto lo aprendimos cuando estudiamos el Algoritmo de Representación Clausal). La semántica del cuantificador universal indica, en otras palabras, lo que indica esta proposición.

Ejemplo. Sea C: {¬q(x),t(x,f(x))}. Esta cláusula es la fbf: ∀(x)(¬q(x) v t(x,f(x))).

Efectivamente, para que la cláusula sea verdadera, deben satisfacerse todas las instancias básicas de la cláusula (como: {¬q(a),t(a,f(a))}, y {¬q(f(a)), t(f(a),f(f(a)))}, y {¬q(f(f(a))), t(f(f(a)),f(f(f(a))))}, y así siguiendo). Esto coincide con la semántica del cuantificador universal.

Definición *a la Herbrand* de la semántica declarativa de un programa lógico

Tengamos en mente que podemos entender a un conjunto de cláusulas Γ como las cláusulas de un programa lógico. Al poder identificar cualquier modelo para un Γ como un subconjunto de la $B_H(\Gamma)$, los $M_H(\Gamma)$ permiten caracterizar la *semántica declarativa* de un programa lógico, esto es, lo que un programa lógico dice, o *declara*.

Veamos la siguiente propiedad.

[76] Ver Definición 3.20 y Proposición 3.27 de *Lógica para Matemáticos*, de A. G. Hamilton.

Propiedad de intersección de modelos[77]

Sea P un programa lógico (por abuso de notación seguidamente reemplazamos a Γ con la letra "P"). Sea {M_{Hi}(P), i ∈ Naturales} un conjunto no vacío de modelos de Herbrand para P. Entonces la intersección de estos modelos de Herbrand: $\bigcap_i M_{Hi}$(P), i ∈ Naturales, es también un modelo de Herbrand y lo llamamos Min_H(P), el mínimo modelo de Herbrand para P.

Ejemplo. Sean las cláusulas de P: {{p(a)}, {s(x),¬q(x)}}. El conjunto de interpretaciones de Herbrand para P es un reticulado organizado bajo la relación de inclusión de conjuntos, completo, finito. Dejamos al lector la construcción del U_H(P), de la B_H(P), del reticulado de interpretaciones de Herbrand para P y la comprobación por definición de Min_H(P) = {p(a)}.

¿Qué tienen de particular los átomos pertenecientes al Min_H(P)[78]?.

Sea A un átomo ground. A ∈ Min_H(P) si y sólo si (por definición de modelo mínimo) A es verdadero en todo modelo de P si y sólo si[79] P ∪ {¬A} no tiene modelo, si y sólo si[80] P ∪ {¬A} es insatisfactible si y solo si A es consecuencia lógica[81] de P (notación: P ⊨ A).

Conclusión[82]

Sea P un conjunto de cláusulas. El Min_H(P) se interpreta naturalmente como el conjunto de consecuencias lógicas (atómicas) de P: Min_H(P) = {A ∈ B_H(Γ) / P ⊨ A}. En otras palabras, lo que un programa lógico *declara* son sus consecuencias lógicas atómicas. *Su semántica declarativa son sus consecuencias lógicas atómicas*.

Nota 1. Las consecuencias lógicas de un programa lógico son siempre *átomos ground* (no aparecen variables).

Nota 2. Si conocemos el Min_H(P) entonces conocemos los átomos de la B_H(Γ) que pueden derivarse de P.

[77] Ver Proposición 6.1 de *Foundations of Logic Programming*, de W. L. Lloyd.
[78] Son átomos ground.
[79] por Proposición 3.3 de *Foundations of Logic Programming*, de W. L. Lloyd.
[80] por Proposición 3.1 de *Foundations of Logic Programming*, de W. L. Lloyd.
[81] Ver Definición 4.2 de *Lógica para Matemáticos*, de A. G. Hamilton.
[82] Ver Teorema 6.2 de *Foundations of Logic Programming*, de W. L. Lloyd.

Nota 3. Con la regla de Resolución podremos saber si un átomo ground es consecuencia lógica de un conjunto de cláusulas.

Nota 4. Con la regla de Resolución no podremos calcular *todas* las consecuencias lógicas de ese conjunto de cláusulas.

En el próximo capítulo veremos como la Teoría de Herbrand que estudiamos constituye la base lógica para la definición de la regla de Resolución para cláusulas de Horn.

Ejercicios para el Capítulo 3

1. Sea Γ = {{r(x),¬(p(x, q(x))}, {s(f(y)), p(y,x)}}.

a. ¿Cuál es el lenguaje de O1 asociado a Γ?

b. Hallar algunos cortes analíticos del $U_H(\Gamma)$.

c. Hallar la $B_H(\Gamma)$.

d. Hallar todas las instancias básicas de la $B_H(\Gamma)$ sobre los cortes $U_{H0}(\Gamma)$ y $U_{H1}(\Gamma)$.

2. Dadas las siguientes cláusulas, escritas en notación de flechas:

Γ= C_1 :le-gusta(Daniel, física) ←

C_2 :le-gusta(Daniel, x) ← le-gusta(x, física).

C_3 : ← le-gusta(x, y), le-gusta(y, y).

a. Hallar $U_H(\Gamma)$ y $B_H(\Gamma)$.

b. Sabiendo que todas las interpretaciones de Herbrand son subconjuntos de la $B_H(\Gamma)$, organizarlas en un reticulado cuya relación de orden es la inclusión de conjuntos.

c. ¿Qué interpretaciones de Herbrand son modelos de C_1 y C_2 ? Justificar.

d. ¿Qué interpretaciones de Herbrand son modelos de C_3? Justificar.

d. ¿Qué interpretaciones de Herbrand son modelos de C_1 , C_2 y C_3 ? Justificar.

3. Sea Γ = {{p(x)}, {¬p(f(y))}}. ¿Es posible encontrar una interpretación que sea modelo de Γ?. Ofrecerla por la afirmativa, fundamentar la negativa.

4. Sea Γ = {{p(a)}, {t(x,f(x)), ¬q(x)}}. Sea I una interpretación para Γ tal que:

D = {1}.

$a \rightarrow \{1\}$, $f(x) = 1$.

$p \rightarrow \{1\}$, $q \rightarrow \{1\}$, $t \rightarrow \{(1, 1)\}$.

Obtener una $I_H(\Gamma)$*, interpretación de Herbrand asociada a I para Γ. ¿La $I_H(\Gamma)$*obtenida es $M_H(\Gamma)$?. Fundamentar.

5. Dado Γ = {{r(x),¬(p(f(x),a)}, {q(y,f(y))}}, y dada una interpretación I tal que:

D = {0, 1}.

a → 1, f(0) = 0, f(1) = 1.

p → {(0, 0)}, r → {1}, q → {(1, 0),(0, 0)}.

Encontrar una interpretación de Herbrand $I_H(\Gamma)$* asociada a I. ¿I es modelo de Γ? Justificar. ¿$I_H(\Gamma)$* es modelo de Γ? Justificar.

6. Dado Γ = {{p(x),¬q(x, g(y)),¬r(f(y))}, {¬p(z),r(z)}, {¬q(v,w)}}:

a. Obtener cortes analíticos del Universo de Herbrand $U_H(\Gamma)$.

b. Expresar por comprensión la $B_H(\Gamma)$.

c. Dada I:

D= {0, 1}.

f(0) = 1, f(1) = 0.

g(0) = g(1) = 0.

p → {0}, q → {(0, 0),(1, 1)}, r → {1}. Obtener una interpretación de Herbrand $I_H(\Gamma)$* asociada a I.

Nota: prestar atención a la alternancia de asignación de valores para f.

7. Sea Γ = {{q(a)}, {p(x),¬q((x))}, {¬p(a)}}. Calcular el $U_H(\Gamma)$ y la $B_H(\Gamma)$. Organizar las $I_H(\Gamma)$ en un reticulado bajo la relación de inclusión. Comprobar que Γ no se satisface.

8. Sea Γ = {{r(g(x))}, {r(x),¬p(x)}}. Calcular el $U_H(\Gamma)$ y la $B_H(\Gamma)$. Organizar las $I_H(\Gamma)$ en un reticulado bajo la relación de inclusión. ¿El átomo r(a) es consecuencia lógica de Γ?. Fundamentar.

9. Proposición: "Dado Γ conjunto de cláusulas de Horn, y dado un átomo A ∈ $B_H(\Gamma)$ tal que A ∈ $M_H(\Gamma)$, donde $M_H(\Gamma)$ es algún modelo de Herbrand para Γ; A es consecuencia lógica de Γ". ¿La proposición enunciada es verdadera o es falsa? Fundamentar, demostrando formalmente la proposición por la afirmativa, o construyendo un contraejemplo por la negativa.

10. La propiedad de intersección de modelos solamente vale para programas *definidos*. Construir un contraejemplo que permita verificar que si las cláusulas no son de Horn, entonces la propiedad de intersección de modelos no se cumple.

11. (de *Foundations of Logic Programming*, de W. L. Lloyd.) Sea P un programa definido. Llamamos $2^{B(P)}$ al reticulado completo de todas las interpretaciones de Herbrand para P bajo la relación de inclusión.

El mapeo T_P: $2^{B(P)} \rightarrow 2^{B(P)}$ se define como sigue. Sea $I_H(\Gamma)$ alguna interpretación de Herbrand para P, entonces:

$T_P(I_H(\Gamma))$ = {A ∈ $B_H(P)$ / A ← A_1, ..., A_n es una instancia ground de una cláusula de P, y A_1, ..., A_n ∈ $I_H(\Gamma)$}.

Para los siguientes programas lógicos, obtener $2^{B(P)}$ y calcular T_P. Mostrar gráficamente $2^{B(P)}$ y T_P.

P_1 : p(x) → q(x).
q(a).

P_2 : p(x) → q(x).
p(x) → s(x).
s(x) → r(x).

P_3 : p(x) → q(x).
q(a).
p(b).

P_4 : p(a).
s(x) → q(x).
q(x).

CAPÍTULO 4
El enfoque orientado a pruebas

Clara Smith

En los capítulos previos hemos estudiado la semántica de un programa lógico desde un enfoque semántico, u *orientado a modelos*. Dicho enfoque constituye la base formal para lo que principalmente nos interesa a nosotros los informáticos: el enfoque *orientado a pruebas* también llamado *orientado a la demostración automática* o enfoque sintáctico. Lo que pretendemos como programadores, esencialmente, es obtener consecuencias lógicas de manera automática.

En este Capítulo 4 nos concentramos en entender la esencia de la ejecución de los programas lógicos. Especialmente nos enfocamos en *aspectos sintácticos* de la ejecución.

Haremos una suposición inicial: que los programas lógicos son una representación adecuada y consistente de *lo que se sabe* de alguna porción del mundo respecto de la cual se quieren obtener conclusiones; esto es, los programas con los que trabajamos y que escribimos *modelan bien*, son un buen modelo del conocimiento asociado al problema. Es una suposición tal vez obvia, pero fundamental. Si el conocimiento asociado al problema no está bien representado, no tenemos garantías de que las conclusiones sean verosímiles.

La Regla de Resolución para cláusulas de Horn

Desde un enfoque orientado a modelos (enfoque semántico) aprendimos que, para saber si una fbf A es consecuencia lógica de un conjunto Γ de fbfs (que constituye la base de conocimiento[83]), es suficiente con encontrar una contradicción entre Γ y la negación de A ($\neg$A).

Desde un enfoque orientado a pruebas (enfoque sintáctico) conocemos las reglas de inferencia Modus Ponens y Resolución, ya las estudiamos para el Cálculo de Enunciados.

Tenemos a Γ, nuestra base de conocimiento, constituida por un conjunto de cláusulas al que a partir de ahora también llamaremos, en ocasiones, "P", especialmente cuando hablemos de programas lógicos. Además tenemos a la cláusula objetivo $\neg$A (escribimos: $\leftarrow$A).

Le preguntaremos al programa P si el objetivo $\leftarrow$A se desprende de él.

[83] En la jerga de la programación lógica la expresión usada es *knowledge base*, y se abrevia por sus siglas KB. Un programa lógico es, de algún modo, una KB sobre alguna porción de conocimiento que se tiene de cierto tema.

Cuestión. Sea el siguiente conjunto de cláusulas: Γ = {{(∀x)p(x)}, {¬p(a)}}[84]. Sólo de contemplar Γ podemos concluir, desde el punto de vista orientado a modelos, que Γ es insatisfactible y por lo tanto inconsistente[85]. Esto lo podemos afirmar porque conocemos la semántica del cuantificador universal[86], vemos rápidamente que la validez de la fbf ∀xp(x) *cae* al necesitar que sea verdadero el literal ¬p(a). Así, vemos que es imposible que las dos fbfs en Γ sean verdaderas (recordemos que cada cláusula en Γ tiene que ser verdadera para que Γ sea verdadero). Por lo tanto, Γ es insatisfactible.

Pasemos ahora al análisis del aspecto orientado a pruebas. El programa lógico con el que estamos tratando es:

p(x).

Y la cláusula objetivo es:

← p(a).

Siguiendo, dado que Γ es insatisfactible (y estamos seguros de ello por haberlo comprobado desde el punto de vista orientado a modelos) podemos intentar aplicar la regla de Resolución en Γ -tal como aprendimos a usarla en el Capítulo 1- para probar instisfactibilidad. Al intentarlo, notaremos que no podemos cancelar *directamente* los literales p(x) y ¬p(a) como si fueran opuestos porque, en términos sintácticos, el átomo p(x) (que es la cláusula de programa que se corresponde con la fbf ∀xp(x)) no es, símbolo a símbolo, "opuesto" al átomo p(a).

La pregunta que nos formulamos entonces es: ¿cómo podríamos estar seguros, en el contexto de la LO1, de que estamos efectivamente manejando átomos sintácticamente opuestos y poder así aplicar Resolución? Veamos a continuación un ejemplo para favorecer nuestra intuición.

Ejemplo

Sea Γ = { {q(x), ¬p(x)}, {¬q(z),s(z)}, {p(y)}, {s(w),¬r(w)}, {¬s(a)}}

si renombramos z como x \ /

{¬p(x), s(x)} /

si renombramos y como x \ /

{s(x)} /

si instanciamos x con a \ /

□

[84] Dicho conjunto de cláusulas es identificable con el programa lógico: p(x) y la cláusula objetivo: ¬p(a).

[85] Ver la demostración de la Proposición 3.1 en *Foundations of Logic Programming*, de W. L. Lloyd.

[86] Ver Proposición 3.27 de *Lógica para Matemáticos*, de A. G. Hamilton.

Analicemos con detalle la cuestión de las variables en el ejemplo previo. A medida que intentamos aplicar, intuitivamente, la regla de Resolución, vamos "renombrando" las variables[87]. Esos "renombres" son, técnicamente, reemplazos (sustituciones) de variables por términos[88]. A esos reemplazos los escribiremos a partir de ahora con la siguiente notación: z/x, y/x ("zeta por equis" e "y por equis", respectivamente).

Nota

En el último paso del ejemplo instanciamos una variable con una constante: "x por a" (escribimos x/a). **Es recién en ese paso que logramos dos átomos sintácticamente opuestos.**

Siguiendo con el análisis del ejemplo previo, notemos que los reemplazos son entre términos que están a un mismo "nivel de generalidad": reemplazamos una variable por otra variable (y/x, leemos: "y por x"), o reemplazamos un término por otro más específico (x/a), es decir, un término que está *más cerca* del nombre de un objeto. Estos reemplazos se hacen para lograr, efectivamente, *átomos sintácticamente iguales*. Y si los átomos son sintácticamente iguales, son fácilmente reconocibles por un algoritmo independientemente de lo que significan.

Avanzando con nuestra intuición, vemos entonces que el modo seguro de poder aplicar[89] la regla de Resolución entre dos cláusulas de Horn es entre átomos sintácticamente iguales y que uno sea la negación del otro. Parecido a como sabemos aplicarla para el Cálculo de Enunciados.

A este mecanismo de reemplazos se lo conoce con el nombre de **Unificación** Fue estudiado por Jacques Herbrand. Luego Alan Robinson retomó el método para enlazarlo con la regla de Resolución y poder automatizar el proceso de detección de contradicciones entre cláusulas.

El mecanismo de Unificación

Necesitamos conocer algunas definiciones técnicas que ayudan a organizar y formalizar el mecanismo de reemplazos de variables por términos que vimos más arriba. Estas definiciones a

---

[87] Como somos informáticos, estamos acostumbrados a "instanciar" variables con datos y con objetos, lo hacemos naturalmente porque ya programamos en diferentes lenguajes de programación. Es conveniente consultar también las Proposiciones 1.10 y 1.14 de *Lógica para Matemáticos*, de A. G. Hamilton; ambas capturan de algún modo el concepto de *symbol pushing* o sustitución en el sentido de que, en una fbf del Cálculo de Enunciados, en el lugar de una subfórmula A puede colocarse otra B reemplazando uniformemente A por B en la fbf. Para el sistema formal del Cálculo de Predicados, ver el axioma (K5) en la sección 4.1 del Capítulo 4 de *Lógica para Matemáticos*, de A. G. Hamilton, que establece que $(\forall x)A(x) \rightarrow A(t)$ si x es una variable del lenguaje y t es un término del lenguaje en el que no aparece x.

[88] Recordemos que las variables son términos según la definición que vimos en el Capítulo 2.

[89] Aplicar *automáticamente*, como queremos hacerlo nosotros, como informáticos, en una computadora.

continuación son útiles para poder entender cómo funcionan, juntos, los mecanismos de Unificación y Resolución. Ambos reglas son la esencia del funcionamiento de Prolog.

Definición. Sustitución

Una sustitución σ es un conjunto de reemplazos simultáneos de variables por términos[90]: $\{x_1/t_1, \ldots, x_n/t_n\}$ con $x_1, \ldots x_n$ variables diferentes entre sí, y con cada término t_i diferente de cada x_i; reemplazos que se realizan sobre una expresión escrita en un lenguaje de O1. Por ejemplo: si tenemos la expresión E: s(z) v ¬q(z), y tenemos la sustitución σ: {z/x} entonces Eσ[91] es: s(x) v ¬q(x). A esta sustitución la llamamos *sustitución pura de variables* (todos los t_i son variables). En otro ejemplo, digamos E: s(z) v ¬q(y) y σ: {z/x, y/a}, conseguimos Eσ: s(x) v ¬q(a).

Motivación desde el punto de vista sintáctico

Queremos usar sustituciones para intentar obtener expresiones de la LO1 *sintácticamente iguales* y poder luego aplicar Resolución que, como hemos aprendido, es una regla *sintáctica*, perfectamente programable.

Certeza desde el punto de vista semántico

La técnica de sustitución nos sugiere sutilmente que una fbf *se desprende* de otra, esto es, dada E y dada σ, tenemos $E \rightarrow E\sigma$. Esta intuición nos hace pensar que la sustitución "funciona bien", en el sentido de que es correcta y que, en principio, podemos usarla sin correr peligro de sacar conclusiones incorrectas[92].

Nota. Aplicación de los reemplazos

Al aplicar una sustitución σ sobre una expresión E, los reemplazos de variables por términos son simultáneos y de izquierda a derecha. Por ejemplo, si E: p(x,y,f(a)) y σ: {x/b, y/x}, conseguimos Eσ: p(b,x,f(a)).

---

[90] Recordemos que los términos son: variables, constantes, y símbolos de función aplicados a términos. Ver el Capítulo 2 de este libro.

[91] Eσ lo leemos "la expresión E bajo la sustitución σ" o simplemente "E bajo σ".

[92] Ver el axioma (K5) en la Sección 4.1 del Capítulo 4 de *Lógica para Matemáticos*, de A. G. Hamilton, que establece que $(\forall x)A(x) \rightarrow A(t)$ si x es una variable del lenguaje y t es un término en el que no aparece x.

Definición. Sustitución ground

Una sustitución σ se dice ground si todos sus términos t_i son ground (no contienen variables). Por ejemplo: si tenemos la expresión E: p(x,y) y tenemos la sustitución σ: {x/a, y/b} entonces Eσ es: p(a,b). En otro ejemplo, si E: q(x, y, z) y σ: {x/a, y/b}, logramos Eσ: q(a,b,z). Observemos en este caso que σ es ground y no obstante ello la expresión resultante no es ground.

Nota

Sea Γ un conjunto de fbfs escritas en un LO1, sea $B_H(\Gamma)$ la base de Herbrand para Γ, y sea A un átomo (alguna subfórmula atómica de alguna cláusula en Γ). Si aplicamos una sustitución σ a A y la expresión resultante Aσ es ground, lo que obtenemos es una "instancia básica" de A. El concepto de instancia básica también aplica a la instancia ground de una cláusula de Γ. La expresión "básica" es por "base" de Herbrand.

Vale el ejemplo de más arriba E: p(x,y), σ: {x/a, y/b}, Eσ: p(a,b).

La llamamos "instancia básica" porque el átomo A, al ser ground (y al ser una subfórmula atómica de una cláusula de Γ) pertenece a la $B_H(\Gamma)$[93].

Definición. Sustitución vacía

También llamada *sustitución identidad* (a izquierda y a derecha), la escribimos ε y opera como sustitución neutra, esto es, para toda expresión E tenemos que Eε = E = εE[94]. Por el momento, podemos pensarla intuitivamente como un recurso de *skip*[95] (una sustitución que no sustituye nada).

Es bueno saber también que existe lo que podríamos denominar un *Cálculo o álgebra de sustituciones*[96] en el que ε trabaja como un *operador identidad* en dicho cálculo. Otras operaciones de este Cálculo aparecen seguidamente.

Definición. Sustitución sobre un conjunto de expresiones

Dado un conjunto de expresiones E_1, …, E_n y dada una sustitución σ, ésta se aplica uniformemente sobre cada una de aquellas, esto es: $(E_1, \ldots, E_n)\sigma = E_1\sigma, \ldots, E_n\sigma$.

---

[93] Ver el Capítulo 3 de este libro.

[94] Ver Proposición 4.1 (a) de la Sección 4 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

[95] Dicho desde una "perspectiva procedural" o de ejecución de programas.

[96] Ver Sección 4 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

Definición. Composición de sustituciones

Dadas dos sustituciones σ: {x_1/t_1, …, x_n/t_n} y θ: {y_1/s_1, …, y_n/s_m} la composición ($\sigma\theta$) es: {(x_1/t_1)θ, …, (x_n/t_n)θ, y_1/s_1, …, y_n/s_m}. Esto es, a cada componente de la primera sustitución le aplicamos la segunda sustitución (en el sentido de la definición de sustitución), y luego continuamos con la segunda sustitución. Por ejemplo, si σ: {x/f(y), y/z} y θ: {x/a, z/b} entonces ($\sigma\theta$): {(x/f(y))θ, (y/z)θ, x/a, z/b} = {x/f(y), (y/z)z/b, x/a, z/b} = {x/f(y), y/b, z/b}.

Nota

Obtendremos el mismo resultado si a una expresión E le aplicamos primero una sustitución σ y luego otra sustitución θ que si a E le aplicamos la composición ($\sigma\theta$), esto es ((Eσ)) = E($\sigma\theta$). Dejamos esta verificación al lector[97].

Definición. Unificador

Una sustitución σ es un unificador de expresiones (E_1, …, E_n) si (E_1, …, E_n)σ es un conjunto unitario. Es decir, luego de aplicar la sustitución sobre las expresiones dadas, nos queda una sola expresión. Ejemplo: (p(x), p(y)){y/x} = p(x), con E_1:p(x), E_2:p(y), y con σ: {y/x} unificador.

Definición. Unificador más general

Un unificador θ de un conjunto de expresiones (E_1, …, E_n) se dice unificador más general (abreviamos u.m.g.) si cualquier otro unificador σ de dicho conjunto de expresiones puede construirse como ese unificador θ compuesto con alguna otra sustitución γ: $\sigma = \theta\gamma$. Por ejemplo, el conjunto de expresiones: (p(f(x),z), p(y,a)) es unificable con σ: {x/a, y/f(a), z/a}. Observemos que σ puede componerse como ($\theta\gamma$), con θ: {y/f(x), z/a}, y γ: {x/a}. θ es *más general* porque unifica, digamos, *más lentamente*, tiene menos términos ground que, por ejemplo, σ.

Observación

Unificación busca conseguir una sintaxis idéntica en las expresiones que recibe. Al intentar unificar, intuitivamente podemos pensar en un algoritmo que coloca al inicio de cada expresión

[97] Ver Proposición 4.1 (b) de la Sección 4 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

un *puntero*; estos punteros se van moviendo (*parseando*) las expresiones[98] de izquierda a derecha hasta encontrar símbolos diferentes. Si las subexpresiones que allí comienzan (allí donde están detenidos los punteros) pueden unificarse con una sustitución, entonces sustituimos y continuamos moviendo los punteros desde aquel lugar donde se sustituyó y hacia el final de las expresiones.

Nota

Un conjunto de expresiones puede no ser unificable. Por ejemplo, para las expresiones: p(f(x),a), p(z,f(w)) no hay unificador. Dejamos esta verificación al lector.

A continuación vemos el algoritmo de Unificación.

Algoritmo de Unificación[99].

Input: un conjunto finito de expresiones (E_1, …, E_n).

Output: el u.m.g. de las expresiones (E_1, …, E_n), o un mensaje de que el conjunto de expresiones no es unificable.

El siguiente es un pseudocódigo para el mecanismo de Unificación:

Paso 1. k:=0, σ_0:=ε, umg:=void, D_k:=void;
Paso 2. if (E_1, …, E_n)σ_k es unitario then umg:= σ_k
else begin
D_k:= conjunto de subexpresiones no coincidentes sintácticamente de (E_1, …, E_n)σ_k
/ • construimos el "conjunto de discordia" • /
if (D_k contiene una variable v y un término t tal que v no aparece en t) then
begin
σ_{k+1}:= {v/t};
k:= k +1;
volver al Paso 2
end
else return:(E_1, …, E_n) no es unificable
end;

[98] *To parse* (traducido a la jerga informática como: "parsear") es una expresión técnica proveniente de la Teoría de Compiladores que implica recorrer una expresión y descomponerla en subexpresiones para identificar y analizar dichos componentes.
[99] Ver Sección 4 del Capítulo 1 de *Foundations of Logic Programming*, de W. L. Lloyd.

Nota

La restricción de que v no ocurra en t es fácil de intuir: si v aparece en t puede generarse un loop infinito de intentos de sustitución por ejemplo de la forma x/t(x) y entonces el algoritmo no terminaría nunca.

Ejemplo de aplicación del algoritmo

Dadas las dos expresiones: E_1: p(f(a),g(x)) y E_2: p(y,y) determinemos si son o no unificables. Paso 1: k:= 0, σ_0 := ε, umg:= void. Paso 2: construimos el conjunto D_0 := {f(a), y} y controlamos que haya en D_0 una variable y un término tal que la variable no aparezca en el término. Esto sucede, porque la variable y no aparece en el término f(a), entonces construimos σ_1 := {y/f(a)}, aumentamos k:= k +1 y volvemos al Paso 2. En este punto aplicamos σ_1 y logramos las dos expresiones p(f(a),g(x)) y E_2: p(f(a),f(a)). *Movemos los punteros imaginarios* por ambas expresiones, hacia la derecha, y encontramos -construimos- el próximo conjunto de discordia D_1 = {g(x), f(a)} que contiene dos términos, con lo que no podemos seguir aplicando el algoritmo. El conjunto original no es unificable.

Ejercicio. Aplicar el algoritmo para comprobar que las expresiones E_1: p(a,x,h(g(z))) y E_2: p(z,h(y),h(y)) son unificables y un u.m.g. es: {z/a, x/h(g(a)), y/g(a))}.

Ahora estamos en condiciones de ver cómo usar Unificación combinada con Resolución.

Unificación y Resolución

Sean C_1 y C_2 cláusulas de Horn, sea el átomo A cabeza de cláusula de C_1, y sea el átomo C cabeza de cláusula de C_2, sea σ un u.m.g. entre el átomo A y el átomo A_k. Tenemos:

$C_1 : A \leftarrow B_1, \ldots B_n$

$\underline{C_2 : C \leftarrow A_1, \ldots, A_k, \ldots, A_m}$

$R: (C \leftarrow A_1, \ldots, A_{k-1}, B_1, \ldots B_n, A_{k+1}, \ldots, A_m)\sigma$

con R la resolvente generada entre C_1 y C_2 bajo el u.m.g. σ.

La secuencia de aplicación de las técnicas es: Unificación primero y luego Resolución.

Primero aplicamos Unificación para encontrar dos átomos sintácticamente iguales: $A\sigma$ y $A_k\sigma$. Observemos que, no obstante ser sintácticamente iguales, $A\sigma$ es “positivo” (es cabeza de cláusula), y A_k es “negativo” (integra el cuerpo de otra cláusula).

Luego aplicamos Resolución como sabemos hacerlo: eliminando esos dos átomos "opuestos".

Ejemplo de aplicación de Unificación seguido de Resolución

Sea el conjunto de cláusulas Γ: {{q(x), ¬p(x)}, {s(z), ¬q(z)}, {p(y)}, {s(w), ¬r(w)}, {¬s(a)}}.

Este conjunto de cláusulas se corresponde con un programa lógico más un objetivo (la última cláusula). Escribimos a continuación el programa lógico P y el objetivo en notación de flechas, notación que además es visualmente muy descriptiva:

q(x) ← p(x).

s(z) ← q(z).

p(y).

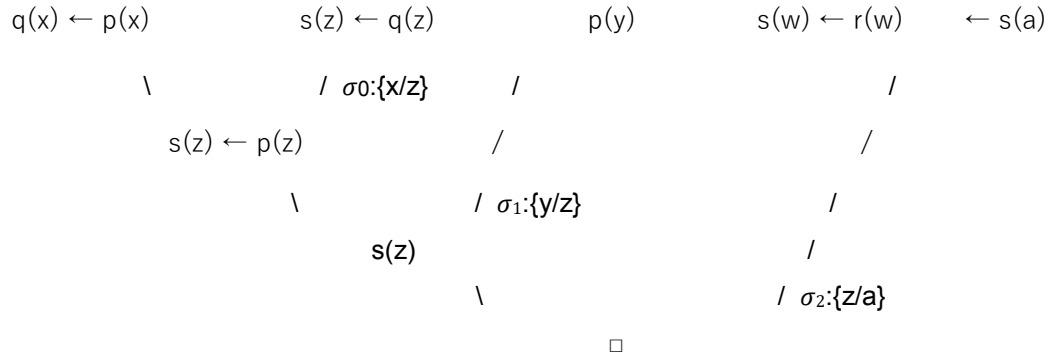
s(w) ← r(w).

← s(a).

así escritas, estas cláusulas se corresponden con un programa lógico, excepto la última cláusula que es la cláusula objetivo a responder (el objetivo no integra el programa).

Aplicaremos ahora Unificación y Resolución. Lo haremos eligiendo de modo directo y a propósito pares de cláusulas, según nos convengan, para poder llegar en el menor número posible de pasos[100] a "□". Elegir las cláusulas a propósito es, en este momento, útil *a los efectos de nuestra presentación*. Pero, como informáticos, debemos saber que el lenguaje de programación lógica que usemos tendrá codificada (*a bajo nivel*) una estrategia puntual de elección de cláusulas para aplicar Resolución, como veremos más adelante. Dicha estrategia de elección debe ser correcta en el sentido de asegurar que si □ existe entonces la encontraremos.

Apliquemos seguidamente Unificación seguida de Resolución en el conjunto de cláusulas de nuestro ejemplo:



[100] Recordemos que como informáticos siempre queremos minimizar los costos de computación.

Vemos que le preguntamos al programa lógico P por el objetivo ← s(a). Por aplicación repetida de Unificación y Resolución llegamos a □. Por lo tanto, el átomo s(a) se desprende de P. Es posible deducir s(a) del programa P.

Notemos que la cláusula s(w) ← r(w) no fue usada. Esto nos pasa siempre cuando pensamos, para sacar conclusiones no usamos todo el conocimiento del que disponemos.

Espacio de resolventes

Organicemos lo visto hasta aquí en este Capítulo.

Tenemos cierto conocimiento que está escrito en cláusulas de un programa lógico. Tenemos además una cláusula objetivo que queremos determinar si se desprende de ese conocimiento, y tenemos un mecanismo de razonamiento subyacente (Unificación más Resolución). Con estos tres elementos -programa lógico, cláusula objetivo y motores de pensamiento- se abre frente a nosotros un *espacio de resolventes posibles*: se nos presenta el conjunto de *todas las resolventes que podemos conseguir*. Este conjunto está formado por: las cláusulas originales del programa lógico, la cláusula objetivo, y las nuevas resolventes que generamos a medida que aplicamos Unificación y Resolución.

Nota

Debe quedar claro que este espacio de resolventes puede ser infinito. Esto puede traernos problemas obvios, si sucede que nos perdemos buscando en dicho espacio. Estamos interesados en encontrar □ lo más rápidamente posible, si es que □ está en ese espacio. Hallar □ detendrá el proceso de búsqueda, como es de suponer, pues estamos buscando exactamente □, es decir, una contradicción.

La pregunta que surge naturalmente para nosotros, que somos informáticos y que estamos interesados en la tarea de automatizar la tarea de probar que una conclusión se desprende de cierto conocimiento, es: ¿cómo podemos tratar ese espacio (ese conjunto potencialmente infinito) de posibles resolventes? ¿Cómo podemos llegar lo más rápidamente posible a □ si es que □ existe dentro de ese espacio? ¿Cómo podemos *navegar* en el espacio de resolventes y asegurarnos de no quedarnos en loop buscando □ para siempre?

Y también: ¿En qué orden tomamos las cláusulas para aplicarles Resolución? Hay muchas maneras de elegir las cláusulas para abrirse camino en el espacio de resolventes e intentar llegar a □ lo más rápidamente posible. Esto significa, para nosotros, en el **menor número de pasos**. Hay distintas estrategias[101], que describimos seguidamente.

[101] Ver Capítulo 5 de *Principles of Artificial Intelligence*. Nils J. Nilsson, Springer-Verlag, 1982.

Resolución por saturación o estrategia "a lo ancho" (BFS[102])

Consiste en la aplicación "ciega", lisa y llana, de la regla de Resolución sobre las cláusulas del programa lógico, más la cláusula objetivo, más las resolventes nuevas que se van generando. Es un método tedioso y costoso aunque seguro, ya que si □ está en el espacio de resolventes entonces la encontraremos porque agotamos todas las aplicaciones posibles de Resolución[103]. Esta estrategia ciega subyace a la presentación de la regla de Resolución para el Cálculo de Enunciados del Capítulo 1.

Resolución con filtrado de tautologías

El fundamento de esta estrategia es el siguiente: como estamos buscando contradicciones, dado que □ representa una contradicción, es prudente pensar que una cláusula que contiene una tautología no nos conducirá a □[104]. Entonces procedemos a eliminarla del conjunto de cláusulas sin temor de perder ninguna posibilidad real de llegar a □. Al eliminar la cláusula, trabajamos con menos cláusulas, que siempre nos interesa. El filtrado de tautologías es un filtrado costoso computacionalmente pues debemos no sólo filtrar las cláusulas del conjunto inicial que contengan tautologías sino que, cada vez que generamos una resolvente, debemos controlar si contiene una tautología.

Ejemplo: supongamos que nuestro conjunto Γ es: {{p(x)}, {¬p(x), p(x)}, {¬p(x)}}[105]. Supongamos que resolvemos la primera cláusula con la segunda: obtenemos la resolvente p(x). Seguido, resolvemos la segunda con la tercera cláusula, obteniendo la resolvente ¬p(x). Notemos que las dos nuevas resolventes en realidad ya las teníamos en el Γ inicial. Observemos que la segunda cláusula, que es una tautología, es inconducente: a través de ella en realidad logramos resolventes que ya existían. En el ejemplo llegamos a □ a partir de Resolución entre la primera y la tercera cláusula (sin usar la segunda cláusula).

Resolución con filtrado de literales puros

En la misma línea que el filtrado anterior, esta técnica o *heurística* elimina aquellas cláusulas que son inconducentes para encontrar □. Llamamos "literal puro" a un literal[106] que aparece en

[102] Breadth First Search o *estrategia a lo ancho*.

[103] Notemos que, en términos teóricos, sin pensar en un algoritmo, aunque el espacio de resolventes sea infinito, si existe en el espacio la encontraremos.

[104] Las tautologías son completamente inútiles en cualquier descripción particular de una porción del mundo o modelización de algún problema específico a resolver pues, por definición, son siempre ciertas en cualquier contexto, tiempo o lugar.

[105] Las dos primeras cláusulas son el programa lógico, la tercera es la cláusula objetivo.

[106] Recordemos que un literal es un átomo o la negación de un átomo.

una cláusula pero cuyo literal opuesto no aparece en ninguna otra cláusula. Es claro que nunca usaremos una cláusula con un literal puro porque no habría otra cláusula con un literal opuesto con el que resolver. Entonces la filtramos, es decir, la eliminamos del conjunto de cláusulas a ser usado. Esta depuración del conjunto de cláusulas no es tan costosa como el filtrado de tautologías ya que se hace una única vez al comienzo (nunca Resolución generará cláusulas con letras de predicado que no aparecen en el conjunto original de cláusulas.)

Ejemplo: si Γ es: {{p(x)}, {¬q(x), r(x)}, {¬p(x), q(x)}, {t(x),¬r(x)}, {¬r(x)}} vemos que la cláusula {t(x),¬r(x)} tiene un literal puro: t(x). Entonces filtramos Γ por única vez y antes de comenzar a aplicar Resolución. Generamos Γ'= Γ - {t(x),¬r(x)} y resolvemos sobre Γ'.

Resolución con preferencia de cláusula unitaria

Con esta heurística privilegiamos, en lo posible, la selección de al menos una cláusula que contenga un solo literal. Naturalmente, si usamos una cláusula con un solo literal tardaremos menos pasos en nuestro camino a □. Esto es evidente pues hay menos literales para consumir cuando las cláusulas son más "breves" (contienen menos literales). Esta heurística es útil no solo porque menos literales pueden ayudar a disminuir la cantidad de pasos hacia □ sino también porque, intuitivamente, lo que estamos haciendo es seleccionar aquellas cláusulas que representan conocimiento con menos restricciones a ser cumplidas. Recordemos que en una cláusula los literales están unidos por disyunciones, entonces entre elegir una cláusula que establece una sola condición y elegir otra cláusula que establece que debe cumplirse una condición u otra, es más conveniente elegir la cláusula que pide menos restricciones.

Así, por ejemplo, si Γ es: {{¬p(x)}, {¬p(x), q(x)}, {p(x),¬r(x)}, {r(x)}} elegimos la primera cláusula (antes que tomar la segunda) para resolver con la tercera cláusula[107].

Analizando otras variantes, podemos establecer heurísticas que privilegien no solo la elección de cláusulas con un literal, sino con dos literales, tres, y así siguiendo.

Método del conjunto soporte

Teniendo presente que estamos buscando □, que es una contradicción, notemos lo siguiente: si en el conjunto Γ de cláusulas hay algún subconjunto consistente, llamémoslo C, entonces podemos afirmar que □ no está en C ni podrá generarse a partir de C[108], por ser □ una

[107] Cuando hay conocimiento representado como en estas cláusulas debemos pensar que la modelización no está del todo bien hecha. Es claro que no es nuestra responsabilidad, pues no somos los ingenieros de software que modelamos el mundo. Pero poder identificar esta deficiencia en la modelización puede ser un punto a favor al momento de depurar las cláusulas (es decir los programas, o los sistemas). Podemos elegir quedarnos con la primera del ejemplo, que es la que tiene menos restricciones. Y eliminar la segunda cláusula.

[108] Un conjunto Γde fbfs es consistente si a partir de él no pueden deducirse a la vez las fbfs A y ¬A. Sabemos que si se pueden deducir las fbfs A y ¬A entonces se puede deducir la contradicción (A ∧ ¬A) y si un conjunto contiene una

contradicción. La estrategia del conjunto soporte consiste en *nunca aplicar Resolución entre dos cláusulas que pertenecen a un subconjunto consistente*. Se aplica Resolución usando al menos una cláusula del subconjunto que llamaremos *conjunto soporte* S y definimos como Γ-C. Esto es, Resolución se aplica entre: o bien dos cláusulas de S, o bien entre una cláusula perteneciente a S y otra cláusula del subconjunto consistente C. Las nuevas resolventes que se van generando ingresan a S.

Por ejemplo, si Γ es: {{p(a)}, {¬r(y), q(a,y)}, {¬p(x),¬t(w), ¬q(x,w)}, {r(b)}, {t(z)}} podemos organizar las cláusulas en los conjuntos S = {¬p(x),¬t(w), ¬q(x,w)} = Γ-C, con C = {{p(a)}, {¬r(y), q(a,y)}, {r(b)}, {t(z)}}. Dejamos para la verificación del lector que C es consistente. Verifique también el lector que Γ es inconsistente, aplicando la técnica del conjunto soporte.

Nota

La complicación puede radicar en la identificación del subconjunto consistente C dentro del original Γ. Intencionalmente, además, podemos querer maximizar el número de cláusulas en C para así minimizar las aplicaciones de Resolución ya que cuantas más cláusulas contenga C menos posibles aplicaciones de Resolución hay, dadas las restricciones de la estrategia. Verifique el lector que C es el máximo conjunto consistente[109] en el Γ del ejemplo.

Técnica de subsunción

Es una técnica de simplificación que consiste en eliminar todas aquellas cláusulas más específicas que otras que ya pertenecen al espacio de resolventes. Esto ayuda a mantener el conocimiento lo más general[110] posible. Por definición, una cláusula C_1 subsume (*cubre*) a otra cláusula C_2 si tenemos una sustitución σ tal que $C_1\sigma \subseteq C_2$. Así, C_1 *es más general que* C_2, y por este motivo privilegiamos a C_1 y descartamos a C_2 por ser más específica.

Por ejemplo, sea C_1: {p(a), r(y)} y sea C_2: {p(a), r(z), ¬q(z)}. Vemos que $C_1\sigma \subseteq C_2$ con σ:{z/y}. Conservamos C_1 y eliminamos C_2 porque C_2 tiene más restricciones que cumplir (ya que debe ser verdadero el átomo ¬q(z)).

---

contradicción entonces el conjunto es inconsistente. Ver Proposiciones 2.16 y 4.42 de *Lógica para Matemáticos*, de A. G. Hamilton.

[109] Esto es, si quisiéramos agregar una cláusula de Γ-C en C, C se vuelve inconsistente.

[110] En el sentido de neutro, o limpio de especificidades. Recordemos que las "instanciaciones" más específicas las logramos con las sustituciones, especialmente con las de la forma *variable por término* (especialmente *ground*).

Nota

Controlar subsunción entre cláusulas es costoso computacionalmente, pues para cada nueva resolvente debemos determinar si queda subsumida por, o si subsume a, otra cláusula.

Resolución Lineal

Su definición es similar a la que dimos al comienzo de este capítulo. Sea C una cláusula de programa y sea O una cláusula objetivo. Se genera una cláusula objetivo O' de la siguiente manera:

C: $A \leftarrow B_1, \ldots B_n$

O: $\leftarrow A_1, \ldots, A_s, \ldots, A_m$

O': $(\leftarrow A_1, \ldots, A_{s-1}, B_1, \ldots B_n, A_{s+1}, \ldots, A_m)\sigma$

con A cabeza de cláusula, A_s átomo seleccionado para unificar con A (seleccionado del cuerpo del objetivo O) y σ u.m.g. entre A y A_s.

SLD-Refutación

Prolog implementa resolución lineal de un modo especial, es una variante conocida como SLD-Refutación[111] (SLD por *Selection Linear Definite*). La presentación de esta variante es ligeramente diferente a la definición de Resolución Lineal, veamos:

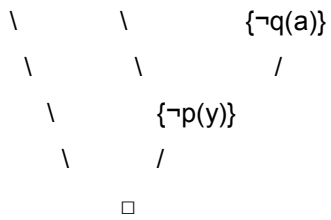
Definición

Una SLD-Refutación para un programa P (cuyas cláusulas son $C_1, \ldots, C_n$) y un objetivo O es: **una secuencia finita de objetivos** $O=O_1, O_2, \ldots, O_m=\square$ y una **secuencia finita de unificadores más generales** $\sigma_1, \sigma_2, \ldots, \sigma_m$ tal que O_{i+1} es una resolvente entre O_i y C_{i+1} con σ_{i+1} u.m.g. entre ellas.

Ejemplo de SLD-Refutación. Dado el siguiente conjunto de cláusulas, aplicamos SLD-Refutación y obtenemos:

{{p(z)}, {¬p(y), q(a)}, {¬q(x), r(a)}, {¬r(x)}}
\ \ \ /

[111] Ver Sección 7 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.



Con $O=O_1= \neg r(x)$, $O_2 = \neg q(a)$, $O_3 = \neg p(y)$, $O_4=\square$, y con $\sigma_1= \{x/a\}$, $\sigma_2=\boldsymbol{\varepsilon}$, $\sigma_1= \{y/z\}$.

Nota. Combinación de estrategias

Podemos decir que Prolog implementa SLD-refutación y que también usa la estrategia de conjunto soporte si asumimos la consistencia del conjunto de cláusulas de programa y consideramos como conjunto soporte inicial S a la cláusula objetivo (los subobjetivos que se van generando ingresan al conjunto soporte S).

Observación

En SLD-Refutación, el átomo que se elige (del objetivo) para intentar unificar con alguna cabeza de cláusula (elección que se llama *regla de computación*[112]) es el de "más a la izquierda". Este criterio de elección del átomo es, en principio, irrelevante, porque todos los átomos del cuerpo de una cláusula deben ser consumidos[113]. Pero al momento de implementar computacionalmente una regla de selección de átomos, debemos, necesariamente, *decidir el orden en el cual tomarlos*. Prolog implementa una regla de computación que toma átomos de izquierda a derecha. Así, un sistema Prolog combina: cláusulas de Horn, SLD-refutación como estrategia de Resolución, y regla de computación de izquierda a derecha.

Correspondencia entre semántica declarativa y procedural

Hemos estudiado mecanismos para poder probar automáticamente consecuencias lógicas a partir de un conjunto de cláusulas. Desde el punto de vista orientado a modelos vimos en el Capítulo 3 que dado un programa P, el $Min_H(P)$ es la semántica declarativa de P. Desde el punto de vista orientado a pruebas, podemos pensar que la contraparte del $Min_H(P)$ es coincidente,

[112] Ver Sección 9 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.

[113] Las estrategias de selección de átomos del cuerpo de una cláusula son irrelevantes desde el punto de vista teórico. Ver Sección 9 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.

esto es, que podemos probar que cada átomo del $Min_H(P)$ es consecuencia lógica de P. Esto es así por definición teórica y tenemos dos teoremas importantes que así lo afirman.

El **Teorema de Clark de la Correctitud de SLD-Refutación**[114] que dice que:

Dado un programa lógico P y un objetivo O
toda respuesta computada con SLD-Refutación para P ∪ {O} es una respuesta correcta.

Y el Teorema de Hill de la Completitud de SLD-Refutación[115], que dice que:

Dado un programa lógico P y un objetivo O,
si P ∪ {O} es insatisfactible entonces existe una SLD-refutación para P ∪ {O}.

Ambos teoremas conectan las semánticas declarativa y procedural de un programa lógico. El Teorema de Clark dice que las respuestas que computamos son verdaderas (correctas), es decir, relaciona el resultado sintáctico con el resultado semántico. El Teorema de Hill vincula *en el otro sentido*, asegurando que la insatisfactibilidad (concepto semántico) se corresponde con la existencia de una SLD-Refutación exitosa (resultado sintáctico).

Semánticas declarativa y procedural en Prolog

Sin embargo, aun a pesar de los resultados teóricos de Hill y Clark, Prolog es un lenguaje de programación. Y la teoría y la práctica no son siempre coincidentes en el campo de la informática. En Prolog es muy fácil encontrar ejemplos de programas cuyas semánticas declarativa y procedural no coinciden.

Veamos: el criterio de selección de cláusulas para intentar aplicar Resolución, en Prolog, es trivial: es el orden de escritura de las cláusulas. Tal el orden en el que las escribamos van a ser elegidas por el sistema para intentar resolver (primero la primera, luego la segunda, y así siguiendo). Por este motivo, si por ejemplo escribimos un programa que incluye la definición de un predicado recursivo, y la cláusula con ese llamado recursivo la ubicamos antes de la cláusula que tiene el caso base, los riesgos de que el programa no termine por entrar en loop son altos.

Contemplemos el siguiente programa:

```
ancestro(x,y) :- ancestro(x,z), progenitor(z,y).
ancestro(x,y) :- progenitor(x,y).
progenitor(Juana,Pedro).
```

[114] Ver Teorema 7.1 de la Sección 7 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.
[115] Ver Teorema 8.4 de la Sección 8 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.

```
progenitor(Pedro,María).
```

Análisis

El programa, en sus dos primeras cláusulas, define el predicado "ser ancestro de". Las siguientes dos cláusulas son dos hechos: se sabe que el progenitor de Juana es Pedro y que el progenitor de Pedro es María.

Notemos que "progenitor" está definido por extensión, mientras que "ser ancestro de" está definido, digamos, "*por comprensión*", su definición está dada de modo "más abstracto".

Supongamos que queremos averiguar si María es ancestro de Juana. El objetivo entonces es: ←`ancestro(Juana,María`. Prolog, que usa SLD-Refutación y selecciona las cláusulas según el orden en que están escritas, unifica ese objetivo con la primera cabeza: `ancestro(x,y)`. Se genera entonces un nuevo subobjetivo: ←`ancestro(Juana,z)`, `progenitor(z,María)`. Seguidamente, como la regla de selección de átomos de Prolog selecciona el átomo de más a la izquierda, el sistema intenta resolver el átomo seleccionado: `ancestro(Juana,z)` y elige para unificar nuevamente la cabeza de la primera cláusula, generando: ← ancestro(Juana,u), progenitor(u,z), repitiendo este patrón de selección de modo tal que el programa entra en un loop infinito, tomando siempre la primera cláusula, que además tiene el llamado recursivo en el primer átomo del cuerpo. Graficamos a continuación, a modo de rama de árbol de búsqueda[116], el armado de los infinitos subobjetivos:

```
                              ← ancestro(Juana,María)
                              /   {x/Juana, y/María}
← ancestro(Juana,z), progenitor(z,María)
        / {z/u}
← ancestro(Juana,u), progenitor(u,z)
                        …
```

Nota

En el gráfico de rama de árbol, z y u son *variables locales*; a medida que avanzan los llamados recursivos se generan nuevas variables locales que nunca se instancian ni unifican.

[116] Ver el ejemplo dado en la Sección 10 del Capítulo 2 de *Foundations of Logic Programming*, de W. L. Lloyd.

Observación

Es fácil ver de las cláusulas tercera y cuarta del programa dado más arriba que María es efectivamente un ancestro de Juana. **Pero tal como está escrito el programa no podremos demostrarlo automáticamente en Prolog.**

Síntesis

Vimos que hay dos resultados teóricos, el Teorema de Hill y el Teorema de Clark, que conectan, dado un programa lógico P, su semántica declarativa (el $Min_H(P)$) con su semántica procedural (la existencia de una SLD-Refutación para un programa P y un objetivo O). Pero al llevar estos resultados al plano de la programación debemos tomar decisiones referidas a la implementación concreta de los lenguajes (por ejemplo, cómo Prolog selecciona las cláusulas o los átomos en el cuerpo de una cláusula), y es en este punto en el que la semántica declarativa y la procedural pueden diferir.

Ejercicios para el Capítulo 4

1- Demostrar que $C \models C\sigma$, siendo C cualquier cláusula escrita en un LO1 y σ cualquier sustitución de la forma $\{x_1/t_1, ..., x_n/t_n\}$, con x_i variable, t_j término; $x_i \neq x_j$, $x_i \neq t_i$.

2- Mostrar con un contraejemplo que la propiedad de composición de sustituciones no es conmutativa.

3- Una sustitución σ es idempotente si $\sigma = \sigma\sigma$. Sea $\sigma = \{x_1/t_1,..., x_n/t_n\}$, supongamos que V es el conjunto de variables que aparecen en los términos $t_1,..., t_n$. Demostrar que σ es idempotente si y sólo si $\{x_1,..., x_n\} \cap V = \emptyset$.

4- Determinar si cada uno de los siguientes conjuntos de expresiones son unificables, dando el u.m.g. cuando corresponda:

a- {q(a), q(b)}.
b- {p(x), q(x)}.
c- {p(f(a),y), p(y,a)}.
d- {r(a,x,f(x)), r(a,y,y)}.
e- {r(x,y,z), r(u,h(v,v),u)}.
f- {s(g(b),x), s(x,b)}.
g- {t(h(x,x),a),t(h(y,h(y,a)),a)}.
h- {p(a,x), p(x,b)}.

5- Para el siguiente: Γ = {{r(x),¬q(y)}, {t(y)}, {p(a,y), ¬r(x)}, {¬p(x,y)}, {r(x),¬t(x)}, {s(x)}}.

Resolver aplicando resolución por saturación (estrategia "a lo ancho" o BFS), mostrando por niveles la traza de resoluciones: $Niv_0 = \Gamma$, Niv_1 = las resolventes resultantes de la aplicación de resolución entre las cláusulas de Niv_0, Niv_{k+1} = las resolventes resultantes de la aplicación de resolución entre cláusulas de $Nivel_k$ con otras en nivel previo. Si Γ es satisfactible sigue hasta que no se pueden generar más nuevas resolventes o se halla la cláusula vacía (*enfoque orientado a pruebas*).

6- Demostrar que el método de Resolución por saturación combinado con la estrategia de filtrado de literales puros es un refinamiento del método de resolución por saturación. Un método, digamos: M' es refinamiento de otro método M cuando el conjunto, espacio o universo de soluciones admisibles según M' está incluido en el conjunto de soluciones admisibles según M.

7- Demostrar que el método del conjunto soporte es un refinamiento del método de Resolución por saturación.

8- Para el conjunto de cláusulas del ejercicio 5, aplicar Resolución con filtrado de literales puros.

9- Dado el siguiente conjunto de cláusulas, probar insatisfactibilidad usando la heurística de *preferencia de cláusula unitaria*, organizando lo más claramente posible las resolventes en los correspondientes niveles de la traza de resolución. Γ = {{p(x), ¬s(x), h(x)}, {¬r(y), s(y)}, {r(a)}, {¬p(a)}, {¬h(z), e(z)}, {¬e(w)}}.

10- Determinar satisfactibilidad o insatisfactibilidad de Γ usando Resolución con saturación con filtrado de tautologías: Γ = { {¬p(x), ¬q(a)} {p(y), q(y)}, {¬p(z), q(z)}, {p(w), ¬q(w)} }.

11- Dado el conjunto de fbfs:

$\exists x ((p(x) \wedge (\forall y\ d(y) \rightarrow r(x,y))), \neg(\exists x\ \exists y\ (p(x) \wedge t(y) \wedge r(x,y))), \exists x\ (d(x)) \wedge (\forall y\ t(y))$

Transformar en notación clausal y resolver usando la estrategia de conjunto soporte.

12- Sea Γ = {{q(a)}, {¬r(y), p(a, y)}, {¬q(x), ¬t(w), ¬p(x, w)}, {r(b)}, {t(z)}}. Probar insatisfactibilidad usando la estrategia de conjunto soporte.

13- El método de Resolución con estrategia basada en el conjunto inicial genera cada resolvente a partir de una cláusula del conjunto inicial. Verificar, construyendo un contraejemplo que este método es incompleto para cláusulas generales (Γ es insatisfactible y no se llega a vacía).

14- Sean C y D dos cláusulas cualesquiera. Si C implica lógicamente a D, ¿entonces C subsume a D?

15- En cada caso, determinar si C subsume a D, si D subsume a C, o si no hay subsunción:

i- C: {r(x,z), p(z)} D: {r(a, b), q(c), p(b)}.

ii- C: {q(x,y), ¬r(y, w)} D: {q(x, d), r(a,b)}

iii- C: {p(a,b), ¬q(a)} D: {¬q(x), p(a,b)}

iv- C: {p(f(x)), q(a)} D: {p(x), q(b)}

v- C: {p(a), ¬r(x)} D: {p(x), ¬r(z), c(b)}.

16- En cada inciso a continuación, ¿qué cláusula subsume a qué otra? Fundamentar.

a) C_1: { p(x), p(f(x))} C_2: {p(f(y)), p(f(f(y))), r(y)} C_3: {p(a), p(f(f(a)))} C_4: {p(f(y)), p(y), p(f(z))}.

b) C_1: {p(x_1),¬p(x_2),¬p(f(x_1,x_2))} C_2:{¬p(f(f(f(a,f(b, c)),c),f(b,a)))}.

C_3: {p(f(x_1,f(f(x_2,f(x_3,x_1)),f(x_3,x_2)))}.

17- Probar que el siguiente conjunto es insatisfactible usando subsunción.

Γ = {{¬t(x), ¬s(a)}, {t(y), s(y)}, {¬t(z), s(z)}, {t(w), ¬s(w)}}.

18- Se dice que C_i subsume a C_j cuando $C_i\sigma \subseteq C_j$. El problema de determinar subsunción es NP. Obtener una reducción a FNC en la Lógica de Enunciados (tal como estudiamos en el Capítulo 1) que se corresponda con el problema de determinar subsunción (¿qué cláusula subsume a qué otra?) en el conjunto de cláusulas siguiente:

C_1: {p(a), p(f(a))}.

C_2: {p(x), p(f(x))}.

C_3: {p(f(y)), p(y)}.

C_4: {p(a), p(f(a)), r(a)}.

Epílogo

En este texto hemos estudiado algunos de los conceptos indispensables para entender los principios generales que sustentan a la programación declarativa, y especialmente comprender cómo funciona un lenguaje básico de la programación lógica, Prolog.

Comprendimos por qué el conocimiento se modela con un subconjunto seleccionado de fórmulas de un lenguaje de O1, las *cláusulas de Horn*, y no con fórmulas cualesquiera. Relacionamos este subconjunto particular de fórmulas con la necesidad de que los programas lógicos, para nosotros como informáticos, finalicen. Vinculado con este punto, aprendimos también que, por ser informáticos, nos importan los dominios finitos y discretos, aunque sabemos que los estudiosos de las matemáticas trabajan cómodos con dominios infinitos.

Estudiamos la semántica declarativa y la semántica procedural de los programas que podemos escribir en Prolog. Concluimos que un programa lógico escrito en Prolog *significa* su mínimo modelo de Herbrand. Aprendimos también que podemos *demostrar automáticamente* átomos de la Base de Herbrand. Comprobamos que las semánticas declarativa y procedural de un programa lógico definido pueden no coincidir, aunque sabemos que existen resultados formales que garantizan que *sí coinciden*. Comprendimos así que, vinculado a esta eventual discordancia entre las semánticas, al momento de implementar (de programar, codificar las bases de) un lenguaje lógico hay que tomar decisiones *de bajo nivel* que pueden comprometer resultados teóricos.

Remisión. En este punto estamos en condiciones de abordar cualquier texto o tutorial que nos *enseñe a programar* en Prolog, aprender técnicas y estilos de programación de Prolog (cómo programar con iteración, con recursión, con negaciones, etcétera), técnicas que si bien están fuera del alcance de este texto inicial, resultarán naturales y de fácil comprensión pues ahora conocemos el sustrato teórico de la programación lógica. El texto *Programming in Prolog Using the ISO Standards*, de William F. Clocksin y Christopher S. Mellish (quinta edición, Springer, 2003) es excelente para adentrarse en modernas técnicas de programación de Prolog, interesantes ejemplos de programas, tratamiento de predicados built-in del lenguaje, y diseño de proyectos en Prolog. Para los aspectos formales más precisos, el lector necesariamente debe enfrentarse a *Foundations of Logic Programming*, de W. L. Lloyd (Springer, reimpresión de la segunda edición original de 1987, 2011). Para aquellos lectores interesados en el estudio de modelización de agentes inteligentes, recomendamos avanzar con la lectura de los Capítulos 1 y 2 de *Modal Logic*, de P. Blackburn, M. de Rijke y Y. Venema (Cambridge University Press, 2001) teniendo presente que a la lógica modal se la puede pensar como una extensión de la Lógica Proposicional, lo que hará fluida la lectura.

La autora

Clara Smith nació el 01/03/1969 en La Plata, Argentina. Ejerce actualmente la profesión como abogada independiente, es asesora legal del Programa Conectar Igualdad del Ministerio de Educación de la Nación, y también es profesora de Programación Lógica en la Facultad de Informática de la Universidad Nacional de La Plata (UNLP), Argentina, luego de obtener su doctorado en la Facultad de Ciencias Exactas de la misma universidad, obtener el título de Docente Universitaria Especializada, ser becaria del CONICET, trabajar para la Suprema Corte de Justicia de la Provincia de Buenos Aires, y obtener una beca de investigación del Ministero degli Affari Esteri (MAE, Italia) para trabajar en la Universidad de Bologna. Ocupó una posición de profesora invitada en la Université des Sciences Sociales en Toulouse (Francia, 2012), una posición de staff en la Università di Bologna (2014), y se desempeñó como fellow Marie Curie en Bologna, en el proyecto MIREL de la Unión Europea (2016-2019).

Ha dirigido proyectos de investigación en UNLP y en la Universidad Católica de La Plata (UCALP), casa de estudios en la que también fue Directora de la Carrera de Sistemas y para la que diseñó las carreras de Licenciatura en Sistemas y el Profesorado en Informática, ambas de la Facultad de Ciencias Exactas e Ingeniería (2010-2015).

Sus principales temas de investigación son la Lógica Computacional, la Filosofía del Derecho y el Diseño de Sistemas Multi-agentes. Fue editora invitada de la revista CMOT de la editorial Springer (Computational and Mathematical Organization Theory, número especial de sistemas multi-agente). Fue co-editora de los anales de los simposios SNAMAS@AISB/IACAP2012 y SNAMAS@AISB2011 (Simposios de Redes Sociales y Sistemas Multi-agente).

Sus publicaciones más relevantes son: Temporal Reasoning and MAS (EUI Working Papers LAW No. 2010/16, con A. Rotolo y G. Sartor; Collective Trust and Normative Agents (Oxford University Press, 2010, doi:10.1093/jigpal/jzp076) con A. Rotolo; y Lógica para Informática, con R. Rosenfeld y C. Pons (http://sedici.unlp.edu.ar/handle/10915/61426), 2017.

Libros de Cátedra

Smith, Clara P.
La lógica como lenguaje de programación : aspectos declarativos y procedurales / Clara P. Smith. - 1a ed - La Plata : Universidad Nacional de La Plata ; EDULP, 2023.
Libro digital, PDF - (Libros de cátedra)

Archivo Digital: descarga
ISBN 978-950-34-2340-0

1. Inteligencia Artificial. 2. Lenguajes de Programación. I. Título.
CDD 005.4

Diseño de tapa: Dirección de Comunicación Visual de la UNLP

Universidad Nacional de La Plata – Editorial de la Universidad de La Plata
48 N.º 551-599 / La Plata B1900AMX / Buenos Aires, Argentina
+54 221 644 7150
edulp.editorial@gmail.com
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias Nacionales (REUN)

Primera edición, 2023
ISBN 978-950-34-2340-0

e exactas

Edulp EDITORIAL DE LA UNLP

UNIVERSIDAD NACIONAL DE LA PLATA