UNIVERSIDAD DE SEVILLA

Facultad de Matemáticas
Grado en Matemáticas
Departamento de Ciencias de la Computación e Inteligencia Artificial

TRABAJO FIN DE GRADO

Fundamentos lógicos de la Inteligencia Artificial: Razonamiento no monótono

Realizado por: Arnaud Darche

Sevilla, septiembre de 2019

Agradecer a Andrés por su empeño en el trabajo aún cuando las circunstancias no acompañaban.

Índice

Resumen

La lógica tradicional resulta muy útil para formalizar razonamientos matemáticos: teoremas, definiciones... en los que se establecen reglas que se cumplen de forma atemporal y para todos los elementos de los que se habla. Por ejemplo, al decir que todas las curvas derivables son continuas, esta afirmación se aplica a todas las curvas bien definidas y derivables; además, es cierta en todo momento, no hay posibilidad de que deje de serlo en el futuro. Sin embargo, cuando queremos formalizar razonamientos más prácticos e "inteligentes" debemos considerar la posibilidad de que las conclusiones que saquemos no se apliquen a todos los elementos del grupo del que hablemos. Por ejemplo, al decir que "todos los pájaros vuelan", sabemos que existen excepciones, como los pingüinos. Además, debemos considerar la opción de que un pájaro que hoy puede volar podría dejar de ser capaz de volar en el futuro si, por ejemplo, pierde un ala. Para razonamientos así la lógica tradicional no es del todo adecuada y es por ello que surge la **lógica no monótona**.

La diferencia principal entre la lógica tradicional y la lógica no monótona es que en la lógica no monótona surge la posibilidad de "retractarse" en el futuro, es decir, algo que consideramos cierto con la información que tenemos podría dejar de serlo en el futuro cuando ampliemos esta información.

En este trabajo veremos varios mecanismos de lógica no monótona, entre los que destacan la hipótesis de mundo cerrado, completación de predicados y la semántica de los modelos estables para los programas lógicos; sus utilidades frente a la lógica tradicional y, finalmente, un sistema que nos permite programar con lógica no monótona: *Clingo*, con el que seremos capaces de resolver problemas aparentemente complejos de una forma bastante simple.

Summary

Traditional logic is very useful for formalizing mathematical reasoning: theorems, definitions ... in which rules are established that are met in a timeless manner and for all the elements that are spoken of, for example, by saying that all derivable curves are continuous, this statement applies to all well-defined and derivable curves, in addition, it is true at all times, there is no possibility of it ceasing to be so in the future. However, when we want to formalize more practical and 'intelligent' reasoning, we must consider the possibility that the conclusions we draw may not apply to all the elements of the group we are talking about, for example, by saying that 'all birds fly,' we know that there are exceptions, such as penguins, in addition, we must consider the option that a bird that can fly today could no longer be able to fly in the future if, for example, it loses a wing. For reasoning like this, traditional logic remains short and that is why the **non-monotonous logic** arises.

The main difference between traditional and non-monotonous logic is that in non-monotonous logic the possibility of 'retracting' in the future arises, that is, something we consider true with the information we have could cease to be in the future when we expand this information.

In this work we will see several mechanisms of non-monotonous logic among which the closed world assumption, completion of predicates and stable models semantics for logic programs stand out, their utilities compared to traditional logic and, finally, a system that allows us to program with non-monotonous logic: Clingo, with which we will be able to solve seemingly complex problems in a fairly simple way.

Capítulo 1

Introducción

El razonamiento no monótono dentro de la aproximación lógica a la Inteligencia Artificial engloba una familia de formalismos creados para representar *conclusiones revisables*, esto es, conclusiones que un agente razonador acepta en un cierto momento como verdaderas pero de las que puede retractarse en el futuro en vista de nueva información que nos haga dudar de su validez. Existen numerosos ejemplos de esto, desde razonamientos básicos como el ensayo-error hasta opiniones de expertos como una diagnosis médica.

El *razonamiento revisable*, al igual que el razonamiento deductivo, puede tener procedimientos bastante complejos y es necesario desarrollar formalismos que aclaren este tipo de razonamiento no monótono. El objetivo del presente trabajo es presentar algunos de esos formalismos para el estudio matemático de las propiedades del razonamiento revisable.

1.1. Razonamiento revisable

Un ejemplo típico de razonamiento revisable es sacar conclusiones basadas en suposiciones por "normalidad" o "generalidad". Por ejemplo, podemos concluir que Piolín vuela basándonos en los hechos de que Piolín es un pájaro, y de que, en general, los pájaros vuelan. Tendríamos, sin embargo, que retractarnos de esa conclusión si aprendemos más tarde que Piolón es aún un bebé pájaro.

Otro ejemplo es concluir que ha llovido al ver las calles mojadas. Sin embargo, al aprender que ese día se han limpiado las calles y ver que los tejados están secos, querremos retirar esa conclusión de nuestra base de conocimiento.

Estas deducciones violan el principio de monotonía del que disfruta la lógica clásica. Este principio dice que dado un conjunto de fórmulas Σ, si

tenemos otro conjunto ampliado Σ', $\Sigma \subseteq \Sigma'$, se tiene

$$\Sigma \models \phi \Rightarrow \Sigma' \models \phi$$

con $\models$ la relación de consecuencia lógica correspondiente. Es decir, que si hacemos una deducción con la información que tenemos, esa información seguirá siendo cierta sin importar la información nueva que podamos tener en el futuro. Es una forma de razonar que se muestra útil en ramas como las matemáticas donde solo hacemos deducciones de las que estemos completamente seguros y que son, en general, reglas universales (Todas las funciones derivables son continuas, toda función continua en un compacto tiene un mínimo,...), pero que, sin embargo, se muestra poco útil en razonamientos del día a día, más basados en el sentido común.

La mayoría de razonamientos revisables están sujetos a conocimientos externos y por tanto violan el principio de monotonía, ya que las consecuencias que se saquen podrían no persistir al obtenerse nueva información. Veremos numerosos ejemplos a lo largo del presente trabajo.

1.2. Creando lógica no monótona

En la lógica tradicional, partiendo de un conjunto Δ de fórmulas que representa una base de conocimiento, se dice que una fórmula α se deduce lógicamente de Δ si α es cierto en *todos* los modelos de Δ.

Cuando nuestro objetivo es formalizar razonamiento no monótono, debemos cambiar este enfoque clásico. Lo primero es que, en lógica no monótona, a la información con la que tratamos Δ la llamaremos un conjunto de creencias, pues somos conscientes de que puede no recopilar toda la información existente sobre el objeto de estudio, y que en un futuro Δ se puede ampliar o modificar. Pero, sobre todo y de manera más importante, modificaremos de una forma u otra la definición clásica de consecuencia lógica arriba descrita. En el presente trabajo, estudiaremos dos maneras bien conocidas de modificar dicha noción clásica de consecuencia lógica para hacer emerger formalismos no monótonos.

La primera de ellas consiste en considerar que una fórmula α se deduce de un conjunto de creencias Δ ya no si α es cierta en todos los modelos de Δ, sino que bastará con que α sea el caso para cierto tipo especial o *preferido* de modelos de Δ. Estos modelos *preferidos* suelen ser modelos minimales que intentan capturar cierta intuición del razonamiento basado en el sentido común. Veremos esto con más detalle cuando estudiemos los modelos mínimos, los modelos estables y los conjuntos de respuesta de los

programas lógicos normales y extendidos en la segunda parte del presente trabajo.

La segunda forma de deducción no monótona que estudiaremos en este trabajo es mediante un mecanismo de *expansión* del conjunto de creencias. Dado Δ un conjunto de creencias, mediante un mecanismo de expansión que intenta capturar cierta intuición de razonamiento basado en sentido común, hallaremos una extensión de Δ: la denotamos en $ext(\Delta)$. A continuación, consideraremos cierto todo aquello que podamos deducir de $\Delta \cup ext(\Delta)$.

Estos mecanismos son no monótonos ya que, en general, al añadirle una información X al conjunto de creencias se tendrá

$$\Delta \cup ext(\Delta) \not\subseteq \Delta \cup X \cup ext(\Delta \cup X)$$

Ejemplos que estudiaremos en este trabajo son: la hipótesis de mundo cerrado y la completación de predicados, hablamos de ellos más a fondo a continuación.

1.3. Mecanismos clásicos de lógica no monótona

Vamos a presentar a continuación algunos de los mecanismos más clásicos de la lógica no monótona, que se estudiarán a fondo más adelante en el trabajo (capítulo 3).

1.3.1. Hipótesis de Mundo Cerrado

La **hipótesis de mundo cerrado** es la formalización de un mecanismo habitual en el estudio de las bases de datos en el que suponemos que idealmente tenemos información completa sobre qué hechos básicos son ciertos en nuestra base de conocimiento. En líneas muy generales, este mecanismo de expansión consiste en la siguiente suposición: Si no nos consta que un cierto hecho básico es verdadero de acuerdo a la base de conocimiento, añadiremos a nuestro conocimiento automáticamente que dicho hecho es falso y emplearemos esta nueva información para obtener nuevas consecuencias.

Veamos un ejemplo típico. Supongamos un cliente que, tras mirar la carta de un restaurante, observa que en ella no se ofrece salmorejo. El cliente podrá entonces deducir que, en dicho restaurante, no sirven salmorejo. El cliente no tiene constancia de que en ese restaurante en específico no se cocine salmorejo, sino que esta deducción parte de la suposición de que en la carta aparecen

todos los platos que ofrece el restaurante (información completa en la base de datos).

Esta hipótesis sirve para reducir el tamaño de la base de datos, pues, de esta manera, el restaurante no necesita especificar en la carta que no sirven salmorejo, sino que se entiende que cualquier plato que no aparezca en la carta es un plato que no se cocina en el restaurante.

Este es un caso de razonamiento no monótona, pues, al aumentar la información se podría cambiar la deducción; por ejemplo, si el camarero cita el salmorejo entre los platos del día.

1.3.2. Herencias y taxonomías

Cuando tenemos una base de datos clasificada en clases con subgrupos específicos tendemos a suponer que dichos subgrupos heredan las propiedades de una clase. Por ejemplo, los vencejos vuelan puesto que son pájaros, y, en general, los *pájaros vuelan*. Sin embargo, se pueden dar excepciones; por ejemplo, las avestruces son un tipo de pájaro que no vuelan. Se dice que las *avestruces* forman un subgrupo de los pájaros con abnormalidad respecto a la propiedad "volar". Podemos imaginar un caso de avestruces que sí vuelen, por ejemplo, si lleva un jetpack; llamemos a esta clase las *avestruces voladores*. Decimos entonces que las avestruces voladores son un subgrupo de las avestruces con abnormalidad respecto a la propiedad " no-volar". Representamos la clasificación en el siguiente diagrama:

$$\text{Avestruz volador} \rightarrow \text{Avestruz} \rightarrow \text{Pájaro}$$

Así, si tenemos un avestruz con un jetpack, tenemos tres informaciones que nos llegan: que vuela por ser un avestruz volador, que no vuela por ser un avestruz, y que vuela por ser un pájaro. Obviamente, la información con la que nos quedaremos es con que vuela, puesto que es un avestruz volador. De manera general, se aplica este procedimiento al consultar una base de datos catalogada, teniendo en cuenta las deducciones asociadas al subgrupo más específico que encontremos, puesto que presenta información más precisa y concreta sobre el objeto de estudio que las clases a las que pertenece.

Ahora bien, cualquiera de estas deducciones basadas en reglas con excepciones pueden presentar anomalías que nos obligarían a retirar alguna de nuestras conclusiones. De ahí su carácter no monótono.

En el presente trabajo, estudiaremos con detalle este mecanismo en el contexto de la completación de predicados (capítulo 3).

1.3.3. Razonamiento por defecto

Uno de los mecanismos más importantes de la lógica no monótona es el **razonamiento por defecto**. Éste consiste en hacer una deducción de la forma "si tenemos a y no nos consta que tengamos b, entonces tenemos c". Se representa mediante lo que llamamos *reglas* que tienen la siguiente forma:

$$c \leftarrow a_1, a_2, ..., a_n, \text{not } b_1, \text{not } b_2, ..., \text{not } b_m$$

Donde $c, a_1, ..., b_m$ son literales de la lógica de primer orden.

Nótese que estas reglas usan la lógica no monótona en el sentido de que, si usando estas reglas deducimos c y, en algún momento posterior deducimos también b_m, perderíamos la deducción y ya no podríamos afirmar c. (Esto puede generar ciclos, si hemos usado c para deducir b_m; veremos más adelante en el trabajo restricciones que nos permitan asegurarnos de que no va a pasar este efecto indeseable).

Usando la negación por defecto podemos trabajar con los mecanismos vistos anteriormente:

Para la hipótesis de mundo cerrado, dada una propiedad $p(X)$ podemos escribir la regla

$$\neg p(X) \leftarrow not\ p(X)$$

que dice que, salvo que se tenga constancia de que X satisface la propiedad p, podemos suponer que no la satisface. Para las herencias y taxonomías podemos escribir, por ejemplo, las reglas siguientes:

$$\begin{aligned}
vuela(X) &\leftarrow pajaro(X), \text{not } raro1(X)\\
\neg vuela(X) &\leftarrow avestruz(X), \text{not } raro2(X)\\
vuela(X) &\leftarrow avestruzVolador(X), \text{not } raro3(X)\\
raro1(X) &\leftarrow avestruz(X)\\
raro2(X) &\leftarrow avestruzVolador(X)
\end{aligned}$$

Donde se explica que los pájaros vuelan salvo que sean abnormales, que los avestruces no vuelan salvo que tengan una abnormalidad, y que las avestruces voladoras vuelan salvo que sean abnormales. Además de especificar que los pájaros y las avestruces son abnormales respecto de su clase.

Notar que el hecho de que el avestruz sea abnormal no implica que no vuele, sino que no podemos afirmar que vuela, en este caso si que sabemos que, en general, vuelan, gracias a la información que nos aporta la segunda regla.

Hablaremos más a fondo de la negación por defecto a partir del capítulo 5 "la negación en los programas lógicos". Dedicaremos bastante espacio a

este tipo de reglas dada su importancia, tanto teórica como práctica, en el estudio del razonamiento no monótono. De hecho, en el último capítulo del trabajo estudiaremos una implementación informática de este tipo de ideas (el sistema CLINGO dentro del paradigma de la programación con conjuntos de respuestas).

1.4. Contenidos del trabajo

En líneas generales, en este trabajo pretendemos explicar la lógica no monótona, su relación con la lógica de primer orden, y el uso que tiene a la hora de resolver diversos problemas. Aparte de este primer capítulo introductorio, este trabajo consta de seis capítulos más.

En el capítulo 2 **Conocimiento declarativo** damos una breve descripción de los elementos más básicos de la lógica de primer orden y de cómo podemos usar la lógica de primer orden como un lenguaje de especificación para formalizar el conocimiento declarativo. Estudiamos además un puñado de ejemplos clásicos: el mundo de las cajas, especificación de un circuito booleano y descripción de una base de datos familiar.

En el capítulo 3 **Razonamiento no monótono** estudiamos los primeros mecanismos de razonamiento no monótono del trabajo: la Hipótesis de Mundo Cerrado y la Completación de Predicados. En este último caso, estudiamos con cierto detalle cómo usar el formalismo para especificar mecanismos de herencias y taxonomías jerárquicas.

A partir del capítulo 4, cambiamos el enfoque y nos restringimos al lenguaje de la Programación Lógica, donde una base de conocimiento se reescribe siempre en forma de cláusulas (reglas) de primer orden.

En el capítulo 4 **Programas lógicos positivos** estudiamos los programas lógicos más fundamentales, los cuales gozan de una semántica clara con muy buenas propiedades matemáticas. Incluimos demostraciones de las propiedad básica de estos programas: poseen siempre un modelo canónico, el modelo de Herbrand mínimo. Ahora bien, es importante destacar que los programas lógicos positivos NO dan lugar a razonamiento no monótono. Es necesario ampliarlos para romper la monotonía (como haremos en los capítulos sucesivos).

El capítulo 5 **Negación en los programas lógicos** introduce formalmente la negación por defecto. Veremos condiciones suficientes para estar seguros de que un programa no entra en bucles y hablaremos de los modelos de estos programas y, en concreto, de los *modelos estables*, que son aquellos

que buscaremos en general y uno de los conceptos fundamentales del trabajo. Tanto incluiremos propiedades teóricas de los programas lógicos con negación por defecto como ilustraremos los principales conceptos con un buen número de ejemplos.

En el capítulo 6 **Extensiones de los programas lógicos normales** extenderemos los programas lógicos mediante la introducción de *restricciones*, *disyunciones* y *negaciones fuertes*. Adaptaremos también los principales resultados y definiciones del capítulo 5 a este contexto más general.

Finalmente, en el capítulo 7 **El paradigma de la programación con conjuntos de respuesta ASP** veremos una implementación informática del modelo matemático estudiado en el capítulo anterior. Veremos asimismo cómo resolver usando este paradigma de programación una serie de problemas esencialmente combinatorios, que irán desde resolución de puzles y rompecabezas (sudoku, buscaminas, n-damas) hasta problemas de grafos (3-coloraciones, caminos eulerianos) y problemas sabor matemático (construcción de un grupo no abeliano, cálculo de los números de Shur).

Capítulo 2

Conocimiento declarativo

El comportamiento inteligente depende en gran medida del conocimiento que tiene el sujeto sobre su entorno. El objetivo de este capítulo es ser capaces de expresar ese conocimiento de forma "declarativa".

2.1. Sintaxis

2.1.1. Conceptualización

Para empezar a formalizar el conocimiento de forma declarativa necesitamos una *conceptualización*. En líneas generales, la conceptualización de un dominio de estudio consiste en fijar los *objetos* del entorno que estudiaremos y las *relaciones* existentes entre ellos que serán de nuestro interés.

La definición de objeto en este caso es bastante general. Un objeto puede ser algo concreto (por ejemplo, un libro o una persona) o abstracto (por ejemplo, el número 2 o un concepto). Llamaremos pues objeto a cualquier entidad del dominio de estudio (concreta o abstracta) sobre la que queramos decir algo. Llamaremos *Universo de trabajo* al conjunto de todos aquellos objetos del dominio que queramos estudiar.

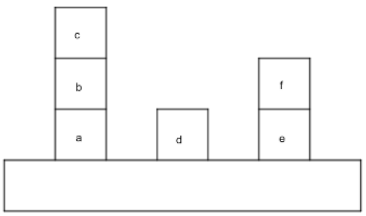

Figura 2.1: *Mundo de las cajas*

Ejemplo 2.1.1 (El mundo de las cajas)**.** Veamos, como un primer ejemplo de conceptualización, el *mundo de las cajas* descrito en la Figura 2.1. En este caso, el universo de trabajo estará compuesto por cada una de las cajas que se muestran en la figura (podría considerarse también como objeto adicional del universo la tabla sobre la que descansan las cajas, pero, por simplicidad, no lo haremos este primer ejemplo). Obtenemos pues el siguiente universo de trabajo:

$$\{\text{a,b,c,d,e,f}\}$$

Una vez fijado el universo de trabajo, el siguiente paso de la conceptualización consistirá en decidir qué relaciones entre los objetos del universo serán de nuestro interés.

Un primer tipo de interrelaciones entre los objetos del universo de trabajo que usaremos habitualmente son las *funciones*. Desde el punto de vista combinatorio, son muchas las funciones (cada una con su correspondiente aridad, esto es, su correspondiente número de argumentos de entrada) que se pueden definir entre los objetos del universo. Sin embargo, típicamente elegiremos un puñado de entre estas funciones que sean de interés por su significación para el dominio que pretendemos estudiar (*funciones base* de la conceptualización) e ignoraremos el resto.

Por ejemplo, en el ejemplo del mundo de las cajas anterior, una función natural que nos podría interesar destacar en la conceptualización es la función 1-aria *sombrero* que relaciona cada caja con la que tiene encima (en caso de tenerla). Las duplas correspondientes a esta función serían:

$$\{\langle \text{a,b}\rangle, \langle \text{b,c}\rangle, \langle \text{e,f}\rangle\}$$

Nótese que la función *sombrero* arriba descrita es una función parcial. Las cajas c, d y f, que se encuentran en lo alto de cada una de las tres pilas de cajas de la Figura 2.1, no tienen imagen definida por la función *sombrero*. Aunque en la descripción informal de una conceptualización tiene perfecto sentido considerar funciones parciales, es importante destacar que cuando pasemos a la contrapartida formal de la conceptualización y utilicemos para ello la lógica de primer orden como lenguaje formal de especificación, todas las funciones consideradas han de ser, por definición, funciones *totales*. Esto no debe suponer un problema. Bastará con extender la definición de la función parcial, teniendo en cuenta que los nuevos valores no serán relevantes para las propiedades de la conceptualización. Por ejemplo, una elección natural para hacer total la función *sombrero* anterior es asignar a cada caja que no tiene ninguna caja encima ella misma como imagen por la función. Las duplas correspondientes a esta nueva función total serían pues:

$$\{\langle \text{a,b}\rangle, \langle \text{b,c}\rangle, \langle \text{e,f}\rangle, \langle \text{c,c}\rangle, \langle \text{d,d}\rangle, \langle \text{f,f}\rangle\}$$

Una función que, a priori, no interesaría considerar en el conjunto de funciones base es la función *siguiente*, la cual asigna a cada caja su caja siguiente en el orden alfabético usual (por ejemplo, *siguiente*(d) = e). Obsérvese que esta función no tiene ningún significado geométrico y no parece aportar información relevante para el mundo de las cajas.

Otro tipo fundamental de interrelaciones entre objetos del universo son las *relaciones*, que nos permiten expresar tanto propiedades de algunos elementos (relaciones de aridad 1) como relacionar dos o más elementos entre ellos (relaciones de aridad n con $n > 1$). Como en el caso de las funciones, elegiremos una serie de relaciones de interés que constituirán las *relaciones base* de la conceptualización.

En este caso, por ejemplo, nos puede interesar la relación binaria *encima* que relaciona dos elementos si el primero está encima del segundo (con o sin cajas de por medio), y la relación binaria *sobre* que relaciona dos objetos si el primero está directamente sobre el segundo. Las duplas asociadas a ambas relaciones son, respectivamente:

$$\{\langle \text{b,a} \rangle, \langle \text{c,b} \rangle, \langle \text{c,a} \rangle, \langle \text{f,e} \rangle\}$$

$$\{\langle \text{b,a} \rangle, \langle \text{c,b} \rangle, \langle \text{f,e} \rangle\}$$

También nos interesan las relaciones 1-arias *despejado* y *tabla*. Un elemento satisface la relación *despejado* si no tiene ninguna caja encima; mientras que un elemento satisface la relación *tabla* si está sobre la tabla. Dichas relaciones corresponden a los siguientes conjuntos, respectivamente:

$$\{\text{c,d,f}\}$$

$$\{\text{a,d,e}\}$$

Formalmente, definimos una *conceptualización* como una terna que incluye el universo de trabajo, un conjunto de funciones y un conjunto de relaciones. Por ejemplo, en este caso nuestra conceptualización sería:

$$\langle \{\text{a,b,c,d,e,f}\}, \{sombrero\}, \{sobre, encima\, despejado, tabla\} \rangle$$

2.1.2. El lenguaje de la lógica de primer orden

Una vez fijada la conceptualización del dominio que se pretende estudiar, podemos empezar a formalizar el conocimiento que tenemos mediante frases en un lenguaje adecuado: el lenguaje de la *Lógica de Primer Orden*, también conocida como *Cálculo de predicados*.

El lenguaje de la lógica de primer orden consta dos tipos de símbolos: los símbolos lógicos, que son independientes del lenguaje y comunes a todos los lenguajes que formemos, y los símbolos propios, que usaremos para definir los elementos concretos del dominio de estudio en cuestión. Más concretamente:

Símbolos lógicos:

- Variables: $x, y, z, x_1, y_1, ...$ Intuitivamente, designarán elementos genéricos del universo de trabajo.
- Conectivas proposicionales: $\neg, \vee, \wedge, \Rightarrow, \Leftarrow, \Leftrightarrow$.
- Cuantificadores: $\exists, \forall$.
- Símbolo de igualdad: $=$.
- Símbolos auxiliares: (,)

Símbolos propios:

- Símbolos de constantes: $a, b, c, juan, lluvia, ...$ Intuitivamente, designarán elementos concretos del universo de trabajo.
- Símbolos de predicado (o relación): $p, q, r, hermanos, ...$ Intuitivamente, designarán relaciones entre los elementos del universo. Cada símbolo de predicado tiene asociada su aridad (un número natural $n \geq 1$ que indica el número de argumentos que relaciona). A veces, se reserva el término predicado para nombrar las relaciones de aridad 1. Nosotros emplearemos indistintamente ambos términos.
- Símbolos de función: $f, s, t, log, edad, ...$ Intuitivamente, designarán funciones totales definidas sobre el universo de trabajo. Cada símbolo de función tiene asociada su aridad (un número natural $n \geq 1$ que indica el número de argumentos de entrada de la función).

Un lenguaje de primer orden L vendrá especificado por tres conjuntos que describen sus símbolos propios: el conjunto de los símbolos de constante de L, el conjunto de los símbolos de predicado de L y el conjunto de los símbolos de función de L. Cualquiera de esos tres conjuntos puede ser vacío.

Definición 2.1.1 (Términos)**.** Se definen los términos de un lenguaje L de la siguiente forma:

- Un símbolo de constante es un término.

- Una variable es un término.
- Si $\tau_1, .., \tau_n$ son términos y f es un símbolo de función con aridad n, entonces $f(\tau_1, ..., \tau_n)$ es un término.

Notación. Para los operadores matemáticos usuales, tales como la suma o el producto, usaremos notación infija y escribiremos $\tau_1 + \tau_2$ para representar $+(\tau_1, \tau_2)$.

Para expresar hechos en un lenguaje L, usaremos las fórmulas del lenguaje, que pasamos a definir a continuación:

Definición 2.1.2 (Fórmulas atómicas)**.** Una *fórmula atómica* o *átomo* de un lenguaje L se compone de un símbolo de predicado p de aridad n y de n términos $\tau_1, .., \tau_n$. Se escribirá

$$p(\tau_1, ..., \tau_n)$$

También incluimos en las fórmulas atómicas la composición del símbolo de igualdad $=$ con dos términos τ_1, τ_2: $\tau_1 = \tau_2$.

Definición 2.1.3 (Fórmulas)**.** Definimos las *fórmulas* de un lenguaje L como sigue:

- Las fórmulas atómicas de L son fórmulas.
- Si F es una fórmula, también lo es $\neg F$.
- Si F y G son fórmulas, también lo son: $(F \vee G)$, $(F \wedge G)$, $(F \Rightarrow G)$, $(F \Leftarrow G)$, $(F \Leftrightarrow G)$.
- Si F es una fórmula y x es una variable, $\exists x\, F$ y $\forall x\, F$ también son fórmulas.

Criterios de reducción de paréntesis.-
El orden de prioridad de los operadores es el siguiente (de más a menos prioritario): $\forall, \exists, \neg, \wedge, \vee, \Rightarrow, \Leftarrow, \Leftrightarrow$.
Se omitirán los paréntesis más externos.
Cuando una conectiva se use varias veces, se asociará por la derecha:

$$\tau_1 \vee \tau_2 \vee \tau_3 \text{ es una abreviatura de} (\tau_1 \vee (\tau_2 \vee \tau_3)).$$
$$\tau_1 \wedge \tau_2 \wedge \tau_3 \text{ es una abreviatura de} (\tau_1 \wedge (\tau_2 \wedge \tau_3)).$$

Definición 2.1.4 (Variables libres y variables ligadas)**.** Una aparición (u ocurrencia) de una variable x en una fórmula F se dirá *ligada* si dicha aparición ocurre bajo el alcance de un cuantificador de la forma $\exists x$ o $\forall x$. En caso contrario, dicha aparición de x se dirá *libre* en la fórmula F. Una variable x se dirá libre en F si tiene, al menos, una aparición libre.

Definición 2.1.5 (Fórmulas cerradas)**.** Diremos que una fórmula F de un lenguaje L es *cerrada* si F no contiene ninguna variable libre.

En líneas generales, fijada una conceptualización, usaremos un conjunto de fórmulas cerradas del lenguaje L asociado para expresar formalmente propiedades del dominio de estudio. Será importante pues también definir formalmente cuándo una fórmula cerrada es verdadera en un determinado dominio.

2.2. Semántica

En esta sección vamos a establecer la relación entre el lenguaje formal que hemos introducido y los elementos de la conceptualización que pretendemos estudiar.

Definición 2.2.1 (Estructuras)**.** Una estructura del lenguaje L es un par (Ω, I) tal que:

- Ω es un conjunto no vacío cualquiera, denominado el *universo* o el *dominio* de la estructura;
- la interpretación I es una función con dominio el conjunto de los símbolos propios de L tal que:
  - si σ es una constante, entonces $\sigma^I \in \Omega$.
  - si π es una símbolo de función con aridad n, entonces $\pi^I : \Omega^n \to \Omega$.
  - si ρ es una símbolo de predicado con aridad n, entonces $\rho^I \subseteq \Omega^n$.

Nota: En la definición anterior hemos escrito σ^I, π^I y ρ^I en lugar de $I(\sigma)$, $I(\pi)$ e $I(\rho)$, respectivamente.

Ejemplo 2.2.1. Consideramos de nuevo el ejemplo del mundo de las cajas que vimos en el apartado de conceptualización. Partimos de un lenguaje L con los símbolos de constante a, b, c, d, e y f, el símbolo de función *sombrero/1* y los símbolos de predicado *sobre/2,encima/2,tabla/1* y *despejado/1.* (Los sufijos $/n$ anteriores indican la aridad de los símbolos de función o predicado el lenguaje). La siguiente L-estructura corresponde a la interpretación esperada de estos símbolos:

- Dominio: $\Omega = \{\text{a,b,c,d,e,f}\}$.
- Interpretación: I dada por

$$a^I = \text{a}$$
$$b^I = \text{b}$$
$$c^I = \text{c}$$
$$d^I = \text{d}$$
$$e^I = \text{e}$$
$$f^I = \text{f}$$
$$\textit{sombrero}^I = \{\langle\text{b,a}\rangle, \langle\text{c,b}\rangle, \langle\text{f,e}\rangle\langle\text{a,a}\rangle, \langle\text{e,e}\rangle, \langle\text{d,d}\rangle\}$$
$$\textit{sobre}^I = \{\langle\text{a,b}\rangle, \langle\text{b,c}\rangle, \langle\text{e,f}\rangle\}$$
$$\textit{encima}^I = \{\langle\text{a,b}\rangle, \langle b, c\rangle, \langle\text{c,a}\rangle, \langle\text{e,f}\rangle\}$$
$$\textit{tabla}^I = \{\text{a,d,e}\}$$
$$\textit{despejado}^I = \{\text{c,d,f}\}$$

Definición 2.2.2 (Asignación de variables)**.** Una *asignación de variables* U en una L-estructura (Ω, I) es una función que asigna a cada variable del lenguaje un elemento del universo Ω. Si σ es una variable, se designará por σ^U o por $U(\sigma)$ la imagen de σ por la asignación U.

Definición 2.2.3 (Asignación de términos)**.** Dadas una L-estructura (Ω, I) y una asignación de variables U, se define la *asignación de términos* T_{IU} como sigue:

- si τ es un símbolo de constante de L, entonces $T_{IU}(\tau) = I(\tau)$,
- si τ es una variable, entonces $T_{IU}(\tau) = U(\tau)$,
- si τ es de la forma $\pi(\tau_1, .., \tau_n)$, donde π es un símbolo de función de L, entonces $T_{IU}(\tau) = I(\pi)(T_{IU}(\tau_1), .., T_{IU}(\tau_n))$.

Las nociones de L-estructura y asignación de variables nos permiten definir de manera formal una noción de verdad o *satisfacción*. Si una interpretación I y una asignación de variables U satisfacen a una fórmula ϕ, entonces diremos que ϕ es verdadera relativa a I y U y se denotará $\models_I \phi[U]$. Veamos la definición formal de esta noción de capital importancia.

Notación. En la siguiente definición, en vez de escribir $\neg(\models_I \phi[U])$ escribiremos $\not\models_I \phi[U]$. (Esta será una notación habitual, a veces evitaremos usar el símbolo $\neg$ si va seguido de otro símbolo y simplemente tacharemos el símbolo que le sigue).

Definición 2.2.4 (Satisfacción)**.** Dadas una L- estructura (Ω, I) y una asignación de variables U diremos que:

- $\models_I (\sigma = \tau)[U]$ si y solo si $T_{IU}(\sigma) = T_{IU}(\tau)$, donde σ y τ son L-términos.
- $\models_I \rho(\tau_1, ..., \tau_n)[U]$ si y solo si $\langle T_{IU}(\tau_1), ..., T_{IU}(\tau_n)\rangle \in I(\rho)$, con ρ un símbolo de predicado de aridad n y τ_i términos.
- $\models_I (\neg\phi)[U]$ si y solo si $\not\models_I \phi[U]$.
- $\models_I (\phi_1 \wedge \phi_2)[U]$ si y solo si $\models_I \phi_i[U]$ para todo i, $1 \leq i \leq 2$.
- $\models_I (\phi_1 \vee \phi_2)[U]$ si y solo si $\models_I \phi_i[U]$ para al menos un i, $1 \leq i \leq 2$.
- $\models_I (\phi \Rightarrow \psi)[U]$ si y solo si $\not\models_I \phi[U]$ o $\models_I \psi[U]$.
- $\models_I (\phi \Leftarrow \psi)[U]$ si y solo si $\not\models_I \psi[U]$ o $\models_I \phi[U]$.
- $\models_I (\phi \Leftrightarrow \psi)[U]$ si y solo si $\models_I (\phi \Rightarrow \psi)[U]$ y $\models_I (\phi \Leftarrow \psi)[U]$.
- $\models_I (\forall v\phi)[U]$ si y solo si para todo $d \in \Omega$ se cumple que $\models_I \phi[V]$, donde $V(v) = d$ y $V(w) = U(w)$ para toda variable w distinta de v.
- $\models_I (\exists v\phi)[U]$ si y solo si para algún $d \in \Omega$ se cumple que $\models_I \phi[V]$, donde $V(v) = d$ y $V(w) = U(w)$ para toda variable w distinta de v.

Definición 2.2.5 (Modelo)**.** Sean $M = (\Omega, I)$ una L-estructura y ϕ una fórmula. Si la interpretación I satisface a ϕ para cualquier asignación de variables, se dice que M es un *modelo* de ϕ y se denota $\models_I \phi$.

Por ejemplo, en el mundo de las cajas, la estructura M definida en el Ejemplo 2.2.1 es un modelo de la fórmula $\phi \equiv sobre(x, y) \Rightarrow encima(x, y)$.

Definición 2.2.6 (Satisfacción y validez)**.** Una L-fórmula ϕ se dice *satisfacible* si existen alguna L-estructura y alguna asignación de variables en dicha estructura que la satisfagan. En caso contrario, se dirá insatisfacible.

Una fórmula L-fórmula ϕ se dice *válida* si es satisfecha por toda L-estructura y toda asignación de variables.

Nota 2.2.1. Podemos extender estas definiciones a un conjunto de fórmulas Γ de manera natural. Diremos que una interpretación I y una asignación de variables U satisfacen a Γ (se denota $\models_I \Gamma[U]$) si I y U satisfacen a todas las fórmulas de Γ. Una estructura M es modelo de Γ si es modelo de cada una de sus fórmulas. Se dice que Γ es satisfacible o consistente si existe una interpretación y una asignación de variables que satisfagan a todos sus miembros, en caso contrario se dice que Γ es insatisfacible o inconsistente.

Definición 2.2.7 (Consecuencia lógica)**.** Sea Γ un conjunto de fórmulas y ϕ una fórmula. Diremos que ϕ es consecuencia lógica de Γ, y lo denotaremos por $\Gamma \models \phi$, si para toda interpretación I y toda asignación de variables U se cumple que:

$$\models_I \Gamma[U] \quad \Rightarrow \quad \models_I \phi[U]$$

Definición 2.2.8 (Equivalencia lógica)**.** Dos fórmulas ϕ y ψ son *lógicamente equivalentes* si para toda interpretación I y toda asignación de variables U se cumple que:

$$\models_I \phi[U] \quad \Leftrightarrow \quad \models_I \psi[U]$$

Proposición 2.2.1 (Monotonía de la lógica clásica)**.** Sean Γ, Γ' conjuntos de fórmulas cerradas y ϕ, ψ fórmulas cerradas.

1. (Reflexiva) Si $\phi \in \Gamma$ entonces $\Gamma \models \phi$.
2. (Corte) Si $\Gamma \models \psi$ y $\Gamma \cup \{\psi\} \models \phi$ entonces $\Gamma \models \phi$.
3. (**Monotonía**) Si $\Gamma \subseteq \Gamma'$ entonces: $\Gamma \models \phi \Rightarrow \Gamma' \models \phi$.

Destacamos la propiedad de monotonía de la relación de consecuencia de la lógica clásica pues, como ya hemos mencionado, el objetivo del presente trabajo es estudiar otros contextos formales donde dicha propiedad de monotonía deja de ser cierta.

2.3. El mundo de las cajas

Como ejemplo para expresar el conocimiento vamos a considerar otra vez el mundo de las cajas como se expresa en la Figura 2.1.

Los elementos de la conceptualización son las seis cajas y las relaciones *sobre, encima, despejado* y *tabla*. Nuestro lenguaje consta de los símbolos de constante *a, b, c, d, e, f* y de los símbolos de predicado *sobre/2*, *encima/2*, *despejado/1*, *tabla/1*. Para interpretarlos, asumimos la interpretación estándar I descrita anteriormente en el Ejemplo 2.2.1.

Las fórmulas que siguen engloban la información esencial sobre este caso (fórmulas atómicas sin variables verdaderas en la interpretación):

$sobre(b,a)$	$encima(b,a)$	$despejado(c)$	$tabla(a)$
$sobre(c,b)$	$encima(c,b)$	$despejado(d)$	$tabla(d)$
$sobre(f,e)$	$encima(c,a)$	$despejado(f)$	$tabla(e)$
	$encima(f,e)$		

Usando fórmulas más complejas, podemos expresar hechos más generales. Por ejemplo, podemos expresar la propiedad transitiva de la relación *encima* y la propiedad "si una caja está sobre otra, entonces está encima"de la siguiente manera:

$$\forall x \forall y \forall z \, (encima(x, y) \wedge encima(y,z) \Rightarrow encima(x, z))$$

$$\forall x \forall y \, (sobre(x, y) \Rightarrow encima(x, y))$$

Nótese que el predicado *despejado* puede definirse a partir del predicado *sobre* y la lógica de primer orden. Esto es:

$$\forall y \, (\neg \exists x \, sobre(x, y) \Leftrightarrow despejado(y))$$

De manera análoga, podemos definir el símbolo de predicad *tabla* a partir del predicado *sobre*:

$$\forall y \, (\neg \exists x \, sobre(y, x) \Leftrightarrow tabla(y))$$

El problema de saber qué funciones y qué relaciones se deben incluir en la conceptualización no tiene una solución general. En el caso anterior, por ejemplo, hemos visto que podría prescindirse de los símbolos de predicado *tabla* y *despejado* sin perder poder expresivo, pues son definibles a partir del resto de símbolos del lenguaje. Sin embargo, en otros escenarios, el considerar o no la presencia de un determinado símbolo de predicado en el lenguaje que usemos para la formalización de una base de conocimiento puede alterar significativamente la capacidad expresiva y las propiedades del lenguaje. Incluso conservando el poder expresivo, el hecho de incluir o no un símbolo en el lenguaje puede hacer que la escritura de las propiedades se convierta en una tarea mucho más laboriosa o menos intuitiva.

2.4. Ejemplo de Circuitos

Consideremos el circuito **f** descrito en la Figura 2.2, donde las puertas x_1, x_2 son puertas XOR, la puerta a_1 es una puerta AND y las puertas o_1, o_2 son puertas OR. Las entradas de cada componente están a la izquierda y el cable de la derecha representa la salida. Cada puerta tiene dos entradas y una salida, mientras que el circuito **f** tiene tres entradas y dos salidas.

El universo de trabajo tendría, en principio, 26 objetos: los 20 puertos de entrada/salida, las 5 puertas y el propio circuito. Uno podría considerar un lenguaje de primer orden con símbolos de constante para cada uno de estos elementos, pero resultaría demasiado prolijo. En su lugar, podemos

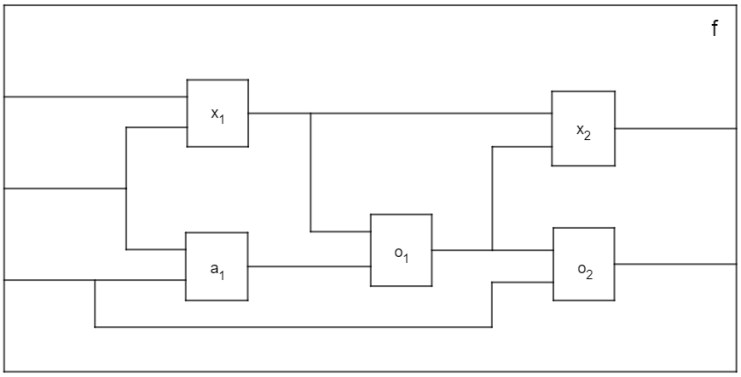

Figura 2.2: Circuito

considerar un lenguaje más reducido y crear funciones que asocien puertos y componentes. Por ejemplo, podemos considerar una función *entrada* $i(n, x)$ que al recibir un entero n y una componente x devuelva la entrada n-ésima de dicha componente, y la función *salida* $o(n, x)$ que al recibir un entero n y una componente x devuelva la salida n-ésima de dicha componente. Ahora bien, esta elección nos obliga a ampliar el universo de trabajo subyacente con el conjunto de números naturales $\{1, 2, 3\}$ para denotar las posibles entradas o salidas de las componentes.

Por otra parte, también necesitamos una relación que nos especifique qué puertos están conectados entre ellos (por ejemplo, "la tercera entrada de f está conectada a la segunda entrada de a_1"). Optamos pues por considerar el siguiente lenguaje de primer orden:

- *f, x1, x2, a1, o1, o2* son símbolos de constante, que representan las seis componentes (el propio circuito y sus cinco puertas).
- *1, 2, 3* son símbolos de constante, que representas las posibles entradas y salidas de las componentes.
- $circuito(x)$ es un símbolo de relación que indica que x es un circuito.
- $xorp(x)$ es un símbolo de relación que indica que x es una puerta xor.
- $andp(x)$ es un símbolo de relación que indica que x es una puerta and.
- $orp(x)$ es un símbolo de relación que indica que x es una puerta or.
- $i(n, x)$ es un símbolo de función que recibe un entero n y una componente x y devuelve la n-ésima entrada de x.

- $o(n, x)$ es un símbolo de función que recibe un entero n y una componente x y devuelve la n-ésima salida de x.
- $con(x, y)$ es un símbolo de relación que indica que el objeto x está conectado con el objeto y.

Dado este vocabulario, y teniendo en mente la interpretación natural de estos símbolos, podemos describir la estructura del circuito con las siguientes fórmulas atómicas. Las seis primeras indican qué tipo de componente es cada componente y el resto indica cómo están conectadas:

circuito(f)
xorp(x1)
xorp(x2)
andp(a1)
orp(o1)
orp(o2)
con(i(1,f),i(1,x1))
con(i(2,f),i(2,x1))
con(i(2,f),i(1,a1))
con(i(3,f),i(2,a1))
con(i(3,f),i(2,o2))
con(o(1,x1),i(1,x2))
con(o(1,x1),i(1,o1))
con(o(1,a1),i(2,o1))
con(o(1,o1),i(1,o2))
con(o(1,o1),i(2,x2))
con(o(1,x2),o(1,f))
con(o(1,o2),o(2,f))

Ahora bien, al ser o e i símbolos de función de un lenguaje de primer orden, pueden ser aplicadas a *todos* los elementos del universo de trabajo. Por ejemplo, ha de existir un valor para el término $o(1, 3)$ o el término $i(o(1, 1), i(o2, 1))$ aunque dichas expresiones no tengan sentido en la práctica. Una posible solución es ampliar el universo de trabajo con el número 0 y usar el 0 como un valor por defecto para todos estos términos sin sentido en la práctica.

Una vez descrita la estructura del circuito, pasamos a tratar la propagación de señales eléctricas a través de él. Podemos describir el estado del circuito **f** expandiendo nuestro lenguaje para incluir bits que indiquen si la frecuencia es alta o baja en una componente y una relación que asocie a cada puerto el valor de dicho puerto. Podemos hacerlo añadiendo a nuestro lenguaje lo siguiente:

- *0* es un símbolo de constante que indican baja frecuencia. 'Alta frecuencia' estará representado por la constante *1* que ya pertenece a nuestro lenguaje.
- $v(x, z)$ es un símbolo de relación que indica que el valor en el puerto x es z ($z \in \{0, 1\}$).

Usando este vocabulario, podemos introducir valores específicos que nos digan la frecuencia en algún puerto, por ejemplo:

v(i(1,f),0)
v(i(2,f),1)
v(i(3,f),1)

Podemos describir ahora el comportamiento de cada componente del circuito con fórmulas cerradas más complejas. En este caso, las dos primeras fórmulas describen el comportamiento de una puerta AND, las siguientes dos describen una puerta OR, las siguientes dos describen una puerta XOR, y la última describe la transmisión de la señal a través del circuito.

$\forall x\,(andp(x) \wedge v(i(1,x),1) \wedge v(i(2,x),1) \Rightarrow v(o(1,x),1))$
$\forall x \forall n\,(andp(x) \wedge v(i(n,x),0) \Rightarrow v(o(1,x),0))$
$\forall x \forall n\,(orp(x) \wedge v(i(n,x),1) \Rightarrow v(o(1,x),1))$
$\forall x\,(orp(x) \wedge v(i(1,x),0) \wedge v(i(2,x),0) \Rightarrow v(o(1,x),0))$
$\forall x \forall z\,(xorp(x) \wedge v(i(1,x),z) \wedge v(i(2,x),z) \Rightarrow v(o(1,x),0))$
$\forall x \forall y \forall z\,(xorp(x) \wedge v(i(1,x),y) \wedge v(i(2,x),z) \wedge y \neq z \Rightarrow v(o(1,x),1))$
$\forall x \forall y \forall z\,(con(x,y) \wedge v(x,z) \Rightarrow v(y,z))$

Con esto hemos descrito completamente el funcionamiento del circuito **f**. Además, si cambiáramos de circuito, sólo tendríamos que cambiar las fórmulas atómicas que dan información explícita del circuito, ya que el funcionamiento sería válido para cualquier circuito.

Nota 2.4.1. En las fórmulas anteriores, al escribir por ejemplo $\forall n$, la variable n puede tomar cualquier valor del universo de la estructura subyacente, es decir, podría ser un número del 1 al 3 (que es el caso para el cual tiene sentido) pero también podría ser una puerta o un valor de encendido/apagado. En este caso concreto esto no da problemas puesto que la relación $v/2$ solo se verifican para ciertos elementos y, por tanto, la implicación solo aporta información en los casos en los que n es un elemento del tipo que esperamos que sea. Ahora bien, en otros casos (veremos un ejemplo en el ejemplo que sigue) puede ser necesario explicitar el *tipo* de una variable de primer orden. En principio, esto no sería posible pues el lenguaje de la lógica de primer orden no contiene tipos. Sin embargo es fácil simularlos de una manera canónica. Se añade un predicado 1-ario para el tipo $p(v)$ que queremos tratar y se relativizan los cuantificadores correspondientes: $\forall v\,(p(v) \Rightarrow \dots)$ y $\exists v\,(p(v) \wedge \dots)$.

2.5. Base de datos familiar

Por último, vamos a ver un ejemplo de una base de datos familiar y representar el conocimiento de ésta. Primero veremos los conceptos básicos como hermano y hermana, luego definiremos el concepto de huérfano y, finalmente, una definición recursiva de ancestro.

2.5.1. Conceptos básicos

Partimos del siguiente vocabulario:

- *juan, pablo, irene, alberto, macho, hembra* son símbolos de constante que representan a cuatro personas y dos géneros.
- *persona(x)* es un símbolo de predicado que indica que x es una persona.
- *genero(x)* es un símbolo de predicado que indica que x es un género.
- *padre(x,y)* es un símbolo de predicado que indica que x es el padre de y.
- *madre(x,y)* es un símbolo de predicado que indica que x es la madre de y.
- *generode(x,y)* es un símbolo de predicado que indica que el género de x es y.

Tenemos la siguiente información explícita:

persona(juan)
persona(pablo)
persona(irene)
persona(alberto)
genero(macho)
genero(hembra)
padre(juan,pablo)
madre(irene,pablo)
padre(juan,alberto)
madre(irene,alberto)
generode(juan,macho)
generode(pablo,macho)
generode(irene,hembra)
generode(alberto,macho)

y añadimos las siguientes fórmulas que dan información sobre las relaciones derivadas *progenitor(x,y)*, *hijo(x,y)* y *hermano(x,y)*.

1. $\forall x \forall y \quad (padre(x,y) \vee madre(x,y) \Rightarrow progenitor(x,y))$
2. $\forall x \forall y \quad (progenitor(x,y) \Rightarrow hijo(y,x))$
3. $\forall x \forall y \quad (\exists z \exists v \ (padre(z,x) \wedge padre(z,y) \wedge madre(v,x) \wedge madre(v,y) \wedge x \neq y) \Rightarrow hermano(x,y))$

Con las fórmulas que tenemos hasta el momento, podríamos añadir sin provocar ninguna contradicción que, por ejemplo, Irene sea el padre de Alberto. Sin embargo, en la vida real, no aceptaríamos dicho hecho en nuestra base de datos familiar pues haríamos uso de la regla implícita de sentido común que afirma que el padre de un individuo ha de ser de género masculino y la madre de un individuo ha de ser de género femenino. Vamos a corregir eso añadiendo fórmulas para que los padres solo puedan ser hombres y las madres sean mujeres, además de una fórmula que especifique que una persona

no puede tener dos géneros simultáneamente.

4. $\forall x \forall y \quad (generode(x,macho) \Rightarrow \neg madre(x,y))$
5. $\forall x \forall y \quad (\ generode(x,hembra) \Rightarrow \neg padre(x,y))$
6. $\forall x \forall y \forall z \quad (\ generode(x,y) \wedge y \neq z \wedge genero(z) \Rightarrow \neg generode(x,z))$

Añadimos también una fórmula para expresar que una persona x no puede tener más de un padre o más de una madre (al menos biológicamente).

7. $\forall x \forall y \forall z \quad (\ padre(y,x) \wedge y \neq z \wedge persona(z) \Rightarrow \neg padre(z,x)\ )$
8. $\forall x \forall y \forall z \quad (\ madre(y,x) \wedge y \neq z \wedge persona(z) \Rightarrow \neg madre(z,x)\ )$

De esta forma, podríamos ya detectar una contradición si añadimos a nuestra base de datos familiar el hecho *padre(irene,alberto)*, así como deducir otros hechos de *sentido común*. Hacer explícita toda esta información implícita que manejamos en los ejemplos de aplicaciones reales es otro gran reto en la formalización del conocimiento declarativo.

2.5.2. Huérfanos

Para definir la relación huérfano, nos ceñimos a la definición del diccionario de la R.A.E. de la palabra huérfano: *adj. Dicho de una persona menor de edad: A quien se le han muerto el padre y la madre o uno de los dos.* Necesitamos por tanto incluir en el vocabulario nuevas relaciones primitivas que indiquen si alguien está muerto y si alguien es menor de edad. Ampliamos pues nuestro lenguaje de primer orden con los símbolos de predicado:

- *menorDeEdad(x)* que indica que x es menor de edad.
- *hafallecido(x)* que indica que x está muerto o muerta.

Añadimos la fórmula cerrada que define la relación derivada huérfano/a:

9. $\forall x \forall y \quad (\ menorDeEdad(x) \wedge hafallecido(y) \wedge progenitor(y,x) \Rightarrow huerfano(x)\ )$

Si ahora añadimos por ejemplo a la información inicial la fórmula atómica $hafallecido(irene)$, podemos deducir $huerfano(pablo)$ y $huerfano(alberto)$.

2.5.3. Ancestros

Queremos definir ahora la relación *ancestro(x,y)* que se cumple si x es un ancestro (padre/madre, abuelo/abuela...) de y, es decir, queremos expresar que nuestros padres son nuestros ancestros, que los padres de nuestros padres también lo son, y así sucesivamente. Para ello, podemos usar recursión codificada en la lógica de primer orden como sigue:

10. $\forall x \forall y$ *(progenitor(x,y)* $\Rightarrow$ *ancestro(x,y))*
11. $\forall x \forall y \forall z$ *(progenitor(x,y)* $\wedge$ *ancestro(y,z)* $\Rightarrow$ *ancestro(x,z))*

Capítulo 3

Razonamiento no monótono

En el presente capítulo introducimos los primeros formalismos de razonamiento no monótono, que estarán basados en la extensión de un conjunto de creencias a partir de un cierto supuesto que pretende describir un esquema de razonamiento basado en el sentido común.

3.1. Hipótesis de Mundo Cerrado

Dado un lenguaje de primer orden L, un *conjunto de creencias* Δ es simplemente un conjunto de fórmulas cerradas de L; mientras que una *L-teoría* τ es un conjunto de fórmulas cerradas de L cerrado bajo consecuencia lógica. Todo conjunto de creencias da lugar de manera natural a una teoría asociada.

Definición 3.1.1. Dado un conjunto de L-fórmulas cerradas Δ, se define la teoría $\tau(\Delta)$ como el conjunto de las fórmulas cerradas de L que se pueden deducir lógicamente de Δ. Esto es:

$$\tau(\Delta) = \{A \,:\, A \text{ fórmula cerrada de } L,\, \Delta \models A\}.$$

Obsérvese que Δ desempeña el papel de los *axiomas propios* de un conjunto de creencias y la teoría $\tau(\Delta)$ no es más que el conjunto de las propiedades de primer orden que se pueden deducir (desde el punto de vista clásico) a partir de Δ. Si no se menciona explícitamente el lenguaje de primer orden en el que se escriben las fórmulas de Δ, entenderemos que el lenguaje asociado será el menor lenguaje que permita escribir las fórmulas de Δ.

Una teoría τ se dice *consistente* si tiene algún modelo e *inconsistente* en caso contrario[1]. Un conjunto de creencias Δ se dice consistente o inconsistente según lo sea su teoría asociada $\tau(\Delta)$.

[1]La noción de consistencia de una teoría suele corresponder con un enfoque sintáctico:

Definición 3.1.2 (Átomos y Literales)**.** Dado un lenguaje de primer orden L, llamamos *átomos* a las fórmulas atómicas de L, y llamamos *literales* a un átomo de L o a su negación. Recordemos que un átomo (resp. un literal) se dirá *básico* si no contiene variables. Por otra parte, un átomo se dirá también un literal *positivo* mientras que la negación de un átomo se dirá un literal *negativo*.

Definición 3.1.3 (Átomo-completitud)**.** Sea L un lenguaje de primer orden. Una L-teoría τ se dice *átomo-completa* si para toda A fórmula atómica básica de L, o bien $A \in \tau$ o bien $\neg A \in \tau$.

Por ejemplo, si consideramos el conjunto de creencias

$$\Delta = \{p(a),\, p(a) \Rightarrow q(a) \wedge p(b)\}$$

entonces la teoría $\tau(\Delta)$ no es átomo-completa; pues, aunque se pueden deducir $p(a)$, $q(a)$ y $p(b)$ a partir de $\tau(\Delta)$, no se deduce ni $q(b)$ ni su negación.

La *Hipótesis de Mundo cerrado* o *CWA* (por sus siglas en inglés) consiste en completar una teoría definida por un conjunto de creencias Δ, incluyendo en dicha teoría la negación de todos los átomos básicos que no se puedan deducir lógicamente de Δ. Dicho de otro modo, si no podemos demostrar que un cierto hecho básico es el caso, entonces añadiremos a nuestro sistema de creencias su negación. En el ejemplo mencionado más arriba, puesto que del conjunto Δ no se deduce el átomo $q(b)$, la aplicación de la hipótesis de mundo cerrado consiste en expandir el conjunto de creencias con el literal $\neg q(b)$ (que convierte dicha expansión en átomo-completa).

Es claro que este método de expansión es no monótono ya que, al añadir nuevas fórmulas a Δ, se podría reducir dicha expansión (siempre que las nuevas fórmulas aporten nuevos literales positivos a nuestra teoría). Veremos ejemplos concretos más adelante. Veamos ahora le definición formal de este mecanismo de expansión.

Definición 3.1.4 (Cierre bajo CWA)**.** Sea Δ un conjunto de creencias en un lenguaje L.

- Definimos su *extensión de mundo cerrado*, Δ_{emc}, como el conjunto de los literales $\neg A$ tal que A es un átomo básico de L y $A \notin \tau(\Delta)$.

---

no existe una prueba en un cierto cálculo deductivo para la lógica de primer orden de una contradicción a partir de los axiomas de la teoría. Sin embargo, por el Teorema de Completitud de la lógica de primer orden, sabemos que dicho enfoque es equivalente a la definición semántica por la que hemos optado en el presente trabajo.

- La teoría aumentada $CWA[\Delta]$, el *cierre de* Δ *bajo la hipótesis de mundo cerrado*, se define como $\tau(\Delta \cup \Delta_{emc})$. Esto es, como el conjunto de todas las L-fórmulas cerradas ϕ tales que $\Delta \cup \Delta_{emc} \models \phi$.
- Sea ϕ un fórmula cerrada de L. Escribiremos $\Delta \models_{CWA} \phi$ para denotar que $\phi \in \tau(\Delta \cup \Delta_{emc})$.

La relación de consecuencia $\models_{CWA}$ que acabamos de introducir es nuestro primer ejemplo de relación de consecuencia lógica no monótona.

Proposición 3.1.1. La relación de consecuencia $\models_{CWA}$ no es monótona.
Demostración: Consideramos el ejemplo anterior dado por

$$\Delta = \{p(a),\, p(a) \Rightarrow q(a) \wedge p(b)\}.$$

Entonces, $\Delta_{emc} = \{\neg q(b)\}$ y, por tanto, $\Delta \models_{CWA} \neg q(b)$.
Ahora bien, $(\Delta + q(b))_{emc} = \emptyset$ y, por tanto, $\Delta + q(b) \not\models_{CWA} \neg q(b)$. □

Ejemplo 3.1.1. Si en una base de datos queremos introducir todas las provincias españolas y establecer una relación de vecindad escribiríamos:

$$\begin{aligned} &vecino(sevilla, huelva)\\ &vecino(huelva, cadiz)\\ &vecino(granada, sevilla)\\ &\quad\vdots\\ &\forall x \forall y\, (vecino(x, y) \Rightarrow vecino(y, x)) \end{aligned}$$

En un tal conjunto de creencias Δ convendría asumir que si dos provincias no aparecen en la lista como vecinos, o no es posible deducir su vecindad aplicando la regla, entonces no lo son. Esta suposición es un ejemplo de Hipótesis de Mundo Cerrado. Si no recurrimos a esto, entonces deberíamos enunciar todas las parejas de provincias no vecinas y el tamaño de la base de datos aumentaría considerablemente. Obsérvese, por ejemplo, que $\Delta \not\models \neg vecino(sevilla, teruel)$; mientras que $\Delta \models_{CWA} \neg vecino(sevilla, teruel)$.

Es importante notar que, en general, el cierre bajo CWA no tiene por qué conservar la consistencia de un conjunto de creencias. Consideremos el siguiente ejemplo simple:

$$\Delta = \{p(a) \vee q(a)\}.$$

Puesto que ni $p(a)$ ni $q(a)$ están en $\tau(\Delta)$, $\neg p(a)$ y $\neg q(a)$ estarán en Δ_{emc}. Ahora bien, esto entra en contradicción con la fórmula de Δ y, por tanto, $CWA[\Delta]$ es inconsistente.

En el ejemplo anterior, el problema viene de tener una disyunción con dos literales positivos y no poder deducir lógicamente ninguno de ellos. Veremos que este hecho no es casual. El siguiente teorema establece los requisitos para que una ampliación por mundo cerrado sea consistente.

Nota 3.1.1. Recordemos que una cláusula es una disyunción de literales (que se interpreta semánticamente como el cierre universal de dicha disyunción de literales si contiene alguna variable) y que una cláusula es positiva si solo contiene literales positivos.

Teorema 3.1.1. Sea Δ un conjunto de creencias consistente en un lenguaje L. Son equivalentes:

1. Su ampliación $CWA[\Delta]$ es consistente.

2. Para cada cláusula positiva sin variables $L_1 \vee ... \vee L_n$ que se deduce de Δ, hay al menos un átomo L_i que se deduce de Δ.

Demostración: Equivalentemente, basta demostrar que la teoría $CWA[\Delta]$ es inconsistente si y solo si existen literales positivos básicos $L_1, ..., L_n$ tales que $\Delta \models L_1 \vee ... \vee L_n$ pero $\Delta \not\models L_i \, \forall i = 1...n$.

La teoría $CWA[\Delta]$ es inconsistente si y solo si $\Delta \cup \Delta_{emc}$ lo es. Supongamos que lo es. Entonces, puesto que Δ es consistente, por el teorema de compacidad de la lógica de primer orden, existe un subconjunto *finito no vacío* de Δ_{emc} que contradice a Δ. Sea $\{\neg L_1, ..., \neg L_n\}$, con L_i átomos básicos y $n \geq 1$, dicho conjunto finito. Puesto que $\Delta \cup \{\neg L_1, ..., \neg L_n\}$ es inconsistente, tenemos que $\Delta \models \neg(\neg L_1 \wedge ... \wedge L_n)$. Esto implica que $\Delta \models L_1 \vee ... \vee L_n$. Por otra parte, como tenemos que $\neg L_i$ está en Δ_{emc} para cada $i \leq n$, ninguno de los L_i es consecuencia lógica de Δ. La otra implicación es inmediata. $\square$

Nota 3.1.2. Nótese que en el teorema anterior es importante fijar qué lenguaje de primer orden se está considerando. Veamos el siguiente ejemplo. Sea Δ el conjunto de creencias dado por las siguientes fórmulas:

$$\forall x \, (p(x) \vee q(x))$$
$$p(a)$$
$$q(b)$$

Si consideramos el lenguaje $L = \{a, b, p/1, q/1\}$, entonces se cumplen las condiciones del teorema y la ampliación $CWA[\Delta]$ es consistente. De hecho,

$\Delta_{emc} = \{\neg q(a), \neg p(b)\}$ y $\Delta \cup \Delta_{emc}$ es claramente consistente (y átomo-completa). Sin embargo, si consideramos el mismo conjunto de fórmulas pero ahora en el lenguaje $L' = \{a, b, c, p/1, q/1\}$, donde c es un nuevo símbolo de constante, la ampliación $CWA[\Delta]$ resulta ahora inconsistente, pues $\Delta \models p(c) \vee q(c)$ y, sin embargo, $\Delta \not\models p(c)$ y $\Delta \not\models q(c)$. La hipótesis de mundo cerrado nos obligaría a extender Δ con los literales $\neg p(c)$ y $\neg q(c)$, lo cual entraría en contradicción con la primera fórmula de Δ.

Nota 3.1.3. En relación con este último ejemplo, el *supuesto de dominio cerrado* es un método de ampliación de un conjunto de creencias Δ en la que asumimos que los únicos objetos en el universo de trabajo son aquellos que se pueden nombrar mediante los términos sin variables del lenguaje. Formalmente se puede representar con la fórmula:

$$\forall x \, (x = t_1 \vee x = t_2 \vee \dots)$$

donde los t_i son las términos sin variables del lenguaje. En caso de tener infinitos términos básicos (ya sea por contar con un número infinito de símbolos de constante o por contar con símbolos de función en el lenguaje), no se podría representar con una fórmula de primer orden.

Otra suposición relacionada que aparece a menudo en la lógica no monótona es el *Supuesto de Nombre Único* (SNU), que asume que dos términos básicos no son iguales entre sí a menos que se pueda deducir lógicamente del conjunto de creencias. Obsérvese que esto no es más que aplicar la Hipótesis de Mundo Cerrado al predicado de igualdad.

Puesto que las condiciones del teorema 3.1.1 anterior pueden ser difíciles de probar, enunciamos un corolario que nos proporciona una condición suficiente para aplicar el teorema que es más fácil de comprobar. Recordemos que una *cláusula de Horn* es una cláusula con, a lo más, un literal positivo.

Proposición 3.1.2. Sea Δ un conjunto de creencias formado por cláusulas de Horn. Supongamos que $\Delta \cup \{\neg L_1, ..., \neg L_n\}$ es inconsistente, con L_i átomos básicos. Entonces, existe $i \in \{1, ..., n\}$ tal que $\Delta \cup \{\neg L_i\}$ es inconsistente.

Demostración: La proposición se deduce de los siguientes hechos: 1) un conjunto de cláusulas es consistente si y solo si tiene un modelo de Herbrand (véase el capítulo 4 para definiciones); y 2) la intersección de un número finito de modelos de Herbrand de un conjunto de cláusulas de Horn Δ es un nuevo modelo de Herbrand de Δ. □

Corolario 3.1.1. Sea Δ un conjunto de creencias consistente que puede expresarse como un conjunto de cláusulas de Horn. Entonces, la ampliación $CWA[\Delta]$ es una teoría consistente y átomo-completa.

Demostración: Lo vemos por reducción al absurdo. Supongamos que nuestro conjunto de creencias Δ es consistente y de Horn y que, sin embargo, su ampliación $CWA[\Delta]$ inconsistente. Entonces, usando el teorema 3.1.1, podemos deducir de Δ una disyunción $L_1 \vee \cdots \vee L_n$ de átomos básicos tal que ninguno de ellos se deduce de Δ. Entonces, $\Delta \cup \{\neg L_1, ..., \neg L_n\}$ es inconsistente. Sin embargo, como Δ solamente contiene cláusulas de Horn, debe existir un i tal que $\Delta \cup \{\neg L_i\}$ es inconsistente (aplicando la proposición anterior). Luego, para algún i, $\Delta \models L_i$; esto entra en contradicción con la elección de los L_i. □

Nótese que el corolario anterior nos proporciona una condición suficiente para la consistencia de $CWA[\Delta]$ pero que dicha condición no es necesaria. Basta considerar de nuevo el primer ejemplo de la nota 3.2.2. anterior para mostrar un ejemplo de un conjunto de creencias Δ que no es de Horn y, sin embargo, su extensión $CWA[\Delta]$ sí es consistente.

Nota 3.1.4. La Hipótesis de Mundo Cerrado en toda su generalidad es demasiado fuerte en muchas aplicaciones. No siempre será natural considerar que *cualquier* átomo básico que no podamos probar sea automáticamente falso. En algunos casos, conviene considerar la Hipótesis de Mundo Cerrado *respecto de un conjunto de predicados* Π. Esto es, supondremos falsos todos los literales positivos básicos que no se deduzcan del conjunto de creencias pero solo para los literales que incluyan algún predicado en el conjunto fijado Π. Consideremos el siguiente ejemplo. Sea Δ el conjunto de creencias

$$\forall x\,(p(x) \vee q(x))$$

en el lenguaje $L = \{a, b\}$. Entonces, $CWA[\Delta]$ es inconsistente (nótese que la cláusula no es de Horn). Sin embargo, si consideramos CWA respecto de $\Pi = \{p\}$, la ampliación de mundo cerrado correspondiente sí es consistente y nos permite inferir $\forall x\, q(x)$ si añadimos el supuesto de dominio cerrado.

Nota 3.1.5. En general, que un conjunto de creencias Δ sea Horn respecto de un conjunto de predicados Π (esto es, cada cláusula de Δ sólo contiene, a lo más, un literal positivo con predicado en Π) no implica que la ampliación de mundo cerrado respecto de Π sea consistente. Esto es, en este contexto no se verifica el Corolario 3.1.1 anterior. Por ejemplo, tomemos Δ dado por

$$p(a) \vee q(c)$$
$$p(b) \vee \neg q(c)$$

Es claro que Δ es de Horn con respecto de $\Pi = \{p\}$, pero la correspondiente ampliación de mundo cerrado respecto de Π es inconsistente (puesto que tanto $\neg p(a)$ como $\neg p(b)$ estarían en ella).

3.2. Completación de Predicados

El objetivo de la completación de un predicado p en un conjunto de creencias Δ es expandir el conjunto de creencias de manera que los únicos elementos que cumplan p son aquellos que sabemos que necesariamente lo cumplen a partir de las fórmulas de Δ. Por ejemplo, en un caso básico en el que tenemos Δ dado por

$$\forall x\,(x = a \Rightarrow p(x))$$

la completación del predicado p sería añadir a Δ la fórmula

$$\forall x\,(p(x) \Rightarrow x = a)$$

En este caso, la ampliación del predicado tiene el mismo efecto que la hipótesis de mundo cerrado respecto de p.

La completación de un predicado en un conjunto de creencias general Δ puede ser muy complicada y dar lugar a inconsistencias o argumentos circulares. Para evitarlo, nos restringiremos a ciertos casos de buen comportamiento.

Definición 3.2.1 (Conjunto de creencias solitario en un predicado)**.** Dado un conjunto de creencias Δ expresado en forma clausal y un predicado p de su lenguaje, se dice que Δ es *solitario* en el predicado p si cada cláusula en Δ con una ocurrencia positiva de p tiene, como mucho, una ocurrencia de p. En otras palabras, si una cláusula de Δ contiene una ocurrencia positiva de p, ésa es la única ocurrencia de p (positiva o negativa) que puede aparecer en ella.

Nota 3.2.1. Si un conjunto de cláusulas es solitario en p, entonces también es de Horn en p. La implicación en el otro sentido no es cierta en general: $p(a) \vee \neg p(b)$ es de Horn en p, pero no es solitaria en p.

Dado un conjunto de creencias Δ finito expresado en forma clausal y solitario en el predicado p, podemos escribir cada cláusula de Δ en la que ocurra p de forma positiva como sigue:

$$\forall \mathtt{y}\,(q_1 \wedge \cdots \wedge q_m \Rightarrow p(\mathtt{t}))$$

con $\mathtt{t}$ una tupla de términos $t_1, ..., t_n$ (siendo n la aridad de p) y q_i literales que no contienen a p. Si no hubiera ningún q_i, la cláusula sería simplemente $p(\mathtt{t})$. Los q_i y $\mathtt{t}$ pueden contener variables, las notamos por $\mathtt{y}$. La expresión es entonces equivalente a

$$\forall \mathtt{x}\,\forall \mathtt{y}\,(\mathtt{x} = \mathtt{t} \wedge q_1 \wedge ... \wedge q_m \Rightarrow p(\mathtt{x}))$$

Donde $\mathtt{x}$ es una tupla de variables nuevas que no están en $\mathtt{t}$, y donde $\mathtt{x=t}$ es una abreviación para $x_1 = t_1 \wedge ... \wedge x_n = t_n$. Finalmente, como las variables $\mathtt{y}$ sólo están presentes en la parte izquierda de la implicación, la fórmula es equivalente a:

$$\forall \mathtt{x}\,(\exists \mathtt{y}\,(\mathtt{x} = \mathtt{t} \wedge q_1 \wedge ... \wedge q_m) \Rightarrow p(\mathtt{x}))$$

Esta forma de reescribir la cláusula se llama la *forma normal de la cláusula*. Supongamos que tenemos k cláusulas con una ocurrencia positiva de p, y sus formas normales:

$$\begin{aligned}
&\forall \mathtt{x}\,(E_1 \Rightarrow p(\mathtt{x}))\\
&\forall \mathtt{x}\,(E_2 \Rightarrow p(\mathtt{x}))\\
&\quad\vdots\\
&\forall \mathtt{x}\,(E_k \Rightarrow p(\mathtt{x}))
\end{aligned}$$

Cada E_i es una conjunción de literales afectada por un cuantificador existencial $\exists \mathtt{y}$, como en el caso anterior. Podemos juntar todas estas cláusulas como una sola implicación de la siguiente forma:

$$\forall \mathtt{x}\,(E_1 \vee ... \vee E_k \Rightarrow p(\mathtt{x})) \tag{3.1}$$

Hemos juntado en una implicación todos los elementos que satisfacen a p de los que tenemos información. La completación del predicado p se puede hacer ahora añadiendo la fórmula siguiente, que implica que estos son los únicos elementos que satisfacen a p:

$$\forall \mathtt{x}\,(p(\mathtt{x}) \Rightarrow E_1 \vee ... \vee E_k)$$

Tenemos entonces una doble implicación y podemos escribirla como sigue:

$$\forall \mathtt{x}\,(E_1 \vee ... \vee E_k \Leftrightarrow p(\mathtt{x}))$$

Esto nos permite dar la siguiente definición:

Definición 3.2.2 (Completación de un predicado)**.** Dado un conjunto de creencias Δ finito expresado en forma clausal y solitario en el predicado p, definimos la completación del predicado p en Δ como

$$COMP[\Delta, p] := \Delta + \forall \mathtt{x}\,(E_1 \vee ... \vee E_k \Leftrightarrow p(\mathtt{x}))$$

Nota 3.2.2.

1. Si en el conjunto Δ aparece la cláusula $\forall \mathtt{x}\, p(\mathtt{x})$, podemos reescribirla como $\forall \mathtt{x}\,(True \Rightarrow p(\mathtt{x}))$. En consecuencia, la completación de p en Δ añadiría una fórmula del tipo $\forall \mathtt{x}\,(p(\mathtt{x}) \Rightarrow True \vee \dots)$ que no aporta ninguna información nueva. Por tanto, $COMP[\Delta, p] = \Delta$.

2. Si en el conjunto Δ no aparece ninguna cláusula en la que el predicado p ocurra de forma positiva, podemos suponer que en Δ ocurre la cláusula $\forall \mathtt{x}\,(False \Rightarrow p(\mathtt{x}))$. En consecuencia, la completación de p en Δ añadiría la fórmula $\forall \mathtt{x}\,(p(\mathtt{x}) \Rightarrow False)$. Por tanto, $COMP[\Delta, p] = \Delta + \forall \mathtt{x}\,\neg p(\mathtt{x})$.

Enunciamos, sin demostración, el principal resultado sobre la consistencia de la completación respecto a un predicado.

Teorema 3.2.1. Sea Δ un conjunto de creencias finito y consistente expresado en forma clausal y solitario en un predicado p. Entonces, $COMP[\Delta, p]$ es también consistente.

Ejemplo 3.2.1. Consideremos un ejemplo simple. El conjunto Δ está formado por las siguientes fórmulas:

$$\forall x\,(pinguino(x) \Rightarrow pajaro(x))$$
$$pajaro(juan)$$
$$\neg pinguino(maria)$$

Tenemos que Δ es solitario en el predicado *pajaro*. Vamos a proceder entonces a completarlo. Escribimos la forma normal de las cláusulas para *pajaro* y obtenemos:

$$\forall x\,(pinguino(x) \vee x = juan \Rightarrow pajaro(x)).$$

Entonces:

$$COMP[\Delta, pajaro] = \Delta + \forall x\,(pinguino(x) \vee x = juan \Leftrightarrow pajaro(x)).$$

Usando la completación del predicado *pajaro* y el supuesto de nombre único (que nos permite suponer que María no es Juan) podemos deducir que María no es un pájaro. La completación del predicado *pajaro* nos dice en este caso que los únicos pájaros que existen son los pingüinos y Juan.

Veamos a continuación que la completación de un predicado nos proporciona un nuevo mecanismo de razonamiento no monótono.

Definición 3.2.3 (Relación de consecuencia asociada a COMP)**.** Dados un conjunto de creencias Δ finito expresado en forma clausal y solitario en el predicado p y una fórmula cerrada ϕ, escribiremos $\Delta \models_{COMP,p} \phi$ para denotar $COMP[\Delta, p] \models \phi$.

Proposición 3.2.1. La relación de consecuencia lógica $\models_{COMP,p}$ anterior no satisface la propiedad de monotonía.

Demostración: Basta considerar de nuevo el ejemplo 3.2.1 anterior. Entonces, $\Delta + (juan \neq maria) \models_{COMP,pajaro} \neg pajaro(maria)$ mientras que, por otra parte, se tiene que $\Delta + (juan \neq maria) + pajaro(maria) \not\models_{COMP,pajaro} \neg pajaro(maria)$. □

Ejemplo 3.2.2. La condición del teorema 3.2.1 anterior es suficiente pero no necesaria. De hecho, aplicando la definición de completación a conjuntos de cláusulas que son de Horn en un predicado p aunque no necesariamente solitarias en p, podemos a veces obtener resultados coherentes y con sentido. Un ejemplo típico es la definición del factorial. La relación *factorial(x,y)* indica que el factorial de x es y. Sea Δ el conjunto de las siguientes fórmulas:

$\forall x\,(x{=}0 \Rightarrow factorial(x,1))$
$\forall x \forall y\,(\ x{\neq}0 \wedge factorial(menos(x,1),y) \Rightarrow factorial(x,producto(x,y)))$

La primera fórmula nos dice que el factorial de 0 es 1, la segunda nos dice que si el factorial de $x-1$ es y, entonces el factorial de x es $x \cdot y$. Escribiéndolo en forma normal nos queda:

$\forall x \forall z(x{=}0 \wedge z{=}1 \Rightarrow factorial(x,z))$
$\forall x \forall z(\exists y\ (x \neq 0 \wedge z{=}producto(x,y) \wedge factorial(menos(x,1),y) \Rightarrow factorial(x,z)))$

Ahora podemos completar el predicado *factorial*:

$\forall x \forall z\ (factorial(x,z) \Rightarrow (x{=}0 \wedge z{=}1) \vee$
$(\exists y\ (x{\neq}0 \wedge z{=}producto(x,y) \wedge factorial(menos(x,1),y)))$

Esto se interpreta como una definición recursiva del factorial que dice que si x es 0, entonces su factorial es 1; y si x no es cero entonces su factorial es x por el factorial de $x-1$.

3.2.1. Completación Paralela

Es posible también realizar la completación de un conjunto de creencias Δ con respecto a varios predicados en paralelo. En el proceso de completación paralela de un conjunto de predicados, cada predicado se trata de manera separada y las fórmulas nuevas que añadimos al completar respecto a uno de los predicados del conjunto no han de considerarse para efectuar la completación del siguiente predicado del conjunto.

Para evitar problemas de referencias circulares en la completación paralela, será necesario imponer ciertas restricciones en cuanto a cómo pueden ocurrir los predicados respectos a los cuales queremos efectuar la completación en el conjunto original de creencias Δ. No bastará con que el conjunto Δ sea solitario en cada uno de los predicados del conjunto. Veamos a continuación cuáles son estas restricciones adicionales.

Recordemos que una cláusula solitaria en un predicado p se puede escribir en su forma normal como sigue:

$$\forall x\,(E_1 \vee ... \vee E_k \Rightarrow p(x))$$

que se puede reescribir compactando la conjunción de los E_i en un solo predicado E:

$$\forall x\,(E \Rightarrow p(x)).$$

A continuación, debemos considerar un orden en el conjunto de los predicados respecto a los cuales queremos efectuar la completación paralela: $\Pi = \{p_1 < ... < p_n\}$, con p_i predicados en Δ. Debemos escribir las cláusulas que contengan literales positivos de Π en su forma normal, y combinar las que tengan los mismos p_i en fórmulas únicas:

$$\begin{aligned}
&\forall x\,(E_1 \Rightarrow p_1(x))\\
&\forall x\,(E_2 \Rightarrow p_2(x))\\
&\vdots\\
&\forall x\,(E_n \Rightarrow p_n(x))
\end{aligned}$$

La completación paralela se ejecuta entonces añadiendo las fórmulas

$$\forall x\,(p_i(x) \Rightarrow E_i)$$

$i = 1, ..., n$. Ahora bien, para evitar definiciones circulares de los predicados p_i, debe verificarse que la ordenación de los p_i considerada haga que ningún E_i contenga ocurrencias de $\{p_i, p_{i+1}, ..., p_n\}$, ni ocurrencias negativas de $\{p_1, ..., p_{i-1}\}$. Cuando se cumple esto, diremos que las cláusulas de Δ están *ordenadas en* Π.

Esta definición nos permite enunciar el siguiente resultado.

Teorema 3.2.2. Sea Δ un conjunto de creencias finito y consistente expresado en forma clausal y sea Π un conjunto ordenado de predicados de Δ. Si Δ es ordenado en Π, entonces la completación paralela de Π en Δ es también consistente.

Veamos un ejemplo muy sencillo de completación paralela en el que es posible aplicar el teorema anterior (en el siguiente parágrafo estudiaremos ejemplos más elaborados). Consideramos el conjunto de creencias Δ compuesto por las siguientes fórmulas:

$$seta(oronja)$$
$$\forall x\,(seta(x) \wedge \neg comestible(x) \Rightarrow venenosa(x))$$
$$venenosa(pipasDeManzana)$$

Haremos la completación con los predicados *seta* y *venenosa*. Debemos aislar pues el predicado *venenosa*, obteniendo:

$$\forall x\,((seta(x) \wedge \neg comestible(x)) \vee x = pipasDeManzana) \Rightarrow venenosa(x))$$

Además debemos reescribir la primera fórmula en su forma normal, es decir:

$$\forall x\,(x = oronja \Rightarrow seta(x))$$

Elegimos el orden $\Pi = \{p1 = seta,\ p2 = venenosa\}$. Entonces, tenemos

$$E_1 \equiv (x = oronja)$$

$$E_2 \equiv (seta(x) \wedge \neg comestible(x)) \vee (x = pipasDeManzana)$$

Se tiene que Δ es ordenado en Π, pues E_1 no contiene ocurrencias ni de *seta* ni de *venenosa*; y E_2 no contiene ocurrencias de *venenosa* ni ocurrencias negativas de *seta*.

La completación paralela consiste entonces en añadir las fórmulas:

$$\forall x\,(seta(x) \Rightarrow x = oronja)$$
$$\forall x\,(venenosa(x) \Rightarrow (x = pipasDeManzana \vee (seta(x) \wedge \neg comestible(x))$$

Al realizar la completación paralela por predicados hemos asumido que la única seta que consideramos es la seta oronja y la única comida venenosa que consideramos son las pipas de manzana y aquellas setas que no son comestibles.

Nota 3.2.3. La completación en paralelo no coincide, en general, con la composición secuencial de completaciones. Es fácil comprobar que en el ejemplo anterior el resultado obtenido es distinto si primero completamos respecto al predicado *venenosa* y, a continuación, completamos el conjunto de creencias extendido respecto del predicado *seta*.

3.3. Taxonomía Jerárquica y Razonamiento por Defecto

El objetivo de esta sección es ilustrar el uso de la completación de predicados para formalizar el razonamiento por defecto. El ejemplo que usaremos es el dado por *los pájaros normalmente vuelan*. Queremos expresar que, en general, los pájaros vuelan, pero que se dan ciertas excepciones. Vamos a empezar definiendo los conceptos con una *taxonomía jerárquica*. Para simplificar la notación, en lo que sigue omitiremos los cuantificadores universales más externos.

1. $cosa(piolin)$
2. $pajaro(x) \Rightarrow cosa(x)$
3. $avestruz(x) \Rightarrow pajaro(x)$
4. $avestruzV(x) \Rightarrow avestruz(x)$

Donde $avestruzV$ hace referencia a las avestruces que pueden volar (si las hubiera). Este conjunto de fórmulas expresa que Piolín es una cosa, todas las cosas son pájaros, todas las avestruces son pájaros y todas las avestruces voladoras son avestruces. Denotaremos (el cierre universal de) estas fórmulas como Δ_H.

Supongamos ahora que queremos expresar que, en general, las cosas no vuelan, que los pájaros en general sí vuelan, y que las avestruces no suelen volar. Escribiríamos:

5. $cosa(x) \wedge \neg pajaro(x) \Rightarrow \neg vuela(x)$
6. $pajaro(x) \wedge \neg avestruz(x) \Rightarrow vuela(x)$
7. $avestruz(x) \wedge \neg avestruzV(x) \Rightarrow \neg vuela(x)$
8. $avestruzV(x) \Rightarrow vuela(x)$

Definiremos este subconjunto de fórmulas como Δ_P, el conjunto de propiedades de cada categoría. Estas fórmulas nos dan unas propiedades taxónomicas de los objetos, es decir, según la clase que le hemos otorgado al objeto podemos saber si vuela o no vuela. Sin embargo sabemos que pueden existir más excepciones a la hora de detectar pájaros que no vuelen (por ejemplo, los pingüinos o un pájaro muerto). Si conociésemos todas las excepciones de antemano, bastaría con rehacer la regla 7 con todas las excepciones juntas. Sin embargo, la idea del razonamiento no monótona es ir descubriendo nuevas excepciones sobre la marcha. Ello nos obligaría a ir modificando las reglas de Δ_P continuamente. Así que desechamos la idea de listar todas las excepciones

al estilo de la regla 7. En su lugar, vamos a utilizar una técnica que nos diga que los pájaros, en general, pueden volar, a no ser que sean una *excepción*. De la misma manera queremos expresar que las cosas en general no pueden volar, pero los pájaros no son la única excepción (mosquitos,aviones...).

Vamos a crear una jerarquía de excepciones:

$$9.\ cosa(x) \land \neg raro1(x) \Rightarrow \neg vuela(x)$$

Donde *raro1* es un predicado que indica una anormalidad en un objeto para la propiedad *volar*. Este enunciado nos dice que un objeto que no tenga una anormalidad puede volar; sin embargo, que el objeto sea anormal no implica que pueda volar. Los pájaros forman parte de esa anormalidad:

$$10.\ pajaro(x) \Rightarrow raro1(x)$$

Llamamos a esta regla una *Regla de cancelación de herencia*. En términos generales, podríamos pensar que el hecho de que los pájaros sean cosas implica que heredan la incapacidad para volar de dichas cosas; sin embargo, con la propiedad *raro1* podemos cancelar esa herencia.

Al ampliar nuestro conocimiento sobre el mundo podríamos añadir reglas que supongan nuevas excepciones. Por ejemplo, una regla que diga que todo avión cumple la propiedad *raro1*. De esta manera, el conjunto de creencias se amplía añadiendo reglas en vez de modificándolas.

Siguiendo con el ejemplo queremos expresar ahora que, en general, los pájaros vuelan:

$$11.\ pajaro(x) \land \neg raro2(x) \Rightarrow vuela(x)$$

El predicado *raro2* sirve para decir que aunque x sea un pájaro, no sabemos con certeza que pueda volar (por ejemplo, las avestruces cumplen este predicado).

$$12.\ avestruz(x) \Rightarrow raro2(x)$$

En principio las avestruces no vuelan:

$$13.\ avestruz(x) \land \neg raro3(x) \Rightarrow \neg vuela(x)$$

Donde *raro3* nos indica una anormalidad que nos impide deducir que el avestruz en cuestión no vuele. Las avestruces voladoras cumplen esta propiedad:

$$14.\ avestruzV(x) \Rightarrow raro3(x)$$

Las avestruces voladoras vuelan:

15. $avestruzV(x) \Rightarrow vuela(x)$

Incluimos estas fórmulas en Δ_H y nos queda:

4. $avestruzV(x) \Rightarrow avestruz(x)$
3. $avestruz(x) \Rightarrow pajaro(x)$
2. $pajaro(x) \Rightarrow cosa(x)$
1. $cosa(piolin)$
10. $pajaro(x) \Rightarrow raro1(x)$
12. $avestruz(x) \Rightarrow raro2(x)$
14. $avestruzV(x) \Rightarrow raro3(x)$

Y reemplazamos Δ_P por las nuevas propiedades que hemos enunciado:

9. $cosa(x) \wedge \neg raro1(x) \Rightarrow \neg vuela(x)$
11. $pajaro(x) \wedge \neg raro2(x) \Rightarrow vuela(x)$
13. $avestruz(x) \wedge \neg raro3(x) \Rightarrow \neg vuela(x)$
15. $avestruzV(x) \Rightarrow vuela(x)$

Podemos ahora realizar una completación paralela de Δ_H con respecto al conjunto de predicados

$$\Pi = \{avestruzV < avestruz < pajaro < cosa < raro3 < raro2 < raro1\}$$

Nótese que Δ_H es ordenado en dicho conjunto. Esto nos hará asumir que los únicos objetos que son raros, pájaros, cosas, avestruces o avestruces voladores son aquellos que Δ_H nos dice que lo son. Al hacerlo nos queda:

16. $cosa(x) \Rightarrow pajaro(x) \vee x{=}piolin$
17. $pajaro(x) \Rightarrow avestruz(x)$
18. $avestruz(x) \Rightarrow avestruzV(x)$
19. $\neg AvestruzV(x)$
20. $raro1(x) \Rightarrow pajaro(x)$
21. $raro2(x) \Rightarrow avestruz(x)$
22. $raro3(x) \Rightarrow avestruzV(x)$

El único objeto que se menciona es Piolín y estas cláusulas nos dicen que no hay otros objetos y, por lo tanto, no hay ni pájaros ni avestruces ni avestruces voladoras, tampoco hay ningún objeto con anormalidad. Usando la regla 19, podemos deducir $\neg avestruzV(piolin)$. Luego, $\neg avestruz(piolin)$, $\neg pajaro(piolin)$, $\neg raro1(piolin)$, y, finalmente, $\neg vuela(piolin)$.

3.3.1. Completación Entera

Nótese que para realizar la completación de predicados hemos omitido por completo el conjunto Δ_P. Sin embargo, éste nos dice tres fórmulas sobre las anormalidades $raro1$, $raro2$ y $raro3$ que podríamos usar para completar el predicado. Por esto se dice que hemos realizado una *completación delimitada*. El motivo por el que las hemos omitido es porque agregarlas nos da una restricción más débil del universo. Veámoslo: las propiedades 9,11 y 13 pueden reescribirse como:

$$9'.\ cosa(x) \wedge vuela(x) \Rightarrow raro1(x)$$
$$11'.\ pajaro(x) \wedge \neg vuela(x) \Rightarrow raro2(x)$$
$$13'.\ avestruz(x) \wedge vuela(x) \Rightarrow raro3(x)$$

Uniendo estas nuevas reglas con las de Δ_H el conjunto sigue siendo ordenado, luego podemos hacer la completación paralela de predicados. Al hacer la completación de predicados tendríamos entonces que las reglas 20,21,22 se reescribirían como:

$$20'.\ raro1(x) \Rightarrow pajaro(x) \vee (cosa(x) \wedge vuela(x))$$
$$21'.\ raro2(x) \Rightarrow avestruz(x) \vee (pajaro(x) \wedge \neg vuela(x))$$
$$22'.\ raro3(x) \Rightarrow avestruzV(x) \vee (avestruz(x) \wedge vuela(x))$$

Comparemos estas nuevas reglas con las que teníamos antes. Por ejemplo, la regla 20' y la regla 20. Con la regla 20 podíamos deducir que todo objeto con la anormalidad 1 era un pájaro. Ahora, sin embargo, también estamos abiertos a la posibilidad de que un objeto con la anormalidad 1 no sea un pájaro, podría ser una cosa con la capacidad de volar.

3.3.2. Ampliando conocimientos

Si ampliamos los conocimientos que tenemos, estos conjuntos se modificarían y podríamos llegar a nuevas conclusiones. Vamos a ver algún ejemplo.

Vamos a añadir nuevas categorías:

- Los patos, son pájaros que, en general, no se ven afectados por la anormalidad 2, se dirá *pato(x)*;
- *avion(x)*: los aviones son objetos con la anormalidad 1, que en principio vuelan;
- *roto(x)*: expresa que un x está estropeado y, por lo tanto, no puede volar, se aplica a aviones;
- *lucas,donald* son dos patos con una anormalidad, de los cuales sabemos que Donald no puede volar;
- *raro4(x)* indica que un x tiene una anormalidad que no nos permite afirmar que vuela.

Vamos entonces a incluir en Δ_H las nuevas fórmulas que expresan esto, además, vamos a actualizar el conocimiento y expresar que Piolín es un pájaro:

23. x=*donald* $\vee$ x=*lucas* $\Rightarrow$ *pato(x)*
24. *pajaro(piolin)*
25. *avion(x)*$\Rightarrow$ *cosa(x)*
26. *roto(x)*$\Rightarrow$ *avion(x)*
27. *pato(x)* $\Rightarrow$ *pajaro(x)*
28. *avion(x)*$\Rightarrow$ *raro1(x)*
29. x=*donald* $\vee$ x=*lucas* $\Rightarrow$ *raro2(x)*
31. *roto(x)* $\Rightarrow$ *raro4(x)*

Al conjunto Δ_P de las propiedades le queremos añadir que los aviones en general vuelan, que un avión roto no vuela, que Lucas y Donald no vuelan:

32. *avion(x)* $\wedge\neg$ *raro4(x)* $\Rightarrow$ *vuela(x)*
33. *roto(x)* $\Rightarrow \neg$ *vuela(x)*
34. $\neg$*vuela(donald)*

Para realizar la completación paralela en Δ_H, necesitamos unir algunas fórmu-

las entre ellas. Tenemos entonces que Δ_H es:

1. x=*piolin* $\vee$ *pajaro(x)* $\vee$ *avion(x)* $\Rightarrow$ *cosa(x)*
2. *roto(x)* $\Rightarrow$ *avion(x)*
3. x=*piolin* $\vee$ *pato(x)* $\vee$ *avestruz(x)* $\Rightarrow$ *pajaro(x)*
4. x=*lucas* $\vee$ x=*donald* $\Rightarrow$ *pato(x)*
5. *avestruzV(x)* $\Rightarrow$ *avestruz(x)*
6. *pajaro(x)* $\vee$ *avion(x)* $\Rightarrow$ *raro1(x)*
7. x=*lucas* $\vee$ x=*donald* $\vee$ *avestruz(x)* $\Rightarrow$ *raro2(x)*
8. *avestruzV(x)* $\Rightarrow$ *raro3(x)*
9. *roto(x)* $\Rightarrow$ *raro4(x)*

Mientras que Δ_P es:

10. *cosa(x)* $\wedge\neg$*raro1(x)* $\Rightarrow$ $\neg$*vuela(x)*
11. *pajaro(x)* $\wedge\neg$*raro2(x)* $\Rightarrow$ *vuela(x)*
12. *avestruz(x)* $\wedge\neg$*raro3(x)* $\Rightarrow$ $\neg$*vuela(x)*
13. *avestruzV(x)* $\Rightarrow$ *vuela(x)*
14. *avion(x)* $\wedge\neg$*raro4(x)* $\Rightarrow$ *vuela(x)*
15. *roto(x)* $\Rightarrow$ $\neg$*vuela(x)*
16. $\neg$*vuela(donald)*

Vamos ahora a realizar la completación paralela de los predicados

$$roto < raro4 < pato < avion < avestruzV < avestruz < pajaro < cosa < raro3 < raro2 < raro1$$

en Δ_H:

17. *cosa(x)* $\Rightarrow$ x=*piolin* $\vee$ *pajaro(x)* $\vee$*avion(x)*
19. *avion(x)* $\Rightarrow$ *roto(x)*
20. $\neg$*roto(x)*
21. *pajaro(x)* $\Rightarrow$ *pato(x)* $\vee$ *avestruz(x)* $\vee$ x=*piolin*
22. *pato(x)* $\Rightarrow$ x=*donald* $\vee$ x=*lucas*
23. *avestruz(x)* $\Rightarrow$ *avestruzV(x)*
24. $\neg$*avestruzV(x)*
25. *raro1(x)* $\Rightarrow$ *pajaro(x)* $\vee$ *avion(x)*
26. *raro2(x)* $\Rightarrow$ *avestruz(x)* $\vee$ x=*donald* $\vee$ x=*lucas*
27. *raro3(x)* $\Rightarrow$ *avestruzV(x)*
28. *raro4(x)* $\Rightarrow$ *roto(x)*

Hecho todo esto nos podemos preguntar si los objetos que tenemos vuelan o no:

- Sabemos que Donald no vuela por la regla 16.
- Sabemos que Lucas es un pato y, por la regla 3, deducimos que es un pájaro. Sin embargo, como tiene la anormalidad 2 (regla 7), no podemos decir que vuele. Tampoco hay una fórmula que nos diga que no vuele, así que no sabemos si vuela.
- Sabemos que Piolín es un pájaro por la regla 2, no es un avestruz volador por la regla 24, y, por tanto, tampoco es un avestruz por la regla 23. Con la suposición de nombre único sabemos que Piolín no es Lucas ni Donald. Con esto podemos deducir que Piolín no tiene la anormalidad 2 (usando la regla 26); luego, con la regla 11, podemos deducir que Piolín vuela.

Capítulo 4

Programas lógicos positivos

En este capítulo nos adentramos en la Programación Lógica, uno de los paradigmas de programación más importantes dentro de la programación declarativa y del estudio del razonamiento no monótono.

En este capítulo nos centraremos en el caso más fundamental y mejor comprendido: los programas lógicos positivos y la semántica del menor punto fijo asociada. Es importante señalar que estos programas aún gozan de la propiedad de monotonía, pero, más adelante, introduciremos casos de lógica no monótona mediante la inclusión de la negación por defecto en este tipo de programas lógicos.

4.1. Programas lógicos positivos. Definiciones

Definición 4.1.1 (Programa lógico positivo)**.** Se define un programa lógico positivo P como un conjunto finito de cláusulas de primer orden, que llamaremos *reglas*, de la forma

$$A \Leftarrow B_1 \wedge ... \wedge B_m$$

donde $m \geq 0$, y $A, B_1, ..., B_m$ representan fórmulas atómicas de un lenguaje de primer orden L. Se llama *cuerpo de la regla* a la conjunción $B_1 \wedge ... \wedge B_m$ (que podría ser vacía si $m = 0$) y *cabeza de la regla* al átomo A. Si una regla no tiene cuerpo, se dirá que es un *hecho* y se denotará simplemente por A.

Nota 4.1.1. Obsérvese que un programa lógico positivo está formado por un conjunto de cláusulas de Horn con, exactamente, un literal positivo. Aunque a primera vista pueda parecer una restricción fuerte en la capacidad expresiva del lenguaje, hemos vista ya en el capítulo anterior como muchos conjuntos

de creencias que modelan parcelas de la realidad pueden reescribirse como conjuntos de cláusulas he Horn.

Notación. En general, representaremos las reglas $A \Leftarrow B_1 \wedge ... \wedge B_m$ de la siguiente forma: $A \leftarrow B_1, ..., B_n$. Por otra parte, como es costumbre en la programación lógica, a partir de ahora escribiremos las variables que puedan aparecer en las reglas de un programa lógico con la letra mayúscula.

Una regla puede ser vista esencialmente como una instrucción de tipo `if ..., then ...`, comúnmente usada en programación. Por ejemplo, la regla

$$conectado(sevilla) \leftarrow llegada(madrid), enlazados(madrid, sevilla)$$

se puede interpretar como "Si un avión llega a Madrid y Madrid y Sevilla están enlazados, entonces el avión está conectado con Sevilla".

Definición 4.1.2 (Reglas básicas)**.** Diremos que una regla de un programa lógico P es *básica* si solo contiene fórmulas atómicas básicas, (es decir, si no contiene variables).

Ejemplo 4.1.1. Dado el lenguaje $L = \{a, b, c, d, p, f\}$, con p un predicado 1-ario y f una función 2-aria, se tiene que:

- a es un término básico.
- $f(c, a)$ es un término básico.
- $f(X, a)$ no es un término básico puesto que contiene una variable.
- $p(d)$ es una fórmula atómica básica.
- $p(X)$ no es una fórmula atómica básica puesto que contiene una variable.
- $p(f(a, b)) \leftarrow p(a), p(b)$ es una regla básica.
- $p(f(X, Y)) \leftarrow p(X), p(Y)$ no es una regla básica puesto que contiene variables

El ejemplo antes dado de conectividad es una regla básica (sin variables), pero un programa lógico puede perfectamente contener reglas con variables; por ejemplo, reescribiendo la misma regla de forma más general como sigue:

$$conectado(Y) \leftarrow llegada(X), enlazados(X, Y)$$

Podemos pensar en un programa lógico como en una descripción declarativa de un determinado escenario en el cual ciertas reglas, de carácter general (con variables) o específico (reglas básicas), han de cumplirse. Determinar con precisión cuál es dicho escenario asociado a un programa P es la tarea de asociar a cada programa su contenido semántico. Para ello, serán fundamentales las siguientes definiciones.

Definición 4.1.3 (Universo de Herbrand)**.** Dado un programa lógico positivo P y su lenguaje L, se define el *universo de Herbrand de* P, denotado $HU(P)$, como el conjunto de todos los términos básicos de L.

Definición 4.1.4 (Base de Herbrand)**.** Dado un programa lógico positivo P y su lenguaje L, se define la *Base de Herbrand de* P, denotado $HB(P)$, como el conjunto de todas las fórmulas atómicas básicas del lenguaje L.

Definición 4.1.5 (Interpretación de Herbrand)**.** Por último, una *interpretación de Herbrand de* P (o, simplemente, interpretación) es cualquier subconjunto I de $HB(P)$.

Ejemplo 4.1.2. Consideremos, de nuevo, el mundo de las cajas. Esta vez formalizado con las siguientes reglas que constituyen un programa lógico positivo P:

caja(a)
caja(b)
caja(c)
caja(d)
caja(e)
caja(f)
sobre(b,a)
sobre(c,b)
sobre(f,e)
$encima(X,Y) \leftarrow sobre(X,Y)$
$encima(X,Z) \leftarrow sobre(X,Y), encima(Y,Z)$

Las nueve primeras reglas son hechos que indican que los objetos son *cajas* y qué cajas están sobre otras. Las dos últimas reglas forman una definición recursiva de la propiedad *encima*.

Salvo que se especifique otra cosa, entenderemos que el lenguaje asociado a un programa P es el menor lenguaje de primer orden que perite escribirlo. En este caso, $L = \{a, b, c, d, e, f, sobre/2, encima/2\}$. Por tanto, $HU(P) =$

$\{a,b,c,d,e,f\}$, no hay más términos puesto que no hay ningún símbolo de función en el lenguaje. Por otra parte $HB(P) =$

$$\{caja(a), caja(b), ..., caja(f),$$
$$sobre(a,a), sobre(a,b), ..., sobre(f,f),$$
$$encima(a,a), encima(a,b), ..., encima(f,f)\}$$

Ejemplos de interpretaciones son:

$$I_1 = \{caja(a), sobre(a,a)\}$$
$$I_2 = \emptyset$$
$$I_3 = HB(P)$$

Puesto que la base de Herbrand está formado por 105=7+49+49 elementos, existen un total de 2^{105} interpretaciones de Herbrand para P.

Definición 4.1.6 (Instancia básica)**.** Sea un lenguaje L. Dada una cláusula C y el conjunto de las variables que aparecen en ella $Var(C)$, se dice que C' es una *instancia básica de* C si se puede obtener de C aplicando una asignación de variables y en C' no hay variables. Se denota $grnd(C)$ al conjunto de todas las instancias básicas de C.

Definición 4.1.7. Dado un programa P, se define $grnd(P) = \bigcup_{C \in P} grnd(C)$.

Nota 4.1.2. Intuitivamente, $grnd(C)$ permite materializar los términos de los que habla una cláusula, de hecho, se puede pensar en C como un atajo que denotamos para hablar de un conjunto de cláusulas $grnd(C)$.

Ejemplo 4.1.3. Volviendo al mundo de las cajas de antes, $grnd(P)$ estaría

formado por:

$$
\begin{array}{l}
\textit{caja(a)} \\
\textit{caja(b)} \\
\textit{caja(c)} \\
\vdots \\
\textit{sobre(b,a)} \\
\textit{sobre(c,b)} \\
\textit{sobre(f,e)} \\
\textit{encima(a,a)} \leftarrow \textit{sobre(a,a)} \\
\textit{encima(a,b)} \leftarrow \textit{sobre(a,b)} \\
\vdots \\
\textit{encima(f,f)} \leftarrow \textit{sobre(f,f)} \\
\textit{encima(a,a)} \leftarrow \textit{sobre(a,a),encima(a,a)} \\
\textit{encima(a,b)} \leftarrow \textit{sobre(a,a),encima(a,b)} \\
\vdots \\
\textit{encima(f,f)} \leftarrow \textit{sobre(f,f),encima(f,f)}
\end{array}
$$

Para calcular el número de fórmulas que hemos generado, obsérvese que los hechos se quedan sin modificar y que las reglas con variables en cambio aparecen ahora x^n veces, siendo x el número de símbolos de constantes que tiene el lenguaje L y n el número de variables que tiene la regla. En este caso, por ejemplo, tenemos 9 hechos (6 nos indican que son cajas y otros 3 son las de la relación *sobre*). La regla *encima(X,Y)* $\leftarrow$ *sobre(X,Y)* nos da $6^2 = 36$ fórmulas; y, por otra parte, la regla *encima(X,Z)* $\leftarrow$ *sobre(X,Y),encima(Y,Z)* nos trae $6^3 = 216$ nuevas reglas. Tenemos entonces 261 reglas en total, con la ventaja de que ninguna contiene variables.

Intuitivamente, una interpretación de Herbrand puede verse como un conjunto que contiene aquellos átomos básicos que son verdaderos en un escenario dado. Ello justifica la siguiente definición.

Definición 4.1.8 (Modelo de Herbrand)**.** Dada una interpretación I se dice que:

- I es un *modelo de Herbrand* de una cláusula básica $C = A \leftarrow B_1, ..., B_m$ si $A \in I$, o si $\{B_1, ..., B_m\} \not\subseteq I$. Se denota $I \models_H C$.
- I es un *modelo de Herbrand* de una cláusula C si $I \models_H C'$ para toda instancia básica C' de C. Se denota $I \models_H C$.

- I es un *modelo de Herbrand* de un programa P si $I \models_H C$ para todas las cláusulas $C \in P$.

Nota 4.1.3. Una interpretación de Herbrand para un programa P puede verse de manera canónica como una L-estructura general en el sentido de la lógica de primer orden tal y como se definió en el capítulo 2. Además es bien conocido que un conjunto de cláusulas tiene un modelo (en el sentido general) si y solo si tiene un modelo de Herbrand.

Volvamos al ejemplo del mundo de las cajas. $I_1 = HB(P)$ claramente es un modelo de Herbrand del programa puesto que la cabeza de cada regla siempre en I.

Otro ejemplo de modelo de Herbrand es $I_2 = HB(P) \setminus \{sobre(X, X) | X \in \{a, b, c, d\}\}$ puesto que ninguna caja está sobre ella misma.

Finalmente, otro ejemplo de modelo de Herbrand es $I_3 =$

$$\{caja(a), caja(b), ..., caja(f),$$

$$sobre(b, a), sobre(c, b), sobre(f, e),$$

$$encima(c, a), encima(c, b), encima(b, a), encima(f, e)\}$$

4.2. Modelos Mínimos

En general, existen varias interpretaciones que modelan un programa P. Sin embargo, una interpretación que modela un programa P puede contener información de más que no está indicada por P. Éste el caso en el ejemplo de las cajas de $I_1 = HB(P)$. I_1 modela a P, pero contiene información que no se deduce directamente de P (por ejemplo, *encima(e,f)*) y, por lo tanto, no modela de manera satisfactoria el dominio de trabajo.

El objetivo de este apartado es encontrar una interpretación I para cada programa lógico positivo P, tal que I modele a P y no contenga información que no se pueda deducir de P. Esto se consigue creando una interpretación que sólo contenga la información que "necesariamente debe ser cierta" por P. Esta noción de "necesidad" se llama comúnmente interpretaciones *justificadas*. En lugar de dar una definición formal, veamos algunos ejemplos.

Ejemplo 4.2.1. Consideremos el programa P:

$$a \leftarrow b.\ b \leftarrow c.\ c.$$

En este ejemplo, la interpretación $I_1 = \{a, b, c\}$ es justificada: c debe ser cierto porque está en un hecho, lo que implica b, y, por tanto, a. Por otra parte, la interpretación $I_2 = \{c\}$ es justificada pero no es un modelo de P.

Consideremos ahora el programa P':

$$a \leftarrow b.\ b \leftarrow a.\ c.$$

En este ejemplo, la interpretación $I_1 = \{a, b, c\}$ es un modelo de P' pero no es justificado, ya que ninguna regla fuerza el hecho de que aparezcan a y b. Sin embargo, el modelo $I_2 = \{c\}$ sí es justificado.

Esta intuición puede trasladarse a una definición formal mediante el concepto de modelo mínimo (preferimos los modelos con el menor número posible de átomos en ellos).

Definición 4.2.1. Un modelo de Herbrand I de un programa P se dirá *mínimo* si no existe un modelo de Herbrand J de P tal que $J \subset I$.

La existencia de modelos mínimos se sigue de manera directa de la siguiente observación.

Proposición 4.2.1. Si I y J son modelos de Herbrand de un programa lógico positivo P, entonces $I \cap J$ es también modelo de P.

Demostración: Sea $C = A \leftarrow B_1, ..., B_m$ una instancia básica de una regla de P. Por hipótesis, tanto $I \models_H C$ como $J \models_H C$.
Caso 1: $A \in I \cap J$. Entonces, es claro que $I \cap J \models_H C$.
Caso 2: $A \notin I \cap J$. Entonces, o bien $\{B_1, ..., B_m\} \not\subseteq I$ o bien $\{B_1, ..., B_m\} \not\subseteq J$. En cualquier caso se tiene que $\{B_1, ..., B_m\} \not\subseteq I \cap J$ y, por tanto, $I \cap J \models_H C$ por definición. □

Teorema 4.2.1. Todo programa lógico positivo P tiene un único modelo mínimo (el cual se denotará por $LM(P)$).

Demostración: Basta observar que

$$LM(P) = \bigcap_i \{I_i \mid I_i \text{ es un modelo de Herbrand de } P\}$$

en vista de la Proposición anterior. □

El menor modelo de Herbrand de un programa lógica positivo P será pues su interpretación canónica.

4.2.1. Operador de consecuencia inmediata

En esta sección veremos cómo obtener el menor modelo de Herbrand de un programa lógico positivo de manera constructiva mediante un proceso iterativo. Para ello, a cada programa P le asociaremos un operador T_P que actúa sobre la base de Herbrand de P.

Definición 4.2.2. Dado un programa P se define el *operador de consecuencia inmediata* $T_P : \mathcal{P}(HB(P)) \longrightarrow \mathcal{P}(HB(P))$ como sigue:

$$T_P(I) = \{A \mid \exists (A \leftarrow B_1, ..., B_m) \in grnd(P) \text{ con } \{B_1, ..., B_m\} \subseteq I\}$$

Asimismo, se define $T_P^0 = \emptyset$ y $T_P^{i+1} = T_P(T_P^i)\ \forall i \geq 0$.

El resultado fundamental es el siguiente teorema.

Teorema 4.2.2. Sea P un programa lógico positivo. El operador T_P tiene un mínimo punto fijo, que denotaremos $mpf(P)$, y la sucesión $\{T_P^i\}_{i\geq 0}$ converge a $mpf(P)$ (es decir, se cumple $mpf(P) = \bigcup_{i\geq 0} T_P^i$).

La demostración de este Teorema se basa en el Teorema del Punto Fijo de Kleene, que veremos más adelante. Antes detallamos algunas propiedades interesantes del operador T_P.

Nota 4.2.1. Dado un programa lógico positivo P, se tiene que:

- una interpretación I es un modelo de P si y solo si $T_P(I) \subseteq I$.
- una interpretación I está justificada por P si y solo si $I \subseteq T_P(I)$.
- combinando estos dos hechos, obtenemos que una interpretación I es un modelo justificado de P si y solo si $I = T_P(I)$.

Corolario 4.2.1. Sea P un programa lógico positivo. Entonces, se cumple que $mpf(P) = LM(P)$.

4.2.2. Mínimo punto fijo de T_P

En esta sección ofrecemos una prueba del Teorema 4.2.2 anterior y obtendremos, en particular, un método iterativo para el cálculo del menor modelo de Herbrand de un programa lógico positivo.

Como mencionamos previamente, el Teorema 4.2.2 es un caso particular del Teorema del Punto Fijo de Kleene, que es válido en el marco general de los retículos completos. Daremos pues la prueba del Teorema 4.2.2 en este contexto más general. Comenzamos con un puñado de definiciones básicas.

Definición 4.2.3 (Orden parcial)**.** Dado un conjunto no vacío U y una relación binaria R sobre U, R se dice *de orden parcial* si cumple las siguientes propiedades:

- Propiedad Reflexiva: $\forall x \in U, xRx$.

- Propiedad Antisimétrica: $\forall x, y \in U, xRy \wedge yRx \Rightarrow x = y$.
- Propiedad Transitiva $\forall x, y, z \in U, xRy \wedge yRz \Rightarrow xRz$.

Se llama *conjunto parcialmente ordenado* a la dupla (U, R).

Nota 4.2.2. En lo que sigue, escribiremos $x \leq y$ en lugar de xRy.

Definición 4.2.4 (Cota superior)**.** Dados un conjunto parcialmente ordenado $(U, \leq)$ y un subconjunto $W \subseteq U$, se llama *cota superior* de W a todo elemento $v \in U$ tal que $w \leq v \ \forall w \in W$.

Definición 4.2.5 (Cota inferior)**.** Dados un conjunto parcialmente ordenado $(U, \leq)$ y un subconjunto $W \subseteq U$ se llama *cota inferior* de W a todo elemento $v \in U$ tal que $v \leq w \ \forall w \in W$.

Definición 4.2.6 (Ínfimo)**.** Dados un conjunto parcialmente ordenado $(U, \leq$), un subconjunto $W \subseteq U$ y una cota inferior de W, α, se dice que α es *ínfimo* de W si cumple que $v \leq \alpha$ para toda cota inferior v de W.

Definición 4.2.7 (Supremo)**.** Dados un conjunto parcialmente ordenado $(U, \leq)$, un subconjunto $W \subseteq U$ y una cota superior de W, β, se dice que β es *supremo* de W si cumple que $\beta \leq v$ para toda cota superior v de W.

Definición 4.2.8 (Retículo completo)**.** Un conjunto parcialmente ordenado $(U, \leq)$ se dice un *retículo completo* si todo $W \subseteq U$ no vacío tiene un supremo y un ínfimo.

Nota 4.2.3. Sea P un programa lógico positivo. Si tomamos $U = \mathcal{P}(HB(P))$ y $\leq = \subseteq$, entonces $(\mathcal{P}(HB(P)), \subseteq)$ es un conjunto parcialmente ordenado. Además, si $W \subseteq \mathcal{P}(HB(P))$ no vacío, se tiene:

- $sup(W) = \bigcup_{I_i \in W} I_i$
- $inf(W) = \bigcap_{I_i \in W} I_i$

Corolario 4.2.2. Sea P un programa lógico positivo. Entonces, se tiene que $(\mathcal{P}(HB(P)), \subseteq)$ es un retículo completo.

Definición 4.2.9 (Función monótona)**.** Una función $T : (U, \leq) \longrightarrow (U, \leq)$ se dice *monótona* si $\forall x, y \in U, \ x \leq y \Rightarrow T(x) \leq T(y)$.

Nota 4.2.4. Sea P un programa lógico positivo. Entonces, el operador de consecuencia inmediata T_P es un operador monótono.

En efecto, sean $I \subseteq J$ dos interpretaciones de Herbrand para P. Sea $A \in T_P(I)$. Por definición de T_P, existe $A \to B_1, ..., B_m \in ground(P)$ tal que $\{B_1, ..., B_m\} \subseteq I$. Entonces, $\{B_1, ..., B_m\} \subseteq J$ y, por tanto, $A \in T_P(J)$.

El Teorema del Punto Fijo de Knaster-Tarski nos permite garantizar ya la existencia del menor punto fijo del operador T_P.

Teorema 4.2.3 (**Knaster-Tarski**)**.** Todo operador monótono T en un retículo completo $(V, \leq)$ tiene un punto fijo mínimo $mfp(T)$, y

$$mpf(T) = inf(\{x \in V | T(x) \leq x\}).$$

Demostración: Sea un retículo completo $(V, \leq)$ y sea $W = \{x \in V \mid T(x) \leq x\}$. Entonces W es no vacío (el elemento máximo de V está en W) y, por definición de retículo completo, existe $I = inf(W)$.

Supongamos primero $T(I) < I$. Entonces, por monotonía de T, $T(T(I)) \leq T(I)$. Luego $T(I) \in W$, lo que entra en contradicción con que I sea el ínfimo de W.

Supongamos ahora $T(I) > I$. Sea $J \neq I, J \in W$ cualquiera. Como I es el ínfimo de W, tenemos $I < J$. Por monotonía de T tenemos $T(I) \leq T(J)$, por tanto:

$$I < T(I) \leq T(J) \leq J$$

Entonces, $T(I)$ es una cota inferior de W, lo que entra en contradicción con que I es el ínfimo de W.

La única posibilidad que queda entonces es $I = T(I)$. Además es fácil ver que I es el menor punto fijo de T. □

Nota 4.2.5. Por el teorema anterior, tenemos entonces asegurada la unicidad y existencia del menor modelo de Herbrand de un programa positivo P. Ahora bien, el método que nos proporciona el teorema para calcularlo no es más que el que ya teníamos antes, en el que decíamos que:

$$LM(P) = \bigcap_i \{I_i : I_i \text{ es un modelo de Herbrand de } P\}$$

puesto que en el retículo completo de las interpretaciones de Herbrand de P el ínfimo coincide con la intersección, y la condición $T_P(I) \leq I$ equivale a que I es modelo de P.

En lo que sigue, vamos a buscar un método más sencillo para encontrar $mpf(P)$.

Definición 4.2.10 (Conjunto dirigido)**.** Sea $(V, \leq)$ un conjunto parcialmente ordenado. Un conjunto $W \subseteq V$ es *dirigido* si para cualquier par $x, y \in W$ existe un $z \in W : x \leq z, y \leq z$.

Definición 4.2.11 (Operador continuo)**.**

1. Un operador $T : (V, \leq) \to (V, \leq)$ en un retículo completo se dice *continuo* si
$$T(sup(W)) = sup(\{T(x)|x \in W\})$$
para todo conjunto dirigido $W \subseteq V$ no vacío.

2. Se define $T^0 = inf(V)$ y $T^{i+1} = T(T^i)$ para cada $i \geq 0$.

Nota 4.2.6. La definición de conjunto dirigido se expresa para un par de elementos x, y de W. Pero es fácil ver por inducción que, dado un conjunto parcialmente ordenado $(V, \leq)$ y un conjunto dirigido W, se tiene la siguiente propiedad más general:

> Dados p elementos de W, $a_1, ..., a_p$, existe un elemento $w \in W$ tal que $w \geq a_i$ para todo $i = 1, ..., p$.

Proposición 4.2.2. Dado un programa lógico positivo P, el operador T_P es continuo.

Demostración: Sea un subconjunto dirigido $W \subseteq \mathcal{P}(HB(P))$ no vacío.

- Por una parte, $sup(W) \geq I \ \forall I \in W$. Luego $T_P(sup(W)) \geq T_P(I)$ $\forall I \in W$ por monotonía de T_P. Por tanto, $T_P(sup(W)) \geq sup(T_P(W))$.

- Por otra parte, sea $A \in T_P(sup(W))$. Existe una regla básica $A \leftarrow B_1, ..., B_m \in ground(P)$ tal que $\{B_1, ..., B_m\} \subseteq sup(W)$. Luego, existen $I_1, ..., I_m \in W$ tales que $B_i \in I_i$. Puesto que W es dirigido, existe $I_0 \in W$ tal que $I_i \subseteq I_0$ para $1 \leq i \leq m$ y, por tanto, $\{B_1, ..., B_m\} \subseteq I_0$. Tenemos entonces que $A \in T_P(I_0)$ con $I_0 \in W$ y, por tanto, $A \in sup(T_P(W))$. Luego se cumple la segunda desigualdad:
$$T_P(sup(W)) \leq sup(T_P(W))$$

Esto concluye la demostración. □

Teorema 4.2.4 (**Kleene**)**.** Todo operador continuo T en un retículo completo $(V, \leq)$ tiene un mínimo punto fijo, que denotamos $mpf(T)$, y

$$mpf(T) = sup\{T^i | i \geq 0\}.$$

Demostración: Primero veamos que $T^i \leq T^{i+1}\ \forall i \geq 0$ por inducción en i:

- $i = 0$: Se tiene $T^1 \geq T^0$ ya que T^0 es el ínfimo de V.
- $i \Rightarrow i + 1$: Se tiene por hipótesis $T^{i+1} \geq T^i$, luego por monotonía de T se tiene $T(T^{i+1}) \geq T(T^i)$. Finalmente, $T^{i+2} \geq T^{i+1}$.

Sabemos entonces que el conjunto $W = \{T^0, T^1, ...\}$ es un conjunto dirigido, pues dados $T^i, T^j \in W$ se tiene que $T^{\text{máx}\{i,j\}}$ es mayor o igual que ambos. Además, W tiene un supremo ya que $(V, \leq)$ es un retículo completo. Sea entonces $m = sup(W)$.

Como W es un conjunto dirigido y T es un operador continuo, se tiene que $T(sup(W)) = sup(T(W))$. Por tanto, $T(m) = sup(T(W))$.

Por otra parte, $W = T(W) \cup \{T^0\}$. Como T^0 es el ínfimo de V, no interviene en la búsqueda de supremo de W, luego $sup(W) = sup(T(W)) = m$. Luego, $m = T(m)$.

Se tiene además que m es el mínimo punto fijo: supongamos que existe un punto fijo k, vamos a ver por inducción en i que $T^i \leq k\ \forall i \geq 0$

- Si $i = 0$ es trivial ya que T^0 es el ínfimo de V.
- Si tenemos $T^i \leq k$ entonces, por monotonía de T, $T(T^i) \leq T(k)$. Luego, $T^{i+1} \leq T(k) = k$.

Esto concluye la demostración. □

Ejemplo 4.2.2. Volviendo al mundo de las cajas y aplicando el algoritmo tenemos:

$T^0 = \emptyset$
$T^1 = \{caja(a), caja(b), caja(c), ..., caja(f), sobre(b,a), sobre(c,b), sobre(f,e)\}$
$T^2 = T^1 \cup \{encima(b,a), encima(c,b), encima(f,e)\}$
$T^3 = T^2 \cup \{encima(c,a)\}$
$T^4 = T^3$

Como $T^4 = T^3$ sabemos que $T^i = T^3\ \forall i \geq 3$

Vemos que el algoritmo se estabiliza después de 3 pasos (que es la altura de la mayor pila de cajas). T^0, siendo el ínfimo de todas las interpretaciones de

Herbrand posibles para P es claramente el vacío, T_1 se compone por tanto de los hechos del programa (esto es cierto para cualquier programa P), después cada T^i añade la relación *encima* de las cajas que están a una altura i de otra caja.

Ejemplo 4.2.3. Veamos un ejemplo con símbolos de funciones. Esto complica el proceso ya que, al poder aplicar la función infinitas veces, tenemos infinitos términos y, por lo tanto, $grnd(P)$ no es finito. Tomamos el programa P con la constante *0*, el símbolo de función 1-ario siguiente s y la relación 1-aria *natural*:

$$\begin{aligned} &natural(0) \\ &natural(s(X)) \leftarrow natural(X) \end{aligned}$$

En este caso, aplicando el algoritmo obtenemos:

$$\begin{aligned} T^0 &= \emptyset \\ T^1 &= \{natural(0)\} \\ T^2 &= \{natural(0), natural(s(0))\} \\ &\vdots \\ T^i &= \{natural(0), ..., natural(s^{i-1}(0))\} \end{aligned}$$

Como la sucesión $\{T^i\}$ es creciente, sabemos que su supremo es $\lim_{i\to\infty} T^i = \{natural(s^n(0)) \mid n \in \mathbb{N}\}$.
Mediante este algoritmo, se garantiza que se pueda encontrar una solución de forma iterativa; sin embargo, el número de pasos necesario para encontrarla puede ser infinito. Esto puede pasar cuando hay símbolos de funciones ya que crea un número de términos básicos infinitos, lo que a su vez crea un universo de Herbrand de tamaño infinito. Es por ello que en el futuro, en general, evitaremos los símbolos de funciones para nuestros ejemplos.

Nota 4.2.7. Cabe destacar que uno de los requisitos para que el operador T_P sea continuo es que su programa asociado P tenga reglas finitas (esto es, con cuerpos formados por un número finito de átomos). De hecho, vemos a continuación un ejemplo de programa lógico positivo "extendido" P con una regla infinita cuya sucesión T_P^i no converge ni siquiera en infinitos pasos:

$$\begin{aligned} &natural(0) \\ &natural(s(X)) \leftarrow natural(X) \\ &exito \leftarrow natural(s(0)), natural(s^2(0)), natural(s^3(0)), ... \end{aligned}$$

Donde el término éxito mide si el programa ha funcionado. Aquí, al aplicar infinitos pasos, obtenemos

$$\bigcup_{n\in\mathbb{N}} T_P^n = \{natural(0), natural(s(0)), natural(s^2(0)), ...\}$$

pero nunca se llega a incluir el átomo *exito* en el menor punto fijo $\bigcup_{n\in\mathbb{N}} T_P^n$. Por tanto, éste no es un modelo de P ya que incumple la regla 3. Se necesitan en este caso " Infinito + 1 " iteraciones del operador T_P para alcanzar el menor modelo de Herbrand.

Capítulo 5

Negación en los programas lógicos

En este capítulo vamos a ampliar los programas lógicos positivos añadiendo la negación por defecto `not`. Esta negación se distingue de la negación clásica de la lógica de primer orden ya que la expresión `not` A no indicará que el átomo A sea falso, sino que no se sabe que sea cierto. Veremos esta diferencia más en profundidad en el siguiente capítulo, donde introduciremos la negación fuerte en los programas lógicos y compararemos ambas.

El operador de negación por defecto `not` será esencial para nuestro propósito pues es su inclusión en los programas lógicos lo que causa la pérdida de la propiedad de monotonía de la relación de consecuencia lógica asociada.

5.1. Definiciones

Definición 5.1.1. Un *programa lógico normal* (o, simplemente, programa lógico) es un conjunto de cláusulas (o reglas) de la forma:

$$a \leftarrow b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n$$

donde $a, b_i, c_j, i = 1...m, j = 1...n$, son átomos de un lenguaje L de primer orden. Aparecen en el cuerpo de la regla ahora: not c_1, ..., not c_n, que llamamos *literales negados por defecto*; mientras que los $b_1, ..., b_m$ son *literales positivos*. Tanto n como m pueden ser 0.

Intuitivamente, se considera que un literal negado por defecto not c es verdadero si no se puede probar c en un número finito de pasos.

Nos interesa buscar modelos mínimos en este tipo de programas, es decir, una interpretación que modele al programa y no tenga ningún subconjunto

propio que modele a P. Pero veremos que no todos los modelos mínimos nos interesan, pues queremos que no incluyan nada que no estén obligados a creer según P, lo cual no será siempre el caso.

Antes que nada, vamos a proceder a ampliar los conceptos de Universo de Herbrand, Base de Herbrand, Interpretación de Herbrand y modelo de Herbrand a los programas lógicos normales. La definición es esencialmente igual que la de los programas lógicos positivos.

Definición 5.1.2 (Universo de Herbrand)**.** Dado un programa lógico P y su lenguaje L, se define el *universo de Herbrand de* P, denotado $HU(P)$, como el conjunto de todos los términos básicos de L.

Definición 5.1.3 (Base de Herbrand)**.** Dado un programa lógico P y su lenguaje L, se define la *base de Herbrand de* P,denotado $HB(P)$, como el conjunto de todas las fórmulas atómicas del lenguaje L.

Definición 5.1.4 (Interpretación de Herbrand)**.** Dado un programa lógico P, una *interpretación de Herbrand de* P es cualquier subconjunto de $HB(P)$, a menudo la llamaremos interpretación de P a secas.

La siguiente definición es la única que presenta alguna novedad.

Definición 5.1.5 (Modelo)**.** Dada una interpretación I, se dice que I es modelo de:

- una cláusula básica $C : a \leftarrow b_1, .., b_m, \text{not } c_1, .., \text{not } c_n$, con $n, m \geq 0$, si $\{b_1, ..., b_m\} \not\subseteq I$ o si $\{a, c_1, ..., c_n\} \cap I \neq \emptyset$. Se denota $I \models C$.
- Una cláusula C, si $I \models C' \, \forall C' \in grnd(C)$.
- Un programa P, si I es modelo de todas las cláusulas de P.

5.2. Estratificación

El objetivo de la estratificación es dar un orden a seguir para evaluar los predicados de un programa de manera que no se caiga en bucles. No todos los programas lógicos se pueden estratificar pero daremos una condición suficiente para ello.

Definición 5.2.1 (Grafo de dependencia)**.** Dado un programa lógico P, definiremos su *Grafo de Dependencia* como un grafo cuyos vértices son todos los predicados del lenguaje de P y sus aristas son de la siguiente forma:

- existe una arista $p \rightarrow q$ si existe una regla $R \in P$ tal que p está en la cabeza de R y q está en el cuerpo de R.

- existe una *arista negativa* $p \to^{-} q$ si existe una regla $R \in P$ tal que p está en la cabeza de R y not q está en el cuerpo de R.

Definición 5.2.2 (Estratificación)**.** Dado un programa lógico P y su grafo de dependencia G, definimos una *Estratificación* de P como $\Sigma = \{S_i | i \in \{1, ..., n\}\}$ una partición de $pred(P)$ (es decir, cada S_i es un conjunto de predicados de P de manera que los S_i son disjuntos entre ellos y $\bigcup_{i=1}^{n} S_i = pred(P)$) que cumple que:

- si $p \in S_i, q \in S_j$ y $p \to q$ es una arista de G, entonces $i \geq j$.
- si $p \in S_i, q \in S_j$ y $p \to^{-} q$ es una arista de G, entonces $i > j$.

Los conjuntos $S_1, ..., S_n$ se llaman los *estratos* de P. Un programa lógico se dice *estratificable* si tiene alguna estratificación.

Definición 5.2.3 (Base de Herbrand restringida)**.** Dados un programa P y una estratificación $\Sigma = \{S_1, ..., S_k\}$, definimos P_{S_i} como el conjunto de reglas cuya cabeza tiene un predicado en S_i y definimos la *base de Herbrand restringida al estrato* S_i como $HB^*(P_{S_i}) = \bigcup_{j \leq i} \{p(t) \in HB(P) | p \in S_j\}$.

Definición 5.2.4 (Modelo mínimo iterativo)**.** Dados un programa lógico P y una estratificación $\Sigma = \{S_1, ..., S_k\}$ de P, definimos el *modelo mínimo iterativo* $M_i \subseteq HB(P), i = 1, ..., k$ como sigue:

- M_1 es el modelo mínimo de P_{S_1}. Por la definición de estratificación tenemos que P_{S_1} es un programa lógico positivo, así que se puede calcular M_1 con el método visto en el capítulo anterior.
- M_i, para $i > 1$, es el menor subconjunto $M \in HB(P)$ tal que M es un modelo de P_{S_i}, y $M \cap HB^*(P_{S_{i-1}}) = M_{i-1} \cap HB^*(P_{S_{i-1}})$.

Es decir, M_i debe modelar las reglas cuya cabeza esté en el estrato S_i, y además debe tener los elementos del modelo mínimo M_{i-1}. Construyendo iterativamente llegamos a que en la última iteración tenemos un modelo mínimo de P. Con esto tenemos asegurada la existencia de modelo mínimo para programas estratificables El modelo M_k se denota $M_{P,\Sigma}$.

Ejemplo 5.2.1. Consideremos el programa lógico P:

$$\begin{aligned} & seta(oronja) \\ comestible(X) \leftarrow \quad & seta(X), buena(X) \\ venenoso(X) \leftarrow \quad & seta(X), \text{not } buena(X) \end{aligned}$$

Una estratificación válida para P es $S_1 = \{seta, buena\}$, $S_2 = \{comestible\}$ y $S_3 = \{venenoso\}$. Es claro que $M_1 = \{seta(oronja)\}$. Nótese ahora que

$$P_{S_2} = \{comestible(X) \leftarrow seta(X), buena(X)\}$$

$$HB^*(P_{S_1}) = \{seta(oronja), buena(oronja)\}$$

Entonces, $M_2 = \{seta(oronja)\}$ es modelo de P_{S_2}, y $M_2 \cap HB^*(P_{S_1}) = M_1 \cap HB^*(P_{S_1})$, además es el menor subconjunto con estas propiedades. Finalmente tenemos

$$P_{S_3} = \{venenoso(X) \leftarrow seta(X), \text{not } buena(X)\}$$

$$HB^*(P_{S_2}) = \{seta(oronja), buena(oronja), comestible(oronja)\}$$

Entonces, $M_3 = \{seta(oronja), venenoso(oronja))\}$ es el menor modelo de P_{S_3} que cumple $M_3 \cap HB^*(P_{S_2}) = M_2 \cap HB^*(P_{S_2})$.

En definitiva, $M_{P,\Sigma} = \{seta(oronja), venenoso(oronja))\}$.

Otra estratificación de P es $S_1 = \{seta, buena, comestible\}, S_2 = \{venenoso\}$. En este caso tenemos

$$P_{S_1} = \{seta(oronja); comestible(X) \leftarrow seta(X), buena(X)\}$$

y obtenemos $M_1 = \{seta(oronja)\}$.

En la siguiente iteración tenemos

$$P_{S_2} = \{venenoso(X) \leftarrow seta(X), \text{ not } buena(X)\}$$

y $M_2 = \{venenoso(oronja)\} \cup M_1$ es el menor modelo de P_{S_2} que cumple $M_2 \cap HB^*(P_{S_1}) = M_1 \cap HB^*(P_{S_1})$.

Nótese que en ambas estratificaciones el resultado final es el mismo. Esto no es casualidad; lo vemos de forma más general en el siguiente teorema.

Teorema 5.2.1. Sea P un programa estratificable. Para cada estratificación Σ y Σ' se tiene $M_{P,\Sigma} = M_{P,\Sigma'}$. Denotaremos entonces M_P el mínimo modelo de P mediante cualquier estratificación Σ. Esto nos garantiza la unicidad del modelo mínimo de un programa estratificable, por eso lo llamamos el *modelo perfecto de* P.

Ejemplo 5.2.2. Consideremos el programa lógico P', que considera que una seta que no es venenosa es comestible, y una seta que no es comestible es venenosa:

$$seta(oronja)$$
$$venenoso(X) \leftarrow seta(X), \text{not } comestible(X)$$
$$comestible(X) \leftarrow seta(X), \text{not } venenoso(X)$$

Este programa no es estratificable ya que, de serlo, *comestible* debería estar en un estrato anterior al estrato de *venenoso* y *venenoso* debería estar en uno anterior al de *comestible*, lo cual es imposible. De hecho, el programa P' no tiene un único modelo minimal. Siguiendo la definición intuitiva de modelo mínimo, serían válidos tanto $N = \{seta(oronja), comestible(oronja)\}$ como $M = \{seta(oronja), venenoso(oronja)\}$.

5.3. Modelos estables

La semántica de los modelos estables para los programas lógicos con negación por defecto fue introducida por M. Gelfond and V. Lifschitz en 1988 (*The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference on Logic Programming (ICLP), pages 1070–1080*) y en la actualidad es una de las aproximaciones clásicas a la semántica de dichos programas. Veamos las definiciones básicas.

Definición 5.3.1 (Reducción de un programa)**.** Dado un programa lógico sin variables P y una interpretación de Herbrand M para P, se define la *reducción o el reducto de P respecto de M*, denotado P^M, como el programa lógico obtenido de la siguiente forma:

- quitando todas las reglas de P que contengan not a en su cuerpo, para cada átomo $a \in M$.
- quitando todos los literales negados por defecto de las restantes reglas de P.

La idea de esta definición es que si un átomo a está en la interpretación M, entonces una regla que contenga not a en su cuerpo es automáticamente verdadera (puesto que su cuerpo será falso). Para el resto de las reglas, todos los literales negados por defecto son verdaderos, así que no aportan nada y pueden ser eliminados.

Es importante notar que P^M siempre va a resultar en un programa lógico positivo, luego tendrá un modelo mínimo $LM(P^M)$ por los resultados del capítulo anterior. Esto justifica la siguiente definición:

Definición 5.3.2 (Modelo estable)**.** Dado un programa lógico P, se dice que una interpretación de Herbrand M para P es un *modelo estable de P* si $M = LM(ground(P)^M)$.

Nótese que si P es un programa lógico positivo, entonces $ground(P) = ground(P)^M$ para cualquier interpretación M; luego esta definición de modelo estable expande la definición de modelo mínimo para los programas lógicos positivos.

Ejemplo 5.3.1. Volviendo al ejemplo P:

$$\begin{aligned} seta(oronja) \\ venenoso(X) \leftarrow seta(X), \text{not } comestible(X) \\ comestible(X) \leftarrow seta(X), \text{not } venenoso(X) \end{aligned}$$

consideremos las siguientes interpretaciones:

- $M_1 = \{seta(oronja), venenoso(oronja)\}$
- $M_2 = \{seta(oronja), comestible(oronja)\}$
- $M_3 = \{seta(oronja), comestible(oronja), venenoso(oronja)\}$
- $M_4 = \{seta(oronja)\}$

Veamos que sólo las dos primeras son modelos estables de P:

- Empezemos con M_1, tenemos que $ground(P)^{M_1}$ es:

  $$\begin{aligned} seta(oronja) \\ venenoso(oronja) \leftarrow seta(oronja) \end{aligned}$$

  cuyo modelo mínimo es $M = \{seta(oronja), venenoso(oronja)\}$. Se cumple $M = M_1$, luego M_1 es un modelo estable de P.

- Análogamente, tenemos que $ground(P)^{M_2}$ es:

  $$\begin{aligned} seta(oronja) \\ comestible(oronja) \leftarrow seta(oronja) \end{aligned}$$

  cuyo modelo mínimo es $M = \{seta(oronja), comestible(oronja)\}$. Se cumple $M = M_2$, luego M_2 es otro modelo estable de P.

- Veamos que M_3 no es un modelo estable: para crear $ground(P)^{M_3}$ debemos quitar la segunda y tercera regla de P, nos queda pues:

  $$seta(oronja)$$

  cuyo modelo mínimo es $M = \{seta(oronja)\}$. Como $M \neq M_3$ tenemos que M_3 no es un modelo estable de P.

- M_4 tampoco es un modelo estable de P, pues $ground(P)^{M_4}$ es:

$$\begin{aligned} seta(oronja) \\ venenoso(oronja) \leftarrow seta(oronja) \\ comestible(oronja) \leftarrow seta(oronja) \end{aligned}$$

  cuyo modelo mínimo es

$$M = \{seta(oronja), venenoso(oronja), comestible(oronja)\}$$

  y $M \neq M_4$.

Ejemplo 5.3.2 (Gelfond&Lifschitz)**.** Este ejemplo fue el usado por M. Gelfond y V. Lifschitz para introducir por primera vez el concepto de modelo estable en su artículo de 1988. Vamos a buscar los modelos estables para el siguiente programa P:

$$p(1,2) \tag{5.1}$$
$$q(X) \leftarrow p(X,Y), \text{not } q(Y) \tag{5.2}$$

Al llevarlo a su forma básica, obtenemos que $ground(P)$ está formado por:

$$p(1,2) \tag{5.3}$$
$$q(1) \leftarrow p(1,1), \text{not } q(1) \tag{5.4}$$
$$q(1) \leftarrow p(1,2), \text{not } q(2) \tag{5.5}$$
$$q(2) \leftarrow p(2,1), \text{not } q(1) \tag{5.6}$$
$$q(2) \leftarrow p(2,2), \text{not } q(2) \tag{5.7}$$

Este programa no es estratificable (ni siquiera de forma "local"), puesto que de serlo, el predicado $q(1)$ debería estar en un estrato anterior a $q(2)$ y, por otra parte, $q(2)$ debería estar en un estrato anterior al de $q(1)$.

Existen 2^6 interpretaciones entre las cuales buscar un modelo estable; sin embargo, para que una interpretación M sea modelo del programa, está claro que debe contener $p(1,2)$, pues aparece como hecho. Como buscamos un modelo mínimo, obviamos los otros $p(X,Y)$ puesto que no salen en la cabeza de ninguna regla y, por lo tanto, no se pueden deducir. Así que la única regla de la que se puede deducir algo es la 5.5. Nuestro candidato a modelo estable es entonces $M = \{q(1), p(1,2)\}$. Para comprobar si realmento lo es, calculemos el reducto $ground(P)^M$:

$$\begin{aligned} p(1,2) \\ q(1) \leftarrow p(1,2) \\ q(2) \leftarrow p(2,2) \end{aligned}$$

cuyo modelo mínimo es M. Luego M es un modelo estable.

Otro candidato sería $M_1 = \{q(2), p(1,2)\}$. Sin embargo, la reducción $grond(P)^{M_1}$ es:

$$p(1,2)$$
$$q(1) \leftarrow p(1,1)$$
$$q(2) \leftarrow p(2,1)$$

cuyo modelo es mínimo es $\{p(1,2)\}$ que no coincide con M_1. Luego M_1 no es modelo estable de P y, por tanto, M es el único modelo estable del programa.

Ejemplo 5.3.3. Como un último ejemplo, consideremos el programa P:

$$p(a) \leftarrow \text{not } p(a)$$

La base de Herbrand de P se reduce a $\{p(a)\}$ y, por tanto, solo hay dos candidatos a modelo estable: $M_1 = \emptyset$ y $M_2 = \{p(a)\}$. Ahora bien, es fácil comprobar que $LM(P^{M_1}) = M_2$ y $LM(P^{M_2}) = M_1$. Luego ninguno de los dos es un modelo estable. Estamos pues ante un programa que tiene modelos de Herbrand minimales pero no tiene ningún modelo estable.

A la vista de los tres ejemplos anteriores, un programa lógico P puede tener un único modelo estable, o bien tener varios modelos estables distintos, o bien no tener ninguno.

5.3.1. Propiedades de los modelos estables

En esta sección destacamos algunas propiedades básicas de la clase de modelos que acabamos de introducir.

Teorema 5.3.1. Sea P un programa lógico.

- Un modelo estable M de P es, en particular, un modelo de Herbrand de P.
- (Minimalidad) Un modelo estable M de P no contiene a ningún modelo propio M' de P.

Este teorema nos dice que un modelo estable de P cumple con las condiciones que se esperan de un modelo de Herbarnd mínimo.

Corolario 5.3.1 (No comparabilidad)**.** Dados dos modelos estables M_1, M_2 distintos de un programa P, se tiene $M_1 \not\subseteq M_2$ y $M_2 \not\subseteq M_1$.

Hemos resaltado anteriormente que la definición de modelo estable extiende a la definición de modelo minimal de un programa lógico positivo. Lo mismo ocurre para programas normales estratificados para la noción de modelo perfecto.

Teorema 5.3.2. Si un programa P es estratificable, entonces P tiene un único modelo estable y éste coincide con su modelo perfecto.

Es posible extender de manera natural el operador de consecuencia inmediata T_P para programas lógicos normales.

Definición 5.3.3 (Operador consecuencia inmediata)**.** Dado un programa P, definimos el operador $T_P : \mathcal{P}(HB(P)) \to \mathcal{P}(HB(P))$ como sigue:

$$T_P(I) = \{a \,|\, \exists\, a \leftarrow b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n \in grnd(P) : b_i \in I \wedge c_j \notin I, 1 \leq i \leq m, 1 \leq j \leq n\}$$

Se puede pensar en este operador como la extensión a programas lógicos normales del operador T_P que usamos en programas lógicos positivos para calcular modelos mínimos. Sin embargo, al añadir la negación por defecto, el operador T_P pierde, en general, las buenas propiedades que nos permitían garantizar unicidad y existencia de modelo mínimo.

Ejemplo 5.3.4. Consideremos el programa P:

$$pajaro(piolin)$$
$$vuela(X) \leftarrow pajaro(X), \text{not } raro(X)$$

Consideremos las interpretaciones $M_1 = \{pajaro(piolin)\}$ y, por otra parte, M_2=$\{pajaro(piolin), raro(piolin)\}$. Entonces, se tiene que:

$$T_P(M_1) = \{pajaro(piolin), vuela(piolin)\}$$
$$T_P(M_2) = \{pajaro(piolin)\}$$

Por tanto, $M_1 \subseteq M_2$ y, sin embargo, $T_P(M_1) \not\subseteq T_P(M_2)$. Luego, T_P no es un operador monótono y no podemos aplicar, en general, los teoremas de punto fijo que mencionamos en el capítulo anterior.

Consideremos ahora la interpretación $M_3 = \{pajaro(piolin), vuela(piolin)\}$ para P. Tenemos que $ground(P)^{M_3}$ es el programa:

$$pajaro(piolin)$$
$$vuela(piolin) \leftarrow pajaro(piolin)$$

cuyo modelo mínimo es M_3, así que M_3 es un modelo estable de P. Nótese que si le aplicamos el operador T_P obtenemos $T_P(M_3) = M_3$. Éste es un hecho general.

Teorema 5.3.3. Sea P un programa lógico y M una interpretación para P. Si M es un modelo estable de P, entonces $T_P(M) = M$.

El teorema anterior nos da una condición necesaria para ser modelo estable. El siguiente ejemplo muestra que la condición no es suficiente.

Ejemplo 5.3.5. Veamos un ejemplo en el cual un mínimo punto fijo del operador T_P no es un modelo estable. Sea P el programa siguiente:

$$a \leftarrow \text{not } b$$
$$b \leftarrow c$$
$$c \leftarrow b$$

Consideremos las interpretaciones $M_1 = \{a\}, M_2 = \{b, c\}$. Ambas son modelos mínimos de P, M_1 es modelo estable de P y, sin embargo, M_2 no es modelo estable de P (ya que el modelo mínimo de P^{M_2} es la interpretación vacía). Aún así, aplicando el operador nos queda $T_P(M_2) = M_2$, y ningún subconjunto propio de M_2 es un punto fijo de T_P. Así pues, M_2 es un mínimo punto fijo de T_P pero no es modelo estable de P.

5.4. No monotonía de los programas lógicos normales

Puesto que un programa P puede tener uno o varios modelos estables, introducimos dos versiones de consecuencia lógica para la semántica de los modelos estables.

Definición 5.4.1. Dado un programa lógico P y su lenguaje L

- Se dice que un átomo básico a de L es consecuencia *crédula* de P si existe algún modelo estable M de P tal que $a \in M$. Se denota $P \models_c a$.
- Se dice que un átomo básico a de L es consecuencia *escéptica* de P si para todo modelo estable M de P se cumple $a \in M$. Se denota $P \models_e a$.

Esta definición nos permite diferenciar las conclusiones que siempre van a ser ciertas en un programa de aquellas que lo serán en escenarios concretos. Nótese que un átomo que esté en la cabeza de un hecho siempre va a ser una consecuencia escéptica de P.

Ejemplo 5.4.1. En el ejemplo de las setas (ejemplo 5.3.1), se tiene que $P \models_c venenoso(oronja)$, $P \models_c comestible(oronja)$ y $P \models_e seta(oronja)$, ya que en todos los modelos tenemos $seta(oronja)$, mientras que para el resto de conocimientos depende del modelo estable que escojamos.

Tanto $\models_c$ como $\models_e$ son relaciones no monótonas. Veámoslo.

Ejemplo 5.4.2. Veamos un ejemplo de la no monotonía de estas relaciones: Consideremos el ejemplo de las setas (ejemplo 5.3.1) con las dos primeras reglas nada más:

$$seta(oronja)$$
$$venenoso(oronja) \leftarrow seta(oronja), \text{not } comestible(oronja)$$

En este caso, tenemos un único modelo estable dado por

$$M = \{seta(oronja), venenoso(oronja)\}$$

Luego se tiene tanto $P \models_e venenoso(oronja)$ como $P \models_c venenoso(oronja)$. Sin embargo, al ampliar la información y añadir la nueva regla

$$comestible(oronja)$$

el único modelo estable del programa aumentado es ahora la interpretación $\{seta(oronja), comestible(oronja)\}$. Por tanto:

$$P \cup \{comestible(oronja)\} \not\models_e venenoso(oronja)$$

$$P \cup \{comestible(oronja)\} \not\models_c venenoso(oronja)$$

Capítulo 6

Extensiones de los programas lógicos normales

En este capítulo introducimos tres formas de ampliar los programas lógicos vistos hasta ahora: las restricciones, la negación fuerte y las disyunciones. Veremos asimismo cómo adaptar las nociones anteriores a este marco más general.

6.1. Extensiones

- Lo primero que vamos a añadir son las *restricciones*. Una restricción es una regla de la forma $falso \leftarrow \text{not } falso, b_1, ..., b_n$ con $falso$ una fórmula atómica libre en el programa (esto es, sólo aparece en las reglas que son restricciones de esta manera y en ninguna otra parte del programa lógico). Denotaremos estas reglas como

  $$\leftarrow b_1, ..., b_n$$

  y diremos que no tienen cabeza. Intuitivamente lo que una restricción implica es que alguno de los b_i es necesariamente falso. Efectivamente, si todos fuesen verdaderos, entonces tendríamos $falso \leftarrow \text{not } falso$, regla que nunca puede satisfacerse en un modelo estable.

- Lo segundo que añadiremos es la *negación fuerte*. Se define el complementario de una fórmula atómica b como $-b$ y se diferencia de la negación por defecto en que $-b$ se interpreta como que sabemos que b es falso, mientras que escribiendo not b no tenemos esa certeza, aunque tampoco la tengamos de que b sea verdadero. Formalmente, la negación fuerte para cada fórmula atómica b, se define como una fórmula

atómica con la misma aridad que b, se denota $-b$, y se introduce la restricción:

$$\leftarrow b, -b$$

para evitar que se den al mismo tiempo b y $-b$. Para abreviar la notación, y como por definición, $-b$ cumple esa restricción y estas reglas se omiten.

Veamos un par de ejemplos en los que la negación fuerte es útil:

Ejemplo 6.1.1. Supongamos que queremos expresar "si la seta no es venenosa, entonces nos la comemos". Podemos utilizar una de las siguientes reglas para formalizar la frase:

$$comer(X) \leftarrow seta(X), -venenoso(X)$$
$$comer(X) \leftarrow seta(X), \text{not } venenoso(X)$$

La primera fórmula expresa que si tenemos la certeza de que X no es venenosa, entonces nos la podemos comer. La segunda, en cambio, dice que nos podemos comer la seta si no tenemos constancia de que sea venenosa. En escenarios críticos como éste conviene hacer uso de la negación fuerte.

Otro uso de la negación fuerte es para formalizar la negación por defecto como la vimos en el capítulo 3. Por ejemplo, para expresar que los pájaros normalmente vuelan podemos escribir:

$$vuela(X) \leftarrow pajaro(X), \text{not } -vuela(X)$$

que expresa que si X es un pájaro, a menos que nos conste que X no vuela (caso del avestruz), entonces podemos suponer que X vuela.

Análogamente, se puede usar la negación fuerte para introducir la suposición de mundo cerrado con reglas del tipo $-p \leftarrow \text{not } p$.

- Finalmente, vamos a introducir la disyunción $\vee$ en la cabeza de una regla. Por ejemplo, la regla $hembra(X) \vee macho(X) \leftarrow animal(X)$ nos dice que todo animal es macho o hembra.

Una utilidad de la disyunción es que puede permitir al programa hacer suposiciones. Por ejemplo en el caso:

$$funciona(X) \vee -funciona(X) \leftarrow ordenador(X)$$

Como buscamos modelos mínimos, en general, solo consideraremos una de las dos opciones y haremos suposiciones a partir de ella (eso sí, dependiendo del resto del programa, habrá veces en la que varias componentes de la cabeza estén en el modelo mínimo).

Ejemplo 6.1.2. Con las disyunciones podemos reescribir el programas de la seta venenosa como:

$$seta(oronja)$$
$$comestible(X) \vee venenoso(X) \leftarrow seta(X)$$

Lo que nos permite reducir el tamaño del programa, (expresa que cada seta es venenosa o comestible, lo que nos ahorra expresar que si una seta no es comestible entonces es venenosa y viceversa).

De los tres nuevos mecanismos de los que hemos hablado, sólo las disyunciones son una ampliación real de la capacidad expresiva lenguaje, puesto que los otros dos mecanismos se pueden emular mediante un programa lógico normal. Desarrollemos esta idea en el siguiente ejemplo.

Ejemplo 6.1.3. Vamos a encontrar el modelo mínimo para un programa P, que incluye negación fuerte, con los mecanismos estudiados para un programa lógico normal. Sea el programa P con las reglas siguientes:

$$-venOjos(juan)$$
$$-sienteCorazon(X) \leftarrow -venOjos(X)$$
$$sienteCorazon(X) \leftarrow \text{not } -sienteCorazon(X)$$

el programa sirve para expresar que si los ojos de cierta persona no ven, entonces su corazón no siente, que, en concreto, los ojos de Juan no ven, y que, por defecto, suponemos que el corazón de una persona siente. Podemos reescribirlo como un programa lógico P' con las siguientes reglas:

$$noVenOjos(juan)$$
$$noSienteCorazon(X) \leftarrow noVenOjos(X)$$
$$sienteCorazon(X) \leftarrow \text{not } noSienteCorazon(X)$$
$$falso \leftarrow \text{not } falso, venOjos(X), noVenOjos(X)$$
$$falso \leftarrow \text{not } falso, sienteCorazon(X), noSienteCorazon(X)$$

Un modelo mínimo, un modelo de P' es

$$M = \{noVenOjos(juan), noSienteCorazon(juan)\}$$

y es un modelo estable puesto que $ground(P')^M$ resulta en el siguiente programa:

$$noVenOjos(juan)$$
$$noSienteCorazon(juan) \leftarrow noVenOjos(juan)$$
$$falso \leftarrow venOjos(juan), noVenOjos(juan)$$
$$falso \leftarrow sienteCorazon(juan), noSienteCorazon(juan)$$

del cual M es también modelo mínimo.

Damos ahora una definición más formal de una programa lógico extendido.

Definición 6.1.1 (Programa Lógico extendido)**.** Un programa lógico extendido P es un conjunto finito de reglas de la forma

$$a_1 \vee ... \vee a_l \leftarrow b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n$$

donde a_i, b_j, c_k son literales (admitimos negación fuerte) de un lenguaje de primer orden L y $l \geq 1$, $m \geq 0$, $n \geq 0$.

6.2. Semántica de los programas lógicos extendidos

La semántica de los programas lógicos extendidos no es muy diferente de la de los programas lógicos normales, pues la mayoría de las definiciones no son más que expansiones naturales. La mayor diferencia con las anteriores es que ahora admitimos en las interpretaciones la negación fuerte de una fórmula atómica, es decir, ahora las interpretaciones se componen de literales en vez de fórmulas atómicas. No debe sorprender esta nueva definición puesto que, como hemos visto anteriormente, al introducir la negación fuerte en un programa estamos introduciendo una nueva fórmula atómica para indicar que no se tiene la fórmula atómica original. La definición de modelo también varía de forma intuitiva y acorde a los nuevos mecanismos.

Definición 6.2.1 (Base de Herbrand extendida)**.** Dado un lenguaje L, se llama *base de Herbrand extendida* de L al conjunto de todos los literales básicos de L.

Definición 6.2.2 (Interpretación de Herbrand extendida)**.** Dado un lenguaje L, se llama *interpretación de Herbrand extendida* de L a cualquier subconjunto de la base de Herbrand extendida de L. Solo consideraremos interpretaciones consistentes, es decir, que no contengan a un átomo y a su negación.

Definición 6.2.3 (Modelo)**.** Dado un lenguaje L y una interpretación de Herbrand extendida I, se dice:

- que es modelo de una cláusula básica

$$C : a_1 \vee ... \vee a_l \leftarrow b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n$$

si $\{b_1, .., b_m\} \notin I$ o si $\{a_1, ..., a_k, c_1, ..., c_n\} \cap I \neq \emptyset$. Se denota $I \models C$.

- que es modelo de una cláusula C si $I \models C'$ $\forall C' \in grnd(C)$. Se denota $I \models C$.
- que es modelo de un programa P si $I \models C$ para todas las cláusulas C del programa P. Se denota $I \models P$.

Ejemplo 6.2.1. Sea P el programa conformado por la siguiente regla:

$$p(a) \vee p(b) \leftarrow q(b), -t(c), \text{not } t(b)$$

Su base de Herbrand extendida es

$$HB(P) = \{p(a), p(b), p(c), q(a), q(b), \\ q(c), t(a), t(b), t(c), -p(a), -p(b), ..., -t(c)\}$$

y sean las siguientes interpretaciones:
$S_1 = \{-p(a), q(b), -t(c)\}$
$S_2 = \emptyset$
$S_3 = \{p(a)\}$
$S_4 = \{t(b)\}$
$S_5 = \{q(b), -t(b)\}$
$S_6 = \{p(a), q(b), -t(c)\}$
$S_7 = \{q(b), -t(c)\}$

Vamos a ver, para cada una, si es modelo de P. Para ser modelo de P, una interpretación debe cumplir que $A = \{q(b), -t(c)\} \not\subseteq I$ o bien que $B = \{p(a), p(b), t(b)\} \cap I \neq \emptyset$.

- S_1 no satisface la regla, puesto que no se cumple que $A \not\in S_1$ ni se cumple $B \cap S_1 \neq \emptyset$.
- S_2 satisface la regla puesto que se cumple $A \not\subseteq S_2$.
- S_3 satisface la regla puesto que se cumple $B \cap S_3 = \{p(a)\} \neq \emptyset$.
- S_4 satisface la regla puesto que se cumple $B \cap S_4 = \{t(b)\} \neq \emptyset$.
- S_5 satisface la regla puesto que se cumple $A \not\subseteq S_5$.
- S_6 satisface la regla pueto que se cumple $B \cap S_6 = \{p(a)\} \neq \emptyset$.
- S_7 no satisface la regla puesto $A = S_7$,y por otra parte $B \cap S_7 = \emptyset$.

Definición 6.2.4 (Reducción de un programa)**.** Dado un programa lógico extendido P sin variables y una interpretación M, definimos P^M, la reducción de P respecto de M, como el programa obtenido de la siguiente forma:

- quitando todas las reglas que contengan not a en su cuerpo, para cada literal $a \in M$.
- quitando todos los literales negados por defecto de las otras reglas.

Definición 6.2.5 (Modelo estable)**.** Dado un programa lógico extendido P, se dice que una interpretación M es un modelo estable de P si M es un modelo minimal de $ground(P)^M$.

Nota 6.2.1. Esta definición varía ligeramente de la de modelo estable para programas lógicos en la que M es modelo estable si $M = LM(ground(P)^M)$. En este caso no podemos usarla ya que no tenemos garantía de unicidad de modelos minimales en general por la presencia de la disyunción. Por ejemplo, el programa P con la regla:

$$b \vee a$$

tiene dos modelos minimales: $M_1 = \{a\}$ y $M_2 = \{b\}$.

Al variar ligeramente la definición, algunos autores prefieren llamar a los modelos estables en programas lógicos extendidos *Conjuntos de respuesta*.

Las propiedades básicas de los modelos estables de los programas extendidos son similares a las anteriormente mencionadas.

Teorema 6.2.1. Dado un programa lógico extendido P:

- Todo modelo estable M de P es un modelo de P.
- Un modelo estable M no contiene a ningún modelo propio M' de P.

Corolario 6.2.1. Dados dos modelos estables M_1, M_2 distintos de P, se tiene $M_1 \not\subseteq M_2$ y $M_2 \not\subseteq M_1$.

Proposición 6.2.1. Dado un programa lógico extendido P y S un conjunto de respuesta de P, se cumple:

- S es un modelo minimal de P.
- S es un conjunto "sustentado" de P. Es decir, para cada literal $l \in S$, existe una regla r en $ground(P)$ tal que S satisface el cuerpo de r y l es el único literal en la cabeza de r que satisface S.

Nota 6.2.2. El recíproco no es cierto. Por ejemplo, tomando el programa P:

$$p(a) \leftarrow p(a)$$
$$p(a) \leftarrow \text{not } p(a)$$

la interpretación $\{p(a)\}$ es minimal y justificada. Sin embargo, no es un conjunto de respuesta.

Finalizamos con un ejemplo.

Ejemplo 6.2.2. Consideramos el programa P:

$$\begin{aligned} s(b) & \\ r(a) & \\ p(a) \vee p(b) & \\ q(X) \leftarrow p(X), r(X), & \text{not } s(X) \end{aligned}$$

Tenemos que $grnd(P)$ es:

$$\begin{aligned} s(b) & \\ r(a) & \\ p(a) \vee p(b) & \\ q(a) \leftarrow p(a), r(a), & \text{not } s(a) \\ q(b) \leftarrow p(b), r(b), & \text{not } s(b) \end{aligned}$$

Tiene dos modelos estables:

$$M_1 = \{s(b), r(a), p(a), q(a)\}$$

$$M_2 = \{s(b), r(a), p(b)\}$$

En efecto, $ground(P)^{M_1}$ es el programa siguiente:

$$\begin{aligned} s(b) & \\ r(a) & \\ p(a) \vee p(b) & \\ q(a) \leftarrow p(a), & r(a) \end{aligned}$$

Que resulta ser el mismo programa que $ground(P)^{M_2}$. Este programa tiene dos modelos mínimos: M_1 y M_2, luego ambos son modelos estables.

Capítulo 7

El paradigma de la programación con conjuntos de respuesta: ASP

Se llama un *conjunto de respuesta* de un programa P a un modelo estable del mismo. El objetivo de este capítulo es formalizar problemas de manera que su conjunto de respuesta sea la respuesta a ese problema. A esta metodología se le llama la programación con conjuntos de respuesta (ASP: answer set programming). Veremos varios ejemplos y métodos para trabajar con diferentes problemas. En particular, se pretende que se puedan resolver puzles como son el sudoku o acertijos, así como también consideraremos problemas de sabor más matemático.

7.1. Clingo

Clingo es una herramienta que recibe un programa lógico extendido P y devuelve el/los conjunto/s de respuesta de P (si los hay). No entraremos en el funcionamiento de la aplicación ya que se escapa del alcance del presente trabajo, pero usaremos Clingo para resolver los ejemplos que veremos a continuación. Explicamos brevemente la notación de Clingo:

- La implicación $\leftarrow$ se escribe como $:-$
- La disyunción or se escribe como ;
- La negación por defecto se escribe como *not*
- La negación fuerte $\neg$ se escribe como $-$

- Toda regla acaba en un punto.
- Las variables se escriben con el primer carácter en mayúscula; las constantes, empezando en minúscula.
- Las operaciones aritméticas básicas están incluidas en el lenguaje: + - * / ; también las relaciones de orden: $\leq, <, >, \geq$.
- La igualdad se escribe como =. Se usa ! = para expresar $\neq$.

Por ejemplo, si queremos decir que una seta es venenosa o comestible y que la oronja es una seta, escribimos en Clingo:

$seta(oronja).$
$venenosa(X); comestible(X) :- seta(X).$

La siguiente regla expresa que los pájaros normalmente vuelan:

$vuela(X) :- pajaro(X), not\ raro1(X).$

Clingo dispone también de varios atajos en le lenguaje que permite escribir los programas de manera más cómoda. Por ejemplo, se permite usar intervalos para describir un dominio numérico. Para intervalos se usan dos puntos (..) separando los extremos del intervalo. Cuando usamos intervalos, Clingo entiende que nos referimos a los números naturales que se encuentran entre dos números, por ejemplo:

$num(1..9).$

establece que los números del 1 al 9, ambos inclusive, satisfacen el predicado 1-ario *num*.

Finalmente, una gran utilidad de Clingo es la **regla de elección**, que permite atajar el código de manera muy cómoda. Se escribe

$$s_1 \alpha \{t_1 : L_1; ...; t_n : L_n\} s_2$$

donde:

- s_1, s_2 son términos del lenguaje con el que se esté tratando, se pueden omitir si no influyen. Por ejemplo, si la restricción no tiene cota inferior, se omite s_2.
- $t_1, .., t_n$ son relaciones, predicados o términos.
- $L_1, .., L_n$ son relaciones o predicados que le indican a cada relación entre qué elementos puede elegir.

- α es una función entre las siguientes $COUNT, SUM, MIN, MAX$; se puede omitir en el caso de $COUNT$.

Se aplica la restricción de la siguiente manera: Se aplica la función α al conjunto de los t_i que quedan después de evaluar las condiciones expresadas por su respectivo L_i, y se exige que se satisfagan $s_1 \leq \alpha\{t_1 : L_1; ...; t_n : L_n\}$ y $\alpha\{t_1 : L_1; ...; t_n : L_n\} \leq s_2$.

Por ejemplo, si escribimos

$num(0..10)$.
$1\{notade(juan, X) : num(X)\}1$.

estamos diciendo que Juan tiene una, y solo una nota, entre los números naturales que van del 0 al 10.

Veamos otro ejemplo. Vamos a crear un programa que a cada alumno le asigne de una a cinco notas comprendidas entre el 0 y el 10

$num(0..10)$.
$1\{notade(A, X) : num(X)\}5 :- alumno(A)$.

7.2. Técnicas varias

Destacaremos varias técnicas de programación en ASP que veremos a continuación:

- la doble negación como técnica para calcular de forma eficiente el máximo de un conjunto de elementos.
- La técnica de generación y filtrado para resolver problemas combinatorios.
- La técnica de forzar el éxito.

7.2.1. Doble negación

Cuando hablamos de doble negación, estamos refiriéndonos a una mezcla de la negación por defecto y la negación fuerte de la forma "not $-a$", no se debe confundir con la doble negación de la lógica de primer orden ($\neg\neg a \equiv a$).

Esta técnica es útil para calcular el máximo de un conjunto. La idea es, en vez de hacer todos los cálculos diciendo que x es el máximo del conjunto si es mayor que todos los demás elementos, decir que x no es el máximo si encontramos algún elemento que sea mayor que x ($-max(X) \leftarrow Y > X$).

Ejemplo 7.2.1. Si en una clase queremos saber quién tiene la nota más alta y tenemos la relación $nota(X, Y)$ que expresa que la nota del alumno X es Y, la relación $maxnota(X)$ nos indica que X es el alumno con mayor nota. Podemos usar el programa P:

$$-maxnota(X) :- nota(X, Y1), nota(Z, Y2), Y1 < Y2.$$
$$maxnota(X) :- nota(X, Y), not\ -maxnota(X).$$

Entrando como base de datos (hechos) todas las relaciones *nota* con los alumnos y sus respectivas notas, el conjunto de respuesta del programa P será el conjunto de todos los alumnos y sus notas y $maxnota(X)$ con X el alumno de mayor nota.

7.2.2. Regla de elección.

La idea de esta técnica es tener una o varias reglas de elección, de donde se puedan escoger varias opciones, luego, usar un conjunto de reglas de *verificación* que nos permitan deducir cuáles de esas opciones son válidas para nuestro problema. Esta técnica también se conoce como programación por *generación y filtrado*. Veamos un ejemplo típico.

Ejemplo 7.2.2. Consideremos el problema de 3-colorear un grafo, usando los colores rojo, verde y azul, denotamos $r(X), v(X)$ y $a(X)$ para asignarle a cada nodo del grafo su respectivo color. La regla de elección es entonces:

$$r(X); v(X); a(X) :- nodo(X).$$

O, con notación de regla de elección:

$$1\{r(X); v(X); a(X)\}1 :- nodo(X).$$

En las reglas de verificación queremos añadir las restricciones del problema: si existe una arista que una dos nodos del grafo, entonces dichos nodos no pueden tener el mismo color:

$$:- r(X), r(Y), arista(X, Y).$$
$$:- v(X), v(Y), arista(X, Y).$$
$$:- a(X), a(Y), arista(X, Y).$$

Tenemos entonces que de todas las posibles combinaciones que nos deja la regla 1 (todos rojos, todos azules, todos rojos menos el primero...) sólo son válidas aquellas que cumplen con las restricciones del problema de 3-coloración, en otro caso, el programa resulta insatisfacible.

7.2.3. Técnica de forzar el éxito

La idea de esta técnica es primero crear reglas de elección para generar un espacio de búsqueda y después forzar el éxito de la busca con una regla de la siguiente forma:

exito $:- a_1(X_1), a_2(X_2), .., a_n(X_n).$

con *exito* un predicado de aridad 0. A continuación añadimos la regla:

$:-$*not* exito.

Añadiendo esta regla, forzamos que *exito* esté en el modelo estable (si existe) y esto fuerza que se cumplan las propiedades $a_1(X_1), .., a_n(X_n)$. En caso de no existir una combinación válida, el programa será insatisfacible.

Ejemplo 7.2.3. Se plantea encontrar un camino euleriano dado un grafo dirigido con nodos $V = \{1, 2, .., n\}$ y aristas que los conectan, teniendo el predicado $nodo(X)$ para expresar que X es un nodo y $arista(X, Y)$ para expresar que hay una arista que va de X a Y. Fijamos la regla de elección para elegir una arista por cada nodo para que conforme el camino:

$1\{camino(X, Y) : arista(X, Y)\}1 :- nodo(X).$

Ahora, añadimos otra regla para precisar que a cada nodo solo le llega una arista en el camino

$1\{camino(X, Y) : arista(X, Y)\}1 :- nodo(Y).$

Finalmente, queremos que el camino pase por todos los nodos, podemos ver los nodos que están unidos por el camino mediante una función iterativa:

$unidos(X, Y) : -camino(X, Y).$
$unidos(X, Y) : -unidos(X, Z), camino(Z, Y).$

Finalmente, forzamos el éxito, que, en este caso, llamaremos *euleriano*:

$euleriano :- unidos(1, 1).$
$:-$*not euleriano.*

7.3. Haciendo matemáticas en Clingo

7.3.1. Encontrar grupos finitos no abelianos

Vamos a usar el paradigma de la programación con conjuntos de respuesta con un fin un poco más teórico: encontrar grupos finitos no abelianos. Como bien sabemos, un grupo está formado por un conjunto de elementos y una operación interna y binaria, lo expresamos con las siguientes reglas:

$num(0..n)$.

$1\{operacion(X,Y,Z) : num(X)\}1 :- num(Z), num(Y)$.

Con el predicado $operacion(X,Y,Z)$ expresamos que $Y * Z = X$, de esta manera estamos diciendo que tenemos $n+1$ elementos que se representarán mediante números y que al efectuar $Y * Z$ obtenemos un solo resultado, que es parte del grupo. A la hora de ejecutar el grupo deberemos cambiar n por el número que corresponda según el orden del grupo que queramos encontrar.

Para que la respuesta que nos devuelva tenga estructura de grupo debe cumplir dos requisitos: existencia de elemento neutro (que será el 0), y existencia de elemento inverso para cada elemento, lo implementamos a continuación:

$operacion(Y,Y,0) :- num(Y)$.
$operacion(Z,0,Z) :- num(Z)$.
$1\{operacion(0,Y,Z) : num(Y)\}1 :- num(Z)$.

Las dos primeras reglas hacen referencia al elemento neutro y la tres exige la existencia y unicidad de elemento inverso para cada elemento del grupo. Finalmente debemos exigir que la operación cumpla la propiedad asociativa, con la siguiente regla

$$operacion(X, AmasB, C) :- operacion(X, A, BmasC), operacion(AmasB, A, B), operacion(BmasC, B, C).$$

Como queremos que el grupo no sea abeliano, vamos a usar la técnica de forzar el éxito con el predicado *noabeliano*:

$noabeliano :- operacion(X1,Y,Z), operacion(X2,Z,Y), X1! = X2$.
$:- not\ noabeliano$.

Ahora, este programa devuelve un grupo no abeliano de orden $n+1$ si existe algún grupo no abeliano con este orden, y es inconsistente si no existe ninguno grupo de orden $n+1$ que no sea abeliano.

Un ejemplo de respuesta para $n = 5$ es:

num(0) num(1) num(2) num(3) num(4) num(5) suma(0,0,0) suma(1,0,1) suma(1,1,0) suma(2,0,2) suma(2,2,0) suma(3,0,3) suma(3,3,0) suma(4,0,4) suma(4,4,0) suma(5,0,5) suma(5,5,0) suma(0,2,5) suma(1,3,5) suma(2,5,5) suma(3,4,5) suma(4,1,5) suma(5,1,4) suma(3,2,4) suma(2,3,4) suma(0,4,4) suma(1,5,4) suma(2,1,3) suma(1,2,3) suma(0,3,3) suma(5,4,3) suma(4,5,3) suma(3,1,2) suma(5,2,2) suma(4,3,2) suma(1,4,2) suma(0,5,2) suma(0,1,1) suma(4,2,1) suma(5,3,1) suma(2,4,1) suma(3,5,1) noabeliano

7.3.2. Números de Schur

Vamos a atacar en esta sección un problema clásico in: el cálculo de los números de Schur, presentado por Vladimir Lifschitz en 2000 (*Programming with CLINGO*). Empezaremos dando un par de definiciones introductorias

Definición 7.3.1 (Conjunto insumable)**.** Se dice que un conjunto X de números naturales positivos es *insumable* si $\forall x, y \in X,\ x + y \notin X$.

Ejemplo 7.3.1.

- El conjunto $\{1, 3, 5\}$ es insumable.
- El conjunto $X = \{1, 3, 4, 5\}$ no es insumable, pues $1 + 4 = 5 \in X$.

El objetivo es descomponer un conjunto de la forma $A_n = \{1, ..., n\}$ en el menor número de subconjuntos insumables posibles. Por ejemplo, si $n = 4$, podemos descomponer A_4 en dos conjuntos insumables: $X = \{1, 4\}$ e $Y = \{2, 3\}$; sin embargo, A_5 no se puede descomponer en dos conjuntos insumables.

Proposición 7.3.1. Si A_n no se puede descomponer en r conjuntos insumables, entonces A_m tampoco se puede descomponer en r conjuntos insumables para todo $m > n$.

Demostración: Si un conjunto es insumable, también lo es al quitarle el mayor de sus elementos, luego, si podemos descomponer A_m en r conjuntos insumables, también podemos descomponer A_{m-1} en r conjuntos insumables. □

Definición 7.3.2 (Números de Shur)**.** Se define $S(r)$ con $r \in \mathbb{N} - \{0\}$ como el mayor número n tal que podemos descomponer A_n en r conjuntos insumables

Ejemplo 7.3.2. Para $r = 2$ ya vimos que se puede descomponer A_4 en dos conjuntos insumables, sin embargo, no se puede decir lo mismo de A_5, por lo tanto $S(2) = 4$.

Para valores mayores de r, vamos a usar conjuntos de respuesta. Creamos un programa, con parámetros r y n que modificaremos según convenga, para que descomponga A_n en r conjuntos insumables. Usaremos la propiedad $en(B, R)$ que nos dice que el número B está en el conjunto I_R:

$1\{en(B, 1..r)\}1 :- B = 1..n.$
$:- en(B, X), en(C, X), en(B + C, X).$

Este programa nos devolverá la partición de A_n en r conjuntos insumables cuando sea posible, y será insatisfacible cuando no se pueda. Vamos a probarlo con diferentes parámetros:

- con $n = 13$, $r = 3$ obtenemos una partición.
- con $n = 14$, $r = 3$ obtenemos insatisfacible, sabemos por tanto que $S(3) = 13$.
- con $n = 44$, $r = 4$ obtenemos una partición, el programa tarda 2.953 segundos.
- con $n = 45$, $r = 4$ obtenemos insatisfacible, sabemos por tanto que $S(4) = 44$. (El programa tarda, vemos que cuando no encuentra respuesta, el programa tarda mucho más que cuando solo tiene que encontrar una).

Nota 7.3.1. Recientemente, se ha demostrado que $S(5) = 160$. Éste es, hasta la fecha, el mayor número de Shur que se conoce con exactitud. (La prueba de que $S(5) = 160$ fue anunciada en 2007 y ocupa hasta 2 petabytes de espacio. Para aumentar la confianza en dicha prueba, la prueba ha sido verificada en el demostrador automático ACL2).

7.4. Resolución de Puzles

Veamos un par de ejemplos de resolución de puzles:

7.4.1. El Sudoku

Vamos a crear un programa para resolver el Sudoku: Describiremos el tablero de sudoku dando a cada casilla (X, Y) un valor N que será el número que le corresponda a la casilla, usando la relación $pos(N, X, Y)$.

Describimos primero el tablero:

$num(1..9).$
$coord(X, Y) :- num(X), num(Y).$

Escribimos ahora una regla para indicar que en cada casilla debe haber un número.

$$1\{pos(N, X, Y) : num(N)\} :- coord(X, Y).$$

Mediante esta regla decimos que hay un número N en cada coordenada.

Debemos añadir ahora las otras tres reglas que nos permiten saber que NO hay un número N en una coordenada (X, Y), estas son: El número N se encuentra en otra casilla en la fila de (X, Y); El número N se encuentra en otra casilla en la columna de (X, Y), el número N se encuentra en otra casilla

en la misma región que (X, Y). Las dos primeras reglas no son difíciles de escribir puesto que basta con cambiar una de las coordenadas, sin embargo, para decir que una casilla está en la misma región vamos a tener que usar una regla auxiliar, que nos diga en que región se encuentra cada coordenada:

$mismoint(X1, X2) :- X1 < 4, X2 < 4, X1 > 0, X2 > 0.$
$mismoint(X1, X2) :- X1 > 3, X1 < 7, X2 > 3, X2 < 7.$
$mismoint(X1, X2) :- X1 > 6, X2 > 6, X1 < 10, X2 < 10.$
$mismaregion(X1, Y1, X2, Y2) :- mismoint(X1, X2), mismoint(Y1, Y2).$

Ahora tenemos una relación $mismaregion(X1, Y1, X2, Y2)$ que se satisface solo si $(X1, Y1)$ están en la misma región que $(X2, Y2)$. Vamos a proceder a escribir las reglas que describan las restricciones antes mencionadas:

$$\begin{aligned} -pos(N, X, Y) :- & \ pos(N, X, Y1), coord(X, Y1), Y1! = Y. \\ -pos(N, X, Y) :- & \ pos(N, X1, Y), coord(X1, Y), X1! = X. \\ -pos(N, X, Y) :- & \ pos(N, X1, Y1), mismaregion(X, Y, X1, Y1), \\ & X1! = X, Y1! = Y \end{aligned}$$

El programa devuelve ahora $pos(N, X, Y)$ con un único N para cada coordenada (X, Y) dando una solución válida al sudoku. Para rellenar un sudoku dado se deben incluir hechos con el número que le corresponde a cada coordenada.

7.4.2. SEND+MORE=MONEY

Vamos a resolver un problema clásico de criptoaritmética: " SEND + MORE = MONEY ". La idea es, dada la suma simbólica anterior y tratando cada letra como un número comprendido entre 0 y 9, asignarle a cada letra un número de manera que la igualdad sea cierta (Suponiendo que si dos letras son la misma se le asigna el mismo valor y si dos letras son distintas se le asigna un valor distinto).

Empezamos introduciendo los datos:

$letra(s).\ letra(e).\ letra(n).\ letra(d).\ letra(m).\ letra(o).\ letra(r).\ letra(y).$

Le decimos al programa que a cada letra se le asigna un solo valor, y un solo valor para cada letra:

$num(0..9).$
$1\{valor(N, L) : num(N)\} :- letra(L).$

y ahora vamos a crear una regla de manera que el programa nos devuelva un modelo mínimo que contenga los valores correctos asociados a cada letra:

$hafuncionado :- valor(S,s), valor(E,e), valor(N,n), valod(D,d),$
$valor(M,m), valor(O,o), valor(R,r), valor(Y,y),$
$S! = 0, M! = 0,$
$S! = E, S! = N, S! = D, S! = M, S! = O, S! = R, S! = Y,$
$E! = N, E! = D, E! = M, E! = O, E! = R, E! = Y,$
$N! = D, N! = M, N! = O, N! = R, N! = Y$
$D! = M, D! = O, D! = R, D! = Y,$
$M! = O, M! = R, M! = Y,$
$O! = R, O! = Y,$
$R! = Y,$
$A = 1000 * S + 100 * E + 10 * N + D,$
$B = 1000 * M + 100 * O + 10 * R, E,$
$C = 10000 * M + 1000 * O + 100 * N + 10 * E + Y,$
$A + B = C.$

Finalmente, nos aseguramos de que se cumpla la propiedad $hafuncionado :- not\ hafuncionado.$

7.4.3. Buscaminas

Vamos a atacar otro puzle clásico como es el buscaminas. Para resolverlo en ASP, nos basta con explicar cómo funciona el juego y darle un tablero de entrada, consideraremos un tablero de la siguiente forma

.	.	2	.	3	.
2	.	.	.	.	.
.	.	2	4	.	3
1	.	3	4	.	.
.	.	.	.	.	3
.	3	.	3	.	.

Donde los puntos representan los espacios en los que no se sabe lo que hay aún y los números el número que hay en esa casilla. El objetivo es saber qué hay en cada punto, si una bomba o un número. Para empezar definiremos el tablero de forma genérica, queremos decirle que hay f filas, c columnas y nueve posibles números en una casilla (del 0 al 8).

$nums(0..8).$
$fila(1..f).$
$col(1..c).$

Cuando usemos un tablero concreto, cambiaremos c y f por los números que convengan, por ejemplo, con el tablero de antes tendríamos $f = 6, c = 6$. Usaremos las siguientes propiedades:

- $numero(f, c, n)$ indica que en la casilla (f, c) se encuentra el número n.
- $bomba(f, c)$ indica que en la casilla (f, c) se encuentra una bomba.

En el buscaminas, que un número n esté en una casilla indica que alrededor de esa casilla se encuentran n bombas, tenemos, por lo tanto, que definir el concepto de adyacencia en el tablero:

$adj(R, C, R1, C1) : -fila(F), fila(F1), col(C), col(C1)$, |R-R1| + |F-F1|==1.
$adj(R, C, R1, C1) : -fila(F), fila(F1), col(C), col(C1)$, |R-R1| ==1, |F-F1|==1.

Indicamos que en cada casilla se encuentra bien una bomba, bien un número:

$1\{numero(F, C, Z) : nums(Z); mina(F, C)\}1 :- fila(F), col(C).$

Finalmente, alrededor de un número N, se encuentran N bombas:

$N\{mina(F2, C2) : adj(F2, C2, F1, C1)\}N :- numero(F1, C1, N).$

Con esto, el programa ya es capaz de resolver un buscaminas. Vamos a ponerle de ejemplo el tablero anterior, para ello, además de cambiar f y c por 6, introducimos los datos:

$numero(1, 3, 2).$
$numero(1, 5, 3).$
$numero(2, 1, 2).$
$numero(3, 3, 2).$
$numero(3, 4, 4).$
$numero(3, 6, 3).$
$numero(4, 1, 1).$
$numero(4, 3, 3).$
$numero(4, 4, 4).$
$numero(5, 6, 3).$
$numero(6, 2, 3).$
$numero(6, 4, 3).$

En tan solo 0.3 segundos, el programa nos devuelve una respuesta, de la que sacamos la siguiente información:

numero(3,1,1) numero(5,1,2) numero(1,2,2) numero(3,2,1) numero(4,2,2) numero(2,3,2) numero(6,3,3) numero(1,4,2) numero(5,5,4) numero(2,6,3) numero(4,6,2) mina(1,1) mina(6,1) mina(2,2) mina(5,2) mina(5,3) mina(2,4) mina(5,4) mina(2,5) mina(3,5) mina(4,5) mina(6,5) mina(1,6) mina(6,6)

Nota 7.4.1. Este programa solo es realmente efectivo cuando el tablero admite una única solución, puesto que trata de encontrar la solución partiendo de la información que tiene, sin embargo, al jugar al buscaminas, se amplía la información y ,finalmente, solo una de las soluciones es válida.

7.4.4. Las n damas

Finalmente, el acertijo de las n damas consiste en colocar n damas en un tablero de ajedrez de tamaño $n \times n$ de manera que no se amenacen entre ellas. Inicialmente nació en 1848 como el problema de las 8 damas y es un ejemplo clásico para explicar algoritmos de backtracking. Con $n = 8$, existen 92 soluciones, que se quedan en 12 al eliminar soluciones simétricas entre ellas. El número de soluciones que tiene el problema tiene a crecer cuando n crece, teniendo más de dos millones de soluciones para $n = 15$. Vamos a buscar alguna solución con programación de conjuntos de respuesta.

Lo primero es establecer el rango n con el que se trabajará y que se ha de sustituir por el número con el que vayamos a tratar:

$num(1..n)$.

A continuación, si dos damas están en una misma fila/columna, entonces se amenazan, debemos impedir que esto pase. Al estar en un tablero $n \times n$ y colocar n damas, es obvio que habrá una en cada fila y una en cada columna, expresamos estas dos exigencias juntas:

$1\{damaen(X,Y) : numero(Y)\}1 : -numero(X)$.
$1\{damaen(X,Y) : numero(X)\}1 : -numero(Y)$.

Finalmente, dos damas no pueden estar en la misma diagonal:

$: -damaen(X1,Y1), damaen(X2,Y2), X1 < X2, Y1 + X1 == Y2 + X2$.
$: -damaen(X1,Y1), damaen(X2,Y2), X1 < X2, Y1 - X1 == Y2 - X2$.

Haciendo pruebas con $n = 8$ obtenemos:

number(1) number(2) number(3) number(4) number(5) number(6) number(7) number(8) damaen(2,3) damaen(1,5) damaen(3,1) damaen(4,7) damaen(5,2) damaen(6,8) damaen(7,6) damaen(8,4)

en 0.065 segundos; usando $n = 50$ obtenemos una respuesta en 10.693 segundos.

Con el programa como lo tenemos, la restricción de la diagonal implica que por cada casilla, se crean del orden de n^2 reglas (una por cada una de las otras casillas), una de las formas de mejorar la eficiencia es sustituir estas reglas por las siguientes:

$: -\#count\{(X,Y) : queen(X,Y), diag1(X,Y,D)\} >= 2, D = 1..2*n-1$.
$: -\#count\{(X,Y) : queen(X,Y), diag2(X,Y,D)\} >= 2, D = 1..2*n-1$.
$diag1(X,Y,X+Y-1) : -numero(X), numero(Y)$.
$diag2(X,Y,X-Y+n) : -numero(X), numero(Y)$.

De esta manera, el programa solo usa $2n$ reglas para cada diagonal, reduciendo la complejidad del problema. Obtenemos con estos cambios una respuesta para el caso $n = 50$ en tan solo 1.067 segundos.

Capítulo 8

Conclusión

En este trabajo hemos estudiado una ampliación de la lógica tradicional que nos permite crear nuevas reglas y mecanismos de deducción lógica que se adapten de manera más adecuada a los razonamientos humanos que, en muchas circunstancias, son razonamientos *no monótonos* basados en el sentido común. Una de las mayores ventajas es poder crear reglas que se cumplen "en general" pero que admiten excepciones sin generar por ello una contradicción lógica y que, además, se pueden rectificar al ampliar la información que se tenga. Uno de los mecanismos que se destacan de este trabajo es el hecho de aceptar algo de forma **escéptica o crédula**, ya que son dos mecanismos no monótonos que nos permiten recopilar información rectificable, uno de ellos mostrando la información de la que se está seguro y el otro enseñando toda la posible información que se puede deducir con la información que se tiene en el momento.

Como hemos podido ver en el capítulo final, usando el paradigma de la programación con conjuntos de respuesta podemos resolver problemas aparentemente complejos en muy poco tiempo y, sobre todo, con un código altamente declarativo y sorprendentemente simple. Este paradigma usa mecanismos algo complejos para optimizar la complejidad con la que se atacan los problemas y que no se han visto en este trabajo. Sin embargo, resulta un lenguaje cómodo para implementar el primer prototipado de un problema. Además, usa como base la lógica no monótona y para utilizarlo no hace falta conocimientos más allá de lo que se estudian en este trabajo. Una línea interesante de trabajo futuro sería profundizar en el estudio del paradigma de la programación con conjuntos de respuesta.

Finalmente, aclarar, por si no quedaba claro, que la seta oronja es perfectamente comestible. Sin embargo, no tengo ni idea de si Piolín vuela ni parece haber documentación al respecto en internet, queda también pendiente para el trabajo futuro.

Bibliografía

[1] Thomas Eiter, Giovambista Ianni, Thomas Krennwallner: Answer Set Programming: A Primer.Institut für Informationssysteme, Universitá della Calabria 2009

[2] Gelfond, M., Kahl,Y.Knowledge representation, reasoning and the design of Intelligent Agents, 2013.

[3] Gelfond, M. Lifschitz, V. The stable model semantics for logic programming, 2000

[4] Genesereth, M., Nilsson, N. Logical foundations of Artificial Intelligence, 1998.

[5] Marijn J.H Heule. Schur Number Five, University of Texas. 2017.

[6] Vladimir Lifschitz, Programming with Clingo, University of Texas.

[7] Marek Sergot. Minimal models and fixpoint semantics for definite logic programs. . 2005