DR. ALEJANDRO GUERRA-HERNÁNDEZ

REPRESENTACIÓN DEL CONOCIMIENTO

Universidad Veracruzana

NOTAS DE CURSO

Instituto de Investigaciones en Inteligencia Artificial
https://www.uv.mx/personal/aguerra/rc
2023, Febrero

Dr. Alejandro Guerra-Hernández. *Representación del Conocimiento.* Universidad Veracruzana, Instituto de Investigaciones en Inteligencia Artificial, Campus Sur, Calle Paseo Lote II, Sección Segunda No 112, Nuevo Xalapa, Xalapa, Ver., México 91097, 2023.

WEBSITE:
http://www.uv.mx/personal/aguerra

E-MAIL:
aguerra@uv.mx

2

ABSTRACT

Welcome to the notes for Knowledge Representation, a course about Knowledge, Reasoning and Planning. The contents correspond roughly to the part III of Artificial Intelligence, a modern approach, although they are also influenced by other texts about logic in computer science and rational agency.

RESUMEN

Bienvenidos a las notas del curso Representación del Conocimiento. Un curso ascerca de la representación, el razonamiento y la planeación en IA. El contenido de este curso corresponde a la parte III del libro Inteligencia Artificial, un enfoque Moderno, aunque está influenciado por otros textos sobre la lógica en la ciencias de la computación y los agentes racionales.

AGRADECIMIENTOS

A José Negrete Martínez†, Amal El-Fallah Seghrouchni y Christian Lemaître León.

Xalapa, Ver., México
Febrero 2023

Alejandro Guerra-Hernández

ÍNDICE GENERAL

1 INTRODUCCIÓN

La Inteligencia Artificial (IA) tiene como objeto de estudio a las entidades inteligentes y su comportamiento; pero a diferencia de la filosofía, la psicología, las neurociencias, y demás disciplinas que comparten este objeto de estudio, su meta no tiene que ver únicamente con la comprensión de tales entidades, sino con su construcción. De hecho, como se muestra en la Figura 1.1, Varela [107] ubica a la IA como parte de las Ciencias Cognitivas resaltando esta asimetría: Su interés, único en el área, por la síntesis de entidades inteligentes. La **construcción de agentes racionales** constituye la idea central del enfoque adoptado en este curso, propuesto por Russell y Norvig [94], y curiosamente calificado de moderno [1].

Agencia e IA

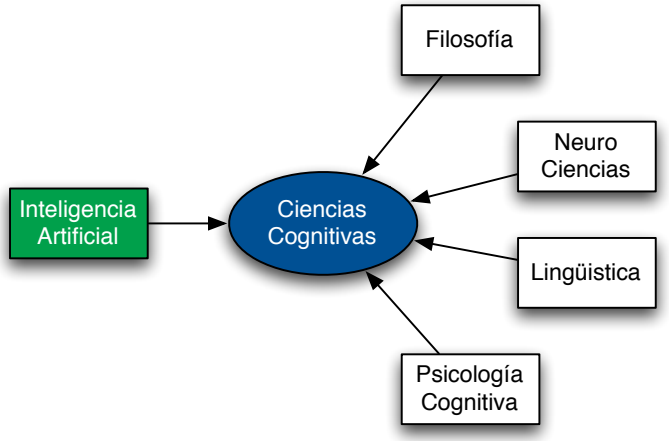


Figura 1.1: La IA como parte de las Ciencias Cognitivas, según Varela [107], resalta por su interés en la síntesis de entidades inteligentes y no solo en su característico análisis, presente en otras disciplinas del área.

Aunque muchos practicantes de la IA se consideran a si mismos científicos de la computación, como bien señala Kayser [56] parafraseando al famoso cuadro de Magritte (Fig. 1.2) –*Ceci n'est pas de l'informatique*. O más precisamente, la no es reducible a la computación. Sus intereses, representación de conocimientos incluida, son anteriores a ella y más amplios.

IA y Computación

A propósito del cuadro, Magritte nos ofrece un comentario sobre el mismo, ideal para iniciar este curso:

> La famosa pipa. ¡Como la gente me reprochó por ello! Y sin embargo, ¿Podría usted rellenar mi pipa? No, sólo es una representación ¿No es así? ¡Así que si hubiera escrito en mi cuadro "Esto es una pipa", habría estado mintiendo! [105].

En esta introducción abordaremos intuitivamente, algunas de las cuestiones planteadas por el cuadro de Magritte y su relación con el tema central de este curso: ¿Qué queremos decir por conocimiento, representación y razonamiento?, ¿Qué es la verdad? y ¿Porqué creemos que estos conceptos son útiles para la construcción de sistemas inteligentes? Para ello, nos guiaremos con el capítulo uno del libro sobre representación del conocimiento de Brachman y Levesque [15] y otras lecturas complementarias [74, 30].

1 AIMA (*Artificial Intelligence: A Modern Approach*) es el libro de texto más usado en el mundo sobre IA. Se tiene registro de su uso en más de 1500 universidades: https://aima.cs.berkeley.edu/index.html

Figura 1.2: De la serie la traición de las imágenes de René Magritte, el texto dice "Esto no es una pipa".

Representación de Conocimiento

Siendo así las cosas, la **Representación del Conocimiento** es el área de la IA que estudia cómo el conocimiento puede ser representado y procesado simbólicamente mediante programas que razonan automáticamente –El estudio del pensamiento como un proceso computacional.

Agencia y Conocimiento

Si bien, tal estudio está ligado a la cuestión de cómo es que un agente utiliza su conocimiento para satisfacer sus metas en su medio ambiente, aquí focalizaremos en el **conocimiento**, no en el conocedor [2]: Nos preguntaremos qué necesita saber un agente –humano, animal, electrónico, ó mecánico, para comportarse inteligentemente; y qué clase de mecanismos computacionales le permiten acceder a ese conocimiento cuando así lo requiere.

Organización

En lo que sigue, la Sección 1.1 introduce una serie de conceptos básicos a los que haremos referencia continuamente: Conocimiento, proposición, valor de verdad, actitud proposicional, creencia, representación, símbolo, razonamiento, e inferencia entre otros. Posteriormente, la Sección 1.2 aborda la cuestión de la relevancia de la representación del conocimiento para la IA, discutiendo la postura intencional e introduciendo una hipótesis de trabajo. La Sección 1.3 introduce lo que será nuestro interés tecnológico principal: Los Sistemas Basados en el Conocimiento. La Sección 1.4 abunda sobre la relevancia de usar una base de conocimiento en estos sistemas y la Sección 1.5 hace lo propio con respecto al razonamiento y los mecanismos de inferencia. El capítulo no podría terminar sin una pequeña nota sobre el uso de la lógica en estas tareas, de lo cual da cuenta la Sección 1.6.

1.1 CONCEPTOS BÁSICOS

Conocimiento

La cuestión sobre qué es el **conocimiento** es anterior a la IA y a la computación. Se ha abordado desde la antigua Grecia y sigue siendo una cuestión de interés. No entraremos aquí en detalles filosóficos, pero si intentaremos ofrecer una versión informal de lo que se supone es el conocimiento, útil para los propósitos del curso.

Proposicíon

Primero, entre otras cosas, el conocimiento es una relación entre un agente y una **proposición** –Una idea expresada como un enunciado declarativo. Observen que es común decir "Ana sabe que..." donde a los puntos suspensivos sigue una proposición. Por ejemplo, "Ana sabe que la clase de representación de conocimiento es martes y jueves".

2 Un enfoque con énfasis en el conocedor, el agente, se ofrece en nuestro curso sobre Sistemas Multi-Agentes: https://www.uv.mx/personal/aguerra/sma/

Una parte del misterio acerca del conocimiento se debe a la naturaleza de las proposiciones. Por ahora, lo que nos interesa de las proposiciones es que son entidades abstractas sujetas a **valores de verdad**: Pueden ser verdaderas o falsas, correctas o incorrectas. Decir que "Ana sabe P" equivale a decir que "Ana sabe que P es verdad". Observen que esto refleja un juicio donde el agente se da cuenta de que su medio ambiente es de cierta forma (Uno donde P es el caso), y no de otras (Donde P no es el caso).

Valor de verdad

Algo similar sucede con frases como "Ana desea que la clase de representación de conocimiento sea los martes y jueves". Se trata de la misma proposición del ejemplo anterior, pero su relación con el agente es diferente. Los verbos como saber, creer, desear, intentar, etc., denotan lo que se conoce como **actitudes proposicionales**. Independientemente de la actitud, lo que importa con respecto a la proposición es su valor de verdad. Si "Ana desea que P", entonces Ana desea que el medio ambiente sea tal, que P es el caso.

Actitudes proposicionales

Las actitudes proposicionales **creer** y saber guardan cierta relación. Usamos la primera cuando deseamos expresar que el juicio del agente no es necesariamente preciso, o que no se sostiene por la razones adecuadas. A veces también se usa para expresar que el agente no está completamente convencido de que la proposición sea el caso. Lo importante aquí es que ambas actitudes comparten una idea básica sobre el conocimiento: Expresan que el agente asume que el mundo es de cierta forma y no de otra.

Creencia

El concepto de representación es tan escurridizo como el de conocimiento. En líneas generales, una **representación** es una relación entre dos dominios, donde se pretende que el primero denote o tome el lugar del segundo. Normalmente el primer dominio es más concreto, inmediato o accesible que el segundo. Por ejemplo, el diagrama de la bomba de gasolina en los avisos de la carretera es más concreto que la frase "Gasolinera".

Representación

De especial interés para nosotros es la representación basada en **símbolos** formales, esto es, un carácter o secuencia de caracteres tomada de algún alfabeto. Por ejemplo, el dígito 6 representa al número seis, lo mismo que la secuencia de letras VI y, como no, seis y six. Como en toda representación, se asume que es más fácil contender con lo símbolos, que con lo que representan. Observen que el caso de las proposiciones es una representación simbólica.

Símbolo

La **representación del conocimiento** es entonces el área de estudio concerniente al uso de los símbolos formales para representar una colección de proposiciones creídas por un agente putativo. No estamos afirmando que los símbolos representarán todas las proposiciones creídas por el agente, podría haber un número infinito de proposiciones creídas y solo un número finito de ellas representadas. Es el razonamiento el que tiende un puente entre lo representado y lo creído.

Representación del conocimiento

Por **razonamiento** entendemos la manipulación formal de los símbolos que representan colecciones de proposiciones creídas, para producir representaciones de nuevas proposiciones. Aquí explotamos el hecho de que el símbolo sea más simple que lo que representa: Las representaciones simbólicas deben ser lo suficientemente concretas para manipularlas (cambiarlas de sitio, eliminarlas, copiarlas, concaternarlas, etc.) de tal manera, que se posible construir nuevas representaciones.

Razonamiento

Brachman y Levesque [15] nos recuerdan una brillante analogía con la aritmética: Podemos pensar en la suma como una manipulación formal, de los símbolos 103 y 8, podemos construir 111 para denotar la suma de ellos. La manipulación (suma por columnas, acarreo, etc.) define el concepto suma. El razonamiento es similar: De proposiciones como "Todos los días martes hay clase de representación de conocimiento" y "El día de hoy es martes" podemos construir el enunciado "Hay clase de representación de conocimiento". A esta forma de razonamiento le solemos llamar **inferencia lógica**, porque la proposición final es una conclusión lógica de las primeras. De forma que, como fue por primera vez expuesto por Leibniz, el razonamiento es una forma de **cálculo** que en lugar de operar sobre números, como la aritmética, lo hace sobre proposiciones.

Inferencia lógica

Cálculo

1.2 REPRESENTACIÓN DEL CONOCIMIENTO E IA

¿Porqué es relevante el conocimiento para la IA? Una primer respuesta sería que pareciera que los humanos somos capaces de exhibir un **comportamiento inteligente**, gracias a lo que sabemos.

Comportamiento

Otra respuesta es que a veces resulta útil describir el comportamiento de sistemas que son suficientemente complejos, usando un vocabulario que incluye términos como creencias, deseos, metas, intenciones, etc. Por ejemplo, al jugar ajedrez contra un programa podemos decir que nuestro contrincante movió su alfil porque creía que su reina era vulnerable, pero aún desea amenazar mi caballo. En términos de cómo el programa está construído, tal explicación se reduciría a que usando el procedimiento de evaluación P con la función de evaluación estática Q, se obtuvo un valor de +9 en una búsqueda alfa-beta minimax de profundidad 4. La segunda descripción, aunque precisa, está expresada en un nivel de detalle incorrecto y no nos ayuda a determinar que movimiento haré yo en consecuencia. La primera, que resulta más útil para este fin, descansa en lo que Dennett [31] denomina la **postura intencional**.

Postura intencional

Lo anterior no significa que la postura intencional sea siempre la adecuada y la clave está en la complejidad del sistema que se quiere representar: Asumir la postura intencional para describir el apagador de luz resulta innecesario, inadecuado y caricaturesco. McCarthy [70] estableció claramente que en computación la postura intencional es útil para contender con nuestra limitación para adquirir conocimiento, usarlo para predecir y establecer generalizaciones en términos de la estructura de un programa; siempre y cuando las actitudes proposicionales reflejen el uso que se les suele dar.

Ahora bien, ¿Es esto todo lo que entendemos por representación de conocimiento? Es solo hablar acerca del conocimiento. Observen que la postura intencional no dice nada acerca de qué está y qué no está representado simbólicamente. El programa de ajedrez representa simbólicamente la posición de las piezas en el tablero, pero, por ejemplo, su deseo de eliminar mi caballo lo más pronto posible puede no estarlo. Puede tratarse de una propiedad emergente de diferentes componentes del programa: sus funciones de evaluación, su librería de movimientos, etc. Y aún así, resulta útil para mi saber que el programa desea despachar a mi caballo. ¿Cual es el rol entonces de la representación de conocimiento?

La **hipótesis** subyacente en el área de representación de conocimiento es que podemos construir sistemas que contengan representaciones simbólicas con dos propiedades importantes: Que desde fuera, nosotros podamos entender qué representan las proposiciones del sistema; y que internamente, el sistema haya sido diseñado para comportarse como lo hace debido a estas representaciones.

Hipótesis

1.3 SISTEMAS BASADOS EN EL CONOCIMIENTO

Para entender en qué consiste un sistema basado en el conocimiento usaremos dos programas muy simples en Prolog. Veamos el primero:

```
1 adivina_color(nieve) :- !, write("Es blanca.").
2 adivina_color(pasto) :- !, write("Es verde.").
3 adivina_color(cielo) :- !, write("Es amarillo.").
4 adivina_color(_) :- write("Me has ganado.").
```

Y he aquí el segundo:

```
1 adivina_color(X) :-
2     color(X,Y), !,
3     write("Es de color "), write(Y), write(".").
4 adivina_color(_) :- write("Me has ganado.").
5
6 color(nieve,blanca).
```

```
7  color(cielo,amarillo).
8  color(vegetacion,verde).
9  color(X,Y) :-
10     hecho_de(X,Z),
11     color(Z,Y).
12
13 hecho_de(pasto,vegetacion).
14 hecho_de(selva, vegetacion).
```

Observen que ambos programas pueden adivinar el color de varios objetos, aunque se equivoque con respecto al cielo. Desde la postura intencional, podemos decir que ambos saben que el color de la nieve es blanco. Sin embargo, como veremos más adelante, solo el segundo programa está diseñado en conformidad con la hipótesis de la representación del conocimiento.

Consideremos la cláusula `color(nieve,blanca)`, por ejemplo. Se trata de una estructura simbólica que puede verse como una representación de la proposición "la nieve es blanca"; mejor aún, gracias a que sabemos como funciona Prolog, sabemos que el sistema imprimirá la frase adecuada debido a que el programa dará con la cláusula en el momento adecuado. Si eliminamos la cláusula el programa dejará de actuar de esa manera. Ahora bien, no existe una cláusula similar en el primer programa. Lo más cercano es la primer cláusula, pero no resulta natural interpretarla como una creencia.

De forma que lo que hace que un sistema esté basado en conocimiento, no es el uso de un formalismo lógico (Prolog en este caso); ni el hecho de que el sistema sea lo suficientemente complejo como para justificar el uso de la postura intencional y sus descripciones sobre el conocimiento; ni el hecho de lo que creemos es verdadero. Lo importante es la presencia de una **base de conocimientos**, una colección de estructuras simbólicas que representan lo que el agente cree y razona, durante su operación.

Base de conocimiento

Diversos sistemas propuestos por la IA funcionan de esta manera. Los **sistemas expertos** son un claro ejemplo de ello, pero se pueden encontrar bases de conocimiento en sistemas para el procesamiento de lenguaje natural, planeación, diagnóstico y aprendizaje. Otros sistemas incluyen en menor grado conocimiento, como algunos juegos y sistemas de visión de alto nivel (reconocimiento de objetos). Y por supuesto, hay una parte de la IA que no está basada en el conocimiento: Reconocimiento de habla de bajo nivel, visión, control motor, entre otros.

Sistemas expertos

1.4 ¿PORQUÉ UNA BASE DE CONOCIMIENTO?

Una pregunta obvia que surge al comparar los dos programas Prolog de la sección anterior es ¿Qué ventajas tiene, si acaso tiene alguna, usar una base de conocimiento? Los defensores del llamado **conocimiento procedimental** defenderían al primer programa: Una especie de compilación de la base de conocimientos del segundo programa que distribuye lo que debe saberse para actuar en cada módulo que así lo requiere. De hecho, el desempeño del sistema será mejor en este caso, ya que buscar los hechos en la base de conocimientos para posteriormente, decidir qué hacer, solo puede hacer que el sistema sea más lento.

Conocimiento procedimental

Sin embargo, desde el punto de vista del diseño de sistemas, la aproximación basada en el conocimiento del segundo programa tiene las siguientes **propiedades deseables**:

Propiedades deseables

- Podemos agregar nuevas tareas y fácilmente hacer que dependan del conocimiento previo. Por ejemplo, en nuestro programa podemos agregar la tarea de enumerar todos los objetos de un color dado e incluso pintar un dibujo, haciendo uso de la base de conocimientos previamente especificada para determinar los colores a usar.

- Podemos extender el comportamiento del programa agregando nuevas creencias. Por ejemplo, al agregar una cláusula que especifique que el color de los canarios es amarillo, automáticamente propagamos esa información a cualquier rutina que la necesite.
- Podemos depurar el comportamiento erróneo del sistema localizando creencias erróneas en el mismo. Por ejemplo, corregir la cláusula que especifica que el color del cielo es amarillo.
- Podemos explicar y justificar de manera concisa el comportamiento del sistema. ¿Porqué dice mi programa que el pasto es verde? Porque cree que el paso es una forma de vegetación y la vegetación es de color verde. Tal explicación se justifica plenamente, de hecho, si eliminamos alguna cláusula del programa, la explicación no se sostiene, pero el comportamiento del mismo será diferente.

Resumiendo, la ventaja de los sistemas basados en el conocimiento es que, por diseño, pueden ser informados de nuevos hechos sobre el mundo y ajustar su comportamiento en consecuencia. La propiedad de algunas de nuestras acciones de ser dependientes de lo que creemos, es lo que Zenon Pylyshyn llama **penetrabilidad cognitiva**. Por ejemplo, nuestra respuesta normal al oir una alarma contra incendios sería levantarnos y salir del lugar donde nos encontramos para ir a un área segura. Sin embargo, la respuesta es diferente si sabemos que la alarma está siendo probada. Nuestro reflejo de retirar la mano del calor, por el contrario, no parece ser cognitivamente penetrable.

Penetrabilidad cognitiva

1.5 ¿PORQUÉ EL RAZONAMIENTO?

La motivación detrás del razonamiento tiene que ver con el hecho de que nos gustaría que la actuación del sistema dependiera no únicamente de lo que está explícitamente representado, sino de lo que el sistema cree como consecuencia de ello. En el segundo programa Prolog, no hay una cláusula que diga que el pasto es verde, y aún así queremos que el sistema sepa esto. Mucho del conocimiento que guardamos en una base tiene que ver con hechos muy generales que necesitarán posteriormente aplicarse en situaciones particulares. Por ejemplo, podríamos querer representar las siguientes dos cláusulas de forma explícita:

1. Paciente X alérgico al medicamento M.
2. Cualquier alérgico al medicamento M también es alérgico al medicamento M'.

Cuando tratemos de decidir si es apropiado darle el medicamento M' al paciente X, ninguno de los hechos representados resuelven la cuestión individualmente. Sin embargo, juntos proveen la información de que X es alérgico a M'. Esto y otros hechos acerca de las alergias, son suficientes para desaconsejar tal medicación. No queremos condicionar el comportamiento solo a aquellos hechos que podamos colectar a manera de una **base de datos**, las creencias de un sistema basado en el conocimiento deben ir más allá.

Base de datos

¿Más allá a donde? Existe una respuesta simple a esta cuestión, aunque no siempre sea práctica: El sistema debe creer P si, conforme a las creencias que tiene representadas, el mundo que se imagina es uno donde P es verdadera. Si las cláusulas (1) y (2) están representadas en el sistema y el mundo es tal que son verdaderas, entonces en ese mundo

3. El paciente X es alérgico al medicamento M'.

también es una cláusula verdadera, aún cuando este hecho está representado solo implícitamente. Esto es ilustra el concepto de **consecuencia lógica**: Decimos que las proposiciones representadas por un conjunto de enunciados Δ tienen como consecuencia la proposición representada por el enunciado P cuando la verdad de P está implícita en la verdad de los enunciados de Δ. En otras palabras, si el mundo es tal que cada enunciado en Δ es verdadero, entonces este es el caso también para P. Todo lo que necesitamos para obtener una noción de consecuencia, es un lenguaje donde esté definido que significa que un enunciado sea falso o verdadero, algo necesario en nuestro caso. De forma que cualquier lenguaje de representación de conocimiento, independientemente de otras características que pueda poseer y sean cuales sean su sintaxis y sus procedimientos de razonamiento, tendría que tener una noción de consecuencia bien definida.

Consecuencia lógica

De manera que una respuesta simple a la cuestión de que creencias debe poseer un sistema basado en el conocimiento, sería: Todas aquellas y solo aquellas que son consecuencia lógica de sus representaciones explícitas. La tarea de un mecanismo de razonamiento es entonces computar las consecuencias lógicas de una base de conocimiento. La razón por la cual tal respuesta no es siempre práctica tiene que ver con que comúnmente hay buenas razones para no computar tales consecuencias. Una razón es que puede ser muy difícil, costoso computacionalmente hablando, calcular que enunciados son consecuencia de la base de conocimiento que nos interesa usar. Todo procedimiento que siempre nos de una respuesta en un tiempo razonable, ocasionalmente perderá algunas consecuencias o nos dará una respuesta equivocada. En el primer caso, decimos que el proceso de razonamiento es lógicamente **incompleto**; en el segundo decimos que el procedimiento no es **sólido** (en inglés, *unsound*).

Completitud
Solidez

Tenemos razones para tomar en consideración el razonamiento no sólido e incompleto. Un ejemplo clásico: Supongamos que todo lo que sé acerca de Piolín es que es un ave. Podría tener un cierto número de hechos acerca de las aves en mi base de conocimiento, pero difícilmente estas tendrían como consecuencia que Piolín vuela (podría ser el caso que Piolín fuese una avestruz). Y sin embargo, es razonable creer que Piolín vuela. Estamos ante un razonamiento no sólido, donde las cláusulas de la base de datos pueden ser verdaderas en el mundo, mientras que Piolín no vuela.

Otro caso a considerar es aquel en donde el sistema integra en su base de conocimiento hechos provenientes de varias fuentes que resultan **inconsistentes**, es decir, no pueden ser verdaderos juntos. En tal caso, es inapropiado computar la consecuencia lógica completa del sistema, porque entonces todo enunciado sería creído: Puesto que no hay mundos donde la base de conocimientos sea verdadera, el consecuente se vuelve trivialmente verdadero. En este caso es evidente que una forma de razonamiento incompleto sería más conveniente, al menos hasta que se pueda resolver la contradicción.

Consistencia

Evidentemente, en algunos casos la respuesta simple es la más adecuada y aunque sea un error equiparar razonamiento en una base de conocimientos con el concepto de inferencia lógica completa y sólida, tal aproximación puede ser un buen comienzo.

1.6 EL PAPEL DE LA LÓGICA

La razón por la cual la **lógica** es relevante para la representación de conocimiento y el razonamiento es que, al menos conforme a un punto de vista, ésta es el estudio de las relaciones de consecuencia –lenguajes, condiciones de verdad y reglas de inferencia. Es por tanto normal, que usemos una gran cantidad de herramientas y técnicas provenientes de la lógica simbólica formal. Particularmente, haremos uso de la **lógica de primer orden** (FOL, por sus siglas en inglés), aunque claro, esto solo es un punto de partida y consideraremos tambien lenguajes bien distintos en forma y significado.

Lógica

FOL

El nivel del conocimiento

Donde la lógica reditúa realmente es en lo que Newell [74] llama *the knowledge level*. La idea es que todo sistema basado en el conocimiento puede entenderse en al menos dos niveles diferentes. En el nivel del conocimiento, hacemos preguntas concernientes a el lenguaje de representación y su semántica. Al nivel del símbolo, por otra parte, hacemos preguntas concernientes a aspectos computacionales. Claramente hay asuntos pertinentes a cada nivel: Al nivel de conocimiento contendemos con la expresividad del lenguaje de representación, las características de su relación de consecuencia, incluyendo su complejidad computacional intrínseca; a nivel del símbolo, contendemos con la arquitectura computacional usada, las propiedades de las estructuras de datos empleadas y los procedimientos de razonamiento, incluyendo su complejidad algorítmica. La Figura 1.3 ilustra las posibles descripciones de una computadora a diferentes niveles.

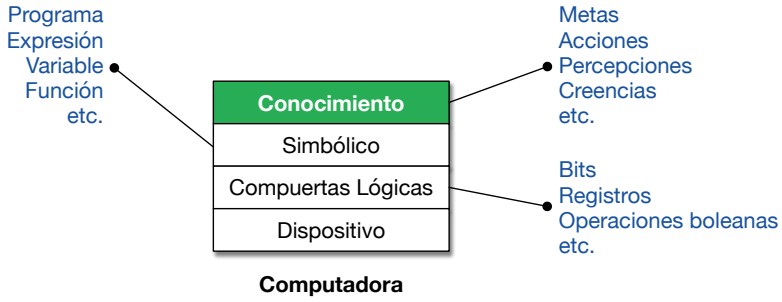


Figura 1.3: Niveles de descripción de una computadora según Newell [74].

Las herramientas de la lógica simbólica parecen adecuadas para el análisis a nivel del conocimiento. En los siguientes capítulos abordaremos la lógica proposicional y de primer orden, dejando de lado por ahora los aspectos computacionales.

1.7 LECTURAS Y EJERCICIOS SUGERIDOS

Esta introducción esta basada en las ideas expresadas por Brachman y Levesque [15], cuyo libro servirá de base a este curso. Una vista rápida de esta visión de la IA puede obtenerse en un artículo previo de Brachman [16]. Levesque [63] ofrece una introducción a nivel de inicio de licenciatura en su libro sobre el pensamiento computacional. Una reseña de mi autoría [3] sobre este texto, puede consultarse en la revista Komputer Sapiens (8):3, de la Sociedad Mexicana de Inteligencia Artificial. Otra perspectiva interesante para introducirse en el área de la representación del conocimiento es la ofrecida por Davis, Shrobe y Szolovits [30]. Harmelen, Lifschitz y Porter [52] proveen una colección de artículos a la manera de un *handbook* muy completo sobre el área. También he tomado en cuenta el libro de Kayser [56] y las conversaciones de pasillo y metro con él, durante mi estancia doctoral en Paris 13.

Casi todos los libros sobre IA incluyen una sección sobre representación del conocimiento. En nuestro caso, el contenido del curso está influenciado por la parte III y parcialmente la IV del libro de Russell y Norvig [95]. El concepto de agente aquí discutido también está influenciado por el libro de Wooldridge [112]. Una excelente introducción al uso de los formalismos lógicos en la IA se encuentra en el libro de Genesereth y Nilsson [43].

La referencia obligada a Prolog es el libro de Bratko [18], que puede complementarse con el de Clocksin y Melish [24]. Otra alternativa es el excelente libro de Sterling y Shapiro [103]. Para revisar la técnicas avanzadas de este lenguaje se recomiendan los libros de Clocksin [23] y de Shoham [99]. Las notas de nuestro curso de Programación para la IA [4] también pueden ser de ayuda en este aspecto.

3 http://smia.mx/komputersapiens/download.php?file=ks83_14MB_extensa.pdf
4 https://www.uv.mx/personal/aguerra/pia/

Negrete-Martínez, González-Pérez y Guerra-Hernández [73] ofrecen una aproximación a los sistemas basados en el conocimiento con implementaciones en Lisp. Una gran parte del libro de Norvig [78] también aborda el uso de Lisp en estas tareas.

Ejercicios

Ejercicio 1.1. *Explique la diferencia entre creencia y conocimiento. Nuestro primer ejemplo de sistema basado en el conocimiento implementado en Prolog ¿Sabe o cree que ciertos objetos son de cierto color?*

Ejercicio 1.2. *¿Porqué razones la cláusula* `bel(color(cielo,amarillo))` *es incorrecta para expresar en Prolog que creemos que el color del cielo es amarillo?*

Ejercicio 1.3. *¿Qué diferencia hay entre la manipulación simbólica que define el razonamiento en este capítulo y la expresada en la hipótesis del sistema simbólico físico propuesta por Newell y Simon [76]?*

Ejercicio 1.4. *Describa un caso en el que la postura intencional nos podría resultar de utilidad de computación.*

Ejercicio 1.5. *Si vemos a Prolog como un sistema basado en el conocimiento, ¿Donde reside su base de conocimientos? ¿Cual es su método de inferencia? ¿Se trata de un método completo, sólido y consistente?*

Ejercicio 1.6. *Describa con detalle las ideas detrás del nivel de conocimiento propuesto por Newell [74] y su relevancia para la IA en general, y este curso en particular.*

2 AGENTES RACIONALES

Dado nuestro señalado interés en la construcción de agentes racionales, en este capítulo delimitaremos el concepto de agente, de manera que podamos utilizarlo para estructurar el resto de este texto. De lo anterior, se sigue que este curso es acerca de la representación de conocimiento y el razonamiento en función del estudio y síntesis de agentes inteligentes. La Sección 2.1 presenta una definición de agente racional con elementos computacionales. La Sección 2.2 introduce una posible caracterización del comportamiento flexible y autónomo que los agentes racionales deben observar, con especial énfasis en la autonomía. La Sección 2.3 introduce el concepto de medio ambiente y su relación con los agentes racionales en él situados. La Sección 2.4 presenta una arquitectura abstracta de agente y su adaptación a diferentes tipos de agentes reportados en la literatura. Finalmente, la Sección 2.5 propone una serie de lecturas suplementarias y ejercicios. Es de esperar que la aproximación informal a estos conceptos facilite este primer encuentro, así como su posterior formalización y estudio a fondo.

Organización

2.1 AGENTES COMPUTACIONALES

Históricamente, fuera de la IA, el término agente ha sido usado con dos acepciones. Primero, a partir de Aristóteles [4] y hasta nuestros días, los **filósofos** usan el término agente para referirse a una entidad que actúa con un **propósito** dentro de un contexto social. Segundo, la noción **legal** de agente, como la persona que actúa en beneficio de otra con un propósito específico, bajo la **delegación** limitada de autoridad y responsabilidad, estaba ya presente en el derecho Romano y ha sido ampliamente utilizada en economía [72]. Como veremos, ambas nociones de agencia han influenciado considerablemente su forma computacional.

Agencia y filosofía

Agencia y derecho

En el contexto de la computación, Wooldridge [112], argumenta que el concepto de agente se consolida como una solución a una demanda propia de los **entornos computacionales actuales**, caracterizados por su: ubicuidad, interconexión, inteligencia, delegación y disponibilidad a amplios sectores de la población (Figura 2.1). Esto es, en entornos donde tenemos una gran variedad de usuarios, con una diversidad de dispositivos de cómputo distribuidos en nuestro entorno e interconectados, los agentes inteligentes emergen como la herramienta para **delegar** adecuadamente nuestro trabajo y abordar esta problemática desde una perspectiva más familiar para usuarios especializados, no especializados, programadores y diseñadores.

Agencia y computación

Pero, **¿Qué es un agente racional artificial?** Como veremos, la respuesta no es sencilla, ni unívoca. Franklin y Graesser [42] argumentan que todas las definiciones del término agente en el contexto de la IA, se basan en alguna de las dos acepciones históricas mencionadas al inicar esta sección. Una definición consensual de agente [113, 95] en este contexto, puede ser:

> Un **agente** es un sistema computacional capaz de **actuar** de manera **autónoma** para satisfacer sus objetivos y **metas**, mientras se encuentra **situado** persistentemente en su medio ambiente.

Definición consensual

Esta definición provee una **abstracción** del concepto de agente basada en su presencia e interacción persistente con el medio ambiente. Como se puede ver en la Figura 2.2, las **acciones** del agente modifican el medio ambiente; mientras que los cambios en el medio ambiente son **percibidos** por el agente. Eventualmente, las ac-

Ventajas de la definición

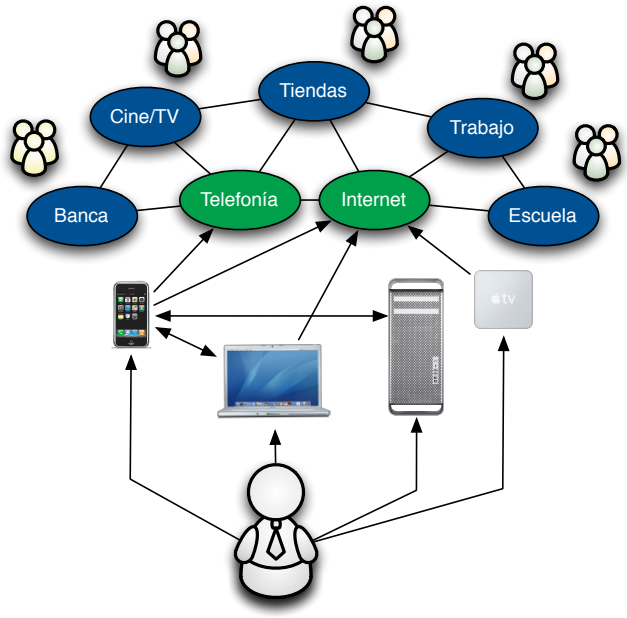


Figura 2.1: Entornos computacionales ubicuos, distribuidos, e interconectados, de amplia disponibilidad, demandan la delegación de tareas en agentes inteligentes. Adaptado de Wooldridge [112].

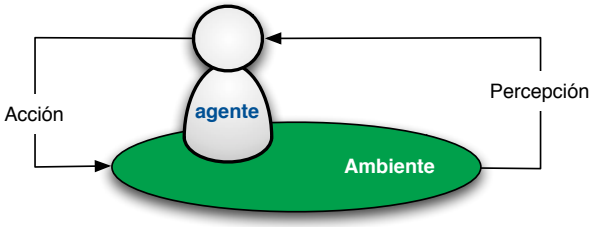


Figura 2.2: Abstracción de un agente a partir de su interacción persistente con el medio ambiente, para la consecución de sus metas.

ciones del agente lo llevan a la consecución de sus objetivos. Russell y Subramanian [96] encuentran que tal abstracción presenta tres ventajas:

1. Nos permite observar las las facultades cognitivas de los agentes al servicio de encontrar cómo hacer lo correcto.
2. Permite considerar diferentes tipos de agente, incluyendo aquellos que no se supone tengan tales facultades cognitivas.
3. Permite considerar diferentes especificaciones sobre los subsistemas que componen los agentes.

Problemas de la definición

Sin embargo, por si misma la definición consensual de agente resulta demasiado **general**. Consideren el Ejemplo 2.1 de un programa UNIX que difícilmente puede ser concebido como un agente, pero se ajusta a la definición consensual del término.

Ejemplo 2.1. *La Figura 2.3 muestra a* `xbiff`*, una pequeña interfaz gráfica para* `biff`*, un sistema de notificaciones de correo electrónico para UNIX. Cuando un mensaje llega al buzón del usuario,* `biff` *emite una alerta. Por su parte* `xbiff` *usa la alerta para desplegar el ícono que se muestra en la ventana superior izquierda.*

Esta pequeña aplicación se las arregla para identificar a su usuario, encontrar su buzón de correo electrónico, buscar mensajes nuevos y comunicar al usuario

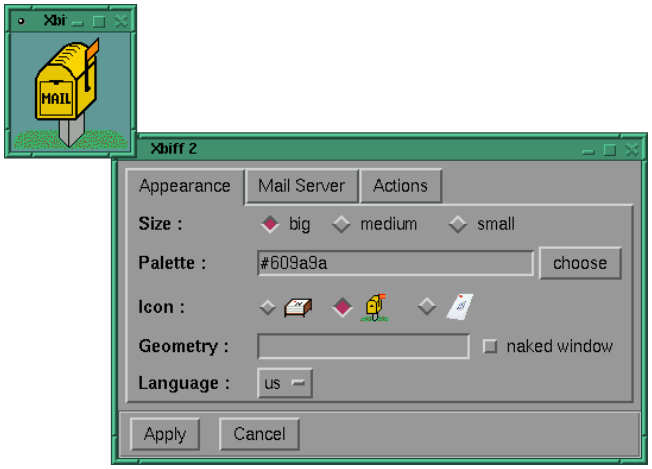


Figura 2.3: ¿Es la aplicación `xbiff` un agente?

la presencia de éstos. Es decir, `xbiff` es un sistema de cómputo que actúa persistentemente en su entorno UNIX, percibiendo la llegada de mensajes nuevos para satisfacer su objetivo: tenernos al tanto de ello. Además, las tres ventajas de la abstracción de agente aplican aquí: Puede que `xbiff` no sea muy listo, pero **hace lo correcto** y su funcionamiento es claro, independientemente de la especificación e implementación subyacentes. ¿Es `xbiff` un agente?

Primero, observen que hacer los correcto nos provee de una aproximación indirecta a la racionalidad, que además resulta funcional. Se dice que un agente es **racional** si hace lo correcto. Esto reduce el problema de establecer cuando un agente es racional, al de definir cómo y cuando evaluar el éxito del agente. El término **medida de desempeño** se refiere al criterio usado para determinar el éxito de un agente. Es preferible manejar una medida de desempeño objetiva impuesta por alguna forma de autoridad. Esto es, nosotros como observadores estableceremos para cada agente, un estándar de lo que significa ser exitoso en un ambiente dado, y usamos ese estándar para medir el desempeño del agente. Bajo ninguna circunstancia, el agente puede manipular [1] tal estándar. El Ejemplo 2.2 nos muestra una medida de desempeño típica de los concursos de robótica, que cumple con las características mencionadas.

Desempeño

Ejemplo 2.2. *En el 3^{er} Torneo Nacional de Robots Limpiadores 2006 (Xalapa, Ver., México), el desempeño del agente fue establecido por un estándar externo al agente, especificado mediante la formula de evaluación diseñada por los organizadores:*

$$PT = \sum_i (CL_i \times DA_i) \times (TMax/TOper)$$

donde: PT es el puntaje total. El segundo término evalúa la persistencia del agente: TMax es el tiempo máximo de la prueba y TOper es el tiempo que el robot se mantuvo en operación. Si el robot trabaja durante toda la prueba no es penalizado, si no la puede terminar, si lo es. Si el robot no arranca, este factor es cero, y por tanto el puntaje total del agente es cero. El primer término evalúa la capacidad limpiadora del agente. Para ello el piso a limpiar, rectangular, se divide conceptualmente en celdas. CL_i es la i-ésima celda limpiada con éxito, o si se prefiere toma valor 1 si la celda se limpio, y 0 en caso contrario. DA_i es el valor de dificultad de acceso a la i-ésima celda limpiada. El robot que limpia más celdas de mayor dificultad y funciona más tiempo es el ganador. La función está acotada entre cero y un máximo, por lo que existe un desempeño óptimo.

Segundo, observen que la racionalidad de un agente se define en relación con el **éxito esperado**, dadas sus circunstancias. Esto es, no podemos exigir a un agente

1 Un agente que manipula la medida de desempeño, suele llegar a razonamientos del tipo –Al fin y al cabo, ¡Quién diablos quiere una maestría!

que tome en cuenta lo que no puede percibir, o haga lo que sus efectores no pueden hacer. Es decir, además de la **medida de desempeño**, otros factores presentes al evaluar la racionalidad de un agente incluyen:

Evaluación de la racionalidad

- La **secuencia de percepciones** del agente, esto es, todo lo que el agente haya percibido hasta el momento de la evaluación.
- El **conocimiento** del agente sobre el medio ambiente en el que está situado y el problema a resolver.
- Las **competencias** del agente, entendidas como el repertorio de acciones que éste puede llevar a cabo en su medio ambiente.

Con los elementos anteriores, podemos definir un **agente racional ideal**, como aquel que para toda secuencia de percepciones posible, selecciona y ejecuta una acción que se espera maximice la medida de desempeño, con base en la información que proveen su percepción y conocimiento sobre el ambiente. Observen que podríamos describir a un agente por medio de una tabla o una **función de comportamiento**, donde se establecen las acciones que el agente toma en respuesta a cada posible secuencia de percepciones. Por lo tanto una **función ideal** describe a un agente ideal y define el diseño de un agente ideal. El ejemplo 2.3 muestra la función ideal de `xbiff`.

Agente ideal

Función de comportamiento

Ejemplo 2.3. *En el caso de `xbiff` la función ideal es muy sencilla. Si `biff` avisa que llegó un mensaje nuevo, desplegar el icono de mensaje nuevo; en cualquier otro caso, desplegar el icono de mensajes leídos.*

Y es aquí donde los problemas de `xbiff` para ser calificado como un agente comienzan. ¿Porqué si esta aplicación cumple con la definición consensual de agente y tiene una función ideal, difícilmente se le ve como un agente? Primero, es muy difícil que un programa con una función de comportamiento ideal tan simple, sea concebido como un agente. De hecho, lo normal es que no sea factible enumerar todas las entradas de la función de comportamiento ideal de un agente, es decir, el número de posibles secuencias de percepción a considerar, suele ser muy grande. De hecho, un problema central en la programación de agentes es la búsqueda de mecanismos que aproximen las funciones ideales que no pueden definirse por extensión. A esto se le conoce como el problema de **selección de acción**. Precisamente, como dice Maes [67], el problema de ¿Cómo hacer lo correcto? Y es que, segundo, el repertorio de acciones de un agente suele mayor que el de `xbiff`, por lo que una selección es necesaria. En particular, el comportamiento de `xbiff` difícilmente puede verse como flexible y autónomo.

Selección de acción

2.2 COMPORTAMIENTO FLEXIBLE Y AUTÓNOMO

Independientemente de la implementación usada para construir a `xbiff`, no resulta natural identificar a los daemons de UNIX como agentes y menos aún como agentes inteligentes. Foner [41] argumenta que para ser percibido como inteligente, un agente debe exhibir cierto tipo de comportamiento, denominado más tarde por Wooldridge y Jennings [113], **comportamiento flexible y autónomo**. Este tipo comportamiento se caracteriza por su:

Flexibilidad y autonomía

- **Reactividad.** Los agentes inteligentes deben ser capaces de percibir su medio ambiente y responder a tiempo a los cambios en él, a través de sus acciones; así como explotar las oportunidades que se presentan para conseguir sus metas.
- **Iniciativa.** Los agentes inteligentes deben exhibir un comportamiento orientado por sus metas, tomando la iniciativa para satisfacer sus objetivos de diseño y posiblemente decidiendo que objetivo atender.

- **Sociabilidad.** Los agentes inteligentes deben ser capaces de interactuar con otros agentes, posiblemente tan complejos como los seres humanos, con miras a la satisfacción de sus objetivos. Esto incluye la capacidad de comunicarse, negociar y alcanzar acuerdos.

Una caracterización más detallada de la autonomía es presentada por Covrigaru y Lindsay [28]. Su *desiderata*, que incluye algunos de los aspectos ya mencionados, expresa que un agente se percibe como **autónomo** en la medida en que:

Autonomía en detalle

1. Su comportamiento está orientado por sus metas y es capaz de seleccionar que meta va a procesar a cada instante.
2. Su existencia se da en un período relativamente mayor al necesario para satisfacer sus metas.
3. Es lo suficientemente robusto como para seguir siendo viable a pesar de los cambios en el ambiente.
4. Puede interaccionar con su ambiente en la modalidad de procesador de información.
5. Es capaz de exhibir una variedad de respuestas, incluyendo movimientos de adaptación fluidos; y su atención a los estímulos es selectiva.
6. Ninguna de sus funciones, acciones o decisiones, está totalmente gobernada por un agente externo.
7. Una vez en operación, el agente no necesita ser programado nuevamente por un agente externo.

El sexto punto requiere una defensa rápida, puesto que la dimensión social de los agentes no será abordada aquí, sino en el curso de Sistemas Multi Agentes [2]. En este punto se puede adivinar una tensión entre autonomía y sociabilidad ¿Para qué necesita un agente autónomo socializar con otros agentes? La filosofía política liberal parece resolver esta cuestión: es aceptado que los agentes solo pueden llegar a ser autónomos si se da un conjunto de condiciones necesarias para ello. Dentro de estas condiciones, Rawls [90] habla de un **contexto de elección**, una pluralidad de bienes primarios que proveen los medios necesarios para que el agente tenga éxito en la satisfacción de sus intenciones. Tal pluralidad es posible únicamente, si el agente tiene una relación cercana con su **ambiente social y cultural**. No solo ambos conceptos no están reñidos, sino que no puede haber autonomía sin sociabilidad. En el contexto de la IA, es Newell [75] quien ofrece argumentos similares.

Autonomía y sociabilidad

Covrigaru y Lindsay argumentan además, que ser autónomo depende no solo de la habilidad para seleccionar metas u objetivos de entre un conjunto de ellos, ni de la habilidad de formularse nuevas metas, sino de tener el tipo adecuado de metas. Los agentes artificiales son usualmente diseñados para llevar a cabo tareas por nosotros, de forma que debemos comunicarles que es lo que esperamos que hagan. En un sistema computacional tradicional esto se reduce a escribir el programa adecuado y ejecutarlo. Un agente puede ser instruido sobre que hacer usando un programa, con la ventaja colateral de que su comportamiento estará libre de incertidumbre. Pero programar un agente de esta forma, atenta contra su autonomía, teniendo como efecto colateral la incapacidad del agente para enfrentar situaciones imprevistas mientras ejecuta su programa. Las **metas** y las funciones de **utilidad** son dos maneras de indicarle a un agente lo que hacer, sin decirle cómo hacerlo.

Metas, utilidad y autonomía

Con los elementos discutidos hasta ahora, es posible definir a un **agente racional** como aquel que exhibe un comportamiento flexible y autónomo, mientras está situado en un sistema de información, por ej., un sistema operativo o el web. Esto constituye lo que Wooldridge y Jennings [113] llaman la **noción débil** de agente,

Agencia débil y fuerte

2 https://www.uv.mx/personal/aguerra/sma

como contrapunto a una **noción fuerte** de agencia, que además de las propiedades mencionadas hasta ahora, utiliza términos que hacen referencia a estados que solemos aplicar exclusivamente a los seres humanos: Creencias, deseos, intenciones.

2.3 MEDIO AMBIENTE

Real o virtual

Por medio ambiente, entendemos el espacio donde un agente, o un grupo de ellos, se encuentra **situado**. Brooks [19] argumenta que el medio ambiente por excelencia es el mundo real, y en su propuesta todo agente toma una forma robótica. Por el contrario, Etzioni [38], considera que no es necesario que los agentes tengan implementaciones robóticas porque los ambientes virtuales, como los sistemas operativos y el web, son igualmente válidos que el mundo real.

Artefactos

Espacio de trabajo

Organización

En este texto asumimos una tercera vía propuesta por Ricci [79, 92], donde la interacción de los agentes con el medio ambiente se concibe como una interfaz, una capa de servicios compuesta por objetos reactivos, más no autónomos, ni con iniciativa, llamados **artefactos** . Como es de esperar, los agentes pueden usar estos artefactos para sensar y actuar en el medio ambiente. Agentes y artefactos pueden organizarse en **espacios de trabajo**, para reflejar la topología del ambiente y proveer una noción de localidad ó sitio. Esto constituye un meta-modelo que puede aproximar ambientes tanto reales, como virtuales. La Figura 2.4 ilustra esto, junto con un componente social propio de los Sistemas Multi Agente: Las **organizaciones**.

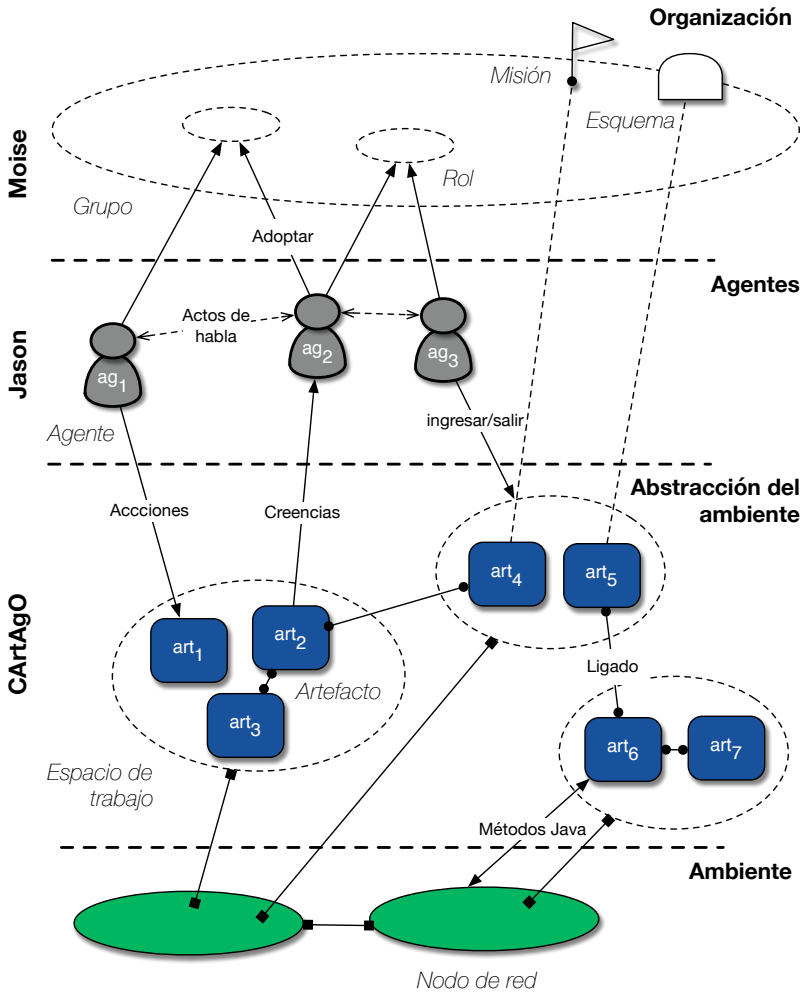


Figura 2.4: Vista general del esquema de Agentes y Artefactos, extendido con organizaciones: JaCaMo.

Entorno de trabajo

El concepto de espacio de trabajo se asemeja al de **entorno de trabajo**, propuesto por Russell y Norvig [95] para especificar tipos de agente con base en la medida de desempeño utilizada, el ambiente donde el agente está situado, los actuadores del agente y sus sensores. Tal especificación suele conocerse como **PEAS** por sus siglas en inglés (*Performance, Environment, Actuators, Sensors*). Consideren el siguiente Ejemplo de un agente taxista:

Descripción PEAS

Ejemplo 2.4. *Un agente taxista tiene la siguiente especificación PEAS:*

- ***Medida de desempeño.*** *Un taxista artificial debería: llegar al destino correcto; minimizar el consumo de gasolina; minimizar el tiempo y costo de la carrera; minimizar las infracciones de tránsito y las molestias ocasionadas a otros conductores; maximizar la seguridad y comodidad del pasajero; maximizar las ganancias. Observen que algunos componentes del desempeño son contradictorios y el agente deberá contender con tales contradicciones.*
- ***Ambiente.*** *La calle, o mejor aún las posibles calles donde conducirá. Esto incluye avenidas, callejones, caminos rurales, autopistas donde suele haber otros autos, peatones, perros, vacas, hoyos; y eventualmente nieve. O un simulador. Evidentemente, entre más restringido sea el ambiente, más sencillo es el diseño del agente.*
- ***Actuadores.*** *Incluye el control de auto: acelerador, freno, cambio de velocidad, puertas, ventanas, limpia brisas, etc.; Y alguna forma de comunicarse con los clientes y otros autos, por ejemplo, una pantalla donde despegar mensajes, una bocina para hablar, etc.*
- ***Sensores.*** *Video cámaras controlables para ver el camino; sensores infra rojos o sonares para medir distancias; velocímetro; acelerómetro; sensores electrónicos de motor, gasolina, sistema eléctrico; GPS; Y posiblemente micrófono, pantalla táctil y/o teclado para comunicarse con el usuario.*

En el caso de los espacios de trabajo, sensores y actuadores están **encapsulados** en los artefactos. Esto tiene una ventaja inherente: se puede usar cualquier metodología de programación para abordar estos componentes de cara al ambiente, por ejemplo una metodología orientada a objetos bajo Java, con su riqueza de librerías y extensibilidad; de cara al agente los artefactos actualizan su representación del ambiente y proveen una serie de operaciones que el agente puede ejecutar al usar sus artefactos.

Agente	Desempeño	Ambiente	Actuadores	Sensores
diagnóstico médico	salud paciente, minimizar costos y demandas	paciente, hospital, persona, tratamientos	preguntas, pruebas, diagnósticos,	lecturas, reportes, respuestas
análisis imágenes de satélite	clasificación correcta	satélite, control	despliegue imagen	color, intensidad
brazo robótico	porcentaje de piezas colocadas	línea de producción	codo, mano	cámara, ángulo de articulación

Cuadro 2.1: Ejemplos de agentes y su descripción PEAS según Russell y Norvig [95].

La descripción PEAS puede verse con una fase previa al diseño de artefactos y espacios de trabajo. De hecho, una descripción PEAS adecuada, es indispensable para el diseño de agentes siguiendo prácticamente cualquier metodología. El Cuadro 2.1 muestra otros ejemplos de descripciones PEAS. Ahora bien, los elementos que componen este tipo de descripciones nos permiten concebir diferentes **propiedades** de los entornos de trabajo:

Propiedades del entorno de trabajo

- **Ambiente y sensores.** Si los sensores de un agente le permiten percibir el estado completo del ambiente en cualquier momento, decimos que el entorno de

trabajo es **observable**. Los juegos formales, por ejemplo: crucigramas, ajedrez, backgammon, etc. constituyen entornos de trabajo observables. Un entorno de trabajo es **eficazmente observable** si los sensores del agente detectan todos los aspectos relevantes para decidir que acción debe llevarse a cabo. Relevancia aquí depende de la definición de la medida de desempeño. Las líneas de montaje donde se situan brazos robóticos suelen ser entornos de trabajo efectivamente observables. Un entorno de trabajo puede ser **parcialmente observable** debido a la imprecisión y el ruido en los sensores; o bien porque algunos aspectos del ambiente caen fuera del rango de lectura de los sensores. El poker, los ambientes donde navegan los robots o conducimos nuestros autos suelen ser parcialmente observables. Los entornos de trabajo observables son los más convenientes, ya que un agente no necesita mantener el historial de estados del ambiente para ser efectivo en ellos. Si el agente no tiene sensores , se dice que el entorno de trabajo es **inobservable**.

Observable

Eficazmente observable

Parcialmente observable

Inobservable

- **Número de agentes y sus interacciones.** Si un agente resuelve solo su tarea, ya sea porque es el único agente en el ambiente, o porque no interactua con los demás agentes, decimos que el entorno de trabajo es **monoagente** . En caso contrario, decimos que se trata de un entorno de trabajo **multiagente**. A pesar de que esta categorización de los ambientes parece obvia, esconde una cuestión en extremo relevante: ¿Qué entidades en el sistema son considerados por el agente como agentes? Como hemos visto, el concepto de artefacto nos permite concebir entidades en el ambiente que no son agentes y el comportamiento autónomo y flexible es la clave para distinguir ambas entidades. Sin embargo, la situación puede ser ambigua. Por ejemplo, en el caso del agente taxista,¿Debe el agente concebir los otros autos como agentes o como artefactos? La clave aquí está en el concepto de concepto de **interacción**. Si el agente ve al otro auto como un objeto que sigue las leyes de la física mientras se mueve en el mismo espacio, posiblemente se le deba modelar como un artefacto. Sin embargo, si el otro auto se concibe como una entidad autónoma y flexible que me puede obstaculizar o ayudar a circular más rápido, definitivamente debe ser modelado como un agente. De forma que puede haber entornos de trabajo **competitivos**, como el ajedrez, o los otros autos buscando el mejor carril; o **colaborativos** cuando por ejemplo, otro auto me deja pasar, o un grupo de agentes busca dar el mejor diagnóstico posible. Los entornos de trabajo multiagente introducen aspectos que no se consideran en el caso monoagente, por ejemplo: comunicación, negociación, colaboración, etc.

  Monoagente

  Multiagente

  Competitivo

  Colaborativo

- **Ambiente y actuadores.** Si el próximo estado del ambiente depende exclusivamente de su estado actual y de la acción que ejecuta el agente, se dice que el ambiente es **determinista**. Evidentemente, todos los juegos formales son deterministas. Si otros factores influyen en el próximo estado del ambiente, éste es **estocástico**. Si el ambiente es parcialmente observable, entonces parecerá ser estocástico. Esto es particularmente cierto en el caso de ambientes complejos, donde es difícil dar seguimiento a los aspectos del ambiente. Si el entorno de trabajo es estocástico o parcialmente observable, decimos que es **incierto**. El término estocástico refuerza aquí el hecho de que en estos entornos, la incertidumbre se cuantifica en términos de probabilidades. Un entorno de trabajo se dice **no determinista** si los actuadores se caracterizan en términos de sus posibles resultados, pero ninguna probabilidad es asociada a estos. El carácter estocástico o no determinista del entorno de trabajo captura dos nociones importantes:

  Determinista

  Estocástico

  Incierto

  No determinista

  1. El hecho de que los agentes tienen una esfera de influencia limitada, es decir, en el mejor de los casos tienen un control parcial de su ambiente;
  2. y el hecho de que las acciones de un agente puede fallar y no lograr el resultado deseado por el agente.

Si el entorno de trabajo es determinista, excepto por las acciones de otros agentes, como en los juegos con contrincante, se dice que es **estratégico**. Es más sencillo construir agentes en entornos de trabajo deterministas.

Estratégico

- **Episódico o secuencial.** Si el desempeño del agente puede evaluarse en rondas, se dice que el ambiente es **episódico**. Las acciones se evalúan en cada episodio o ronda, esto es, la calidad de la acción en los episodios subsecuentes no depende de las acciones ocurridas en episodios previos. Por ejemplo, el detector de basura en las botellas de una cervecería es episódico: la decisión de si una botella esta sucia o no, no depende de los casos anteriores. Dada la persistencia temporal de los agentes, estos tienen que hacer continuamente decisiones locales que tienen consecuencias globales. Los ambientes episódicos reducen el impacto de estas consecuencias, y por lo tanto es más fácil construir agentes en ambientes episódicos. Si el ambiente no es episódico, se dice que es **secuencial**.

  Episódico

  Secuencial

- **Estático o Dinámico.** Si el ambiente puede cambiar mientras el agente se encuentra deliberando, se dice que es **dinámico**; de otra forma, se dice que es **estático**. Si el ambiente no cambia con el paso del tiempo, pero si lo hace con las acciones del agente, se dice que el ambiente es **semi-dinámico**. Los ambientes dinámicos tienen dos consecuencias importantes:

  Dinámico

  Estático

  Semi-dinámico

  1. Un agente debe percibir continuamente, porque aún si no ha ejecutado ninguna acción entre los tiempos t_0 y t_1, el agente no puede asumir que el estado del ambiente sea el mismo en t_0 que en t_1;
  2. Otros procesos en el ambiente pueden interferir con las acciones del agente, incluyendo las acciones de otros agentes.

  Es más sencillo diseñar agentes en ambientes estáticos.

- **Discreto o Contínuo.** Si hay un número limitado de posibles estados del ambiente, distintos y claramente definidos, se dice que el ambiente es **discreto**; de otra forma se dice que es **contínuo**. Esta propiedad puede aplicarse al estado del ambiente; a la forma en que se registra el tiempo y; a las percepciones y acciones de los agentes.Es más fácil construir agentes en ambientes discretos, porque las computadoras también son sistemas discretos y aunque es posible simular sistemas continuos con el grado de precisión deseado, una parte de la información disponible se pierde al hacer esta aproximación. Por lo tanto, la información que manejan los agentes discretos en ambientes continuos es inherentemente aproximada.

  Discreto

  Continuo

Ambiente	Observ.	Determ.	Episódico	Estático	Discreto	SMA
Crucigrama	si	si	no	si	si	mono
Ajedrez con reloj	si	estratégico	no	semi	si	multi
Backgammon	si	estocástico	no	si	si	multi
Poker	parcial	estocástico	no	si	si	multi
Tutor inglés	parcial	estocástico	no	no	si	multi
Brazo robótico	efectivo	estocástico	si	no	no	mono
Control refinería	parcial	estocástico	no	no	no	mono
Robot navegador	parcial	estocástico	no	no	no	mono
Análisis imágenes	si	si	si	semi	no	mono
Manejo de autos	parcial	estocástico	no	no	no	multi
Diagnóstico	parcial	estocástico	no	no	no	mono

Cuadro 2.2: Ejemplos de ambientes estudiados en IA y sus propiedades según Russell y Norvig [95].

Esta categorización sugiere que es posible encontrar diferentes clases de entornos de trabajo. Russell y Norvig [95] presentan algunos ejemplos de ambientes bien

estudiados en Inteligencia Artificial y sus propiedades (ver Cuadro 2.2). Cada ambiente, o clase de ambientes, requiere de alguna forma agentes diferentes para que estos tengan éxito. La clase más compleja de ambientes corresponde a aquellos que son inaccesibles, no episódicos, dinámicos, continuos y multi-agente, lo que desgraciadamente corresponde a nuestro ambiente cotidiano.

2.4 ARQUITECTURAS DE AGENTE

Es posible formalizar la definición abstracta de agente que hemos presentado (pág. 11), en la forma de una **arquitectura abstracta** de agentes [112]. Primero asumiremos que el ambiente puede caracterizarse por medio de un conjunto finito de **estados** discretos posibles, definido como:

Arquitectura abstracta
Estados

$$E = \{e, e', \dots\}$$

Como hemos visto, no todos los ambientes son discretos, en el sentido estricto del término. Sin embargo, asumir que el ambiente es discreto facilitará esta primera aproximación formal al concepto de agente. Por otra parte, este supuesto se justifica por el hecho de que aún cuando el ambiente no fuese discreto, éste puede ser modelado como un ambiente discreto con cierto nivel deseable de precisión.

Definamos la **competencia** o **capacidad** de un agente, como el conjunto finito de **acciones** que éste puede ejecutar:

Acciones

$$Ac = \{\alpha, \alpha', \dots\}$$

Ahora, supongamos que el ambiente se encuentra en algún estado inicial e_0 y que el agente comienza su actividad eligiendo una acción α_0 en ese estado. Como resultado de esta acción, el ambiente puede responder con cierto número de estados posibles. Eventualmente, uno de estos estados posibles resulta de esta interacción, digamos e_1. Inmerso en este nuevo estado del ambiente, el agente selecciona otra acción a ejecutar, y así continua. Una **corrida** r de un agente en un ambiente es, por lo tanto, una secuencia finita de estados y acciones intercalados:

Corrida

$$r = e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \dots \xrightarrow{\alpha_{u-1}} e_u$$

Sea R el conjunto de todas las **posibles corridas finitas** sobre E y Ac. Definimos R^{Ac} como el subconjunto de las corridas que terminan en una acción y R^E como el subconjunto de las corridas que terminan en un estado del ambiente.

Corridas posibles

Para modelar el efecto de una acción en el ambiente, usamos una **función de transición** entre estados [3] que mapea una corrida terminando en acción a un conjunto de posibles estados del ambiente:

Función de transición

$$\tau : R^{Ac} \to \wp(E)$$

En esta forma, el ambiente se asume como sensible a su historia, esto es, su próximo estado no está determinado exclusivamente por la acción ejecutada por el agente en el estado actual. Las acciones ejecutadas por el agente en el pasado, también afectan esta transición. Esta definición introduce un grado de indeterminismo en el ambiente, y por lo tanto, incertidumbre sobre el resultado de una acción ejecutada en cierto estado. Como vimos, considerar que el ambiente es no determinista, es importante para entender las limitaciones de los agentes.

Si $\tau(r) = \emptyset$ para todo $r \in R^{Ac}$, entonces no hay posibles estados sucesores para r y se dice que el sistema ha **terminado** su corrida. Asumimos que todas las corridas terminan eventualmente.

Condición de paro

Un **ambiente** se define como $Env = \langle E, e_0, \tau \rangle$ donde E es el conjunto de los posibles estados del ambiente, $e_0 \in E$ es un estado inicial y τ es la función de transición de estados.

Ambiente

Los **agentes** se modelan como funciones que mapean corridas que terminan en un estado del ambiente, a acciones: *Agente*

$$Ag : R^E \rightarrow Ac$$

Observen que mientras los ambientes se asumen no deterministas, asumimos que los agentes son deterministas.

Un **Sistema agente** es una tupla conformada por un agente y un ambiente. El conjunto de posibles **corridas** del agente Ag en el ambiente Env se denota como $R(Ag, Env)$. Por simplicidad, consideraremos por ahora solo corridas finitas. Una secuencia de la forma $(e_0, \alpha_0, e_1, \alpha_1, e_2, \dots)$ representa una corrida del agente Ag en el ambiente $Env = \langle E, e_0, \tau \rangle$ si y solo si e_0 es el estado inicial del ambiente Env; $\alpha_0 = Ag(e_0)$; y para $u > 0$: *Sistema agente*

$$e_u \in \tau((e_0, \alpha_0, \dots, \alpha_{u-1})) \text{ y } \alpha_u = Ag((e_0, \alpha_0, \dots, e_u))$$

Puesto que nuestra tarea es implementar **programas de agente**, podemos usar la formalización propuesta para definir un programa de agente que acepte percepciones de su ambiente y regrese acciones sobre éste. Como habíamos mencionado, la forma más sencilla de implementar un programa de agente es a través de su **mapeo ideal**. El Algoritmo 2.1 muestra la función que implementa un agente de este tipo. *Programa de agente* *Agente mapeo ideal*

Algoritmo 2.1 Programa de agente basado en mapeo ideal

```
1: function AGENTE-MAPEO-IDEAL(p)          ▷ p es una percepción del ambiente.
2:     push(p, perceps)                    ▷ perceps es una cola de percepciones.
3:     acción ← tabla(perceps)             ▷ tabla : perceps → acciones predefinida.
4:     return acción
5: end function
```

Aunque ésta es también la forma más sencilla de fracasar en la construcción de un agente inteligente. Aún para agentes en ambientes controlados como el ajedrez, la tabla del mapeo alcanza ordenes de 10^{150} entradas. Si consideramos que el número de átomos en el universo observable es menor que 10^{80}, entenderemos las razones del fracaso ineludible de esta estrategia.

Algoritmo 2.2 Programa de ambiente

```
1: procedure AMBIENTE(e, τ, ags, fin)      ▷ e es el estado inicial del ambiente.
2:     repeat
3:         for all ag ∈ ags do             ▷ ags es un conjunto de agentes.
4:             p(ag) ← percibir(ag, e)
5:         end for
6:         for all ag ∈ ags do
7:             acción(ag) ← ag(p(ag))
8:         end for
9:         e ← τ(⋃_{ag∈ags} acción(ag))    ▷ τ es una función de transición del ambiente.
10:    until fin(e)                        ▷ fin es un predicado de fin de corrida.
11: end procedure
```

Podemos también implementar un programa básico de ambiente que ilustre la relación entre éste y los agentes situados en él. El Algoritmo 2.2 muestra el programa del **ambiente**. La historicidad del ambiente queda oculta en las percepciones del agente. *Programa del ambiente*

La función *percibir*/2 mapea estados del ambiente a alguna entrada perceptiva. Los agentes con formas robóticas, pueden implementar *percibir*/2 usando hardware, por ejemplo, cámaras de vídeo, sensores infrarrojos, sonares, etc. Los agentes software pueden implementar esta función usando comandos del sistema operativo. El Ejemplo 2.5 muestra diferentes opciones de percepción para el agente `xbiff`.

3 Véase Fagin et al. [39], p. 154, pero aquí τ es aplicada a un solo agente.

Ejemplo 2.5. *Un agente del estilo de* `xbiff` *puede usar el comando de UNIX* `whoami` *para determinar la identidad de su usuario y encontrar así su buzón de correo. Luego puede usar el comando* `grep` *para contar el número total de mensajes, el número de mensajes leídos, y el número de mensajes abiertos pero no leídos* [4].

Ahora bien, sea *Per* un conjunto no vacío de percepciones, la función de *percibir*/2 se define como el mapeo del conjunto de estados del ambiente *E* al conjunto de percepciones posibles *Per*: *Percepción*

$$percibir : E \rightarrow Per$$

El Ejemplo 2.6 muestra la diferencia entre percepción y estados posibles del ambiente, en el caso del agente `xbiff`.

Ejemplo 2.6. *Para el agente de estilo* `xbiff`*, podemos definir es conjunto de percepciones posibles como* $Per = \{mensajesNuevos, noMensajesNuevos\}$*, mientras que los estados posibles del ambiente en E son más complejos, incluyendo como mencioné, el* `login` *del usuario, el número de cada tipo de mensaje en el buzón, etc.*

Levitin [64], en su libro sobre nuestro cerebro y la música, ilustra esta diferencia. La palabra tono se refiere a la representación mental que un organismo tiene de la frecuencia fundamental de un sonido. Un fenómeno enteramente psicológico relacionado con la frecuencia de las moléculas de aire vibrando a nuestro alrededor. Por psicológico entendemos que es resultado de una serie de procesos físico químicos en el agente que percibe el tono; que dan lugar a esa representación interna enteramente subjetiva. Isaac Newton fue el primero en darse cuenta que el color se percibe de manera similar –la luz no tiene color, ésta es una construcción de nuestro cerebro. El filósofo Georges Berkeley planteo esta curiosa pregunta: Si un árbol cae en el bosque y nadie está ahí para escucharlo caer ¿Produjo algún sonido?

La función *acción*/1 se define entonces como el mapeo entre conjuntos de percepciones *Per* y el conjunto de acciones posibles *Acc* del agente: *Acción*

$$acción : Per \rightarrow Acc$$

Un **agente** puede definirse ahora como $Ag = \langle percibir, acción \rangle$. Usando esta extensión, es posible expresar interesantes propiedades sobre la percepción de los agentes. Supongamos dos diferentes estados del mismo ambiente, $e_1 \in E$ y $e_2 \in E$, tal que $e_1 \neq e_2$ pero $percibir(e_1) = percibir(e_2)$. Entonces tenemos dos estados diferentes del ambiente que mapean a la misma percepción, y desde el punto de vista del agente e_1 y e_2 son **indistinguibles**. *Agente*

Ejemplo 2.7. *Observe que la definición de Per para el agente estilo* `xbiff`*, oculta al agente la identidad del usuario! Todo lo que el agente "cree" es que hay un buzón con o sin mensajes nuevos. Si son mis mensajes o los mensajes del usuario que entró más tarde al sistema, resulta indistinguible para nuestro agente.*

Dados dos estados del ambiente $e, e' \in E$, $percibir(e) = percibir(e')$ será abreviado como $e \sim e'$. Observen que "$\sim$" es una relación de equivalencia sobre *E*. El ambiente es accesible para el agente, si y solo si $|E| = | \sim |$ y entonces se dice que el agente es **omnisciente**. Si $| \sim | = 1$ entonces se dice que el agente no tiene capacidad de percepción, es decir, el ambiente es percibido por el agente como si tuviera un estado único, porque todos los estados de éste son idénticos desde la perspectiva del agente. *Omnisciencia*

2.4.1 Agentes reactivos

Consideren ahora nuestro primer programa de agente concreto: Un **agente reacti-** *Agente reactivo*

4 Todas estas etiquetas para los mensajes de correo electrónico se encuentran especificadas por el el estándar de formato para mensajes de texto en ARPA Internet (RFC# 822).

vo [19], o reflex [95], selecciona sus acciones con base en su percepción actual del ambiente, ignorando el resto de su historia perceptiva. La Figura 2.5 ilustra la interacción de estos agentes con su ambiente. El Ejemplo 2.8 nos muestra la formulación del programa `xbiff` como si fuese un agente reactivo.

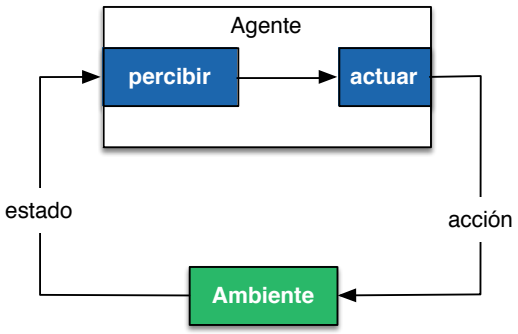


Figura 2.5: Abstracción de un agente reactivo.

Ejemplo 2.8. *Los estados posibles del ambiente, las acciones y la función de selección de acción para* `xbiff`*:*

$$E = \{correoNuevo\}$$

$$Ac = \{despliegaIconoCorreoNuevo, despliegaIconoCorreoViejo\}$$

$$Ag(e) = \begin{cases} despliegaIconoCorreoNuevo & Si\ e = correoNuevo \\ despliegaIconoCorreoViejo & En\ otro\ caso \end{cases}$$

La manera más sencilla de implementar este tipo de programa de agente es mediante reglas **condición-acción**. El Algoritmo 2.3 muestra la forma general de un programa de agente reactivo.

Algoritmo 2.3 Programa de agente reactivo o reflex

```
1: function AGENTE-REACTIVO(e)
2:     estado ← percibir(e)
3:     regla ← selecciónAcción(estado, reglas)      ▷ reglas condición-acción predefinidas.
4:     acción ← cons(regla)                          ▷ cons, el consecuente de la regla.
5:     return acción
6: end function
```

De manera que la formulación de `xbiff` puede transformarse, utilizando reglas condición-acción, en:

1. If $p = correoNuevo$ Then $despliegaIconoCorreoNuevo$.
2. If $true$ Then $despliegaIconoCorreoViejo$.

Aunque hay otras maneras de implementar agentes reactivos, todas comparten una limitación formal: producen un comportamiento racional, solo si la decisión correcta puede obtenerse a partir de la percepción actual del agente. Esto es, su comportamiento es correcto si, y solo si, el ambiente es accesible o efectivamente accesible.

2.4.2 Agentes con estado

La forma más eficiente de enfrentar un ambiente inaccesible es llevando un registro de lo percibido, de forma que el agente tenga acceso mediante este registro, a lo que en cierto momento ya no puede percibir. Esto es, el agente debe mantener

una forma de **estado interno** que depende de su historia de percepciones y por lo tanto refleja al menos algunos de los aspectos del ambiente no observados en la percepción actual del agente. Sea I el conjunto de estados internos posibles de un agente. Redefinimos la función **acción** para mapear estados internos a acciones posibles:

Estado interno

Acción

$$acción : I \rightarrow Ac$$

Una nueva función $siguiente/2$, mapea estados internos y percepciones a estados internos. Se usa para actualizar el estado interno del agente:

$$siguiente : I \times Per \rightarrow I$$

Un agente inicia con algún estado interno $i_0 \in I$, observa entonces el estado del ambiente e generando así un percepto $p \leftarrow percibir(e)$. El estado interno del agente se actualiza, $i_1 \leftarrow siguiente(i_0, p)$. La acción seleccionada por el agente es entonces ejecutada y el agente repite este ciclo.

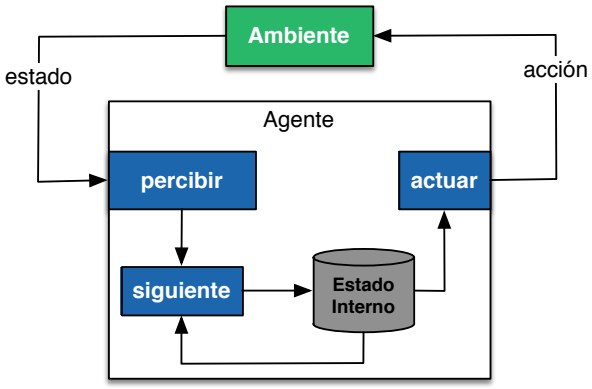


Figura 2.6: Agente con un estado interno.

La Figura 2.6 nos muestra un agente con estado interno, cuyo algoritmo se muestra en el Cuadro 2.4. La diferencia con el agente puramente reactivo, es que la función $siguiente/2$ modifica el estado interno del agente interpretando su entrada perceptiva, usando el conocimiento que el agente tenía sobre el ambiente.

Algoritmo 2.4 Programa de agente con estado

```
1: function AGENTECONESTADO(e)
2:     p ← percibir(e)
3:     estado ← siguiente(estado, p)
4:     regla ← selecciónAcción(estado, reglas)
5:     acción ← AcciónRegla(regla)
6:     return acción
7: end function
```

2.4.3 Agentes lógicos

La aproximación tradicional de la IA a los sistemas inteligentes, nos dice que el comportamiento racional puede obtenerse a partir de una representación simbólica [76] del ambiente y el comportamiento deseado. El agente manipulará sintácticamente esta representación para decidir como actuar. Llevada al extremo, esta aproximación nos lleva a formular el estado de un agente como un conjunto fórmulas lógicas; y la selección de acción como demostración de teoremas o deducción lógica.

El concepto de agente como un demostrador de teoremas es extremadamente seductor. Supongamos que disponemos de una teoría que establece cómo debe comportarse un agente racional. Tal teoría debe explicar cómo un agente genera sus

metas, cómo explota su comportamiento orientado por las metas y su reactividad, etc. Lo importante es que esta teoría ϕ puede considerarse como una **especificación** del comportamiento racional de un agente. Esta idea es centra en la concepción de **lenguajes de programación orientados a agentes**.

Especificación

Programación Orientada a Agentes

La manera tradicional de implementar un agente que cumpla con la especificación de la teoría ϕ es **refinar** paso a paso la teoría, concretando sus conceptos en estructuras de datos y procedimientos computacionales ejecutables. Sin embargo, en el enfoque basado en demostración de teoremas, tal refinamiento no se lleva a cabo. La teoría ϕ se concibe como una especificación ejecutable directamente por el agente.

Refinamiento

Sea L el conjunto de fórmulas bien formadas en una lógica, esto es, todas las expresiones sintácticamente correctas de la lógica en cuestión. Normalmente se asume una lógica de primer orden clásica, pero puede tratarse de una proposicional; incluso en el área de SMA, es común utilizar complejas lógicas multi-modales. El conjunto de **bases de conocimiento** en L se define como $D = \wp(L)$, es decir, el conjunto de todos los conjuntos de fbf en L. Los elementos de D se denotan $\Delta, \Delta_1, \ldots$ El estado interno del agente es siempre un miembro de D. El proceso de decisión del agente se especifica mediante un conjunto de **reglas de inferencia** ρ. Escribimos $\Delta \vdash_\rho \psi$ si la fbf ψ puede ser probada en Δ. Definimos la función *siguiente*/2 del agente como:

Bases de Conocimiento

Reglas de Inferencia

$$siguiente : D \times Per \to D$$

Algoritmo 2.5 Programa de agente lógico

```
1:  function Selección-Acción(Δ : D, Ac)                ▷ Ac Acciones.
2:      for all a ∈ Ac do
3:          if Δ ⊢_ρ ejecuta(a) then                    ▷ ρ predefinida, Δ base de conocimientos.
4:              return a
5:          end if
6:      end for
7:      for all  a ∈ Ac do
8:          if Δ ⊬_ρ ¬ejecuta(a) then
9:              return a
10:         end if
11:     end for
12:     return null
13: end function
```

El Algoritmo 2.5 muestra una función de selección de acción para un agente lógico. La primera parte de esta función (líneas 2 a 6) iteran sobre las acciones del agente para encontrar una acción a que valide $ejecuta(a)$. Si tal acción no se encuentra, se busca una acción a (líneas 7 a 11) que al menos sea **consistente** con el estado interno del agente. Si no se encuentra ninguna acción, se regresa *null* para indicar el fallo.

Consistencia

2.4.4 Agentes basados en Metas

En la tradición de la IA, las **metas** describen situaciones que son deseables para un agente, y son definidas como cuerpos de conocimiento acerca de los estados del medio ambiente que el agente desea ver realizados [74]. Esta concepción de las metas está relacionada con el concepto de **espacio de estados de un problema** compuesto por un estado inicial del ambiente, $e_0 \in E$; por un conjunto de operadores o acciones que el agente puede ejecutar para cambiar de estado; un espacio de estados alcanzables a partir del estado inicial e_0 al aplicar cualquier secuencia de operadores; y un test para verificar si el estado alcanzado es una meta, por ej., pertenece a un subconjunto de E que incluye los estados meta o cumple con determinadas propiedades especificadas en algún lenguaje lógico.

Metas

Espacio de Estados

Implícita en la arquitectura del agente, está su "intención" de ejecutar las acciones que el "cree" le garantizan satisfacer cualquiera de sus metas. Esto se conoce en filosofía como **silogismo práctico** y constituye la definición de **racionalidad** usada en IA [69, 74].

Silogismo práctico

Racionalidad

Es posible especificar las metas que el agente debe satisfacer usando una especificación basada en predicados. Un predicado en este contexto es definido como la función:

$$\Psi : R \rightarrow \{0, 1\}$$

esto es, una función de las corridas del agente a 1 (verdadero) o 0 (falso). Se considerará que una corrida $r \in R$ satisface la especificación si y solo si $\Psi(r) = 1$. Un **ambiente de tareas** se define entonces como el par $\langle Env, \Psi \rangle$. Dado un ambiente de tareas:

Ambiente de tareas

$$R_\Psi(Ag, Env) = \{r | r \in R(Ag, Env) \wedge \Psi(r)\}$$

denota el conjunto de todas las corridas del agente Ag en el ambiente Env que satisfacen la tarea especificada por Ψ. Podemos expresar que un agente Ag tiene éxito en el ambiente de tareas $\langle Env, \Psi \rangle$ de dos maneras diferentes: i) $\forall r \in R(Ag, Env)$ tenemos que $\Psi(r)$, lo que puede verse como una especificación pesimista de éxito, puesto que el agente tiene éxito únicamente si todas sus corridas satisfacen Ψ; ii) $\exists r \in R(Ag, Env)$ tal que $\Psi(r)$, lo cual es una versión optimista de la definición de éxito, puesto que especifica que el agente tiene éxito si al menos una de sus corridas satisface Ψ.

Observen que Ψ opera sobre corridas del agente, de forma que si queremos especificar un test de meta, como se define en el espacio de estados de un problema, algunos cambios son necesarios. Recuerde que el ambiente se define como $Env = \langle E, e_0, \tau \rangle$ donde E es el conjunto finito de estados discretos del ambiente, y que el agente es definido como $Ag = R^E \rightarrow Ac$, un mapeo de corridas que terminan en un estado del ambiente a acciones. El conjunto E corresponde al espacio de estados en la definición del espacio de estados del problema. Es posible identificar un subconjunto $G \subset E$ de estados finales, tal que $\Psi(r)$ es verdadera solo si uno o más $e \in G$ ocurren en $r \in R$. Ψ especifica la prueba de meta y el conjunto de acciones Ac define los operadores.

Una vez que un agente alcanza un estado que satisface el test de meta, se dice que ésta ha sido satisfecha. Las metas definidas de esta forma, constituyen un subconjunto de las metas que un agente puede enfrentar. Como ya lo comentamos, Covrigaru y Lindsay [28] concluyen que un sistema se percibe más fácilmente como autónomo si tiene un conjunto de **metas de alto nivel** (aquellas que no son submetas de otra meta) y algunas de ellas son **homeostáticas** (la satisfacción de la meta no termina al alcanzar el estado que la define, el sistema verifica continuamente si ese estado ha sido abandonado, para buscar satisfacerlo nuevamente).

Metas y autonomía

Ejemplo 2.9. *La meta de* `xbiff`*, comunicar que hay mensajes en el buzón, es homeostática. Cada vez que el estado del medio ambiente le indica al agente la presencia de mensajes, este ejecutará la acción de cambiar su icono. Sin embargo,* `xbiff` *tiene una única meta.*

2.4.5 Agentes basados en funciones de utilidad

Otra forma de comunicarle al agente que hacer, sin decirle como hacerlo, es definiendo sus metas indirectamente, por medio de alguna medida de desempeño, por ej., utilidades. Una **utilidad** es un valor numérico que denota la bondad de un estado del ambiente. Implícitamente, un agente tiene la "intención" de alcanzar aquellos estados que maximizan su utilidad a largo término. Por lo tanto, la especificación de una tarea en este enfoque corresponde simplemente a una función $u : E \rightarrow \Re$ la cual asocia valores reales a cada estado del ambiente. Para poder especificar una

Utilidad

visión a largo plazo del desempeño del agente, algunas veces las utilidades no se asignan a los estados del ambiente, sino a las corridas del agente.

Ejemplo 2.10. *Pensemos en un agente de correo electrónico más complejo que el programa* `xbiff`*, uno que filtre el* spam *en el correo recibido. La utilidad para una corrida r de este agente, puede definirse como:*

$$u(r) = \frac{SpamFiltrado(r)}{SpamRecibido(r)}$$

Esto da una medida normalizada del desempeño del agente en el rango de 0 (el agente no logró filtrar ningún mensaje spam) a 1 (el agente filtró todo el spam recibido) durante la corrida r.

Asumiendo que la función de utilidad u tiene algún límite superior en los valores que asigna, por ej., existe un $k \in \Re$ tal que $\forall r \in R.u(r) \leq k$, entonces es posible hablar de **agentes óptimos**, agentes que maximizan la utilidad esperada. Definamos $P(r|Ag, Env)$ como la probabilidad de que una corrida r ocurra cuando el agente Ag está situado en el ambiente Env, y sea R el conjunto de todas las posibles corridas de Ag en Env. Es evidente que:

Agentes óptimos

$$\sum_{r \in R(Ag,Env)} P(r|Ag, Env) = 1$$

Entonces el agente óptimo Ag_{opt} entre el conjunto de agentes posibles Ags en el ambiente Env está definido como:

$$Ag_{opt} = \arg \max_{Ag \in Ags} \sum_{r \in R(Ag,Env)} u(r)P(r|Ag, Env)$$

Esta ecuación es idéntica a la noción de maximizar la utilidad esperada, tal y como lo expresan Von Neumann y Morgesten [109] en su libro sobre Teoría de la Decisión. La utilidad esperada se toma conforme a las "creencias" del agente, por lo que la racionalidad perfecta no requiere omnisciencia. De cualquier forma, como señala Simon [101], los agentes enfrentan limitaciones temporales y tienen capacidades limitadas de deliberación, por lo que propone el estudio de una **racionalidad acotada**.

Racionalidad acotada

Russell y Subramanian [96], introducen el concepto de **agente óptimo acotado**, donde el conjunto de agentes posibles Ags es remplazado por un subconjunto de él, identificado como Ags_m, que representa el conjunto de agentes que pueden ser implementados en una máquina m con características específicas, por ej., memoria, almacenamiento, velocidad de procesador, etc., de forma que las restricciones de optimización se imponen en el programa del agente y no en las funciones del agente, sus deliberaciones o su comportamiento.

Agente óptimo acotado

Esta concepción de agente racional nos dice las propiedades del agente deseable Ag_{opt}, pero no nos dice cómo implementar tal agente. En la práctica normalmente resulta difícil derivar la función de utilidad u apropiada para un agente. Por otra parte, los formalismos asociados a esta aproximación a los agentes racionales, son más ajenos a los propósitos de este curso.

2.5 LECTURAS Y EJERCICIOS SUGERIDOS

La formalización del concepto de agente está basada en el texto de Wooldridge [111], aunque una fuente más accesible a este material se encuentra en su libro introductorio a los Sistemas Multiagente [112]. La clasificación de los agentes y sus programas está basada en el texto de Russell y Norvig [95]. El artículo de Covrigaru y Lindsay [28] ofrece una discusión profunda sobre la autonomía en los agentes artificiales. El antecedente directo al uso de funciones de utilidad en agentes se encuentra en el

trabajo de Von Neumann y Morgesten [109], donde queda de manifiesto una de las principales ventajas de este enfoque: su formalización. El concepto de racionalidad acotada fue introducido por Simon [101] siendo uno de sus temas preferidos. Una discusión más reciente sobre el tema de la racionalidad acotada, puede encontrarse el artículo de Russell y Subramanian [96].

Actualmente contamos con referencias bibliográficas coherentes y completas sobre el tema de la programación de agentes y los SMA. Se sugiere una lectura rápida de los capítulos uno y dos del libro de Russell y Norvig [95] para una panorámica del tema de los agentes racionales con respecto a la IA. Huns y Singh [53] ofrecen una colección de artículos fundamentales en el área, algunos de los cuales serán revisados en las tareas del curso. Weiß [110] estructuró el primer texto completo y coherente sobre el estudio de los SMA. Wooldridge [112] nos ofrece un texto de introducción básico a los SMA, imprescindible de leer a lo largo del curso, por su cobertura y complejidad moderada. El texto de Shoham [100] se situa un marco de referencia económico, más próximo a la Teoría de Juegos, pero con un componente algorítmico importante. Un excelente complemento para este texto.

Ejercicio 2.1. *Piense un poco en su tema de tesis y describa un agente y su medio ambiente en ese contexto. Base su descripción en el esquema PEAS.*

Ejercicio 2.2. *¿Existe una medida de desempeño para su agente? ¿Puede formularla?*

Ejercicio 2.3. *¿Es posible agrupar las percepciones y acciones de su agente en artefactos? Describa un ejemplo.*

Ejercicio 2.4. *¿Qué arquitectura elegiría para su agente? Justifique la respuesta.*

3 DEDUCCIÓN NATURAL

El objetivo de la lógica en las Ciencias de la Computación es el desarrollo de lenguajes para modelar las situaciones que un profesional del área encuentra comúnmente, y razonar formalmente acerca de ellas [54]. Razonar acerca de estas situaciones, significa construir **argumentos** acerca de ellas. Lo ideal es que esta construcción sea formal, de manera que los argumentos sean válidos y puedan ser defendidos rigurosamente; o ejecutados por una máquina. Esto último, es de particular importancia desde la perspectiva de la Inteligencia Artificial. Consideren el siguiente ejemplo:

Argumentación

Ejemplo 3.1. *Si el autobus llega tarde y no hay taxis en la estación, llegaré tarde a clase. No llegué tarde a clase. El autobus llegó tarde.* ***Por lo tanto****, había taxis en la estación.*

De manera intuitiva, el argumento anterior es válido, ya que si consideramos el primer enunciado junto con el tercero, nos dicen que si no hay taxis llegaré tarde a clase. El segundo enunciado nos dice que no llegué tarde, por lo que debe ser el caso que en la estación hubiese taxis.

Una gran parte del contenido de este texto está relacionado con argumentos que tienen esta estructura: una secuencia de enunciados seguida de la expresión "por lo tanto" y un enunciado más. El argumento se dice **válido**, si el enunciado final se sigue lógicamente de los enunciados previos al "por lo tanto". Qué queremos decir por "se sigue de", es el tema de este capítulo y el siguiente. Consideren otro ejemplo:

Validez

Ejemplo 3.2. *Si está lloviendo y Ana no trae su sombrilla, se mojará. Ana no está mojada. Está lloviendo.* ***Por lo tanto****, Ana trae su sombrilla.*

Este argumento también es válido y, de hecho, tiene la misma estructura que el del ejemplo anterior. Solo hemos remplazado algunos enunciados por otros:

Ejemplo 1	Ejemplo 2
Si el autobus llega tarde y	Si está lloviendo y
no hay taxis en la estación entonces	Ana no trae su paraguas entonces
llegaré tarde a clase	se mojará
El autobus llegó tarde	Está lloviendo
No llegué tarde	No se mojó
Había taxis en la estación	Traía su paraguas

De hecho, el argumento podría expresarse sin hablar específicamente de autobuses, taxis, lluvia y sombrillas: Si p y no q entonces r. No r. Por lo tanto q. Al desarrollar una lógica, no nos concierne que es lo que los enunciados realmente significan, sino su **estructura lógica**. Aunque claro, cuando aplicamos el razonamiento, como en los ejemplos anteriores, el significado será de gran interés.

Estructura

3.1 ENUNCIADOS DECLARATIVOS

Para poder argumentar de forma rigurosa, necesitamos desarrollar un lenguaje en el cual podamos expresar nuestros enunciados resaltando su estructura lógica. Comenzaremos por el lenguaje de la lógica proposicional. Este lenguaje está basado en **proposiciones** o **enunciados declarativos** que uno puede, en principio, argumentar que son verdaderos o falsos. Los siguientes enunciados son ejemplos de proposiciones:

Proposiciones

1. La suma de 2 y 3 es 5.
2. Mariano reaccionó violentamente ante las acusaciones.
3. Todo número par mayor que dos es la suma de dos números primos.
4. François-Marie Banier était un écrivain français.

El enunciado (1) puede ser declarado como verdadero en el contexto de la aritmética y la representación arábica décimal para los números naturales. El enunciado (2) es un poco más problemático. Para evaluar este enunciado necesitaríamos de un testigo de la reacción de Marianio y posiblemente conocer las acusaciones. En todo caso, si hubiésemos presenciado la escena descrita, podríamos asignar un **valor de verdad** al enunciado en cuestión. El enunciado (3) se conoce como la Conjetura de Goldbach y es uno de los problemas abiertos más antiguos de las matemáticas. Aunque la conjetura no ha sido demostrada, evidentemente toda proposición para los números mayores que dos resulta falsa o verdadera. La cuestión aquí es que no solo desconocemos el valor de verdad del enunciado, sino que ignoramos si aún siendo verdad, ésta puede ser probada por medios finitos. El enunciado (4) se puede declarar verdadero hablando francés y siendo un poco snob. En todo caso, los enunciados declarativos pueden realizarse en cualquier lenguaje, natural o artificial.

Valor de verdad

Los siguientes enunciados no son declarativos:

- Pásame la sal.
- Listo, ¡vamonos!
- ¿Qué pesa más un kilo de plomo o de algodón?
- No uses el teléfono en clase.

Como es de esperarse, nuestro interés se centra en establecer enunciados declarativos acerca del comportamiento de los sistemas computaciones, o sus programas. No solo queremos especificar esos enunciados, sino verificar si dado un programa o sistema, este satisface su especificación. De forma que necesitamos desarrollar un **cálculo** del razonamiento, que nos permita obtener conclusiones a partir de un conjunto dado de supuestos. Una cuestión más complicada es, dada cualquier propiedad verdadera sobre un programa, encontrar un argumento en nuestro cálculo, que tiene como conclusión a la propiedad en cuestión... ¡como la conjetura de Goldbach [1]!

Lógica y Cómputo

Las lógicas que vamos a estudiar, tienen una naturaleza **simbólica** [2]. Se trata de poder traducir un buen número de enunciados declarativos del español, o cualquier otro lenguaje natural, a cadenas de símbolos; de forma que obtengamos un lenguaje que, aunque reducido, sea los suficientemente **expresivo** y al mismo tiempo, nos permita concentrarnos en los **mecanismos** de nuestra argumentación.

Lógica e IA

Una estrategia común es considerar que ciertos enunciados declarativos son **atómicos**, es decir, que no pueden descomponerse en otros enunciados. Por ejemplo: El número 5 es impar. Estos enunciados serán denotados por los símbolos $p, q, r, \ldots$ o a veces por $p_1, p_2, p_3, \ldots$ de forma que componer enunciados complejos con base en ellos.

Proposiciones atómicas

Ejemplo 3.3. *Los siguientes enunciados son declarativos:*

p : *Gané la lotería la semana pasada.*

q : *Compré un boleto de lotería.*

r : *Gané en los caballos la semana pasada.*

Operadores y proposiciones compuestas

Los enunciados **compuestos** pueden construirse usando **operadores lógicos**, de acuerdo a las siguientes reglas:

- $\neg$: La **negación** de p se denota por $\neg p$. Siguiendo el ejemplo 3.3, este enunciado expresa: No gané la lotería la semana pasada; o su equivalente: No es verdad que gané la lotería la semana pasada.
- $\vee$: La **disyunción** de p y r se denota por $p \vee r$ y expresa que al menos uno de los dos enunciados es verdadero: Gané la lotería la semana pasada o gané en los caballos la semana pasada.
- $\wedge$: La **conjunción** es el dual de la regla anterior. $p \wedge r$ expresa que ambos enunciados son verdaderos: La semana pasada gané la lotería y en los caballos.
- $\rightarrow$: La **implicación** $p \rightarrow q$ expresa: **Si** gané la lotería la semana pasada, **entonces** compré un boleto de lotería. El enunciado sugiere que q es consecuencia lógica de p. A p se le suele llamar **premisa** y a q **conclusión**.

Podemos usar estas reglas para construir proposiciones complejas. Por ejemplo, ahora podemos formar la proposición:

$$p \wedge q \rightarrow \neg r \vee q$$

que expresa: Si p y q entonces no r o q. Observen que existe cierta ambigüedad sobre como debe leerse este enunciado. En las ciencias de la computación solemos eliminar la ambigüedad con el uso de paréntesis:

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

Precedencia de los operadores

Otra posibilidad es adoptar alguna convención sobre la **precedencia** de los operadores, es decir, establecer que operadores aplican primero. La opción de los paréntesis es sintácticamente más clara.

Definición 3.1. *Precedencia de los operadores: $\neg$ liga más fuerte que $\vee$ y $\wedge$; que a su vez ligan más fuerte que $\rightarrow$. La implicación es asociativa a la derecha: $p \rightarrow q \rightarrow r$ denota $p \rightarrow (q \rightarrow r)$.*

3.2 REGLAS PARA LA DEDUCCIÓN NATURAL

¿Cómo procederemos para construir un cálculo de razonamiento que permita establecer la validez de, digamos, los ejemplos 3.1 y 3.2 de este documento? Sería deseable contar con un conjunto de reglas que permitan establecer si una conclusión se sigue de ciertas premisas.

Reglas de inferencia

En la deducción natural se cuenta con una colección de **reglas de inferencia** (o de prueba) que nos permiten establecer que fórmulas se pueden derivar a partir de otras. Supongamos que tenemos un conjunto de fórmulas $\Delta = \{\phi_1, \phi_2, \ldots, \phi_n\}$ [3] a las que llamaremos **premisas**. Y otra fórmula ψ a la que llamaremos **conclusión**. Al aplicar las reglas de prueba a las fórmulas en Δ, esperamos obtener nuevas fórmulas hasta obtener eventualmente la conclusión. Esto suele denotarse de la siguiente manera:

Premisas y conclusiones

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$$

Consecuencia

Tal expresión suele conocerse como una **consecuencia** [4]. Se dice que la conse-

1 Por cierto, Doxiadis [36] nos ofrece una divertida novela que gira en torno a la conjetura y sus personajes.

2 Observen la relevancia del carácter simbólico de la lógica para nosotros, de acuerdo a la hipótesis del sistema simbólico físico de Newell y Simon [76]: Tal sistema tiene los medios necesarios y suficientes para la acción inteligente general.

3 Tradicionalmente se usan letras minúsculas del alfabeto griego ($\phi, \psi, \chi, \eta, \alpha, \beta, \gamma, \ldots$) para denotar fórmulas; y mayúsculas ($\Delta, \Gamma, \Phi, \Psi, \ldots$) para denotar conjuntos de ellas. De alguna forma son metavariables, denotan expresiones de la lógica proposicional, pero no forman parte de ella.

cuencia es válida si se puede encontrar prueba para ella. La consecuencia para los ejemplos 1 y 2 tiene la forma:

$$p \wedge \neg q \rightarrow r, \neg r, p \vdash q$$

Construir estas pruebas es un ejercicio creativo, semejante a programar. No es necesariamente obvio que regla aplicar, ni en qué orden. Aunque como veremos, la estructura lógica de lo que queremos concluir puede guiar este proceso. Las reglas de inferencia deben ser cuidadosamente elegidas, de lo contrario podemos "probar" argumentaciones inválidas. Por ejemplo, es deseable que no seamos capaces de probar que la consecuencia $p, q \vdash p \wedge \neg q$ es válida. Asuman que p denota el clavel es una flor y q que la rosa es una flor. No tiene sentido inferir que el clavel es flor y la rosa no lo es. A continuación revisaremos en detalle las reglas usadas en la deducción natural.

3.2.1 Las reglas de la conjunción

Esta regla permite concluir $\phi \wedge \psi$ dado que ϕ y ψ son el caso. La regla se escribe así:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \ (\wedge i)$$

Sobre las líneas están las dos premisas de la regla. Abajo de la línea se encuentra la conclusión. Observen que ésta no es, generalmente, la conclusión que buscamos. Para ello habrá que aplicar más reglas. A la derecha de la regla tenemos el nombre de la misma abreviado $(\wedge i)$ o introducción de la conjunción. El resultado de aplicar esta regla es que hemos introducido una conjunción que no teníamos en las premisas.

Existen también dos reglas sobre la eliminación de la conjunción:

$$\frac{\phi \wedge \psi}{\phi} \ (\wedge e_1) \qquad \frac{\phi \wedge \psi}{\psi} \ (\wedge e_2)$$

La regla $(\wedge e_1)$ indica que si hemos probado $\phi \wedge \psi$, podemos concluir que ϕ. La regla $(\wedge e_2)$ es parecida solo que concluímos el segundo argumento de la conjunción, ψ.

Ejemplo 3.4. *Usemos las reglas definidas para probar que $p \wedge q, r \vdash q \wedge r$ es válida. Comenzamos escribiendo las premisas, una por línea y, dejando un espacio vacío, escribimos la conclusión:*

$$\begin{array}{l} p \wedge q \\ \quad r \\ \\ \\ q \wedge r \end{array}$$

Construir la prueba consiste en llenar la brecha entre las premisas y la conclusión aplicando la secuencia adecuada de reglas de inferencia. En este caso, aplicaremos $(\wedge e_2)$ a la primer premisa para obtener q. Posteriormente, aplicaremos $(\wedge i)$ a q y la segunda premisa para obtener $q \wedge r$. Usualmente numeramos las líneas de la prueba para poder referenciar los resultados parciales obtenidos:

4 Del inglés *sequent*. No se debe confundir consecuencia con conclusión, tal y como se han definido: La expresión $\Delta \vdash \phi$ es una consecuencia, donde ϕ es la conclusión y Δ las premisas.

1. $p \wedge q$ *premisa*
2. r *premisa*
3. q $(\wedge e_2)\ 1$
4. $q \wedge r$ $(\wedge i)\ 3,2$

Observen que esta demostración puede representarse tambien de la siguiente forma:

$$\dfrac{\dfrac{p \wedge q}{q}\ (\wedge e_2) \quad r}{q \wedge r}\ (\wedge i)$$

Sin embargo, hemos optado por una representación lineal que aplana el árbol anterior mediante el uso de las líneas numeradas para hacer referencia a entradas previas en la demostración. En todo caso, si una inferencia es válida, puede haber muchas formas de probarla. Lo que es importante resaltar es que podemos detectar cualquier prueba putativa, verificando que cada línea de la demostración se ha producido mediante una aplicación válida de una regla de prueba.

Ejemplo 3.5. *Pruebe ahora que* $(p \wedge q) \wedge r, s \wedge r \vdash q \wedge s$*:*

1. $(p \wedge q) \wedge r$ *premisa*
2. $s \wedge t$ *premisa*
3. $p \wedge q$ $(\wedge e_1)\ 1$
4. q $(\wedge e_2)\ 3$
5. s $(\wedge e_1)\ 2$
6. $q \wedge s$ $(\wedge i)\ 4,5$

3.2.2 Las reglas de la doble negación

Intuitivamente, una fórmula es equivalente a su doble negación. Si expresamos, no es cierto que no está lloviendo; estamos expresando que está lloviendo. De forma que las reglas de eliminación e introducción de la doble negación son como sigue:

$$\dfrac{\neg\neg\phi}{\phi}\ (\neg\neg e) \qquad \dfrac{\phi}{\neg\neg\phi}\ (\neg\neg i)$$

Ejemplo 3.6. *La prueba de la inferencia* $p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$ *usa la mayoría de las reglas introducidas hasta ahora:*

1. p *premisa*
2. $\neg\neg(q \wedge r)$ *premisa*
3. $\neg\neg p$ $(\neg\neg i)\ 1$
4. $q \wedge r$ $(\neg\neg e)\ 2$
5. r $(\wedge e_2)\ 4$
6. $\neg p \wedge r$ $(\wedge i)\ 3,5$

3.2.3 La eliminación de la implicación

En nuestro cálculo, el *modus ponens* se introduce con la siguiente regla de prueba, conocida como eliminación de la implicación: *Modus ponens*

$$\frac{\phi \quad \phi \to \psi}{\psi} \ (\to e)$$

Consideren los siguientes enunciados declarativos:

p : Llueve

q : La calle está mojada

entonces la fórmula $p \to q$ expresa que si llueve la calle está mojada. Ahora, si conocemos que es el caso que p, llueve; podemos combinar esas dos piezas de información con nuestra regla de eliminación de la implicación, para concluir que q, la calle está mojada. Sentido común. Observen que p es importante, si p no se satisface, no podemos concluir q.

Ejemplo 3.7. *Consideren los siguientes enunciados declarativos:*

p : *La entrada del programa es un entero.*

q : *La salida del programa es un booleano.*

entonces la expresión $p \to q$ expresa que si la entrada del programa es un entero, entonces su salida es un boleano. Luego si p se satisface, la entrada del programa es un entero, entonces podemos concluir q que su salida es un boleano. Nuevamente, si la entrada del programa es una cadena de caracteres, p no se satisface y no podemos concluir q.

Cabe mencionar que los parámetros como ϕ y ψ en estas reglas, pueden recibir proposiciones compuestas, además de atómicas:

1.	$\neg p \wedge q$	*premisa*
2.	$\neg p \wedge q \to r \vee \neg p$	*premisa*
3.	$r \vee \neg p$	$(\to e)$ 2, 1

Evidentemente, podemos aplicar estas reglas tantas veces queramos:

1.	$p \to (q \to r)$	*premisa*
2.	$p \to q$	*premisa*
3.	p	*premisa*
4.	$q \to r$	$(\to e)$ 3, 1
5.	q	$(\to e)$ 3, 2
6.	r	$(\to e)$ 5, 4

Existe otra forma de eliminación de la implicación que puede ser derivada a partir de las reglas básicas que definiremos en este capítulo. Por ello suele decirse que es una regla híbrida. A la regla en cuestión se le suele conocer como *modus tollens*. *Modus tollens*
Supongan que $p \to q$ y $\neg q$ son el caso. Entonces, si p se satisface, podríamos concluir que q $(\to e)$. Pero esto resulta contradictorio con $\neg q$ que es el caso. Por

lo tanto, debemos inferir que $\neg p$ se satisface. Esto puede expresarse en la siguiente regla de prueba:

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ (MT)$$

Eso aplicado al ejemplo 3.7 nos daría la siguiente inferencia. Si la entrada del programa es un entero, su salida es un boleano. No es el caso que la salida sea un boleano. Entonces podemos inferir que no es el caso que la entrada del programa sea un entero.

Ejemplo 3.8. *Pruebe que la inferencia* $p \to (q \to r), p, \neg r \vdash \neg q$ *es válida.*

1.	$p \to (q \to r)$	*premisa*
2.	p	*premisa*
3.	$\neg r$	*premisa*
4.	$q \to r$	$(\to e)$ 1, 2
5.	$\neg q$	(MT) 4, 3

Ejemplo 3.9. *Las siguientes pruebas combinan el* modus tollens *con la eliminación e introducción de la doble negación:*

1.	$\neg p \to q$	*premisa*
2.	$\neg q$	*premisa*
3.	$\neg\neg p$	(MT) 1, 2
4.	p	$(\neg\neg e)$ 3

prueba que la inferencia $\neg p \to q, \neg q \vdash p$ *es válida, y:*

1.	$p \to \neg q$	*premisa*
2.	q	*premisa*
3.	$\neg\neg q$	$(\neg\neg i)$ 2
4.	$\neg p$	(MT) 1, 3

prueba que la inferencia $p \to \neg q, q \vdash \neg p$.

3.2.4 La introducción de la implicación

La regla del *modus tollens* nos permitió probar que la inferencia $p \to q, \neg q \vdash \neg p$ es válida. Pero la validez de la inferencia $p \to q \vdash \neg q \to \neg p$ solo parece plausible. Observen que en cierto sentido, ambas expresiones dicen lo mismo. Sin embargo, no contamos aún con ninguna regla de prueba que nos permita introducir implicaciones, más allá de las que se establecen en las premisas. La mecánica de la regla con este propósito es más complicada que las anteriores, primero asumamos que $p \to q$ es el caso. Si **asumimos temporalmente** que $\neg q$ es válida, entones podría- *Supuestos*

mos usar (MP) para inferir $\neg q$. Por lo tanto, asumiendo $p \to q$, podemos probar que $\neg q \to \neg p$:

1.	$p \to q$	*premisa*
2.	$\neg q$	*supuesto*
3.	$\neg p$	(MT) 1, 2
4.	$\neg q \to \neg p$	$(\to i)$ $2-3$

La caja en esta prueba sirve para delimitar el alcance del supuesto temporal $\neg q$. Estamos expresando lo siguiente: Supongamos que $\neg q$ (para ello abrimos una caja). Entonces seguimos aplicando reglas de prueba, por ejemplo, para obtener $\neg p$. Como esto depende de nuestro supuesto inicial, lo colocamos dentro de la misma caja. El paso final de la demostración no depende del supuesto que habíamos adoptado, por eso lo colocamos fuera de la caja.

Lo anterior también funciona si pensamos que $p \to q$ es el tipo de un procedimiento de cómputo. Por ejemplo, p puede decir que el procedimiento espera un valor entero x como entrada; y q puede expresar que la salida del procedimeinto es un valor boleano y. Probar que esto es cierto es lo que se conoce como **verificación de tipos**, un tema importante en la construcción de compiladores para los lenguajes de programación fuertemente tipificados, por ejemplo CAML, Haskell, etc. La formula de introducción de la implicación, que definimos a continuación, computa precisamente esto:

Verificación de tipos

$$\frac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \psi \\ \hline \end{array}}{\phi \to \psi}\ (\to i)$$

La regla expresa que para probar $\phi \to \psi$, debemos asumir temporalmente ϕ y entonces probar ψ. En la prueba de ψ se pueden usar cualquiera de las fórmulas obtenidas hasta el momento, como premisas y conclusiones provisionales.

Ejemplo 3.10. *Una prueba usando la introducción de la implicación y otras reglas definidas anteriormente:*

1.	$\neg q \to \neg p$	*premisa*
2.	p	*supuesto*
3.	$\neg\neg p$	$(\neg\neg i)$ 2
4.	$\neg\neg q$	(MT) 1, 3
5.	q	$(\neg\neg e)$ 4
6.	$p \to q$	$(\to i)$ $2-5$

La siguiente prueba es válida:

1.		
2.	p	*supuesto*
3.	$p \to p$	$(\to i)\ 2-3$

Podemos decir que la prueba no necesita argumentación, es decir, no tenemos premisas. Esto suele escribirse $\vdash p \to p$.

Definición 3.2. *Las fórmulas lógicas con una inferencia válida $\vdash \phi$ se conocen como **teoremas**.*

Ejemplo 3.11. *Esta es la prueba de un teorema que utiliza las reglas definidas hasta ahora:*

1.		
2.	$q \to r$	*supuesto*
3.	$\neg q \to \neg p$	*supuesto*
4.	p	*supuesto*
5.	$\neg\neg p$	$(\neg\neg i)\ 4$
6.	$\neg\neg q$	$(MT)\ 3,5$
7.	q	$(\neg\neg e)\ 6$
8.	r	$(\to i)\ 2,7$
9.	$p \to r$	$(\to i)\ 4-8$
10.	$(\neg q \to \neg p) \to (p \to r)$	$(\to i)\ 3-9$
11.	$(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$	$(\to i)\ 1-10$

Por lo tanto, la inferencia $\vdash (q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ es válida, mostrando que $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ es otro teorema.

Observen que este ejemplo sugiere que es posible transformar de esta manera toda conseuencia de la forma $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ en una prueba de un teorema de la forma $\vdash \phi_1 \to (\phi_2 \to (\ldots \to (\phi_n \to \psi \ldots)))$ gracias a la introducción de la implicación. Basta con agregar n líneas de la regla $\to i$ aplicada a $\phi_n, \phi_{n-1}, \ldots, \phi$, en ese orden.

Las cajas anidadas en la demostración del teorema del ejemplo 3.11 revelan un patrón en la demostración: el uso de las reglas de eliminación para deconstruir los supuestos; y posteriormente, el uso de las reglas de introducción para construir la conclusión final. Hay más aún, parte de esa demostración está determinada por la estructura de las fórmulas a demostrar, mientras que otra parte es creativa. La estructura lógica de la fórmula en cuestión se muestra enla figura 3.1. Observen que la raíz del árbol es una implicación, pero la única forma de construir implicaciones es por medio de la regla $\to i$, de modo que debemos expresar el supuesto de la implicación como tal (línea 2) y mostrar su conclusión (línea 10). Si tenemos éxito, habremos terminado la demostración en la línea 11. De forma que las líneas 2, 10 y 11 están determinadas por el hecho de que la fórmula en cuestión sea una implicación. Lo mismo sucede con la fórmula de la línea 10, por lo que debemos asumir su premisa en la línea 3 y tratar de probar su conclusión en la línea 9, etc. La única cuestión pendiente es ¿Cómo probar r a partir de los supuestos introducidos

en las líneas 2-4? Esta parte es la única parte creativa de la prueba. La línea 2 introduce $q \to r$ y sabemos que podemos obtener r gracias a la regla de eliminación de la implicación $\to e$, si es que podemos tener q. ¿Cómo obtenemos q? Las líneas 4 y 5 semejan un patrón de modus tollens, excepto que necesitamos $\neg\neg p$ para poder aplicar esa regla y eso lo conseguimos con la regla $\neg\neg i$ en la línea 5. De forma que la estructura lógica de las fórmulas que queremos probar nos dicen mucho acerca de la estructura de la posible prueba. Es deseable que explotemos esa información al hacer nuestras pruebas. Terminaremos con unos ejemplos más que involucran también las reglas de la conjunción.

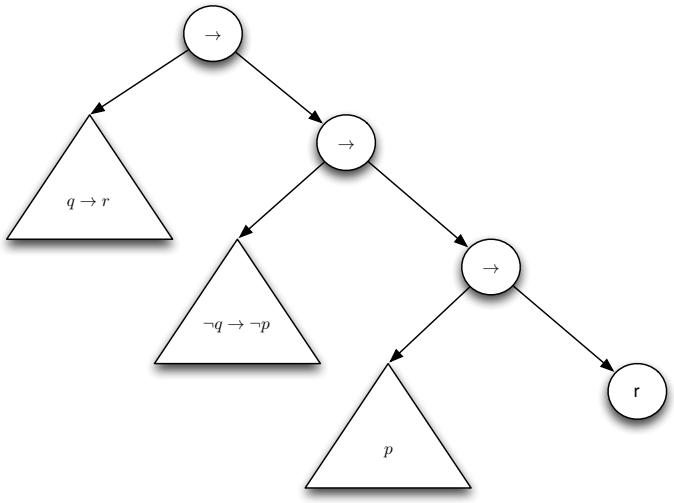


Figura 3.1: La estructura lógica del teorema del ejemplo 3.11.

Ejemplo 3.12. *Utilizando la regla* $(\wedge i)$ *prueba la validez de la inferencia* $p \wedge q \to r \vdash p \to (q \to r)$*:*

1.	$p \wedge q \to r$	*premisa*
2.	p	*supuesto*
3.	q	*supuesto*
4.	$p \wedge r$	$(\wedge i)$ 2, 3
5.	r	$(\to e)$ 1, 4
6.	$q \to r$	$(\to i)$ 3 – 5
7.	$p \to (q \to r)$	$(\to i)$ 2 – 6

Ejemplo 3.13. *Utilize las reglas de eliminación de la conjunción para provar el converso del ejemplo anterior:* $p \to (q \to r) \vdash p \wedge q \to r$*:*

1.	$p \to (q \to r)$	*premisa*
2.	$p \wedge q$	*supuesto*
3.	p	$(\wedge e_1)$ 2
4.	q	$(\wedge e_2)$ 2
5.	$q \to r$	$(\to e)$ 1,3
6.	r	$(\to e)$ 4,5
7.	$p \wedge q \to r$	$(\to i)$ 2 – 6

Lo anterior quiere decir que estas fórmula son **equivalentes** en el sentido que pueden ser probadas a partir de la otra. Esto se denota de la siguiente manera: *Equivalencia lógica*

$$p \wedge q \to r \dashv\vdash p \to (q \to r)$$

Como solo podemos tener una fórmula a la derecha de $\vdash$, cada ocurrencia de $\dashv\vdash$ solo puede relacionar dos fórmulas.

Ejemplo 3.14. *Aquí hacemos uso de la introducción y eliminación de la conjunción para probar la validez de la inferencia $p \to q \vdash p \wedge r \to q \wedge r$:*

1.	$p \to q$	*premisa*
2.	$p \wedge r$	*supuesto*
3.	p	$(\wedge e_1)$ 2
4.	r	$(\wedge e_2)$ 2
5.	q	$(\to e)$ 1,3
6.	$q \wedge r$	$(\wedge i)$ 5,4
7.	$p \wedge r \to q \wedge r$	$(\to i)$ 2 – 6

3.2.5 Las reglas de la disyunción

Comencemos por la introducción de la disyunción. A partir de la premisa ϕ podemos concluir $\phi \vee \psi$, sea cual sea la elección de ψ. De igual manera podemos concluir ϕ si hemos probado ψ:

$$\frac{\phi}{\phi \vee \psi}\ (\vee i_1) \qquad \frac{\psi}{\phi \vee \psi}\ (\vee i_2)$$

La eliminación es un poco más complicada. Supongamos que queremos probar una proposición χ asumiendo que $\phi \vee \psi$. Puesto que no sabemos que argumento de la disyunción es verdadero, debemos obtener dos pruebas por separado que deberemos combinar en un solo argumento:

1. Primero asumimos que ϕ es verdadera y obtenemos la prueba de χ.
2. Entonces asumimos que ψ es verdadera y que da una prueba para χ también.
3. Dadas las dos pruebas anteriores, podemos inferir χ de la verdad de $\phi \vee \psi$, puesto que el análisis de casos ha sido exhaustivo.

La regla se escribe como sigue:

$$\dfrac{\phi \vee \psi \qquad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline \end{array} \qquad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline \end{array}}{\chi} \; (\vee e)$$

Ejemplo 3.15. *Probar que la inferencia $p \vee q \vdash q \vee p$ es válida:*

1.	$p \vee q$	*premisa*
2.	p	*supuesto*
3.	$p \vee q$	$(\vee i_2)$ 2
4.	q	*supuesto*
5.	$q \vee p$	$(\vee i_1)$ 4
6.	$q \vee p$	$(\vee e)$ $1, 2-3, 4-5$

Observen que las sub pruebas son ortogonales, no podemos usar los supuestos de una, en la otra. La idea es que si usamos $\phi \vee \psi$ en un argumento donde aparece como premisa o supuesto, estamos perdiendo información. Sabemos que ϕ es el caso, o lo es ψ pero no sabemos cual de los dos es. Por ello, debemos construir evidencia sólida para cada una de las posibilidades.

Ejemplo 3.16. *Pruebe que la inferencia $q \to r \vdash p \vee q \to p \vee r$ es válida:*

1.	$q \to r$	*premisa*
2.	$p \vee q$	*supuesto*
3.	p	*supuesto*
4.	$p \vee r$	$(\vee i_1)$ 3
5.	q	*supuesto*
6.	r	$(\to e)$ $1, 5$
7.	$p \vee r$	$(\vee i_2)$ 6
8.	$p \vee r$	$(\vee e)$ $2, 3-4, 5-7$
9.	$p \vee q \to p \vee r$	$(\to i)$ $2, 8$

Ejemplo 3.17. *Del algebra boleana y la teoría de circuitos sabemos que la conjunción se distribuye sobre la disyunción. Pruébelo. Esto equivale a probar que la inferencia $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$ y vice-versa. Comencemos por este caso:*

1.	$p \wedge (q \vee r)$	*premisa*
2.	p	$(\wedge e_1)$ 1
3.	$q \vee r$	$(\wedge e_2)$ 1
4.	q	*supuesto*
5.	$p \wedge q$	$(\wedge i)2, 4$
6.	$(p \wedge q) \vee (p \wedge r)$	$(\vee i_1)$ 5
7.	r	*supuesto*
8.	$p \wedge r$	$(\wedge i)$ 2, 7
9.	$(p \wedge q) \vee (p \wedge r)$	$(\vee i_2)$ 8
10.	$(p \wedge q) \vee (p \wedge r)$	$(\vee e)$ $3, 4-6, 7-9$

Una regla final extra lógica, nos permite reutilizar resultados obtenidos previamente en cualquier parte donde sean visibles:

1.		
2.	p	*supuesto*
3.	q	*supuesto*
4.	p	*copia* 2
5.	$q \to p$	$(\to i)$ $2-3$
6.	$p \to (q \to p)$	$(\to i)$ $2-5$

3.2.6 Las reglas de la negación

Hay un problema para introducir las reglas de negación en este juego que tiene que ver con la inferencia, y por tanto con la preservación de la verdad. No podemos inferir directamente $\neg\phi$ a partir de ϕ. Para ello necesitamos notación adicional.

Definición 3.3 (Contradicción)**.** *Las contradicciones son expresiones de la forma $\phi \wedge \neg\phi$ ó $\neg\phi \wedge \phi$, donde ϕ es cualquier fórmula. Una contradicción se denota $\bot$.*

Por ejemplo, $p \wedge \neg p$ es una contradicción; y también lo es $(p \to q) \wedge \neg(p \to q)$. Las contradicciones son un concepto importante en la lógica. Con respecto a la verdad, todas son equivalentes. Esto significa que deberíamos ser capaces de probar que:

$$\neg(r \vee s \to q) \wedge (r \vee s \to q \dashv\vdash (p \to q) \wedge \neg(p \to q)$$

puesto que ambos lados de la equivalencia son una contradicción. De hecho podremos probar esto luego de introducir las reglas de la negación. Lo curioso es que no solo las contradicciones pueden derivarse de las contradicciones, sino que ¡cualquier fórmula puede derivarse de una contradicción! De hecho, el principio suele conocerse por su nombre en latín *ex contradictore quodlibet* (ECQ), a partir de una contradicción cualquier cosa se sigue. Esto puede resultar contrintuitivo, ¿Porqué tendría que argumentar que $p \wedge \neg p \vdash q$, donde:

p: La luna es de queso azul.

q: Me gusta el peperoni en mi pizza.

considerando que mis preferencias gastronómicas no tienen nada que ver con la constitución de la luna? La razón está en $\vdash$ que nos dice todas las cosas que podemos inferir, asumiendo las fórmulas a su izquierda, en un proceso que no toma en cuenta si tales premisas tienen sentido o no. Si quisiera que todas las premisas fuesen relevantes para la concusión, tendría que exlcuir este principio y estaríamos hablando de otro tipo de lógica no clásica, la lógica relevante [68]. El hecho de que $\bot$ pueda probar lo que sea, se expresa en la regla de eliminación de la contradicción:

$$\frac{\bot}{\phi}\ (\bot e)$$

El hecho de que $\bot$ mismo representa una contradicción queda expresado en la regla de eliminación de la negación:

$$\frac{\phi \quad \neg\phi}{\bot}\ (\neg e)$$

Ejemplo 3.18. *Pruebe que la inferencia* $\neg p \vee q \vdash p \to q$ *es válida:*

1.	$\neg p \vee q$	*premisa*
2.	$\neg p$	*supuesto*
3.	p	*supuesto*
4.	$\bot$	$(\neg e)$ 3, 2
5.	q	$(\bot e)$ 4
6.	$p \to q$	$(\to i)$ $3-5$
7.	q	*supuesto*
8.	p	*supuesto*
9.	q	*copia* 7
10.	$p \to q$	$(\to i)$ $8-9$
11.	$p \to q$	$(\vee e)$ $1, 2-10$

Ahora, supongamos que asumimos un supuesto que nos lleva a una contradicción. Entonces, nuestro supuesto no puede ser verdadero, de forma que debería ser falso. Esta intuición es la base de la introducción de la negación:

$$\frac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline \end{array}}{\neg\phi}\ (\neg i)$$

Ejemplo 3.19. *Demuestre que la inferencia* $p \to q, p \to \neg q \vdash \neg p$ *es válida:*

1.	$p \to q$	*premisa*
2.	$p \to \neg q$	*premisa*
3.	p	*supuesto*
4.	q	$(\to e)$ 1, 3
5.	$\neg q$	$(\to e)$ 2, 3
6.	$\bot$	$(\neg e)$ 4, 5
7.	$\neg p$	$(\neg i) 3 - 6$

Ejemplo 3.20. *Pruebe que la inferencia* $p \to \neg p \vdash \neg p$ *es válida:*

1.	$p \to \neg p$	*premisa*
2.	p	*supuesto*
3.	$\neg p$	$(\to e)$ 1, 2
4.	$\bot$	$(\neg e)$ 2, 3
5.	$\neg p$	$(\neg i) 2 - 4$

Ejemplo 3.21. *Pruebe que la inferencia* $p \to (q \to r), p, \neg r \vdash \neg q$ *es válida (sin usar* modus tollens*):*

1.	$p \to (q \to r)$	*premisa*
2.	p	*premisa*
3.	$\neg r$	*premisa*
4.	q	*supuesto*
5.	$q \to r$	$(\to e)$ 1, 2
6.	r	$(\to e) 5, 4$
7.	$\bot$	$(\neg e) 6, 3$
8.	$\neg q$	$(\neg i) 4 - 7$

Ejemplo 3.22. *Volvamos a los ejemplos 3.1 y 3.2 del inicio del capítulo. Pruebe que la inferencia* $p \wedge \neg q \to r, \neg r, p \vdash q$ *es válida:*

1.	$p \wedge \neg q \rightarrow r$	*premisa*
2.	$\neg r$	*premisa*
3.	p	*premisa*
4.	$\neg q$	*supuesto*
5.	$p \wedge \neg q$	$(\wedge i)$ 3, 4
6.	r	$(\rightarrow e)$ 1, 5
7.	$\bot$	$(\neg e)$ 6, 2
8.	$\neg\neg q$	$(\neg i)$ 4 – 7
9.	q	$(\neg\neg e)$ 8

3.2.7 Reglas derivadas

He mencionado que la regla de *modus tollens* no es primitiva, en el sentido que puede derivarse de otras reglas. He aquí su demostración:

1.	$\phi \rightarrow \psi$	*premisa*
2.	$\neg\psi$	*premisa*
3.	ϕ	*supuesto*
4.	ψ	$(\rightarrow e)$ 1, 3
5.	$\bot$	$(\neg e)$ 4, 2
6.	$\neg\phi$	$(\neg i)$ 3 – 5

Podemos replantear las demostraciones que hemos hecho usando *modus tollens* en términos de esta combinación de $(\rightarrow e)$, $(\neg e)$ y $(\neg i)$. Sin embargo, es conveniente pensar en la regla (MT) como una abreviatura de dicha combinación, o mejor aún, como una **macro**.

La introducción de la doble negación también es una regla derivada que se puede demostrar como sigue:

1.	ϕ	*premisa*
2.	$\neg\phi$	*supuesto*
3.	$\bot$	$(\neg e)$ 1, 2
4.	$\neg\neg\phi$	$(\neg i)$ 2 – 3

Hay una cantidad ilimitada de reglas derivadas que podríamos incluir en nuestro cálculo del razonamiento, sin embargo, no es deseable que éste se vuelva voluminoso y pesado. De hecho, algunos puristas nos dirían que debemos asumir el conjunto mínimo de reglas de prueba que son independientes entre ellas. Por cuestiones prácticas, aquí adoptaremos una posición intermedia, dándole nombre a las reglas

derivadas que hemos introducido porque se usan frencuentemente como patrones de razonamiento.

Reducción al absurdo

De hecho introduciremos dos más. La primera de ellas recibe el nombre de *reduction ad absurdum* o prueba por contradicción (PBC) (del inglés *proof by contradiction*). La regla expresa que si a partir de $\neg\phi$ obtenemos una contradicción, podemos deducir que ϕ:

$$\dfrac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\ (PBC)$$

Ley del medio excluído

La segunda regla derivada es muy útil, pues su derivación es larga y complicada. En latín se le conoce como *tertium non datur* o ley del medio excluído (LEM) (del inglés *law of the excluded medium*). La ley expresa que si $\phi \vee \neg\phi$ es verdadera, sea lo que sea ϕ, ésta es falsa o verdadera y nada más. No hay una tercera posibilidad. Nosotros asumimos este principio en los condicionales de los lenguajes de programación: expresiones como `if B then {C} else {D}` asumen que $B \vee \neg B$ es verdadera. He aquí la prueba del medio excluído:

1.		
2.	$\neg(\phi \vee \neg\phi)$	*supuesto*
3.	ϕ	*supuesto*
4.	$\phi \vee \neg\phi$	$(\vee i)$ 3
5.	$\bot$	$(\neg e)$ 4, 2
6.	$\neg\phi$	$(\neg i)$ 3 – 5
7.	$\phi \vee \neg\phi$	$(\vee i_2)$ 6
8.	$\bot$	$(\neg e)$ 7, 2
9.	$\neg\neg(\phi \vee \neg\phi)$	$(\neg e)$ 2 – 8
10.	$\phi \vee \neg\phi$	$(\neg\neg e)$ 9

Ejemplo 3.23. *Use (LEM) para probar que la inferencia $p \to q \vdash \neg p \vee q$ es válida:*

1.	$p \to q$	*premisa*
2.	$\neg p \vee p$	*LEM*
3.	$\neg p$	*supuesto*
4.	$\neg p \vee q$	$(\vee i_1)$ 3
5.	p	*supuesto*
6.	q	$(\to e)$ 1, 5
7.	$\neg p \vee q$	$(\vee i_2)$ 6
8.	$\neg p \vee q$	$(\vee e)$ 2, 3 – 4, 5 – 7

Aunque útil, resulta difícil decidir que caso de (LEM) puede ayudarnos a avanzar en una prueba. Por ejemplo, pudimos haber elegido introducir $q \vee \neg q$ en la línea 2 de la demostración anterior ¿Cual hubiera sido entonces su posterior desarrollo?

3.2.8 Las reglas como procedimientos

La manera como hemos definido las reglas de prueba en esta sección se conoce como una definición **declarativa**: hemos presentado cada regla y la hemos justificado en términos de nuestra intuición sobre los conectores lógicos. Sin embargo una definición más **procedural**, que nos diga cómo se usa cada regla, es posible:

$(\wedge i)$ Para probar $\phi \wedge \psi$, primero probamos ϕ y luego ψ, por separado. Entonces usamos la regla $(\wedge i)$.

$(\wedge e_1)$ Para probar ϕ, comienza por probar $\phi \wedge \psi$ y luego usa la regla $(\wedge e_1)$. Aunque esto no es un buen consejo pues probar la conjunción puede ser más complicado que probar ϕ; lo que puede ser el caso es que $\phi \wedge \psi$ ya haya sido generada previamente y entonces la regla es realmente útil.

$(\vee i_1)$ Para probar $\phi \vee \psi$, intente probar ϕ.

$(\vee e)$ Si es el caso que $\phi \vee \psi$ y se quiere probar χ, entonces intenta probar χ a partir de ϕ y de ψ.

$(\rightarrow i)$ Si se quiere probar $\phi \rightarrow \psi$, pruebe ψ a partir de ϕ (y el resto de las premisas).

$(\neg i)$ Para probar $\neg\phi$, pruebe $\perp$ a partir de ϕ (y el resto de las premisas).

3.2.9 Equivalencia demostrable

Definición 3.4. *Sean ϕ y ψ dos fórmulas de la lógica proposicional. Decimos que ϕ y ψ forman una equivalencia demostrable, si y sólo si* (ssi) $\phi \vdash \psi$ y $\psi \vdash \phi$ *son inferencias válidas. Esto se denota como $\phi \dashv\vdash \psi$.*

Otra forma de expresar la equivalencia demostrable es con el teorema $\vdash (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. Ejemplos de equivalencias demostrables incluyen:

- $\neg(p \wedge q) \dashv\vdash \neg p \vee \neg q$
- $p \rightarrow q \dashv\vdash \neg q \rightarrow \neg p$
- $p \wedge q \rightarrow p \dashv\vdash r \vee \neg r$
- $\neg(p \vee q) \dashv\vdash \neg p \wedge \neg q$
- $p \rightarrow q \dashv\vdash \neg p \vee q$
- $p \wedge q \rightarrow r \dashv\vdash p \rightarrow (q \rightarrow r)$

3.3 LECTURAS Y EJERCICIOS SUGERIDOS

La presentación de las pruebas de inferencia que se ha hecho en este capítulo está basada en el libro de Huth y Ryan [54]. Este material se puede complementar con la lectura de la sección 7.5.1 del libro de Russell y Norvig [95]. La deducción natural fue introducida por Gerhard Karl Erich Gentzen entre 1934-1935: Untersuchungen über das logisch Schliessen. *Mathematische Zeitschrift* (39)2–3:176–210;405–431. Para más detalles, Pelletier [82] presenta una breve historia de la deducción natural. Dalen [29] en la sección 2.4, provee una aproximación más formal a la deducción natural.

Ejercicios

Ejercicio 3.1. *¿Es válida la argumentación de su tema de tesis? Intente una demostración.*

Ejercicio 3.2. *Busque un ejemplo de argumentación válida e inválida en el diario.*

Ejercicio 3.3. *La intepretación procedural de las reglas de prueba, introducida en la sección 3.2.8 sugiere que el proceso de prueba se puede automatizar. Abunde al respecto.*

Ejercicio 3.4. *Demuestre las equivalencias listadas en la sección 3.2.9.*

4 LA LÓGICA PROPOSICIONAL

Las reglas de derivación que se han introducido en el capítulo precedente, son válidas para cualquier fórmula que podamos formar con el lenguaje de la lógica proposicional. Aunque claro, por ahora lo único que sabemos sobre las posibles fórmulas del lenguaje es que, a partir de los átomos proposicionales podemos usar conectivos lógicos para crear fórmulas lógicas compuestas. Esto ha sido intencional, pues el objetivo del capítulo anterior era entender la mecánica de las reglas de la **deducción natural**. Es importante recordar la validez de estos patrones de razonamiento, antes de abordar la lógica proposicional como un lenguaje formal.

Consideren este caso, una aplicación de la regla de derivación $(\rightarrow e)$:

1.	$p \rightarrow q$	*premisa*
2.	p	*premisa*
3.	q	$(\rightarrow e)$ 1,2

su aplicación es válida aún si substituimos $p \vee \neg r$ por p y q con $r \rightarrow p$:

1.	$p \vee \neg r \rightarrow (r \rightarrow p)$	*premisa*
2.	$p \vee \neg r$	*premisa*
3.	$r \rightarrow p$	

Esta es la razón por la cual escribimos las reglas de prueba como esquemas de razonamiento, donde los símbolos griegos se usan como **meta variables** que pueden ser substituidas por fórmulas del lenguaje.

Meta variables

Ahora debemos precisar lo que queremos decir por *cualquier* fórmula del lenguaje. Lo primero que necesitamos es una fuente no acotada de **variables proposicionales**: $p, q, r, \ldots$ o $p_1, p_2, p_3, \ldots$. La cuestión del si tal conjunto es infinito no debería preocuparnos. El carácter no acotado de la fuente es una forma de confrontar que si bien, normalmente necesitaremos una gran cantidad finita de proposiciones para describir un programa de computadora, no sabemos de antemano cuantos vamos a necesitar. Lo mismo sucede con los lenguajes naturales, como el español. La cantidad de frases que puedo expresar en español es potencialmente infinita, pero basta escuchar a los presentadores de noticias de la televisión para saber que podemos usar una cantidad potencialmente finita de ellas.

Variables proposicionales

Que las fórmulas de nuestra lógica proposicional deben ser por tanto cadenas de caracteres formadas a partir del **alfabeto**:

Alfabeto

$$\{p, q, r, \ldots\} \cup \{p_1, p_2, p_3, \ldots\} \cup \{\neg, \wedge, \vee, \rightarrow, (,)\}$$

es una observación trivial, que no provee la información que necesitamos. Por ejemplo, $(\neg)$ y $pq \rightarrow$ son cadenas construídas a partir de este alfabeto, aunque no tienen sentido en la lógica proposicional. Necesitamos especificar cuales de esas cadenas son fórmulas bien formadas.

Definición 4.1 (Fórmula bien formada)**.** *Las fórmulas bien formadas (fbf) de la lógica proposicional son aquellas, y solo aquelas, que se obtienen aplicando finitamente las reglas de construcción siguientes:*

1. *Todo átomo proposicional* $p, q, r, \ldots$ *y* $p_1, p_2, p_3, \ldots$ *es una fórmula bien formada.*
2. *Si* ϕ *es una fórmula bien formada, también lo es* $(\neg\phi)$.
3. *Si* ϕ *y* ψ *son fbf, también lo es* $(\phi \wedge \psi)$.
4. *Si* ϕ *y* ψ *son fbf, también lo es* $(\phi \vee \psi)$.
5. *Si* ϕ *y* ψ *son fbf, también lo es* $(\phi \rightarrow \psi)$.

Este es el tipo de reglas que le gusta a las computadoras. Una fórmula es bien formada si es posible deducir tal cosa con las reglas anteriores. De hecho, las definiciones inductivas de las fórmulas bien formadas de un lenguaje son tan comunes que suelen expresarse en un formalismo diseñado *ad hoc* para este propósito, las gramáticas en notación **Backus-Naur** [1] (BNF). La definición anterior se expresa de manera más compacta como una gramática BNF:

Notación BNF

$$\phi \quad ::= \quad p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \tag{4.1}$$

donde p denota cualquier átomo proposicional y cada ocurrencia de ϕ a la derecha del símbolo ::= denota una fórmula bien formada previamente construida.

Ejemplo 4.1. *¿Cómo podemos probar que la cadena* $(((\neg p) \wedge q) \rightarrow (p \wedge (q \vee (\neg r))))$ *es una fbf? Afortunadamente, la gramática de la definición 4.1 satisface el* ***principio de inversión****: aunque la gramática permite la aplicación de cinco cláusulas, solo una de ellas se ha aplicado a lo último. En nuestro caso la última cláusula aplicada fue la cinco dado que la fórmula es una implicación con* $((\neg p) \wedge q)$ *como antecedente y* $(p \wedge (q \vee (\neg r)))$ *como conclusión. Siguiendo el principio de inversión podemos ver que el antecedente es una conjunción (cláusula tres) entre* $(\neg p)$ *y* q*, donde el primer operando es una negación (cláusula 2) de* p *que es una fbf (cláusula uno); al igual que* q*. De igual forma podemos proceder con el consecuente de la expresión original y demostrar que ésta es una fbf.*

Principio de inversión

Los paréntesis en estas expresiones suelen confundirnos un poco, aunque son necesarios pues reflejan el hecho de que las fórmulas tienen una **estructura de árbol**, como se muestra en la figura 4.1. Observen que los paréntesis se vuelven innecesarios en esa estructura, pero son necesarios sin linearizamos el árbol en un cadena de caracteres.

Estructura de árbol

Probablemente la forma más fácil de saber si una expresión es una fórmula bien formada, sea analizando su árbol sintáctico. En la figura 4.1 podemos observar que la raíz del árbol es una implicación, de manera que la expresión en cuestión es, a su nivel más alto una implicación. Ahora basta probar recursivamente que los sub-árboles izquierdo y derecho son también fórmulas bien formadas. Observen que las fórmulas bien formadas en el árbol o bien tienen como raíz un átomo proposicional; o un operador con el número adecuado de operandos. Esto ilustra el carácter inductivo de la gramática definida para la lógica proposicional.

Pensar en términos de árboles sintácticos ayuda a visualizar algunos conceptos estándar de la lógica proposicional, por ejemplo, el concepto de **sub-fórmula**. Dada la expresión del ejemplo 4.1, sus sub-fórmulas son aquellas que corresponden a los sub-árboles de la figura 4.1. Esto incluye las hojas p y q que ocurren dos veces; así como r. Luego $(\neg p)$ y $((\neg p) \wedge q)$ en el sub-árbol izquierdo de la implicación. Así como $(\neg r)$, $(p \vee (\neg r))$ y $((p \wedge (q \vee (\neg p))))$ en el sub-árbol derecho de la implicación. El árbol entero es un sub-árbol de si mismo. Así que las nueve sub-fórmulas de la expresión son:

Sub-fórmula

- p
- q
- r

1 John Backus nos dio otros regalitos: Fortran, Algol y los principios de la programación funcional expresados en su lenguaje FP.

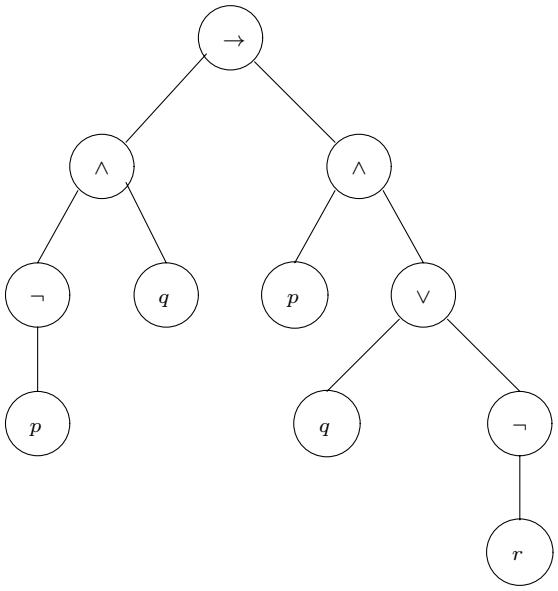


Figura 4.1: El árbol sintáctico de la fórmula bien formada del ejemplo 4.1.

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

Ejemplo 4.2. *Consideren el árbol de la figura 4.2 ¿Porqué representa una fórmula bien formada? Todas sus hojas son átomos proposicionales: p dos veces; q y r una. Todos sus nodos internos son operadores lógicos (¬ dos veces, $\wedge$, $\vee$ y $\rightarrow$) y el número de sus sub-árboles es el correcto en todos los casos (un sub-árbol para la negación, dos para los demás operadores). La expresión linearizada del árbol puede obtenerse recorriendo el árbol de manera en orden [2]:* $((\neg(p \vee (q \rightarrow (\neg p)))) \vee r)$.

Ejemplo 4.3. *El árbol de la figura 4.3 no representa una fórmula bien formada. Hay dos razones para ello: primero, las hojas $\wedge$ y ¬, lo cual puede arreglarse diciendo que el árbol está parcialmente construído; segundo, y definitivo, el nodo para el átomo proposicional p no es una hoja, es un nodo interno.*

4.1 SEMÁNTICA

En el capítulo anterior desarrollamos un cálculo del razonamiento que nos permite verificar si las inferencias de la forma $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ son válidas, lo cual quiere decir que a partir de las premisas $\phi_1, \phi_2, \ldots, \phi_n$ podemos concluir ψ. En esta sección abordaremos otra faceta de la relación entre las premisas y la consecuencia de las inferencias, conocida como **consecuencia lógica**. Para contrastar, denotaremos esta relación como:

Consecuencia lógica

$$\phi_1, \phi_2, \ldots, \phi_n \models \psi$$

2 También es posible obtener representación en pre-orden, como las usadas en Lisp, o post-orden; cambiando el recorrido del árbol.

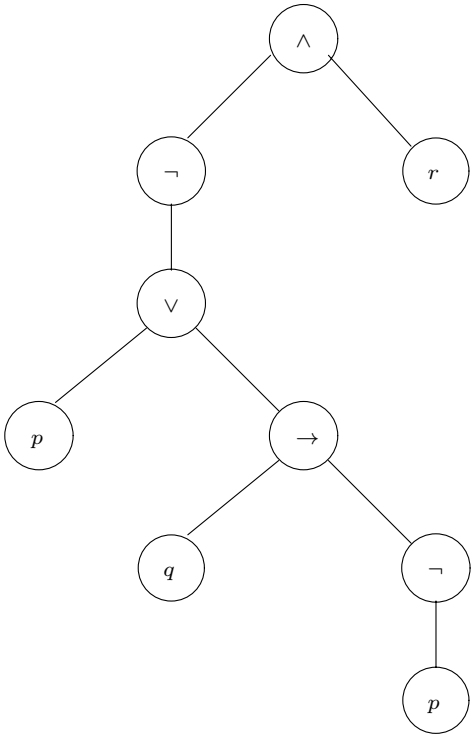


Figura 4.2: Otro ejemplo de un árbol sintáctico de una fórmula bien formada.

La consecuencia lógica se basa en los valores de verdad de las proposiciones atómicas que ocurren en las premisas y la conclusión; así como la forma en que los operadores lógicos manipulan estos valores de verdad. Pero ¿Qué es un valor de verdad? Recuerden que los enunciados declarativos que representan las proposiciones atómicas se corresponden con la realidad, decimos que son verdaderos; mientras que cuando este no es el caso decimos que son falsos.

Si combinamos los enunciados declarativos p y q con un conectivo lógico como $\wedge$, entonces el valor de verdad de $p \wedge q$ está determinado por tres aspectos: el valor de verdad de p, el valor de verdad de q y el significado de $\wedge$. El significado de la conjunción captura la observación de que $p \wedge q$ es verdadera si y sólo si (ssi) p y q son ambas verdaderas; de otra forma $p \wedge q$ es falsa. De forma que, desde la perspectiva de $\wedge$ todo lo que debemos saber es si p y q son verdaderos. No es necesario saber que es lo que dicen p y q acerca del mundo. Esto también es el caso para el resto de los operadores lógicos y es la razón por la cual podemos computar el valor de verdad de una expresión con solo saber el valor de verdad de las proposiciones atómicas que ocurren en ella. Formalicemos estos conceptos:

Definición 4.2 (Valores de verdad)**.** *El conjunto de valores de verdad contiene dos elementos T y F donde T representa verdadero, y F falso.*

Definición 4.3 (Modelo)**.** *Un modelo o valuación de una fórmula ϕ es una asignación de valores de verdad a las proposiciones atómicas que ocurren en ϕ.*

Ejemplo 4.4. *La función que asigna $T \mapsto q$ y $F \mapsto p$ es un modelo de la fórmula $p \vee \neg q$.*

Podemos pensar que el significado de la conjunción es una función de dos argumentos. Cada argumento es un valor de verdad y el resultado de la función es también un valor de verdad. Podemos especificar esto en una tabla, llamada **tabla de verdad** de la conjunción (Ver cuadro 4.1). *Tabla de verdad*

La tabla de verdad de la disyunción se muestra en el cuadro 4.2. La de la implicación se muestra en el cuadro 4.3.

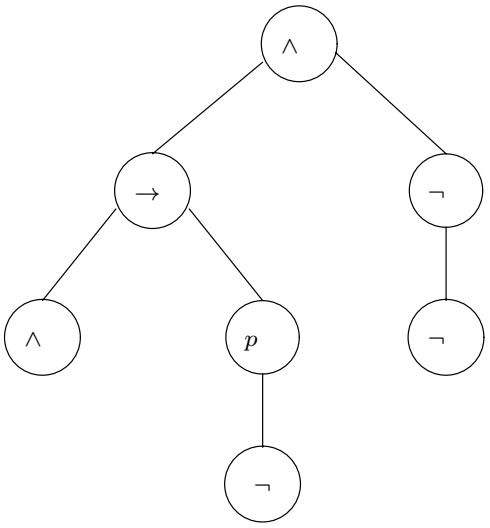


Figura 4.3: El árbol sintáctico de una expresión que no es una fórmula bien formada.

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

Cuadro 4.1: La tabla de verdad de la conjunción ($\wedge$).

Observen que, como cada argumento puede tener dos valores de verdad, el número de combinaciones posibles es 2^n donde n es el número de argumentos del operador. Por ejemplo, $\phi \wedge \psi$ tiene $2^2 = 4$ casos a considerar, los que se listan en la tabla de verdad del cuadro 4.1. Pero $\phi \wedge \psi \wedge \chi$ tendría $2^3 = 8$ casos a considerar.

La semántica de la implicación (Cuadro 4.3) es poco intuitiva. Piensen en ella como una relación que preserva la verdad. Es evidente que $T \rightarrow F$ no preserva la verdad puesto que inferimos algo que es falso a partir de algo que es verdadero. Por ello, la entrada correspondiente en la tabla de verdad de la implicación da como salida falso. También es evidente que $T \rightarrow T$ preserva la verdad; pero los casos donde el primer argumento tiene valor de verdad F también lo hacen, porque no tienen verdad a preservar dado que el supuesto de la implicación es falso.

Si la semántica de la implicación sigue siendo incomoda, consideren que, al menos en cuanto a los valores de verdad se refiere, la expresión $\phi \rightarrow \psi$ es una abreviatura de $\neg\phi \vee \psi$. Las tablas de verdad pueden usarse para probar que cuando la primer expresión mapea a verdadero, también lo hace la segunda. Esto quiere decir que ambas expresiones son **semánticamente equivalentes**. Aunque claro, las reglas de la deducción natural tratan a ambas fórmulas de manera muy diferente, dado que su sintaxis es bien diferente.

Equivalencia semántica

La tabla de verdad de la negación se muestra en el cuadro 4.4. Observen que en este caso tenemos $2^1 = 2$ casos que considerar. También pueden definirse tablas de verdad para la contradicción y su negación: $\bot \mapsto F$ y $\top \mapsto T$.

Ejemplo 4.5. *¿Cual es el valor de verdad de la expresión $\neg p \wedge q \rightarrow p \wedge (q \vee \neg r)$, si el modelo es $F \mapsto q$, $T \mapsto p$ y $T \mapsto r$? Una de las ventajas de nuestra semántica es que es **composicional**. Si sabemos los valores de verdad de $\neg p \wedge q$ y $p \wedge (q \vee \neg r)$, entonces podemos usar la tabla de verdad de la implicación para saber el valor de verdad de toda la expresión. De tal forma que podemos recorrer el árbol sintáctico de la expresión en forma ascendente para computar su valor de verdad. Primero, sabemos el valor de las hojas dado*

Semántica composicional

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

Cuadro 4.2: La tabla de verdad de la disyunción ($\vee$).

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

Cuadro 4.3: La tabla de verdad de la implicación ($\rightarrow$).

el modelo inicial. Puesto que el valor de verdad de p es verdadero, el valor de $\neg p$ es falso, q es falsa y la conjunción de falso y falso da falso. De forma que el lado izquierdo de la conjunción evalúa a falso. Para el lado derecho tenemos que el modelo inicial asigna a r el valor de verdad verdadero, de forma que $\neg r$ es falso. La disyunción de q que tiene un valor falso, con falso, da falso. Y la conjunción de p, cuyo valor es verdadero con falso, da falso. Finalmente, la implicación de dos valores de verdad falsos, da verdadero. La figura 4.4 ilustra la propagación de valores de verdad por este árbol sintáctico, como se ha explicado en el ejemplo.

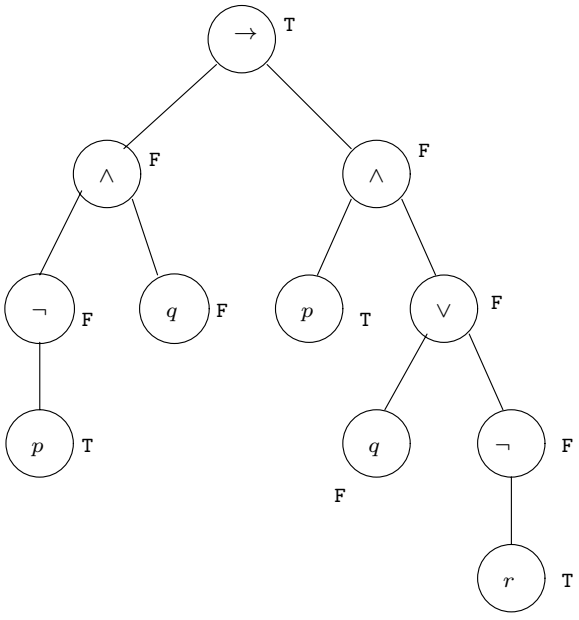


Figura 4.4: La evaluación de una fórmula lógica a partir de su árbol sintáctico, dado un modelo inicial.

Ejemplo 4.6. *Construya la tabla de verdad de la expresión* $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$*:*

ϕ	$\neg\phi$
T	F
F	T

Cuadro 4.4: La tabla de verdad de la negación ($\neg$).

p	q	$\neg p$	$\neg q$	$p \to \neg q$	$q \vee \neg p$	$(p \to \neg q) \to (q \vee \neg p)$
T	T	F	F	F	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	T
F	F	T	T	T	T	T

4.2 INDUCCIÓN MATEMÁTICA

A la edad de ocho años, Gauss se aburría en clase y no prestaba la atención debida a su profesor. Como es de imaginarse, al pequeño se le impuso un castigo que consistía en sumar los números del uno al cien. Un burdo rumor dice que Gauss [3] respondió 5050 a los pocos segundos de haber sido castigado, para sorpresa de su profesor ¿Cómo logró Gauss librarse tan fácilmente del castigo? Quizá sabía que:

$$1+2+3+4+\cdots+n = \frac{n(n+1)}{2} \tag{4.2}$$

de forma que:

$$\begin{aligned} 1+2+3+4+\cdots+100 &= \frac{100(101)}{2} \\ &= 5050 \end{aligned}$$

Inducción matemática

La **inducción matemática** nos permite probar que la ecuación 4.2 se sostiene para valores arbitrarios de n. De manera más general, podemos decir que nos permite probar que **todo** número natural satisface cierta propiedad. Supongamos que tenemos una propiedad M que creemos es verdadera para todo número natural. Escribimos $M(5)$ para denotar que la propiedad es verdadera para 5. Supongamos ahora que sabemos lo siguiente de M:

1. **Caso base:** El número natural 1 tiene la propiedad M, es decir, tenemos una prueba de que $M(1)$.
2. **Paso inductivo:** Si n es un número natural que asumimos tiene la propiedad $M(n)$, entonces podemos probar que $n+1$ tiene la propiedad $M(n+1)$; es decir, tenemos una prueba de que $M(n) \to M(n+1)$.

Definición 4.4. *El principio de inducción matemática dice que, basados en estas dos piezas de información, todo número natural n tiene la propiedad $M(n)$. Al hecho de suponer $M(n)$ en el paso inductivo, se le conoce como* ***hipótesis de la inducción****.*

¿Porqué tiene sentido este principio? Tomemos cualquier número natural k. Si k es igual a 1, entonces usando el caso base podemos probar que k tienen la propiedad $M(1)$. En cualquier otro caso, podemos usar el paso inductivo aplicado a $n = 1$ para inferir que $2 = 1+1$ tiene la propiedad $M(2)$. Podemos hacer esto usando la eliminación de la implicación ($\to e$) puesto que sabemos que 1 tiene la propiedad en cuestión. Igualmente podemos probar $M(3)$ y así hasta llegar a k. Regresemos al ejemplo de Gauss.

3 Lo que no es un rumor, es su precocidad: terminó sus *Disquisitiones Arithmeticae* a los 21 años.

Teorema 4.1. *La suma de* $1+2+3+4+\cdots+n$ *es igual a* $n(n+1)/2$ *para todo número natural* n*.*

La prueba es como sigue: Denotamos por LIE_n el lado izquierdo de la igualdad, $1+2+3+4+\cdots+n$; y por LDE_n el lado derecho de la igualdad $n(n+1)/2$ para todo $n \geq 1$.

- **Caso base:** Si $n = 1$ entonces $LIE_1 = 1$ (solo hay un sumando), que es igual a $LDE_1 = 1(1+1)/2 = 1$.
- **Paso inductivo:** Asumamos que $LIE_n = LDE_n$. Recuerden que este supuesto es llamado hipótesis inductiva; es la guía de nuestro argumento. Necesitamos probar que $LIE_{n+1} = LDE_{n+1}$, esto es, que $1+2+3+4+\cdots+(n+1)$ es igual que $(n+1)(((n+1)+1)/2)$. La clave está en observar que la suma de $1+2+3+4+\cdots+(n+1)$ no es otra cosa que la suma de dos sumandos $1+2+3+4+\cdots+n$ y $(n+1)$; y que el primer sumando es nuestra hipótesis inductiva, de forma que:

$$\begin{aligned}
LIE_{n+1} &= 1+2+3+4+\cdots+n+(n+1) \\
&= LIE_n + (n+1) \quad \text{reagrupando la suma} \\
&= LDE_n + (n+1) \quad \text{por la hipótesis inductiva} \\
&= \frac{n(n+1)}{2} + (n+1) \\
&= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\
&= \frac{(n+2)(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2} \\
&= \frac{(n+1)((n+1)+1)}{2} \\
&= LDE_{n+1}
\end{aligned}$$

De forma que, como hemos probado el caso base y el paso inductivo, podemos inferir matemáticamente que todo número natural n satisface el teorema anterior. □

Existe una variante de inducción matemática en la que la hipótesis inductiva para probar $M(n+1)$ no es solo $M(n)$, sino la conjunción $M(1) \wedge M(2) \wedge \ldots \wedge M(n)$. En esta variante, llamada **curso de valores**, no es necesario tener un caso base explícito, todo puede hacerse en el paso inductivo. *Curso de valores*

¿Cómo puede funcionar la inducción sin un caso base? La respuesta es que el caso base se incluye implicitamente en el paso inductivo. Consideren el caso de $n = 3$: el paso inductivo es $M(1) \wedge M(2) \wedge M(3) \rightarrow M(4)$. Ahora consideren el caso $n = 1$ entonces el paso inductivo es $M(1) \rightarrow M(2)$ ¿Qué sucede cuando el caso es $n = 0$? En ese caso no hay fórmulas a la izquierda de la implicación, por lo que se debe probar $M(1)$ a partir de nada. Veamos algunos ejemplos.

En las Ciencias de la Computación solemos trabajar con estructuras finitas de algún tipo: estructuras de datos, programas, archivos, etc. En muchas ocasiones debemos probar que toda ocurrencia de una estructura de datos tiene cierta propiedad. Por ejemplo, la definición 4.1 de fórmula bien formada tiene la propiedad de que el número de paréntesis que abren es igual al de los paréntesis que cierran. Podemos usar la inducción matemática en el dominio de los números naturales para probar esto. Para tener éxito, debemos ligar de alguna forma las fórmulas bien formadas con los números naturales.

Definición 4.5. *Dada una fórmula bien formada* ϕ*, definimos su* ***altura*** *como 1 más la longitud de la rama más larga del árbol.*

Por ejemplo, la rama más larga del árbol que se muestra en la figura 4.5 es de longitud 4 por lo que su altura es 5. Observen que la altura de un átomo proposicional es $1 + 0 = 1$. Puesto que toda fórmula bien formada tiene una altura finita, podemos demostrar enunciados sobre las fórmulas bien formadas haciendo inducción matemática sobre su altura. Este truco suele conocerse como **inducción estructural**, una importante técnica de razonamiento de Ciencias de la Computación. Usando la noción de altura de un árbol sintáctico nos damos cuenta de que la inducción estructural es solo un caso especial de la inducción por cursos-de-valores.

Inducción estructural

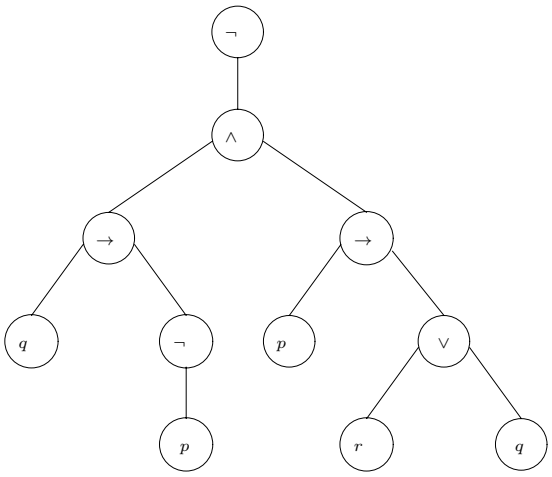


Figura 4.5: Un árbol sintáctico de altura 5.

Teorema 4.2. *Para toda fórmula bien formada de la lógica proposicional, el número de paréntesis que abre es igual al número de paréntesis que cierran.*

Prueba: Procederemos por inducción por curso de valores sobre la altura del árbol sintáctico de las fórmulas bien formadas ϕ . Denotemos por $M(n)$ que "todas las fórmulas de altura n tienen el mismo número de paréntesis que abren y cierran." Asumimos $M(k)$ para cada $k < n$ y tratamos de probar $M(n)$. Tomemos una fórmula ϕ de altura n.

- **Caso base:** Cuando $n = 1$, ϕ es un átomo proposicional por lo que no hay paréntesis en la expresión y $M(1)$ se satisface: $0 = 0$.
- **Paso inductivo por curso-de-valores:** Cuando $n > 1$ la raíz del árbol sintáctico de ϕ debe ser $\neg$, $\rightarrow$, $\vee$ o $\wedge$. Supongamos que es $\rightarrow$ (los otros tres casos se argumentan de manera similar). Entonces ϕ es igual a $(\phi_1 \rightarrow \phi_2)$ para las fórmulas bien formadas ϕ_1 y ϕ_2. Es claro que la altura de los árboles sintácticos de estas dos expresiones es menor que n. Usando la hipótesis de inducción concluímos que ϕ_1 tiene el mismo número de paréntesis que abren y cierran; lo mismo que ϕ_2. Pero en $(\phi_1 \rightarrow \phi_2)$ agregamos un paréntesis que abre y uno que cierra por lo que el número de ocurrencias de los paréntesis en ϕ es el mismo. □

4.3 ROBUSTEZ DE LA LÓGICA PROPOSICIONAL

Las reglas de la deducción natural permiten desarrollar rigurosos hilos de argumentación a través de los cuales concluímos que ψ asumiendo otras proposiciones como $\phi_1, \phi_2, \ldots, \phi_n$. En ese caso decimos que la inferencia $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es válida ¿Tenemos evidencia de que las reglas son todas **correctas** en el sentido de que preserven la verdad computada con nuestras semántica basada en tablas de verdad?

Robustez

Dada una prueba de $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ ¿Es concebible que exista un modelo donde ψ es falso aunque las proposiciones $\phi_1, \phi_2, \ldots, \phi_n$ sean verdaderas? Afortunadamente este no es el caso y en esta sección lo demostraremos formalmente. Supongamos que tenemos una prueba en nuestro cálculo de deducción natural que establece que la inferencia $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es válida. Necesitamos probar que para todo modelo donde las proposiciones $\phi_1, \phi_2, \ldots, \phi_n$ son verdaderas, ψ también lo es.

Definición 4.6. *Si, para todos los modelos donde $\phi_1, \phi_2, \ldots, \phi_n$ son verdaderas, ψ también lo es, entonces decimos que:*

$$\phi_1, \phi_2, \ldots, \phi_n \models \psi$$

*El símbolo $\models$ se llama relación de **consecuencia lógica** o vinculación lógica (*logical entailment*).*

Veamos algunos ejemplos de esta noción:

1. ¿Es el caso que $p \wedge q \models p$? Para responder debemos inspeccionar todas las asignaciones de verdad para p y q. Cuando la asignación de valores compute T para $p \wedge q$ debemos asegurarnos de que ese también es el caso para p. Pero $p \wedge q$ solo computa T cuando p y q son verdaderas, por lo que p es consecuencia lógica de $p \wedge q$.

2. ¿Qué hay acerca de $p \vee q \models p$? Hay tres asignaciones de verdad donde $p \vee q$ es T, de forma que p debería ser T en todas ellas. Sin embargo, si asignamos T a q y F a p la disyunción computa T pero p es falsa. De forma que la relación $p \vee q \models p$ no se mantiene.

3. ¿Qué sucede si modificamos la inferencia anterior para que sea $p \vee q \models p$. Observe que esto obliga a focalizar en evaluaciones donde q es falsa, lo cual obliga a que p sea verdadera si queremos que la disyunción lo sea. Por tanto, es el caso que $p \vee q \models p$

4. Observen que $p \models q \vee \neg q$ se da, aún cuando no existen ocurrencias de las proposiciones atómicas del antecedente en el consecuente.

De la discusión anterior podemos adivinar que el argumento de la robustez consiste en probar que si $\phi_1, \phi_2, \ldots \phi_n \vdash \psi$, entonces se da que $\phi_1, \phi_2, \ldots \phi_n \models \psi$.

Teorema 4.3 (Robustez)**.** *Sean ϕ_1, ϕ_2, ϕ_n y ψ fórmulas lógicas proposicionales. Si la inferencia $\phi_1, \phi_2, \ldots \phi_n \vdash \psi$ es valida, entonces es el caso que $\phi_1, \phi_2, \ldots \phi_n \models \psi$.*

Prueba: Puesto que $\phi_1, \phi_2, \ldots \phi_n \vdash \psi$ es válida, conocemos una prueba de ψ a partir de las premisas $\phi_1, \phi_2, \ldots \phi_n$. Ahora aplicamos un truco estructural, razonaremos por **inducción matemática sobre la longitud de esta prueba**. La longitud de una prueba es el número de líneas en ella. De forma que intentamos mostrar $M(k)$: para toda consecuencia $\phi_1, \phi_2, \ldots \phi_n \vdash \psi \quad (n \geq 0)$ que tiene una prueba de longitud k, es el caso que $\phi_1, \phi_2, \ldots \phi_n \models \psi$ se da. Veamos:

La consecuencia $p \wedge q \rightarrow r \vdash p \rightarrow (q \rightarrow r)$ tiene una prueba:

1.	$p \wedge q \rightarrow r$	*premisa*
2.	p	*supuesto*
3.	q	*supuesto*
4.	$p \wedge q$	$(\wedge i)$ 2, 3
5.	r	$(\rightarrow e)$ 1, 4
6.	$q \rightarrow r$	$(\rightarrow i)$ $3-5$
7.	$p \rightarrow (q \rightarrow r)$	$(\rightarrow i)$ $2-6$

Si eliminamos las dos últimas líneas ya no tenemos una prueba, pues la caja más externa quedaría sin cerrar. Sin embargo, podemos obtener una prueba removiendo la última línea y escribiendo el supuesto de la caja más externa como una premisa:

1.	$p \wedge q \rightarrow r$	*premisa*
2.	p	*premisa*
3.	q	*premisa*
4.	$p \wedge q$	$(\wedge i)$ 2, 3
5.	r	$(\rightarrow e)$ 1, 4
6.	$q \rightarrow r$	$(\rightarrow i)$ $3-5$

Esto es una prueba de la inferencia $p \wedge q \rightarrow r, p \vdash q \rightarrow r$. La hipótesis inductiva garantiza entonces que $p \wedge q \rightarrow r, p \models q \rightarrow r$. Entonces podemos razonar que también es el caso que $p \wedge q \rightarrow r \models p \rightarrow (q \rightarrow r)$ ¿Porqué?

Procedamos con la prueba por inducción. Asumamos que $M(k')$ para cada $k' < k$ y tratemos de probar que $M(k)$.

- **Caso base:** una prueba de longitud 1. Si $k = 1$ entonces la prueba debe ser de la forma:

1.	ϕ	premisa

  puesto que todas las demas reglas involucran más de una linea. Este es el caso cuando $n = 1$ y ϕ_1 y ψ son iguales a ϕ, es decir, la inferencia es $\phi \vdash \phi$. Evidentemente si ϕ evalua a T, es el caso que $\phi \models \phi$.

- **Paso inductivo por curso de valores:** Asumamos que la prueba de la consecuencia $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ tiene una longitud k y que el enunciado que queremos probar es verdadero para todo número menor que k. Nuestra prueba tiene la siguiente estructura:

1.	ϕ_1	premisa
2.	ϕ_2	premisa
	$\vdots$	
n.	ϕ_n	premisa
	$\vdots$	
k.	ψ	justificación

  Hay dos cosas que no sabemos en este punto. Primero, ¿Qué está pasando en los puntos suspensivos? Segundo, ¿Cual fue la última regla aplicada? Es decir, ¿Cual fue la justificación de la última línea? El primer punto no es de nuestro interés, es aquí que la inducción matemática muestra su poder. El segundo punto es donde todo el trabajo de esta prueba descansa. En general, no hay una forma simple de saber que regla fue aplicada a lo último, por lo que necesitamos considerar todos los casos posibles:

  1. Supongamos que la última regla es $(\wedge i)$, entonces sabemos que ψ tiene la forma $\psi_1 \wedge \psi_2$ y la justificación de la línea k hace referencia a dos líneas precedentes que tienen a ψ_1 y ψ_2 respectivamente, como conclusiones. Supongamos que esas líneas son k_1 y k_2. Dado que k_1 y k_2 son menores que k observamos que existen pruebas de las inferencias $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_1$

y $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_2$ con una longitud menor que k. Usando la hipótesis inductiva concluimos que $\phi_1, \phi_2, \ldots, \phi_n \models \psi_1$ y $\phi_1, \phi_2, \ldots, \phi_n \models \psi_2$. Estas dos relaciones implican que $\phi_1, \phi_2, \ldots, \phi_n \models \psi_1 \wedge \psi_2$.

2. Supongamos que la última regla para demostrar ψ es $(\vee e)$, entonces debimos probar, asumiendo o adoptando la premisa de que alguna fórmula $\eta_1 \vee \eta_2$ en alguna línea k' dond $k' < k$ que es referenciada por $(\vee e)$ en la línea k. Por lo tanto, tendremos una prueba más corta de la consecuencia $\phi_1, \phi_2, \ldots \phi_n \vdash \eta_1 \vee \eta_2$, obtenida al convertir los supuestos de las cajas que se abren en la linea k' en premisas. De forma similar obtenemos pruebas de las consecuencias $\phi_1, \phi_2, \ldots \phi_n, \eta_1 \vdash \psi$ y $\phi_1, \phi_2, \ldots \phi_n, \eta_2 \vdash \psi$. Por la hipótesis inductiva concluímos que $\phi_1, \phi_2, \ldots \phi_n \models \eta_1 \vee \eta_2$, $\phi_1, \phi_2, \ldots \phi_n, \eta_1 \models \psi$ y $\phi_1, \phi_2, \ldots \phi_n, \eta_2 \models \psi$. En su conjunto, estas tres relaciones hacen que $\phi_1, \phi_2, \ldots \phi_n \models \psi$.
3. La argumentación continua probando que todas las reglas de prueba se comportan semánticamente en la misma forma que las tablas de verdad correspondiente. □

No existencia de demostración

La robustez de la lógica proposicional es útil para garantizar la **no existencia** de una prueba para una consecuencia dada. Digamos que queremos probar que $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es válida, pero no lo conseguimos ¿Cómo podemos estar seguros de que no hay una prueba para ese caso? Basta con encontrar un modelo en donde ϕ_i evalua a T mientras que ψ evalua a F. Entonces por la definición de consecuencia lógica, no es el caso que $\phi_1, \phi_2, \ldots \phi_n \models \psi$ y, usando la robustez, esto significa que la consecuencia $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ no es válida; y por tanto, no hay una prueba para ella.

4.4 COMPLETITUD DE LA LÓGICA PROPOSICIONAL

Completitud

En esta sección probaremos que las reglas de la deducción natural de la lógica proposicional son **completas**. Cuando es el caso que $\phi_1, \phi_2, \ldots, \phi_n \models \psi$, entonces existe una prueba de deducción natural para la consecuencia $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Combinando esto con el resultado anterior obtenemos que $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es válida, si y sólo si $\phi_1, \phi_2, \ldots, \phi_n \models \psi$.

Búsqueda de prueba

Esto nos da cierta libertad con respecto al método que preferiremos usar. El primer método involucra una **búsqueda de prueba**, que es la base del paradigma de la **programación lógica**. El segundo método nos obliga a computar una tabla de verdad que es exponencial en el número de proposiciones atómicas involucradas en una fórmula. Ambos métodos son intratables en lo general, pero ciertas fórmulas particulares responden diferente a ellos.

El argumento que construiremos en esta sección se da en tres pasos:

1. Mostraremos que $\models \phi_1 \rightarrow (\phi_2 \rightarrow (\ldots (\phi_n \rightarrow \psi)))$ es el caso.
2. Mostraremos que $\vdash \phi_1 \rightarrow (\phi_2 \rightarrow (\ldots (\phi_n \rightarrow \psi)))$ es válida.
3. Finalmente, mostraremos que $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es válida.

El primer paso y el tercero son bastante fáciles, todo el trabajo real se concentra en el segundo paso.

4.4.1 Paso 1

Definición 4.7 (Tautología)**.** *Una fórmula de la lógica proposicional ϕ es llamada una tautología si y sólo si evalua T bajo cualquier modelo. Es decir, si $\models \phi$.*

Supongamos que $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ es el caso. Verifiquemos que $\phi_1 \rightarrow (\phi_2 \rightarrow (\ldots(\phi_n \rightarrow \psi)))$ es una tautología. Como la fórmula en cuestión es una implicación, solo puede evaluar F si y sólo si todas las ϕ_i evalúan a T y ψ evalua a F. Pero esto contradice el hecho de que $\phi_1, \phi_2, \ldots, \phi_n \models \psi$; por lo tanto $\models \phi_1 \rightarrow (\phi_2 \rightarrow (\ldots(\phi_n \rightarrow \psi)))$.

4.4.2 Paso 2

Definición 4.8. *Si $\models \eta$, entonces $\vdash \eta$ es válida. En otras palabras, si η es una tautología, entonces η es un teorema.*

Asumamos que $\models \eta$. Dado que η contiene n distintos átomos proposicionales $p_1, p_2, \ldots, p_n$ sabemos que η es T para todas las 2^n lineas de su tabla de verdad. Cada línea lista un modelo de η ¿Cómo podemos usar esta información para construir una prueba de η? En algunos casos esto puede hacerse facilmente, observando cuidadosamentela estructura de η, pero aquí deberemos contender con una forma **uniforme** de construir tal prueba. La clave está en codificar cada linea de la tabla de verdad de η como una consecuencia. Entonces construiremos pruebas para las 2^n inferencias y las ensamblaremos en la prueba de η.

Proposición 4.1. *Sea ϕ una fórmula tal que $p_1, p_2, \ldots, p_n$ son sus únicos átomos proposicionales. Sea l cualquier número de línea en la tabla de verdad de ϕ. Para todo $1 \leq i \leq n$, $\hat{p}_i$ es p_i si la entrada de la línea l de p_i es T; en cualquier otro caso $\hat{p}_i$ es $\neg p_i$. Entonces tenemos:*

1. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi$ es demostrable si la entrada para ϕ en la línea l es verdadera.*
2. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi$ es demostrable si la entrada para ϕ en la línea l es falsa.*

Prueba: Esta prueba se lleva a cabo por inducción estructural sobre la fórmula ϕ, que es una inducción matemática sobre la altura del árbol sintáctico de ϕ.

- Si ϕ es un átomo proposicional p, debemos mostrar que $p \vdash p$ y que $\neg p \vdash \neg p$. Se puede hacer directamente la prueba.
- Si ϕ es de la forma $\neg\phi_1$, nuevamente tenemos dos casos a considerar. Primero, asumamos que ϕ es verdadera. En ese caso ϕ_1 es falsa. Observen que ϕ_1 tiene las mismas proposiciones atómicas que ϕ. Debemos usar la hipótesis de inducción sobre ϕ_1 para concluir que $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi_1$, pero $\neg\phi_1$ es ϕ; Segundo si ϕ es falsa, entonces ϕ_1 es verdadera y tenemos que, por inducción $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$. Usando $(\neg\neg i)$ podemos extener esta prueba en $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\neg\phi_1$. Pero $\neg\neg\phi_1$ es $\neg\phi$.

El resto de los casos trabajan con dos sub-fórmulas: ϕ es igual a $\phi_1 \circ \phi_2$, donde $\circ$ es $\rightarrow$, $\wedge$ ó $\vee$. En todos estos casos, sea $q_1, q_2, \ldots, q_l$ los átomos proposicionales en ϕ_1 y $r_1, r_2, \ldots, r_k$ los átomos proposicionales en ϕ_2. Entonces la siguiente igualdad se satisface $\{q_1, q_2, \ldots, q_l\} \cup \{r_1, r_2, \ldots, r_k\} = \{p_1, \ldots, p_n\}$. De forma que siempre que $\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_l \vdash \phi_1$ y $\hat{r}_1, \hat{r}_2, \ldots, \hat{r}_k \vdash \phi_2$ son válidas, también lo es $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ usando la regla de $(\wedge i)$.

- Sea ϕ de la forma $\phi_1 \rightarrow \phi_2$. Si ϕ es falsa, entonces sabemos que ϕ_1 también lo es, mientras que ϕ_2 es verdadera. Usando nuestra hipótesis inductiva tenemos que $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ y que $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$. De forma que $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$. El resto consiste entonces en probar que la inferencia $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \rightarrow \phi_2)$ que queda como ejercicio sugerido. Si ϕ es verdadero entonces hay tres casos a considerar. Primero, si ϕ_1 y ϕ_2 son falsos. Por hipótesis inductiva tenemos que: $\hat{q}_1, \ldots, \hat{q}_l \vdash \neg\phi_1$ y $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$. De forma que $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$. Solo queda probar que la inferencia $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \rightarrow \phi_2$ es válida. Segundo, si ϕ_1 es falso y ϕ_2 es verdadero usamos la hipótesis inductiva para llegar

a $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ y tendríamos que probar que $\neg\phi_1 \wedge \phi 2 \vdash \phi_1 \rightarrow \phi_2$ es una inferencia válida. Tercero, si ϕ_1 y ϕ_2 son verdaderos, llegamos a $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ y solo resta probar que $\phi_1 \wedge \phi 2 \vdash \phi_1 \rightarrow \phi_2$.

Veamos un ejemplo de cómo funcionan estas pruebas basadas en la tabla de verdad de una consecuencia. Si queremos probar de esta manera que la fórmula bien formada $p \wedge q \rightarrow p$ es una tautología, tendremos que considerar que las cuatro líneas (2^2) de su tabla de verdad deberían ser verdaderas. Por ello tendríamos cuatro pruebas del tipo $\hat{p_1}, \hat{p_2} \vdash \eta$:

$$
\begin{aligned}
p, q &\vdash p \wedge q \rightarrow p \\
\neg p, q &\vdash p \wedge q \rightarrow p \\
p, \neg q &\vdash p \wedge q \rightarrow p \\
\neg p, \neg q &\vdash p \wedge q \rightarrow p
\end{aligned}
$$

Puesto que queremos probar que esta proposición es una tautología, debemos encargarnos de que la demostración de que es un teorema no tenga premisas. Para contender con el lado izquierdo de las cuatro consecuencias, recurriremos al *LEM*. La prueba es como sigue:

1.	$p \vee \neg p$	*LEM*
2.	p	*supuesto*
3.	$q \vee \neg q$	*LEM*
4.	q	*supuesto*
5.	$\vdots$	$\vdots$
6.	$p \wedge q \rightarrow p$	
7.	$\neg q$	*supuesto*
8.	$\vdots$	$\vdots$
9.	$p \wedge q \rightarrow p$	
10.	$p \wedge q \rightarrow p$	$(\vee e) 3, 4-6, 7-9$
11.	$\neg p$	*supuesto*
12.	$q \vee \neg q$	*LEM*
13.	q	*supuesto*
14.	$\vdots$	$\vdots$
15.	$p \wedge q \rightarrow p$	
16.	$\neg q$	*supuesto*
17.	$\vdots$	$\vdots$
18.	$p \wedge q \rightarrow p$	
19.	$p \wedge q \rightarrow p$	$(\vee e) 12, 13-15, 16-18$
20.	$p \wedge q \rightarrow p$	$(\vee e) 1, 2-10, 11-19$

4.4.3 Paso 3

Finalmente, necesitamos encontrar una prueba de que $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ es una inferencia válida. Tomamos la prueba de que $\vdash \phi_1 \rightarrow (\phi_2 \rightarrow (\ldots(\phi_n \rightarrow \psi)))$ obtenida en el paso 2 y aumentamos la prueba introduciendo $\phi_1, \ldots, \phi_n$ como premisas. Entonces aplicamos $(\rightarrow e)$ n veces sobre cada una de las premisas y llegaremos a la conclusión que ψ lo cual nos da la prueba buscada.

Corolario 4.1 (Robustez y Completitud)**.** *Sean $\phi_1, \ldots, \phi_n$ y ψ fórmulas de la lógica proposicional. Entonces $\phi_1, \ldots \phi_n \models \psi$ si y sólo si $\phi_1, \ldots \phi_n \vdash \psi$ es válida.*

4.5 FORMAS NORMAL CONJUNTIVA

En la sección anterior probamos que nuestro sistema de prueba para la lógica proposicional es robusto y completo para la semántica basada en tablas de verdad. Esto quiere decir que todo lo que probemos es un hecho verdadero; y que, toda consecuencia válida tiene una prueba en el sistema de deducción natural. Esta conexión nos permitirá movernos libremente entre el nivel de prueba ($\vdash$) y el de consecuencia lógica ($\models$). En lo qu sigue exploraremos formas alternativas de probar que $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ basados en la transformación sintáctica de estas fbfs en equivalentes sintácticos que puedan resolverse algoritmicamente. Esto requiere definir con precisión el concepto de **equivalencia**.

Equivalencia semántica

Definición 4.9 (Equivalencia semántica)**.** *Sean ϕ y ψ fbfs de la lógica proposicional. $\phi \equiv psi$, ϕ es semátincamente equivalente a ψ ssi $\phi \models \psi$ y $\psi \models \phi$.*

Decimos que ϕ es válida, ssi $\models \phi$. También pudimos definir $\phi \equiv \psi$ para denotar que $\models (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. Dadas la robustez y la completitud, la equivalencia semántica y de prueba son idénticas.

Ejemplo 4.7. *Las siguientes equivalencias semáticas son válidas:*

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

En lo que sigue, necesitaremos una prueba de que todo procedimiento de decisión para tautologías, lo es también para consecuencias válidas:

Lema 4.1. *Dadas la fórmulas $\phi_1, \phi_2, \ldots, \phi_n$ y ψ de la lógica proposicional, $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ ssi $\models \phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi))$.*

Su prueba es como sigue: Suponemos que $\models \phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi))$ es el caso. Si las ϕ_i son verdaderas bajo una valuación, también lo debe ser ψ. A la inversa, es el paso 1 de la prueba de completitud. □

Definición 4.10 (Forma Normal Conjuntiva)**.** *Una literal L es un átomo p o su negación (4.3). Una fórmula C está en **Formal Normal Conjuntiva** (CNF) si es una conjunción de cláusulas (4.5), donde cada cláusula es una disyunción de literales (4.4).*

Forma Normal Conjuntiva

$$L ::= p \mid \neg p \tag{4.3}$$
$$D ::= L \mid L \vee D \tag{4.4}$$
$$C ::= D \mid D \wedge C \tag{4.5}$$

Ejemplo 4.8. *Las siguientes fbfs están en forma nornal conjuntiva:*

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

La **relevancia** de la CNF es que permite verificar la validez de una fbf fácilmente, proceso que de otra forma es exponencial en el número de átomos de la fbf a verificar. *Relevancia*

Lema 4.2. *Una disyunción de literales $L_1 \vee ... \vee L_m$ es válida ssi existe un $L_i = \neg L_j$ para $1 \leq i, j \leq m$.*

Definición 4.11 (Satisfacción)**.** *Sea ϕ una fbf proposicional, ϕ es **satisfacible** ssi $\neg\phi$ no es válida.* *Satisfacción*

La prueba es como sigue: Asumimos que ϕ es satisfacible. Por definición, existe una valuación donde ϕ es verdadera, pero eso implica que $\neg\phi$ sería falsa en la misma valuación, por lo que $\neg\phi$ no pude ser válida; Asumimos que $\neg\phi$ no es valida, debe haber una valuación donde $\neg\phi$ sea falsa, en cuyo caso ϕ es verdadera y por tanto satisfacible. □

Este resultado es muy interesante, pues implica que solo necesitamos un procedimiento de decisión para ambos conceptos.

4.5.1 Conversión de una fbf a CNF

La ganancia en la eficiencia del procedimiento de prueba, se paga con el precio de convertir una fbf proposicional a su equivalente en CNF. Comenzaremos por un algoritmo **determinista** que siempre computa la misma CNF para una fbf ϕ de entrada. Sus características deseables incluyen:

1. CNF termina para toda fbf de la lógica proposicional.
2. Para cada fbf de la lógica proposicional CNF computa una fbf equivalente.
3. La fbf resultante está en forma normal conjuntiva.

La **estrategia** a seguir consiste en proceder por inducción estructural sobre la fbf ϕ. La inducción estructural garantiza las características deseables del algoritmo. *Estrategia*

Ejemplo 4.9. *Si ϕ es de la forma $\phi_1 \wedge \phi_2$ computar la CNF η_i para ϕ_i, $i = 1, 2$; de forma que $\eta_1 \wedge \eta_2$ es la CNF equivalente a ϕ.*

De forma que la CNF se resuelve por casos:

1. Si ϕ es una literal, por definición está en CNF y la salida es ϕ.
2. Si ϕ es de la forma $\phi_1 \wedge \phi_2$ se llama a CNF recursivamente sobre cada ϕ_i para obtener η_1 y η_2. La CNF es $\eta_1 \wedge \eta_2$.
3. Si ϕ tiene la forma $\phi_1 \vee \phi_2$ se llama a CNF sobre cada ϕ_i, pero no regresamos $\eta_1 \vee \eta_2$ puesto que esta solo es una forma normal si η_i son literales.
4. Se debe distribuir la disyunción sobre la conjunción, garantizando que la disyunciones generadas sean de literales. Una función $distr(\eta_1, \eta_2)$ haría ese trabajo.

El Algoritmo 4.1 muestra el procedimiento principal para convertir una fbf a su equivalente CNF. El primer paso conciste en una fase de preprocesamiento para eliminar implicaciones y posteriormente convertir la fbf resultante a su forma normal bajo negación NNF (Algoritmo 4.2). También es nesario distribuir las conjunciones para obtener CNF donde los argumentos de la disyunción son exclusivamente literales (Algoritmo 4.3). El algoritmo para eliminar la implicación IMPL_FREE, se deja como un ejercicio sugerido.

Algoritmo 4.1 CNF

```
1: function CNF(φ)                                   ▷ φ es una fbf proposicional
2:     ψ ← NNF(IMPL_FREE(φ))
3:     switch ψ do
4:         case literal(ψ)
5:             return ψ
6:         case ψ1 ∧ ψ2
7:             return CNF(ψ1) ∧ CNF(ψ2)
8:         case ψ1 ∨ ψ2
9:             return DISTR(CNF(ψ1), CNF(ψ2))
10: end function
```

Algoritmo 4.2 NNF

```
1: function NNF(φ)                                   ▷ φ es libre de implicaciones
2:     switch φ do
3:         case literal(φ)
4:             return φ
5:         case ¬¬φ1
6:             return φ1
7:         case φ1 ∧ φ2
8:             return NNF(φ1) ∧ NNF(φ2)
9:         case φ1 ∨ φ2
10:            return NNF(φ1) ∨ NNF(φ2)
11:        case ¬(φ1 ∧ φ2)
12:            return NNF(¬φ1) ∨ NNF(¬φ2)
13:        case ¬(φ1 ∨ φ2)
14:            return NNF(¬φ1) ∧ NNF(¬φ2)
15: end function
```

Ejemplo 4.10. *Obtenga la CNF de*$(\neg p \wedge q \rightarrow p \wedge (r \rightarrow q))$. *Una vez implementado el procedimiento de eliminación de la implicación, en mi caso en Prolog, podríamos probarlo:*

```
?- impl_free(~p ^ q => p ^ (r => q), IMPLFREE).
IMPLFREE = ( ~ (~p^q)v p^(~r v q))
```

El resultado es correcto, como se demuestra en la Figura 4.6, que además sugiere la estrategia que IMPL_FREE debería seguir. La forma libre de implicación se le puede pasar a NNF:

```
?- impl_free(~p ^ q => p ^ (r => q), IMPLFREE), nnf(IMPLFREE,NNF).
IMPLFREE = ( ~ (~p^q)v p^(~r v q)),
NNF = (( p v ~q)v p^(~r v q))
```

Algoritmo 4.3 DISTR

```
1: function DISTR(η1, η2)                            ▷ η1 y η2 están en CNF
2:     if η1 = η11 ∧ η12 then
3:         return DISTR(η11, η2) ∧ DISTR(η12, η2)
4:     else if η2 = η21 ∧ η22 then
5:         return DISTR(η1, η21) ∧ DISTR(η1, η22)
6:     else                                          ▷ No hay conjunciones
7:         return η1 ∨ η2
8:     end if
9: end function
```

La Figura 4.7 muestra su ejecución.

$$
\begin{aligned}
\texttt{IMPL\_FREE}\,(\phi) &= \neg\texttt{IMPL\_FREE}\,(\neg p \wedge q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\texttt{IMPL\_FREE}\,\neg p) \wedge (\texttt{IMPL\_FREE}\,q)) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\neg p) \wedge \texttt{IMPL\_FREE}\,q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee \texttt{IMPL\_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee ((\texttt{IMPL\_FREE}\,p) \wedge \texttt{IMPL\_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge \texttt{IMPL\_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg(\texttt{IMPL\_FREE}\,r) \vee (\texttt{IMPL\_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee (\texttt{IMPL\_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}
$$

Figura 4.6: Eliminación de la implicación con IMPL_FREE

$$
\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg(\neg p \wedge q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= \texttt{NNF}\ (\neg(\neg p) \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (\texttt{NNF}\ (\neg\neg p)) \vee (\texttt{NNF}\ (\neg q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee (\texttt{NNF}\ (\neg q))) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee ((\texttt{NNF}\ p) \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge ((\texttt{NNF}\ (\neg r)) \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}
$$

Figura 4.7: La aplicación de NNF para obtener la forna normal bajo negación.

4.6 EL MUNDO DE TARSKI

Mundo de Tarski

Para evaluar la expresividad de la lógica proposicional usaremos el **Mundo de Tarski** [5]. Se trata de un programa que permite representar medios ambientes bidimensionales poblados de figuras geométricas de diferentes tipos y tamaños. La idea es proponer proposiciones, o conjuntos de ellas, y ver si son falsas o verdaderas en un mundo dado. La ventana principal del programa se muestra en la Figura 4.8.

Como pueden observar, la aplicación tiene dos paneles: El superior, con el título momentáneo de *Untitled World* se usa para crear mundos como los descritos; y el inferior, con el título momentáneo de *Untitled Sentences*, se usa para escribir enunciados en lógica proposicional (aunque como veremos, también lo podemos hacer en primer orden), con ayuda del teclado virtual. Este sugiere que las **proposiciones** que podemos formar se refieren al tipo, tamaño y posición relativa de las figuras en el mundo: *Tet*, *Cube*, *Dodec*, *Small*, *Medium*, *Large*, etc. Por ejemplo, la proposición 1 enuncia que la figura a es un tetraedro y la figura b es un cubo y la figura c es un dodecaedro. Observen que, independientemente de su sintaxis, por ejemplo $Tet(a)$, se trata de proposiciones: "a es un tetraedro".

Proposiciones

El teclado virtual también sugiere los **operadores lógicos** que podemos usar. Por el momento, usaremos la conjunción ($\wedge$), la disyunción ($\vee$), la negación ($\neg$), la implicación ($\to$), la equivalencia ($\leftrightarrow$) y la contradicción ($\bot$). También se sugiere que las figuras en el mundo pueden **etiquetarse** como a, b, c, d, e, f. También conta-

Operadores lógicos

Etiquetas

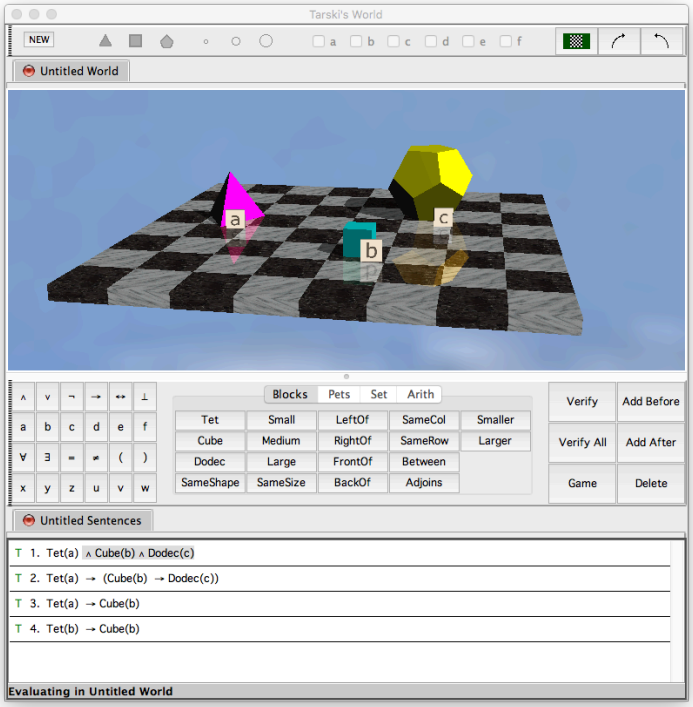


Figura 4.8: Ventana principal del Mundo de Tarski.

mos con paréntesis. El resto de los elementos del teclado, no los usaremos hasta el próximo capítulo.

Tanto el mundo como los enunciados, pueden guardarse en archivos separados para su posterior evaluación. De hecho, la aplicación se instala con una serie de mundos y enunciados que se usarán en los ejercicios propuestos.

4.7 LECTURAS Y EJERCICIOS SUGERIDOS

Este capítulo se basa en las técnicas utilizadas por Huth y Ryan [54] para introducir la lógica proposicional (secciones 1.1 a la 1.4). Como he mencionado, dados los objetivos de este curso; pero sobre todo, dada la duración del mismo, las formas normales de la lógica proposicional y los problemas de satisfacción no han sido abordados aquí. Esta decisión nos permitirá avanzar hacía la lógica de primer orden para abordar temas estrechamente relacionados con estos dos puntos. Para los impacientes, recomiendo las secciones 1.5 y 1.6 del texto en cuestión. Otra referencia al respecto sería el capítulo 7 del libro de Russell y Norvig [95]. Ben-Ari [7] ofrece un texto complementario al de Huth, el material presentado aquí se correspondería con los capítulos 3 y 2 de éste texto. Los capítulos 4,5 y 6 cubren el material que no hemos abordado en nuestro curso. Una presentación más concisa, y por tanto más técnica, de los temas tratados puede encontrarse en el capítulo 1 del libro de Dalen [29]. Por último, estos temas siempre pueden complementarse con la lectura de los fundamentos lógicos de la IA, de Genesereth y Nilsson [43].

Ejercicios sugeridos

Ejercicio 4.1. *Implemente el algoritmo CNF. Para ello es necesario completar las definiciones con el procedimiento para eliminar la implicación.*

5 LA LÓGICA DE PRIMER ORDEN

La lógica proposicional, abordada en el capítulo anterior, nos permite expresar conocimiento sobre situaciones que son de nuestro interés, mediante **enunciados declarativos**. Decimos que estos enunciados son declarativos en el sentido lingüístico del término, esto es, se trata de expresiones del lenguaje natural que son o bien verdaderas, o bien falsas; en contraposición a los enunciados imperativos e interrogativos. La lógica proposicional es declarativa en este sentido, las proposiciones representan **hechos** que se dan o no en la realidad. La lógica de primer orden tienen un compromiso ontólogico más fuerte [95], donde la realidad implica además, **objetos** y **relaciones** entre ellos. Consideren los siguientes ejemplos de enunciados declarativos:

Enunciados declarativos

Hechos

Objetos y relaciones

1. Julia es madre y Luis es hijo de Julia.
2. Toda madre ama a sus hijos.

donde el enunciado (1) se refiere a los objetos de discurso *Julia* y *Luis*, usando propiedades de estos objetos, como ser *madre*; así como relaciones entre éstos, como ser *hijo* de alguien. El enunciado (2) se refiere a relaciones que aplican a todas las madres, en tanto que objetos de discurso. A esto nos referimos cuando hablamos de **representación** de conocimiento en el contexto de la Programación Lógica, a describir una situación en términos de objetos y relaciones entre ellos.

Representación

Al igual que en el caso proposicional, si se aplican ciertas reglas de prueba a tales representaciones, es posible obtener nuevas conclusiones. Por ejemplo, conociendo (1) y (2) es posible **inferir** (vía una versión de *Modus Ponens*, un poco distinta de la proposicional) que:

Inferencia

3. Julia ama a Luis.

De forma que, sería deseable poder describir los objetos que conforman un **universo de discurso**, personas en el ejemplo; así como las relaciones que se dan entre ellos, *hijo* y *madre* en el ejemplo; y computar tales descripciones para obtener conclusiones como (3). Al describir el problema que queremos resolver, también podemos hacer uso de **funciones**, relaciones en las cuales uno o más objetos del universo de discurso se relacionan con un objeto único.

Universo de discurso

Funciones

Ejemplo 5.1. *Por ejemplo, "madre de" puede representarse como una función (todo hijo tiene una sola madre genética), pero "hijo de" no. Esto se ilustra en la Figura 5.1.*

Figura 5.1: La relación *madre de* es una función; mientras que *hijo de* no lo es.

Como en todo **sistema formal**, tal y como lo hicimos con la lógica proposicional, será necesario especificar cuidadosamente los siguientes elementos, que se ilustran en la Figura 5.2:

Sistemas formales

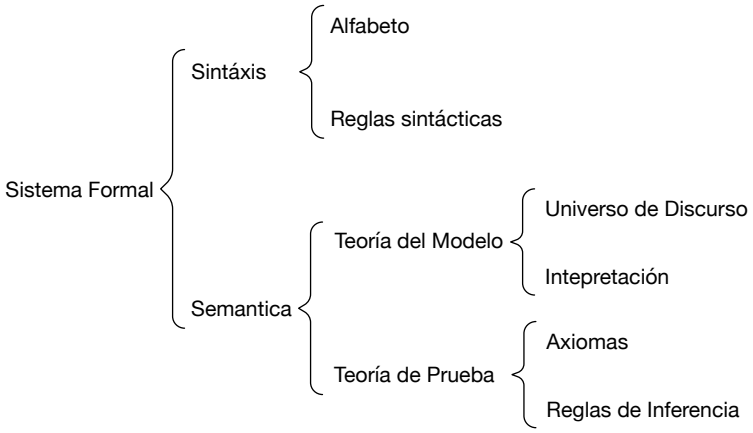


Figura 5.2: Los componentes de un sistema formal.

- **Lenguaje.** La sintaxis del sistema formal está dada por un conjunto de símbolos conocido como **alfabeto** y una serie de **reglas de sintácticas**. Una expresión es cualquier secuencia de símbolos pertenecientes al alfabeto. Cualquier expresión es, o no, una **fórmula bien formada** (fbf). Las fbfs son las expresiones que pueden formarse con los símbolos del alfabeto a partir de las reglas sintácticas.

- **Teoría de prueba.** Este elemento está asociado con el razonamiento deductivo. Su objetivo es hacer de cada enunciado matemático, una fórmula demostrable y rigurosamente deducible. Para ello, la actividad matemática debería quedar reducida a la manipulación de símbolos y sucesiones de símbolos regulada por un conjunto de instrucciones dadas al respecto. Su construcción implica un subconjunto de fbf que tendrán el papel **axiomas** en el sistema, y un conjunto de **reglas de inferencia** que regulen diversas operaciones sobre los axiomas. Las fbf obtenidas mediante la aplicación sucesiva de las reglas de inferencia a partir de los axiomas se conocen como **teoremas** del sistema.

  Axiomas
  Reglas de inferencia
  Teoremas

- **Teoría de modelo.** Establece la interpretación de las fbfs en un sistema formal. Su función es relacionar las fbfs con alguna representación simplificada de la realidad que nos interesa, para establecer cuando una fbf es falsa y cuando verdadera. Esta versión de realidad corresponde a lo que informalmente llamamos "modelo". Sin embargo, en lógica, el significado de "modelo" está íntimamente relacionado con el lenguaje del sistema formal: si la interpretación M hace que la fbf ϕ [1] sea verdadera, se dice que M es un **modelo** de ϕ o que M **satisface** ϕ, y se escribe $M \models \phi$. Una fbf es **válida** si toda interpretación es un modelo para ella.

Este capítulo introduce la lógica de primer orden, que restringiremos sintácticamente para dar lugar a los programas definitivos. Como en el caso de la CNF, tal restricción optimiza la inferencia. En este caso, hace posible el uso de una única regla de inferencia: La **resolución-sld**; tal y como se implementa en el lenguaje de programación Prolog [17, 25].

Resolución

5.1 SINTAXIS

Básicamente, la lógica de primer orden introduce un conjunto de símbolos que nos permiten expresarnos acerca de los **objetos** en un dominio de discurso dado. El conjunto de todos estos objetos se conoce como **Dominio** o **Universo de discurso**

Objetos
Universo de discurso

1 Los símbolos griegos se siguen usando como meta variables.

($\mathcal{U}$). Los miembros del universo de discurso pueden ser objetos concretos, ej., un libro, un robot, etc; abstractos, ej., números; e incluso, ficticios, ej., unicornios, etc. Un objeto es algo sobre lo cual queremos expresarnos. Como ejemplo, consideren el multi citado mundo de los bloques [43] que se muestra en la Figura 5.3. El universo de discurso para tal escenario es el conjunto que incluye los cinco bloques, la el brazo robótico y la mesa: $\mathcal{U} = \{a, b, c, d, e, brazo, mesa\}$.

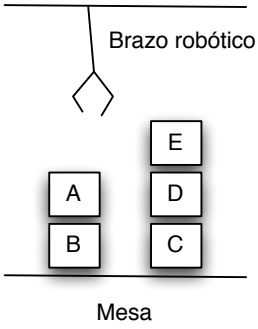


Figura 5.3: *El mundo de los bloques, usado en los ejemplos subsiguientes.*

Funciones

Una **función** es un tipo especial de relación entre los objetos del dominio de discurso. Este tipo de relaciones mapea un conjunto de objetos de entrada a un objeto único de salida. Por ejemplo, es posible definir la función parcial *sombrero* que mapea un bloque al bloque que se encuentra encima de él, si tal bloque existe. Las parejas correspondientes a esta función parcial, dado el escenario mostrado en la Figura 5.3 son: $\{(b, a), (c, d), (d, e)\}$. El conjunto de todas las funciones consideradas en la conceptualización del mundo se conoce como **base funcional**.

Base funcional

Predicados

Un segundo tipo de relación sobre los objetos del dominio de discurso son los **predicados**. Diferentes predicados pueden definirse en el mundo de los bloques, por ejemplo, el predicado *sobre* que se cumple para dos bloques, si y sólo si el primero está inmediatamente encima del segundo. Para la escena mostrada en la Figura 5.3, $sobre/2$ se define por los pares $\{(a, b), (d, c), (e, d)\}$. Otro predicado puede ser $libre/1$, que se cumple para un bloque si y sólo si éste no tiene ningún bloque encima. Este predicado tiene los siguientes elementos $\{a, e\}$. El conjunto de todos los predicados usados en la conceptuación se conoce como **base relacional**.

Base relacional

Cardinalidad

Para universos de discurso finitos, existe un límite superior en el número posible de predicados n-arios que pueden ser definidos. Para un universo de **cardinalidad** b (cardinalidad es el número de elementos de un conjunto), existen b^n distintas n-tuplas. Cualquier predicado n-ario es un subconjunto de estas b^n tuplas. Por lo tanto, un predicado n-ario debe corresponder a uno de máximo $2^{(b^n)}$ conjuntos posibles.

Variables

Cuantificadores

Además de las funciones y predicados, la flexibilidad de la Lógica de primer orden resulta del uso de variables y cuantificadores. Las **variables**, cuyos valores son objetos del universo de discurso, se suelen representar por cualquier secuencia de caracteres que inicie con una mayúscula [2]. El **cuantificador "para todo"** ($\forall$) nos permite expresar hechos acerca de todos los objetos en el universo del discurso, sin necesidad de enumerarlos. Por ejemplo, toda madre ... El **cuantificador "existe"** ($\exists$) nos permite expresar la existencia de un objeto en el universo de discurso con cierta propiedad en particular, por ejemplo, $\exists X\ libre(X) \wedge enLaMesa(X)$ expresa que hay al menos un objeto que no tiene bloques sobre él y aue se encuentra sobre la mesa. Revisemos estos conceptos formalmente.

Alfabeto

El **alfabeto** de la Lógica de primer orden se obtiene al extender la lógica proposicional con un conjunto numerable de símbolos de predicados ($Pred$) y funciones ($Func$). Se asume un conjunto infinito de variables (Var) que toman valores en el

2 En algunos textos encontrarán que las variables se representan como cadenas de texto de inician con minúscula y los símbolos de predicado y funciones, con mayúsculas. He optado por lo opuesto para seguir desde ahora la notación usada en Prolog y Jason.

universo de discurso. $|f|$ denota la **aridad** del predicado o función f, es decir, su número de argumentos. *Aridad*

Los componentes de nuestro lenguaje que nos permiten denotar objetos del universo de discurso se conocen como **términos**; y en la Lógica de primer orden se forman con variables, constantes y funciones aplicadas a arguments, que a su vez son términos. Por ejemplo, $calif(hermano(alex), sma)$ es un término que denota la calificación obtenida por el hermano de Álex en el curso de Sistemas Multi-Agentes, es decir un entero entre cero y cien. Utilizando notación BNF: *Términos*

Definición 5.1 (Términos)**.** *Un término se define como:*

$$t ::= x \mid c \mid f(t, \ldots, t)$$

donde $x \in Var$; $c \in Func$ tal que $|c| = 0$; y $f \in Func$ tal que $|f| > 0$.

Observen que los constructores básicos de los términos son las variables y las funciones. Las funciones de aridad cero se asumen como **constantes**. Se pueden formar términos más complejos usando functores de aridad mayor a cero, cuyos argumentos son a su vez términos. *Constantes*

Definición 5.2 (Sintaxis)**.** *Las fbf del lenguaje de la Lógica de primer orden se construye a partir de las variables Var, los functores $Func$ y los predicados $Pred$ como sigue:*

$$\phi ::= P(t_1, \ldots, t_n) \mid \neg(\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\forall x\ \phi) \mid (\exists x\ \phi)$$

donde $P \in Pred$ es un símbolo de predicado de aridad $n \geq 1$; t_i denota términos sobre $Func$; y $x \in Var$.

Es posible adoptar como fbf a los predicados de aridad cero, que entonces se asumen como **variables proposicionales**. Esto tiene sentido si observamos que las fbf de la lógica de primer orden denotan enunciados declarativos sobre la relaciones que se dan entre los objetos del universo de discurso. De forma que las variables proposicionales son enunciados declarativos donde no nos interesa referirnos explícitamente a los objetos y sus relaciones, pero si posiblemente a su valor de verdad. La Lógica de primer orden subsume a la proposicional. *Variables proposicionales*

Asumiremos que los cuantificadores tienen la misma **precedencia** que la negación. El resto de los operadores tiene la misma precedencia que en la lógica proposicional. Las fbf de la lógica de primer orden pueden representarse como árboles sintácticos. Por ejemplo, el árbol de la fbf $\forall X((p(X) \rightarrow q(X)) \wedge s(X, Y))$ se muestra en la figura 5.4. *Precedencia de los operadores*

5.2 SEMÁNTICA

En la lógica proposicional, una interpretación es una función que asigna valores de verdad a las variables proposicionales de una fbf. En la lógica de primer orden, el concepto análogo debe asignar valores de verdad a las fórmulas atómicas del lenguaje; pero esto involucra normalmente la substitución de variables por objetos en el universo de discurso, de forma que las fbf puedan ser interpretadas como relaciones sobre $\mathcal{U}$ que se satisfacen.

Ejemplo 5.2. *En el escenario del mundo de los bloques que se muestra en la Figura 5.3, si queremos expresar que al menos algún bloque no tiene nada encima, podríamos usar los predicados bloque y libre, con su significado pretendido evidente, en la siguiente expresión: $\exists X\ bloque(X) \wedge libre(X)$. Esta fbf expresa que existe un X tal que X es un bloque y X está libre (no tiene otro bloque encima).*

Cuando usamos cuantificadores, siempre tenemos en mente el universo de discurso, en este caso $\mathcal{U} = \{a, b, c, d, e, mesa, mano\}$. En el caso del predicado $bloque/1$

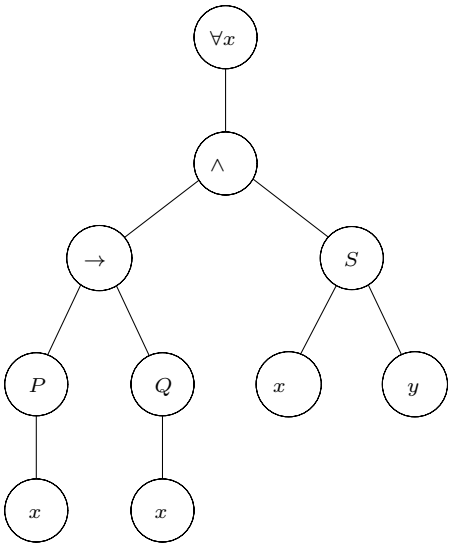


Figura 5.4: El árbol sintáctico de una fbf de la lógica de primer orden.

es de esperar que su **interpretación** sea un subconjunto de $\mathcal{U}$ como es el caso, ya que los objetos del universo de discurso que satisfacen la relación bloque son $\{a, b, c, d, e\} \subset \mathcal{U}$. En general, para un predicado de aridad n, su interpretación será un subconjunto de $\mathcal{U}^n$. Por ejemplo *sobre*/2, en la escena considerada, tiene una interpretación $\{(a, b), (e, d), (d, c)\} \subset \mathcal{U}^2$. Para una función de aridad n la interpretación será un mapeo, posiblemente parcial, $\mathcal{U}^n \mapsto \mathcal{U}$. *Interpretación*

Ejemplo 5.3. *sombrero*/1 *tiene como interpretación* $\{(b) \mapsto a, (d) \mapsto e, (c) \mapsto d\}$, *de forma que podemos computar expresiones como* $sombrero(b) \mapsto a$.

Para definir una interpretación en la lógica de primer orden necesitamos una tupla $\langle D, V \rangle$, donde D es el universo de discurso; y V es una función, tal que para cualquier predicado de aridad n se obtiene el conjunto de las n-tuplas que corresponden a la interpretación del predicado. Para una constante, la función V regresa la misma constante, ej. $V(a) = a$. Algunas veces la expresión $V(\phi)$, se abrevia ϕ^V.

Ejemplo 5.4. *Una posible interpretación para la escena mostrada en al Figura 5.3 y sus bases relacional y funcional es:*

$$\begin{aligned}
a^V &= a \\
b^V &= b \\
c^V &= c \\
d^V &= d \\
e^V &= e \\
sobre^V &= \{(a, b), (e, d), (d, c)\} \\
enLaMesa^V &= \{b, c\} \\
libre^V &= \{a, e\} \\
porEncima^V &= \{(a, b), (e, d), (e, c), (d, c)\} \\
sombrero^V &= \{(b) \mapsto a, (d) \mapsto e, (c) \mapsto d\}
\end{aligned}$$

Todo esto puede especificarse formalmente con la siguiente regla semántica:

Definición 5.3 (Interpretación)**.** *Una interpretación V, con respecto a un dominio de discurso D es una función que satisface las siguientes propiedades: i) Si $\phi \in Const$, entonces $V(\phi) = \phi$; ii) Si $\phi \in Pred$ es de aridad $n > 0$, entonces $V(\phi) \subseteq D^n$; iii) Si $\phi \in Func$ es de aridad $n > 0$, entonces $V(\phi) \subseteq D^n \mapsto D$.*

Variables libres y acotadas

En lo que sigue, se asume que las variables en las fbf están **acotadas**, es decir, bajo el alcance de un cuantificador. En otro caso se dice que la variable es **libre**. Consideren la siguiente fbf cuyo árbol se muestra en la Figura 5.4. Primero, los cuantificadores, al igual que la negación, tienen un solo sub-árbol sintáctico. Segundo, los predicados de aridad n tienen n sub-arboles. En un árbol sintáctico de una fbf las variables ocurren en dos sitios: al lado de un cuantificador y como hojas del árbol. Si subimos por el árbol a partir de las hojas etiquetadas con la variable X, llegaremos al nodo $\forall X$, por tanto decimos que las ocurrencias de X está acotadas por el cuantificador universal, o que X está bajo el **alcance** de $\forall X$; en el caso de la variable Y llegamos al mismo nodo, pero Y no tienen nada que ver con $\forall X$, por lo que decimos que es una variable libre.

Alcance

Definición 5.4 (Variable libre y acotada)**.** *Sea ϕ una fbf en la lógica de primer orden. Una ocurrencia de la variable X en ϕ es libre si X es un nodo hoja en el árbol sintáctico de ϕ tal que no hay caminos hacía arriba del árbol que lleven a un nodo etiquetado como $\forall X$ o $\exists X$. De otra forma se dice que la ocurrencia de la variable es acotada. Para las fbf $\forall X\phi$ y $\exists X\phi$ se dice que ϕ (menos toda sub-fórmula de ϕ, $\forall X\psi$ o $\exists X\psi$) es el alcance del cuantificador $\forall X$ y $\exists X$, respectivamente.*

Fórmulas cerradas

Si una fbf no tiene variables libres, se dice que es una fbf **cerrada**. Si no tiene variables libres, ni acotadas, se dice que es una **fórmula de base**.

Fórmula de base

Asignación de variables

Por razones que serán evidentes más adelante, conviene interpretar las variables de una fbf de forma separada. Una **asignación de variables** es una función de las variables del lenguaje a objetos en el universo de discurso. Decimos que U es una asignación de variables basada en el modelo $M = \langle D, V\rangle$ si para todo $\phi \in Var$, $U(\phi) \in D$. Al igual que en el caso de las interpretaciones ϕ^U es una abreviatura de $U(\phi)$.

Ejemplo 5.5. *En lo que sigue, a la variable X se le asigna el bloque a y a la variable Y el bloque b:*

$$X^U = a$$
$$Y^U = b$$

Asignación de términos

Una interpretación V y una asignación de variables U pueden combinarse en una **asignación de términos**, donde los términos que no son variables son procesados por la interpretación, y las variables por la asignación de variables.

Definición 5.5 (Asignación de términos)**.** *Dadas una interpretación V y una asignación de términos U, la asignación de términos T_{VU} es un mapeo de términos a objetos en el universo de discurso, como sigue:*

1. *Si $\phi \in Const$ entonces $T_{VU}(\phi) = \phi^V$*
2. *Si $\phi \in Var$ entonces $T_{VU}(\phi) = \phi^U$*
3. *Si ϕ es un término de la forma $\phi(\tau_1, \ldots, \tau_n)$; y $\phi^V = g$, mientras que $T_{VU}(\tau_i) = x_i$, entonces $T_{VU}(\phi) = g(x_1, \ldots, x_n)$.*

Satisfacción

Los conceptos de interpretación y asignación de variables son importantes porque nos permiten definir una noción relativa de verdad llamada **satisfacción**. El hecho de que la fbf ϕ se satisfaga en una interpretación V y una asignación de variables U se denota como $\models_I \phi[U]$. Si ese es el caso, se dice que ϕ es **verdadera** en relación a la interpretación V y la asignación de variables U. Como suele ser el caso, la definición de la relación de satisfacción depende de la forma de ϕ, así que se define por casos:

Definición 5.6 (Satisfacción)**.** *Dada una asignación de términos T_{VU} y un modelo $M = \langle D, V\rangle$ la satisfacción se define como sigue:*

1. $\models_V (\phi = \psi)[U]$ *si y sólo si* $T_{VU}(\phi) = T_{VU}(\psi)$.

2. $\models_V \phi(\tau_1, \ldots, \tau_n)[U]$ *si y sólo si* $(T_{VU}(\tau_1), \ldots, T_{VU}(\tau_n)) \in \phi^V$.
3. $\models_V (\neg\phi)[U]$ *si y sólo si* $\not\models_V \phi[U]$.
4. $\models_V (\phi \wedge \psi)[U]$ *si y sólo si* $\models_V \phi[U]$ *y* $\models_V \psi[U]$.
5. $\models_V (\phi \vee \psi)[U]$ *si y sólo si* $\models_V \phi[U]$ *o* $\models_V \psi[U]$.
6. $\models_V (\phi \rightarrow \psi)[U]$ *si y sólo si* $\not\models_V \phi[U]$ *o* $\models_V \psi[U]$.
7. $\models_V (\forall v\phi)[U]$ *si y sólo si para todo* $d \in D$ *es el caso que* $\models_V \phi[W]$, *donde* $v^W = d$ *y* $\mu^W = \mu^U$ *para* $\mu \neq v$.
8. $\models_V (\exists v\phi)[U]$ *si y sólo si para algún* $d \in D$ *es el caso que* $\models_V \phi[W]$, *donde* $v^W = d$ *y* $\mu^W = \mu^U$ *para* $\mu \neq v$.

La igualdad de dos términos (1) se satisface si ambos denotan al mismo objeto bajo la asignación de términos. Una fbf atómica se satisface (2) si la tupla de sus argumentos bajo la asignación de términos está en la interpretación de su predicado. La regla de satisfacción para fbf cuantificadas universalmente (7) requieren que la fórmula ϕ se satisfaga para todas las asignaciones posibles de la variable cuantificada v. Las asignaciones de variables que son idénticas para todas las variables, excepto posiblemente para v, se dicen **v-alternativas**. La regla de satisfacción para fbf cuantificadas existencialmente (8) requieren que la fórmula ϕ se satisfaga en al menos una asignación posible de la variable cuantificada v. Las reglas de satisfacción para los operadores lógicos (3-6) son muy similares a los de la lógica proposicional.

Asignaciones alternativas

Como recordaran, de una interpretación V que satisface una fbf ϕ para toda asignación de variables, se dice que es un **modelo** de ϕ; lo cual se denota por $\models_V \phi$. Por ejemplo, siguiendo con la escena del mundo de los cubos: $\models_V enLaMesa(X) \vee \neg enLaMesa(X)$. Observen que la asignación de variables no tiene efectos en la satisfacción de una fbf cerrada o de base. Por consiguiente, toda interpretación que satisface una de estas fórmulas para una asignación de variables, es un modelo de la fórmula.

Modelo

Una fbf se dice **satisfacible** si y sólo si existe alguna interpretación y asignación de variables que la satisfacen. De otra forma se dice que la fórmula es insatisfacible. Una fbf es **válida** si y sólo si se satisface en toda interpretación y asignación de variables. Al igual que en el caso proposicional, las fórmulas válidas lo son en virtud de su estructura lógica y por tanto, no proveen información sobre el dominio descrito.

Fórmula satisfacible

Fórmula válida

Ejemplo 5.6. $p(X) \vee \neg p(X)$ *es una fbf válida.*

5.2.1 Inferencia

Volvamos al ejemplo de la introducción:

1. Toda madre ama a sus hijos.
2. Julia es madre y Luis es hijo de Julia.

Conociendo (1) y (2) es posible concluir que:

3. Julia ama a Luis.

Podemos formalizar este ejemplo en Lógica de Primer Orden como sigue:

1. $\forall X\ \forall Y\ madre(X) \wedge hijoDe(Y, X) \rightarrow ama(X, Y)$
2. $madre(julia) \wedge hijoDe(luis, julia)$
3. $ama(julia, luis)$

Una vez que hemos formalizado nuestros enunciados, el proceso de inferencia puede verse como un proceso de manipulación de fbf, de la misma naturaleza que la selección natural, donde a partir de fbfs como (1) y (2), llamadas **premisas**, se produce la nueva fbf (3) llamada **conclusión**. Estas manipulaciones se pueden formalizar mediante **reglas de inferencia**. Al igual que en la lógica proposicional, es importante que en una consecuencia, la conclusión se siga de las premisas y las reglas de inferencia adoptadas. Entre las reglas de inferencia de la lógica de primer orden encontramos:

- **Modus Ponens.** O regla de eliminación de la implicación. Esta regla dice que siempre que las fbfs de la forma ϕ y $\phi \rightarrow \psi$ pertenezcan a las premisas o sean concluidas a partir de ellas, podemos inferir ψ:

$$\frac{\phi \quad \phi \rightarrow \psi}{\psi} \quad (\rightarrow E)$$

- **Eliminación de cuantificador universal.** Esta regla expresa que siempre que una fbf de la forma $\forall X \phi$ pertenezca a las premisas o sea concluida a partir de ellas, una nueva fbf puede ser concluida al remplazar todas las ocurrencias libres de X en ϕ por algún término t que es libre con respecto a X (todas las variables en t quedan libres al substituir X por t. La regla se presenta como sigue:

$$\frac{\forall X \phi(X)}{\phi(t)} \quad (\forall E)$$

- **Introducción de conjunción.** Cuando las fbf ϕ y ψ pertenezcan a las premisas o sean concluidas a partir de ellas, podemos inferir $\phi \wedge \psi$:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \quad (\wedge I)$$

La correctez de estas reglas puede ser demostrada directamente a partir de la definición de la semántica de las fbf del lenguaje. El uso de las reglas de inferencia puede ilustrarse con el ejemplo formalizado. Las premisas son:

1. $\forall X \forall Y madre(X) \wedge hijoDe(Y, X) \rightarrow ama(X, Y)$
2. $madre(julia) \wedge hijoDe(luis, julia)$

Al aplicar la eliminación de cuantificador universal $(\forall X)$ a (1) obtenemos:

3. $\forall Y (madre(julia) \wedge hijoDe(Y, julia) \rightarrow ama(julia, Y)$

Al aplicar nuevamente $(\forall E)$ a (3) obtenemos:

4. $madre(julia) \wedge hijoDe(luis, julia) \rightarrow ama(julia, luis)$

Finalmente, al aplicar Modus Ponens a (2) y (4):

5. $ama(julia, luis)$

La conclusión (5) ha sido obtenida rigurosamente, aplicando las reglas de inferencia. Esto ilustra el concepto de **derivación**. El hecho de que una fórmula ϕ sea derivable a partir de un conjunto de fórmulas Δ se escribe $\Delta \vdash \phi$. Si las reglas de inferencia son sólidas (en el sentido técnico de *soundness*), siempre que $\Delta \vdash \phi$ entonces $\Delta \models \phi$. Esto es, si nuestra lógica es sólida, cualquier fbf que puede ser derivada de otra fbf, es tambien una consecuencia lógica de ésta última.

Derivación

5.3 PROGRAMAS DEFINITIVOS

La idea central de la Programación Lógica es usar la computadora para obtener conclusiones a partir de descripciones declarativas, como las introducidas en el capítulo anterior. Estas descripciones, llamadas **programas lógicos**, consisten en un conjunto finito de fórmulas bien formadas (fbfs) de la lógica de primer orden. Esta idea tiene sus raíces en la demostración automática de teoremas, sin embargo, pasar de la demostración automática de teoremas experimental a la programación lógica aplicada, requiere mejoras con respecto a la eficiencia del sistema propuesto. Tales mejoras se logran imponiendo restricciones sobre las fbfs del lenguaje utilizado, de forma que podamos usar una poderosa regla de inferencia conocida como principio de **resolución-SLD**. Este capítulo introduce el concepto de cláusula y programa lógico definitivos. Más adelante se introducirá el concepto menos restrictivo de programas generales, pero el paso por los programas definitivos es necesario para comprender las bases teóricas de la programación lógica, que incluye lenguajes como Prolog y el razonamiento epistémico de Jason. El aparato técnico aquí presentado se basa principalmente en el texto de Nilsson y Maluszynski [77].

Programas lógicos

Resolución-SLD

5.3.1 Cláusulas definitivas

Consideremos una clase especial de enunciados declarativos del lenguaje natural, que utilizamos para describir hechos y reglas positivos. Un enunciado de este tipo puede especificar:

- Que una relación se mantiene entre elementos del universo de discurso (hechos).
- Que una relación se mantiene entre elementos del universo de discurso, si otras relaciones se mantienen (reglas).

Consideren los siguientes enunciados en lenguaje natural:

1. Antonio es hijo de Juan.
2. Ana es hija de Antonio.
3. Juan es hijo de Marcos.
4. Alicia es hija de Juan.
5. El nieto de una persona es el hijo del hijo de esa persona.

Estos enunciados pueden formalizarse en dos pasos. Primero, procedemos con las fbf atómicas que describen hechos:

1. hijo_de(antonio, juan)
2. hijo_de(ana,antonio)
3. hijo_de(juan,marcos)
4. hijo_de(alicia,juan)

El último enunciado puede aproximarse como: Para toda X e Y, X es nieto de Y si existe alguna Z tal que Z es hijo de Y y X es hijo de Z. En lógica de primer orden, esto se escribiría [3]:

$$\forall X \forall Y (nieto\_de(X, Y) \leftarrow \exists Z (hijo\_de(Z, Y) \land hijo\_de(X, Z)))$$

3 Observen que la implicación está invertida ($\leftarrow$) a la usanza de Prolog.

Usando las equivalencias de la lógica de primer orden (en particular $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$; y la equivalencia entre cuantificadores $\forall X\phi \equiv \neg\exists X\neg\phi$), esta fbf puede escribirse de diversas maneras:

$$\forall X\forall Y(nieto\_de(X,Y) \vee \neg\exists Z(hijo\_de(Z,Y) \wedge hijo\_de(X,Z)))$$
$$\forall X\forall Y(nieto\_de(X,Y) \vee \forall Z\neg(hijo\_de(Z,Y) \wedge hijo\_de(X,Z)))$$
$$\forall X\forall Y\forall Z(nieto\_de(X,Y) \vee \neg(hijo\_de(Z,Y) \wedge hijo\_de(X,Z)))$$
$$\forall X\forall Y\forall Z(nieto\_de(X,Y) \leftarrow (hijo\_de(Z,Y) \wedge hijo\_de(X,Z)))$$

Observen que estas fbf están cerradas (no contienen variables fuera del alcance de los cuantificadores) bajo el cuantificador universal. Además, la regla tiene la siguiente estructura:

$$\phi_0 \leftarrow \phi_1 \wedge \cdots \wedge \phi_n \quad (n \geq 0)$$

Los bloques de construcción ϕ_i de estas fbf, se conocen como literales.

Definición 5.7 (Literal)**.** *Una literal es un átomo o la negación de un átomo. Una literal positiva es un átomo. Una literal negativa es la negación de un átomo.*

Un ejemplo de literal positiva es $hijo\_de(juan, marcos)$. Un ejemplo de literal negativa es $\neg hijo\_de(juan, alicia)$. Si p y q son predicados y f es un functor, entonces $p(X, alicia)$ y $q(Y)$ son literales positivas. $\neg q(alicia, f(Y))$ es una literal negativa.

Definición 5.8 (Cláusula)**.** *Una cláusula es una disyunción finita de cero o más literales.*

Definición 5.9 (Cláusula definitiva)**.** *Una cláusula se dice definitiva, si tiene exactamente una literal positiva.*

$$\phi_0 \vee \neg\phi_1 \vee \cdots \vee \neg\phi_n \quad (n \geq 0)$$

lo cual es equivalente a la forma general de fbf que nos interesaba:

$$\phi_0 \leftarrow \phi_1 \wedge \cdots \wedge \phi_n \quad (n \geq 0)$$

Si $n = 0$, tenemos por definición que la literal ϕ_0 será una literal positiva, por lo que la cláusula definitiva toma la forma de un **hecho**. El cuerpo vacío puede representarse por el conectivo nulo ■, que es verdadero en toda interpretación (por simetría también se asume un conectivo nulo □, que es falso en toda interpretación). Si $n > 0$ la cláusula definitiva toma la forma de una **regla**, donde ϕ_0 se conoce como **cabeza** de la regla; y la conjunción $\phi_1 \wedge \cdots \wedge \phi_n$ se conoce como **cuerpo** de la regla.

Hecho

Regla

El ejemplo de la relación $nieto\_de/2$ y la regla que lo define, muestra que las cláusulas definitivas usan una forma restringida de cuantificación existencial, las variables que ocurren sólo en el cuerpo de la cláusula están cuantificadas existencialmente en el cuerpo de la cláusula (el mismo ejemplo muestra que esto equivale a que tales variables estén cuantificadas universalmente sobre toda la fbf).

5.3.2 Programas definitivos y Metas

La definición de programa definitivo es ahora directa:

Definición 5.10 (Programa definitivo)**.** *Un programa definitivo es un conjunto finito de cláusulas definitivas.*

Si una cláusula tiene sólo literales negativas, estamos hablando de una meta definitiva:

Definición 5.11 (Meta definitiva)**.** *Una cláusula sin literales positivas es una meta definitiva.*

$$\leftarrow \phi_1 \wedge \cdots \wedge \phi_n \quad (n \geq 1)$$

Definición 5.12 (Cláusula de Horn)**.** *Una cláusula de Horn es una cláusula definitiva ó una meta definitiva.*

Observen que a partir de estas definiciones, la cláusula vacía $\square$ [4] es una meta definitiva y, por lo tanto, una cláusula de Horn.

Adoptar a las cláusulas de Horn para abordar los programas y metas definitivos, constituye una restricción. Por ejemplo, no podemos expresar $p(a) \vee p(b)$. Esta perdida en expresividad se ve compensada por la ganancia en tratabilidad. Debido a su estructura restringida, las cláusulas de Horn son más fáciles de manipular que las cláusulas generales. En particular, esto es cierto para la deducción basada en resolución-SLD, que resulta completa para las cláusulas de Horn.

El significado lógico de las metas puede explicarse haciéndo referencia a la fbf equivalente cuantificada universalmente:

$$\forall X_1 \dots X_m \neg(\phi_1 \wedge \dots \wedge \phi_n)$$

donde las X_i son todas variables que ocurren en la meta. Esto es equivalente a:

$$\neg\exists X_1 \dots X_m(\phi_1 \wedge \dots \wedge \phi_n)$$

Esto puede verse como una pregunta existencial que el sistema tratará de negar, mediante la construcción de un contra ejemplo. Esto es, el sistema tratará de encontrar términos $t_i \dots t_m$ tales que las fbf obtenidas a partir de $\phi_1 \wedge \dots \wedge \phi_m$ al remplazar la variable X_i por t_i $(1 \leq i \leq m)$ son verdaderas en todo modelo del programa. Es decir, el sistema construirá una consecuencia lógica del programa que es un caso de la conjunción de todas las submetas de la meta.

Al dar una meta definitiva, el usuario selecciona un conjunto de conclusiones a ser construídas. Este conjunto puede ser finito o infinito. El problema de como construir tal conjunto lo veremos al tratar la resolución SLD.

Ejemplo 5.7. *Tomando en cuenta los hechos y reglas sobre una familia presentados al principio de esta sesión, el usuario podría estar interesado en las siguientes consultas (se muestra también la meta definitiva correspondiente):*

Consulta	*Meta definitiva*
¿Es Ana hija de Antonio?	$\leftarrow hijo(ana, antonio)$
¿Quién es nieto de Ana?	$\leftarrow nieto(X, ana)$
¿De quién es nieto Antonio?	$\leftarrow nieto(antonio, X)$
¿Quién es nieto de quién?	$\leftarrow nieto(X, Y)$

Las respuestas obtenidas serían:

- *Puesto que la primer meta no contiene variables, la respuesta sería Si (Yes).*
- *Puesto que el programa no contiene información sobre los nietos de Ana, la respueta a la segunda consulta es No (o ninguno).*
- *Puesto que Antonio es nieto de Marcos, la respuesta obtenida sería* $X = marcos$.
- *La consulta final obtiene tres respuestas:* $X = antonio$ $Y = alicia$, $X = alicia$ $Y = marcos$, $X = ana$ $Y = juan$.

Es posible hacer consultas más elaboradas como ¿Hay alguna persona cuyos nietos son Antonio y Alicia?

$$\leftarrow nieto(antonio, X) \wedge nieto(alicia, X)$$

cuya respuesta esperada es $X = marcos$.

5.3.3 El modelo mínimo de Herbrand

Conocimiento positivo

Los programas definitivos solo pueden expresar **conocimiento positivo**, tanto los hechos, como las reglas, nos dicen que elementos de una estructura están en una relación, pero no nos dicen cuales no. Por lo tanto, al usar el lenguaje de los programas definitivos, no es posible construir descripciones contradictorias, es decir, conjuntos de fbf no satisfacibles. En otras palabras, todo programa definitivo tiene un modelo. Veremos que todo programa definitivo tiene un **modelo mínimo** bien definido, que refleja toda la información expresada por el programa y nada más que eso. Recordemos que una interpretación que hace verdadera una fbf es su modelo.

Modelo mínimo

Existe una clase interesante de modelos, llamados de Herbrand en honor del francés Jacques Herbrand. En esta sección estudiaremos algunas propiedades de los modelos de Herbrand que explican porque son útiles y necesarios en el contexto de la Programación Lógica. Además, constataremos que los modelos de Herbrand proveen una semántica natural para los programas definitivos. Comenzaremos definiendo el **Universo y la Base de Herbrand**:

Universo y Base de Herbrand

Definición 5.13 (Universo y Base de Herbrand)**.** *Sea L un alfabeto que contiene al menos un símbolo de constante ($|Const| \geq 1$). El Universo de Herbrand U_L es el conjunto de todos los términos formados con las constantes y functores de L. La Base de Herbrand B_L es el conjunto de todos los átomos que pueden formarse con los predicados y los términos en el Universo de Herbrand U_L.*

El universo y la base de Herbrand se definen normalmente para un programa dado. En ese caso, se asume que el alfabeto L consiste exactamente de aquellos símbolos que aparecen en el programa. Se asume también que el programa tiene al menos una constante (de otra forma el dominio estaría vacío).

Ejemplo 5.8. *Consideren el siguiente programa definitivo:*

$$\Delta = \{impar(s(0)), impar(s(s(X))) \leftarrow impar(X)\}$$

Si restringimos el lenguaje L a los símbolos que aparecen en este programa definitivo, tenemos que el universo de Herbrand es:

$$U_L = \{0, s(0), s(s(0)), s(s(s(0))), \dots\}$$

Puesto que el programa sólo incluye al predicado $impar$, la base de Herbrand se define como:

$$B_L = \{impar(0), impar(s(0)), impar(s(s(0))), \dots\}$$

Ejemplo 5.9. *Consideren este otro programa $\Delta = \{p(a), q(a, f(b), q(X, X) \leftarrow p(X)\}$. Sea L es lenguaje de primer orden dado por los símbolos en Δ. El Universo de Herbrand U_L es el conjunto infinito:*

$$U_L = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \dots\}$$

Y la base de Herbrand es:

$$B_L = \{p(a), p(b), q(a, b), p(f(a)), p(f(b)), q(a, f(a)), q(a, f(b)), \dots\}$$

Lo que hace especial a una intrepretación de Herbrand es que se toma el conjunto de todos los términos sin variables (U_L) como el dominio de la interpretación; y la interpretación de todo término de base, es el término mismo.

Definición 5.14 (Interpretación de Herbrand)**.** *Una interpretación de Herbrand V de un programa definitivo Δ, es una interpretación donde:*

4 En realidad, la cláusula vacía tiene la forma $\square \leftarrow \blacksquare$ que equivale a $\square$ (Recuerden la tabla de verdad de la implicación). En algunos textos pueden encontrar esta expresión denotada por $\bot \leftarrow \top$.

- *El dominio de V es U_Δ.*
- *Para cada constante $c \in \Delta$, $c^V = c$.*
- *Para cada functor $f/n \in \Delta$, se tiene un mapeo f^V de U_L^n a U_L definido por f^V.*
- *Para cada predicado $p/n \in \Delta$, $p^V \subseteq U_\Delta^n$.*

Definición 5.15 (Modelo de Herbrand)**.** *Un modelo de Herbrand de un conjunto de fbf (cerradas) Δ es una interpertación de Herbrand que es un modelo para todas las fbf en Δ.*

Observen que una interpretación de Herbrand V está completamente especificada por el conjunto de todas las $\phi \in B_\Delta$ que son verdaderas bajo V. En otras palabras, una interpretación de Herbrand, es un subconjunto de la Base de Herbrand $\phi \in B_\Delta | \models_V \phi$.

Ejemplo 5.10. *Consideren el programa Δ en el ejemplo 5.8. Una posible interpretación de este programa es $impar^V = \{s(0), s(s(s(0))), \dots\}$. Una intepretación de Herbrand se puede especificar mediante una familia de tales relaciones (una por cada símbolo de predicado).*

Ejemplo 5.11. *Consideren ahora algunas interpretaciones de Herbrand de Δ tal y como se definió en el ejemplo 5.9:*

$$\begin{aligned} V_1 &= \{p(a), p(b), q(a,b,), q(b,b)\} \\ V_2 &= \{p(a), q(a,a), q(a,f(b))\} \\ V_3 &= \{p(f(f(a))), p(b), q(a,a), q(a,f(b))\} \\ V_4 &= \{p(a), p(b), q(a,a), q(b,b), q(a,f(b))\} \end{aligned}$$

V_2 y V_4 son modelos de Herbrand de $\Delta = \{p(a), q(a,f(b), q(X,X) \leftarrow p(X)\}$. V_1 y V_3 no lo son.

Resultados concernientes a los modelos de Herbrand

Las interpretaciones y los modelos de Herbrand tienen dos propiedades atractivas. La primera es pragmática: para poder determinar si una interpretación de Herbrand V es un modelo de una fbf cuantificada universalmente $\forall\phi$, es suficiente verificar si ϕ es verdadera en V, para todas las asignaciones posibles de las variables de ϕ.

La segunda razón para considerar las interpretaciones de Herbrand es más teórica. Para el lenguaje restringido de cláusulas definitivas, si queremos verificar que una fbf atómica ϕ es consecuencia de un programa definitivo Δ basta con verificar que todo modelo de Herbrand de Δ es también un modelo de Herbrand de ϕ.

Teorema 5.1. *Sea Δ un programa definitivo y γ una meta definitiva. Si V' es un modelo de $\Delta \cup \{\gamma\}$, entonces $V = \{\phi \in B_\Delta | \models_{V'} \phi\}$ es un modelo de Herbrand de $\Delta \cup \{\gamma\}$.*

Claramente V es una interpretación de Herbrand, bajo las definiciones precedentes. Ahora, asumanos que V no es un modelo de $\Delta \cup \{\gamma\}$. Es decir, existe un caso de base de una cláusula en $\Delta \cup \{\gamma\}$, que no es verdadera en V. Puesto que la cláusula es falsa, su cuerpo es verdadero y su cabeza falsa en V'. Pero V' es un modelo de $\Delta \cup \{\gamma\}$, por lo que, por contradicción V es un modelo $\Delta \cup \{\gamma\}$.

Ahora bien, observen que este teorema es válido sólo para conjuntos de cláusulas definitivas. En el caso general, la no existencia del modelo de Herbrand es insuficiente para probar que Δ es no satisfacibe. Por ejemplo, consideren a L como un lenguaje de primer orden formado por los símbolos en $\Delta = \{\exists X p(X), \neg p(a)\}$. Δ tiene un modelo, pero este no es un modelo de Herbrand (Una interpretación posible es aquella cuyo significado pretendido es que o no es impar y existe un número natural que lo es, p. ej., 1).

Hemos mencionado la importancia del concepto de consecuencia lógica. Es común que a partir de un conjunto Δ y una fbf ϕ, queremos encontrar si $\Delta \models \phi$. Esto es cierto si cada modelo de Δ es también un modelo de ϕ. Lo interesante viene ahora:

Proposición 5.1. *Sea Δ un programa definitivo y ϕ una cláusula definitiva. Sea $\Delta' = \Delta \cup \{\neg\phi\}$. Entonces $\Delta \models \phi$ si y sólo si Δ' no tiene modelo de Herbrand.*

La prueba de esta proposición es como sigue: $\Delta \models \phi$ si y sólo si $\Delta \cup \{\neg\phi\}$ no es satisfacible. Esto es cierto sólo si Δ no tiene modelos y por lo tanto, no tiene modelo de Herbrand.

Lo que esta proposición nos dice es que si queremos probar que $\Delta \models \phi$, sólo debemos considerar modelos de Herbrand de la forma V. Aunque el número de interpretaciones de Herbrand es normalmente infinito, la tarea de investigar interpretaciones de Herbrand es más tratable que la de investigar cualquier interpretación arbitraria, puesto que nos restringimos a un dominio único definido por el Universo de Herbrand U_Δ.

Observen que la base de Herbrand de un programa definitivo Δ es siempre un modelo de Herbrand del programa. Sin embargo, es un modelo nada interesante, esto es, cada predicado n-ario en el programa es interpretado como la relación n-aria completa sobre el dominio de los terminos cerrados. ¿Qué es lo que hace a un modelo de programa interesante? En lo que sigue demostraremos la existencia de un modelo mínimo único, llamado el **modelo mínimo de Herbrand** de un programa definitivo. Luego mostraremos que este modelo contiene toda la información positiva presente en el programa.

Modelo mínimo de Herbrand

Los modelos de Herbrand de un programa definitivo son subconjuntos de su base de Herbrand. Por lo tanto, la inclusión en conjuntos establece un orden natural en tales modelos. Para poder demostrar la existencia de modelos mínimos con respecto a la inclusión es suficiente demostrar que la intersección de todos los modelos de Herbrand es también un modelo de Herbrand.

Teorema 5.2 (Intersección de modelos)**.** *Sea M una familia no vacía de modelos de Herbrand de un programa definitivo Δ. Entonces la intersección $V = \bigcap M$ es un modelo de Herbrand de Δ.*

La demostración es como sigue: Supongamos que V no es un modelo de Δ. Por lo tanto existe una una cláusula de base en Δ, de la forma:

$$\phi_0 \leftarrow \phi_1, \ldots, \phi_n \quad (n \geq 0)$$

que no es verdera en V. Esto ímplica que V contiene a $\phi_1, \ldots \phi_n$, pero no a ϕ_0. Luego, $\phi_1, \ldots, \phi_n$ son miembros de toda interpretación en la familia M. Más importante aún, debe existir un modelo $V_i \in M$ tal que $\phi_0 \notin V_i$, de forma que la cláusula $\phi_0 \leftarrow \phi_1, \ldots, \phi_n (n \geq 0)$ no es verdadera en ese V_i. Por lo tanto V_i no es un modelo del programa Δ, lo que contradice nuestro supuesto.

Al tomar la intersección de todos los modelos de Herbrand (se sabe que todo programa definitivo tiene un modelo de Herbrand B_L) de un programa definitivo, obtenemos el modelo mínimo de Herbrand el programa.

Ejemplo 5.12. *Sea Δ el programa definitivo $\{masculino(adan), femenino(eva)\}$ con su interpretación obvia. Δ tiene los siguientes modelos de Herbrand:*

- $\{masculino(adan), femenino(eva)\}$
- $\{masculino(adan), masculino(eva), femenino(eva)\}$
- $\{masculino(adan), masculino(eva), femenino(adan)\}$
- $\{masculino(adan), masculino(eva), femenino(eva), femenino(adan)\}$

No es complicado confirmar que la intersección de estos modelos produce un modelo de Herbrand. Coincidentemente, todos los modelos salvo el primero, contienen átomos incompatibles con el significado esperado del programa. Este ejemplo nos muestra la conexión entre los modelos mínimos de Herbrand y el modelo intentado de un programa definitivo. Este modelo es una abstracción del mundo a ser descrita por el programa. El mundo puede ser más rico que el modelo mínimo de Herbrand. Por ejemplo hay más $femenino$s que *eva*. Sin embargo, aquella información que no se provea explícitamente (hechos) o implícitamente (reglas) no puede ser obtenida como respuesta a una meta. Las respuestas corresponden a las consecuencias lógicas del programa.

Teorema 5.3. *El modelo mínimo de Herbrand M_Δ de un programa definitivo Δ es el conjunto de todas las consecuencias lógicas atómicas de base del programa. Esto es: $M_\Delta = \{\phi \in B_\Delta \mid \Delta \models \phi\}$.*

La prueba de este teorema pasa por demostrar que $M_\Delta \supseteq \{\phi \in B_L | \Delta \models \phi\}$: Es fácil ver que todo átomo de base que es consecuencia lógica de Δ es un elemento de M_Δ. De hecho, por la definición de consecuencia lógica $\phi \in M_\Delta$. Por otra parte, la definición de modelo de Herbrand dice que ϕ es verdadera en M_Δ ssi $\phi \in M_\Delta$; y que $M_\Delta \subseteq \{\phi \in B_\Delta | \Delta \models \phi\}$: Supongamos que $\phi \in M_\Delta$, entonces es verdadera en todo modelo de Herbrand de Δ. Supongamos que no es verdadera en I', un modelo de Δ que no es de Herbrand. Sabemos que el conjunto I de todos los átomos que son verdad en I' es un modelo de Herbrad de Δ. Por lo tanto $\phi \notin I$, lo cual contradice el supuesto de que existe un modelo de Δ donde ϕ es falsa. Entonces ϕ es verdadera en todo modelo de Δ, i.e., $\Delta \models \phi$. □

Construcción del modelo mínimo de Herbrand

¿Cómo podemos construir el modelo mínimo de Herbrand? o ¿Cómo puede aproximarse sucesivamente por medio de la enumeración de sus elementos? La respuesta a estas preguntas se da mediante una aproximación de punto fijo [5] a la semántica de los programas definitivos.

Un programa definitivo está compuesto de hechos y reglas. Es evidente que todos los hechos deben incluirse en cualquier modelo de Herbrand. Si la interpretación V no incluye el hecho ϕ del programa Δ, entonces V no es un modelo de Herbrand de Δ.

Ahora consideremos una regla de la forma $\phi_0 \leftarrow \phi_1, \ldots, \phi_n (n > 0)$. La regla especifica que siempre que $\phi_1, \ldots, \phi_n$ son verdaderas, también lo es ϕ_0. Esto es, tomando cualquier asignacion de valores θ que haga que la regla no tenga variables sin valor $(\phi_0 \leftarrow \phi_1, \ldots, \phi_n)\theta$. Si la interpretación V incluye a $\phi_1\theta, \ldots \phi_n\theta$, deberá incluir también a $\phi_0\theta$ para ser un modelo.

Consideren ahora el conjunto V_1 de todos los hechos sin variables de el programa. Es posible utilizar cada regla para aumentar V_1 con nuevos elementos que necesariamente pertenencen a todo modelo. De modo que se obtiene un nuevo conjunto V_2 que puede usarse para generar más elementos que pertenecen a todo modelo. El proceso se repite mientras puedan generarse nuevos elementos. Los elementos agregados a V_{i+1} son aquellos que se siguen inmediatamente de V_i.

La construcción así obtenida puede formalizarse como la iteración de una transformación T_Δ sobre las interpretaciones de Herbrand de un programa Δ. La operación se llama **operador de consecuencia inmediata** y se define como sigue:

Operador de consecuencia inmediata

Definición 5.16 (Operador de consecuencia inmediata)**.** *Sea base(Δ) el conjunto de todas las cláusulas de base en Δ. T_Δ es una función sobre las interpretaciones de Herbrand de Δ definida como sigue:*

$$T_\Delta(V) = \{\phi_0 \mid \phi_0 \leftarrow \phi_1, \ldots, \phi_n \in ground(\Delta) \wedge \{\phi_1, \ldots, \phi_n\} \subseteq V\}$$

5 El punto fijo de una función $f : D \to D$ es un elemento $x \in D$ tal que $f(x) = x$.

Para los programas definitivos, se puede mostrar que existe una interpretación mínima V tal que $T_\Delta(V) = V$ y que V es idéntica al modelo mínimo de Herbrand de Δ. Más aún, el modelo mínimo de Herbrand es el limite de la creciente, posiblemente infinita, secuencia de iteraciones:

$$\emptyset, T_\Delta(\emptyset), T_\Delta(T_\Delta(\emptyset)), \ldots$$

Existe una notación estándar para denotar a los miembros de esta secuencia de interpretaciones construídas a partir de Δ:

$$\begin{aligned} T_\Delta \uparrow 0 &= \emptyset \\ T_\Delta \uparrow (i+1) &= T_\Delta(T_\Delta \uparrow i) \\ T_\Delta \uparrow n &= \bigcup_{i=0}^{n} T_\Delta \uparrow i \end{aligned}$$

Ejemplo 5.13. *Tomando Δ como el programa de impar (ej. 5.8, tenemos:*

$$\begin{aligned} T_\Delta \uparrow 0 &= \emptyset \\ T_\Delta \uparrow 1 &= \{impar(s(0))\} \\ T_\Delta \uparrow 2 &= \{impar(s(0)), impar(s(s(s(0))))\} \\ &\vdots \\ T_\Delta \uparrow m &= \{impar(s^n(0)) \mid n \in \{1,3,5,\ldots\}\} \end{aligned}$$

Como mencionamos, el conjunto construído de esta manera es identico al modelo mínimo de Herbrand de Δ.

Teorema 5.4. *Sea Δ un programa definitivo y V_Δ su modelo mínimo de Herbrand. Entonces:*

- *V_Δ es la interpretación mínima de Herbrand tal que $T_\Delta(V_\Delta) = V_\Delta$.*
- *$V_\Delta = T_\Delta \uparrow n$.*

5.4 PRINCIPIO DE RESOLUCIÓN

Esta sección introduce un procedimiento de prueba utilizado ampliamente en la programación lógica: el principio de resolución propuesto por Robinson [93]. Si bien este procedimiento está orientado a un lenguaje más expresivo, nosotros nos concentraremos en una versión del principio que aplica a programas definitivos y se conoce como **resolución-SLD** [59] (resolución lineal con función de selección para cláusulas definitivas). De esta forma, definiremos la teoría de prueba para los programas definitivos, cuyo lenguaje y teoría de modelo fueron definidos en el capítulo anterior. Para ello, la sección 5.4.1 recuerda los conceptos de sistema lógico robusto y completo: e introduce el concepto de consecuencia lógica decidible.

Resolución-SLD

5.4.1 Robustez y completez semi-decidibles

La programación lógica, en este caso restringida a programas definitivos, concierne el uso de la lógica para **representar** y resolver problemas. Este uso es ampliamente aceptado en la IA, donde la idea se resume como sigue: Un problema o sujeto de investigación puede describirse mediante un conjunto de cláusulas. Si tal descripción es lo suficientemente precisa, la solución al problema o la respuesta a la pregunta planteada en la investigación, es una consecuencia lógica del conjunto de cláusulas

Representación de conocimiento

que describen el problema. De forma que nos gustaría tener un procedimiento algorítmico, que permita establecer si $\Delta \models \phi$ es el caso, esto es, si ϕ es consecuencia lógica de Δ.

En el caso de la lógica proposicional, el número de interpretaciones posibles para $\Delta \cup \{\phi\}$ es finito, de hecho es 2^n donde n es el número de átomos distintos que ocurren en $\Delta \cup \{\phi\}$. Para computar si $\Delta \models \phi$ simplemente debemos buscar si los modelos de Δ, lo son también de ϕ. Por tanto, se dice que la consecuencia en la lógica proposicional es **decidible**, es decir, existe un algoritmo que puede resolver el problema. Pero ¿Qué sucede en el contexto de la lógica de primer orden?

Procedimiento de prueba decidible

La intuición nos dice que el procedimiento de decisión de la lógica proposicional no es adecuado para las representaciones en primer orden, pues en este caso podemos tener una cantidad infinita de interpretaciones diferentes. Lo que es peor, el teorema de Church [21, 106], expresa que la lógica de primer orden es **no decidible** en lo general:

Procedimiento de prueba no decidible

Teorema 5.5 (Church)**.** *El problema de si $\Delta \models \phi$, cuando Δ es un conjunto finito arbitrario de fbf, y ϕ es una fbf arbitraria, es* ***no decidible****.*

Observen que el problema es no decidible para fórmulas de primer orden arbitrarias. No existe un algoritmo que en un número finito de pasos, de la respuesta correcta a la pregunta ¿Es ϕ consecuencia lógica de Δ? para el caso general. Afortunadamente existen procedimientos de prueba que pueden responder a esta cuestión en un número finito de pasos, cuando es el caso que $\Delta \models \phi$. Pero si es el caso que $\Delta \not\models \phi$ obtendremos la respuesta "no" (en el mejor de los casos) o el procedimiento no terminará nunca. Por ello suele decirse que la consecuencia en estos casos es **semi-decidible**.

Procedimiento de prueba semi-decidible

Hemos abordado el concepto de **procedimiento de prueba en la lógica proposicional**, al introducir el concepto de consecuencia válida y probar que el sistema es robusto y consistente. Recuerden que una consecuencia válida es aquella donde su conclusión ϕ es derivable a partir de sus premisas Δ, mediante el uso de las reglas de inferencia del sistema. Tal derivación, o prueba, suele consistir de un pequeño número de transformaciones, en las cuales nuevas fbf son derivadas de las premisas y de fbf previamente derivadas. Para el caso proposicional, si ϕ es derivable de Δ entonces es el caso que $\Delta \models \phi$ (Robustez) y su ese es el caso, entonces existe una prueba de ello (Completez).

Procedimiento de prueba proposicional

Ahora bien, con nuestro conocido *modus ponens*:

$$\frac{\phi, \quad \phi \rightarrow \psi}{\psi}$$

y el siguiente programa lógico:

$$\Delta = \{p(a), q(b) \leftarrow p(a), r(b) \leftarrow q(b)\}$$

es posible construir la prueba de $r(b)$ como sigue:

1. Derivar $q(b)$ a partir de $p(a)$ y $q(b) \leftarrow p(a)$.
2. Derivar $r(b)$ a partir de $q(b)$ y $r(b) \leftarrow q(b)$.

Es evidente que si usamos *modus ponens*, la conclusión ψ es una consecuencia lógica de las premisas: $\{\phi, \phi \rightarrow \psi\} \models \psi$. A esta propiedad del *modus ponens* se le conoce como robustez (*soundness*). En general un procedimiento de prueba es **robusto** si todas las fbf ψ que pueden ser derivadas de algún conjunto de fbfs Δ usando el procedimiento, son consecuencias lógicas de Δ. En otras palabras, un procedimiento de prueba es robusto si y sólo si sólo permite derivar consecuencias lógicas de las premisas.

Robustez

Una segunda propiedad deseable de los procedimientos de prueba es su completez. Un procedimiento de prueba es **completo** si toda fbf que es una consecuencia

Completez

lógica de las premisas Δ, puede ser derivada usando el procedimiento en cuestión. El *modus ponens* por si mismo, no es completo. Por ejemplo, no existe secuencia alguna de aplicaciones del *modus ponens* que deriven la fbf $p(a)$ de $\Delta = \{p(a) \wedge p(b)\}$, cuando es evidente que $\Delta \models p(a)$.

La regla $\frac{\phi}{\psi}$ es completa, pero no robusta. !Nos permite extraer cualquier conclusión, a partir de cualquier premisa! Esto ejemplifica que obtener completez es sencillo, pero obtener completez y robustez, no lo es.

5.4.2 Pruebas y programas lógicos

Recordemos que las cláusulas definitivas tienen la estructura general de la implicación lógica:

$$\phi_0 \leftarrow \phi_1, \ldots, \phi_n \quad (n \geq 0)$$

donde $\phi_0, \ldots, \phi_n$ son átomos. Consideren el siguiente programa definitivo Δ que describe un mundo donde los padres de un recién nacido están orgullosos, Juan es el padre de Ana y Ana es una recién nacida:

$$\begin{aligned} orgulloso(X) &\leftarrow padre(X,Y), recien\_nacido(Y). \\ padre(X,Y) &\leftarrow papa(X,Y). \\ padre(X,Y) &\leftarrow mama(X,Y). \\ papa(juan, ana). & \\ recien\_nacido(ana). & \end{aligned}$$

Observen que el programa describe únicamente conocimiento positivo, es decir, no especifica quién no está orgulloso. Tampoco que significa para alguien no ser padre. Ahora, supongamos que deseamos contestar la pregunta ¿Quién está orgulloso? Esta pregunta concierne al mundo descrito por nuestro programa, esto es, concierne al modelo previsto para Δ. La respuesta que esperamos es, por supuesto, *juan*. Ahora, recuerden que la lógica de primer orden no nos permite expresar enunciados interrogativos, por lo que nuestra pregunta debe formalizarse como una cláusula meta (enunciado declarativo):

$$\leftarrow orgulloso(Z).$$

que es una abreviatura de $\forall Z \neg orgulloso(Z)$ (una cláusula definitiva sin cabeza), que a su vez es equivalente de:

$$\neg \exists Z \; orgulloso(Z).$$

cuya lectura es –Nadie está orgulloso; esto es, la respuesta negativa a la consulta original –¿Quién está orgulloso? La meta ahora es probar que este enunciado es falso en todo modelo del programa Δ y en particular, es falso en el modelo previsto para Δ, puesto que esto es una forma de probar que $\Delta \models \exists Z \; orgulloso(Z)$. En general para todo conjunto de fbf cerradas Δ y una fbf cerrada γ, tenemos que $\Delta \models \gamma$ si $\Delta \cup \{\neg\gamma\}$ es no satisfacerle (no tiene modelo). Por lo tanto, nuestro objetivo es encontrar una substitución θ tal que el conjunto $\Delta \cup \{\neg orgulloso(Z)\theta\}$ sea no satisfacible, o de manera equivalente, $\Delta \models \exists Z \; orgulloso(Z)\theta$.

El punto inicial de nuestro razonamiento es asumir la meta G_0 – Para cualquier Z, Z no está orgulloso. La inspección del programa Δ revela que una regla describe una condición para que alguien esté orgulloso:

$$orgulloso(X) \leftarrow padre(X,Y), recien\_nacido(Y).$$

lo cual es lógicamente equivalente a:

$$\forall(\neg orgulloso(X) \rightarrow \neg(padre(X,Y) \wedge recien\_nacido(Y)))$$

Al renombrar X por Z, eliminar el cuantificador universal y usar *modus ponens* con respecto a G_0, obtenemos:

$$\neg(padre(Z, Y) \wedge recien\_nacido(Y))$$

o su equivalente:

$$\leftarrow padre(Z, Y), recien\_nacido(Y).$$

al que identificaremos como G_1. Un paso en nuestro razonamiento resulta en remplazar la meta G_0 por la meta G_1 que es verdadera en todo modelo $\Delta \cup \{G_0\}$. Ahora solo queda probar que $\Delta \cup \{G_1\}$ es no satisfacible. Observen que G_1 es equivalente a la fbf:

$$\forall Z \forall Y(\neg padre(Z, Y) \vee \neg recien\_nacido(Y))$$

Por lo tanto, puede probarse que la meta G_1 es no satisfacible para Δ, si en todo modelo de Δ hay una persona que es padre de un recién nacido. Entonces, verificamos primero si hay padres con estas condiciones. El programa contiene la cláusula:

$$padre(X, Y) \leftarrow papa(X, Y).$$

que es equivalente a:

$$\forall(\neg padre(X, Y) \rightarrow \neg papa(X, Y))$$

por lo que G_1 se reduce a:

$$\leftarrow papa(Z, Y), recien\_nacido(Y).$$

que identificaremos como G_2. Se puede mostrar que no es posible satisfacer la nueva meta G_2 con el programa Δ, si en todo modelo de Δ hay una persona que es papá de un recién nacido. El programa declara que *juan* es padre de *ana*:

$$papa(juan, ana).$$

así que sólo resta probar que "*ana* no es una recién nacida" no se puede satisfacer junto con Δ:

$$\leftarrow recien\_nacido(ana).$$

pero el programa contiene el hecho:

$$recien\_nacido(ana).$$

equivalente a $\neg recien\_nacido(ana) \rightarrow \square$, lo que conduce a una refutación. Este razonamiento puede resumirse de la siguiente manera: para probar la existencia de algo, suponer lo contrario y usar *modus ponens* y la regla de eliminación del cuantificador universal, para encontrar un contra ejemplo al supuesto.

Observen que la meta definitiva fue convertida en un conjunto de átomos a ser probados. Para ello, se seleccionó una fbf atómica de la meta $\phi_0(\sigma_1, \ldots, \sigma_n)$ y una cláusula de la forma $\phi_0(\tau_1, \ldots, \tau_n) \leftarrow \phi_1, \ldots \phi_n$ para encontrar una instancia común de la submeta y la cabeza de la cláusula seleccionada; es decir, una substitución θ ambas expresiones sean idénticas. Tal substitución se conoce como **unificador**. La nueva meta se construye remplazando el átomo seleccionado en la meta original, por el cuerpo de la cláusula seleccionada, bajo la substitución θ.

Unificador

El paso de computación básico de nuestro ejemplo, puede verse como una regla de inferencia puesto que transforma fórmulas lógicas. Lo llamaremos **principio de resolución SLD** para programas definitivos. Como mencionamos, el procedimiento combina *modus ponens*, *eliminación del cuantificador universal* y en el paso final un *reductio ad absurdum*.

Cada paso de razonamiento produce una substitución, si se prueba en k pasos que la meta definida en cuestión no puede satisfacerse, probamos que:

$$\leftarrow (\phi_1, \ldots \phi_n)\theta_1 \ldots \theta_k$$

es una instancia que no puede satisfacerse. De manera equivalente, que:

$$\Delta \models (\phi_1 \wedge \cdots \wedge \phi_n)\theta_1 \ldots \theta_k$$

Observen que generalmente, la computación de estos pasos de razonamiento no es determinista: cualquier átomo de la meta puede ser seleccionado y pueden haber varias cláusulas del programa que unifiquen con el átomo seleccionado. Otra fuente de indeterminismo es la existencia de unificadores alternativos para dos átomos. Esto sugiere que es posible construir muchas soluciones (algunas veces, una cantidad infinita de ellas).

Por otra parte, es posible también que el atomo seleccionado no unifique con ninguna cláusula en el programa. Esto indica que no es posible construir un contra ejemplo para la meta definida inicial. Finalmente, la computación puede caer en un ciclo y de esta manera no producir solución alguna.

5.4.3 Substitución

Recordemos que una substitución es un conjunto de asignaciones de términos a variables. Por ejemplo, podemos remplazar la variable X por el término $f(a)$ en la cláusula $p(X) \vee q(X)$, y así obtener la nueva cláusula $p(f(a)) \vee q(f(a))$. Si asumimos que las cláusulas están cuantificadas universalmente, decimos que está substitución hace a la cláusula original *menos general* ó *más específica*, en el sentido que mientras la cláusula original dice que para toda X la cláusula se satisface, la segunda versión es sólo cierta cuando X es substituida por a. Observen que la segunda cláusula es consecuencia lógia de la primera: $p(X) \vee q(X) \models p(f(a)) \vee q(f(a))$

Definición 5.17 (Substitución)**.** *Una substitución θ es un conjunto finito de la forma:*

$$\{X_1/t_1, \ldots, X_n/t_n\}, \quad (n \geq 0)$$

donde las X_i son variables, distintas entre si, y los t_i son términos. Decimos que t_i substituye a X_i. La forma X_i/t_i se conoce como ligadura de X_i.

Substitución de base

Una substitución se dice **de base** si todos los términos en ella son de base. Es claro, que éste es el caso de las interpretaciones de Herband.

Substitución identidad

La substitución definida como el conjunto vacío, se conoce como **substitución de identidad** o **substitución vacía** y se denota por ϵ. La restricción de θ sobre un conjunto de variables *Var* es la substitución $\{X/t \in \theta \mid X \in Var\}$.

Ejemplo 5.14. *$\{Y/X, X/g(X,Y)\}$ y $\{X/a, Y/f(Z), Z/(f(a), X_1/b\}$ son substituciones. La restricción de la segunda substitución sobre $\{X, Z\}$ es $\{X/a, Z/f(a)\}$.*

Definición 5.18 (Expresión)**.** *Una expresión es un término, una literal, o una conjunción o disyunción de literales. Una expresión simple es un término o una literal.*

Observen que una cláusula es una expresión. Las substituciones pueden aplicarse a las expresiones, lo que significa que las variables en las expresiones serán remplazadas de acuerdo a la substitución.

Definición 5.19. *Sea $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ una substitución y ϕ una expresión. Entonces $\phi\theta$, la ocurrencia (*instance*) de ϕ por θ, es la expresión obtenida al substituir simultáneamente X_i por t_i para $1 \leq i \leq n$. Si $\phi\theta$ es una expresión de base, se dice que es una ocurrencia base y se dice que θ es una substitución de base para ϕ. Si $\Sigma = \{\phi_1, \ldots, \phi_n\}$ es un conjunto finito de expresiones, entonces $\Sigma\theta$ denota $\{\phi_1\theta, \ldots, \phi_n\theta\}$.*

Ejemplo 5.15. *Sea ϕ la expresión $p(Y, f(X))$ y sea θ la substitución $\{X/a, Y/g(g(X))\}$. La ocurrencia de ϕ por θ es $\phi\theta = p(g(g(X)), f(a)$. Observen que X e Y son simultáneamente remplazados por sus respectivos términos, lo que implica que X en $g(g(X))$ no es afectada por X/a.*

Si ϕ es una expresión cerrada que no es un término, por ejemplo, una literal, o una conjunción o disyunción de literales, y θ es una substitución, lo siguiente se cumple:

$$\phi \models \phi\theta$$

por ejemplo: $p(X) \vee \neg q(Y) \models p(a) \vee \neg q(Y)$ donde hemos usado la substitución $\{X/a\}$.

Podemos aplicar una substitución θ y luego aplicar una substitución σ, a lo cual se llama composición de las substituciones θ y σ. Si ese es el caso, primero se aplica θ y luego σ. Las composiciones pueden verse como mapeos del conjunto de variables en el lenguaje, al conjunto de términos.

Definición 5.20 (Composición)**.** *Sean $\theta = \{X_1/s_1, \ldots, X_m/s_m\}$ y $\sigma = \{Y_1/t_1, \ldots Y_n/t_n\}$ dos substituciones. Consideren la secuencia:*

$$X_1/(s_1\sigma), \ldots, X_m/(s_m\sigma), Y_1/t_1, \ldots, Y_n/t_n$$

Si se borran de esta sencuencia las ligaduras $X_i/s_i\sigma$ cuando $X_i = s_i\sigma$ y cualquier ligadura Y_j/t_j donde $Y_j \in \{X_1, \ldots, X_m\}$. La substitución consistente en las ligaduras de la secuencia resultante es llamada composición de θ y σ, se denota por $\theta\sigma$.

Ejemplo 5.16. *Sea $\theta = \{X/f(Y), Z/U\}$ y $\sigma = \{Y/b, U/Z\}$. Construimos la secuencia de ligaduras $X/(f(Y)\sigma), Z/(u)\sigma, Y/b, U/Z$ lo cual es $X/f(b), Z/Z, Y/b, U/Z$. Al borrar la ligadura Z/Z obtenemos la secuencia $X/f(b), Y/b, U/Z = \theta\sigma$.*

Definición 5.21 (Ocurrencia)**.** *Sean θ y σ dos substituciones. Se dice que θ es una ocurrencia de σ, si existe una substitución γ, tal que $\sigma\gamma = \theta$.*

Ejemplo 5.17. *La substitución $\theta = \{X/f(b), Y/a\}$ es una ocurrencia de la substitución $\sigma = \{X/f(X), Y/a\}$, puesto que $\sigma\{X/b\} = \theta$.*

Algunas propiedades sobre las substituciones incluyen:

Proposición 5.2. *Sea ϕ una expresión, y sea θ, σ y γ substituciones. Las siguientes relaciones se cumplen:*

1. $\theta = \theta\epsilon = \epsilon\theta$
2. $(\phi\theta)\sigma = \phi(\theta\sigma)$
3. $\theta\sigma)\gamma = \theta(\sigma\gamma)$

5.4.4 Unificación

Uno de los pasos principales en el ejemplo de la sección 5.4.2, consistió en hacer que dos fbf atómicas se vuelvan sintácticamente equivalentes. Este proceso se conoce como **unificación** y posee una solución algorítmica. *Unificación*

Definición 5.22 (Unificador)**.** *Sean ϕ y ψ términos. Una substitución θ tal que ϕ y ψ sean idénticos ($\phi\theta = \psi\theta$) es llamada unificador de ϕ y ψ.*

Ejemplo 5.18.

$$unifica(conoce(juan, X), conoce(juan, maria)) = \{X/maria\}$$

$$unifica(conoce(juan, X), conoce(Y, Z)) = \{Y/juan, X/Z\}$$
$$= \{Y/juan, X/Z, W/pedro\}$$
$$= \{Y/juan, X/juan, Z/juan\}$$

Definición 5.23 (Generalidad entre substituciones)**.** *Una substitución θ se dice más general que una substitución σ, si y sólo si existe una substitución γ tal que $\sigma = \theta\gamma$.*

Definición 5.24 (MGU)**.** *Un unificador θ se dice el unificador más general (MGU) de dos términos, si y sólo si θ es más general que cualquier otro unificador entre esos términos.*

Ejemplo 5.19. *Consideren los términos $f(X)$ y $f(g(Y))$, el MGU de ellos es $\{X/g(Y)\}$, pero existen otros muchos unificadores no MGU, por ejemplo $\{X/g(a), Y/a\}$. Intuitivamente, el MGU de dos términos es el más simple de todos sus unificadores.*

Definición 5.25 (Forma resuelta)**.** *Un conjunto de ecuaciones $\{X_1 = t_1, \ldots, X_n = t_n\}$ está en forma resuelta, si y sólo si $X_1, \ldots, X_n$ son variables distintas que no ocurren en $t_1, \ldots, t_n$.*

Existe una relación cercana entre un conjunto de ecuaciones en forma resuelta y el unificador más general de ese conjunto: Sea $\{X_1 = t_1, \ldots, X_n = t_n\}$ un conjunto de ecuaciones en forma resuelta. Entonces $\{X_1/t_1, \ldots, X_n/t_n\}$ es un MGU idempotente de la forma resuelta.

Definición 5.26 (Equivalencia en conjuntos de ecuaciones)**.** *Dos conjuntos de ecuaciones E_1 y E_2 se dicen equivalentes, si tienen el mismo conjunto de unificadores.*

La definición puede usarse como sigue: para computar el MGU de dos términos ϕ y ψ, primero intente transformar la ecuación $\{\phi = \psi\}$ en una forma resuelta equivalente. Si esto falla, entonces $mgu(\phi, \psi) = fallo$. Sin embargo, si una forma resuelta $\{X_1 = t_1, \ldots, X_n = t_n\}$ existe, entonces $mgu(\phi, \psi) = \{X_1/t_1, \ldots, X_n/t_n\}$. Un algoritmo para encontrar la forma resuelta de un conjunto de ecuaciones es como sigue:

Algoritmo 5.1 Unifica(E)

```
1:  function UNIFICA(E)                          ▷ E es un conjunto de ecuaciones
2:      repeat
3:          (s = t) ← seleccionar(E)
4:          if f(s_1, ..., s_n) = f(t_1, ..., t_n) (n ≥ 0) then
5:              remplazar (s = t) por s_1 = t_1, ..., s_n = t_n
6:          else if f(s_1, ..., s_m) = g(t_1, ..., t_n) (f/m ≠ g/n) then
7:              return(fallo)
8:          else if X = X then
9:              remover la X = X
10:         else if t = X then
11:             remplazar t = X por X = t
12:         else if X = t then
13:             if subtermino(X,t) then
14:                 return(fallo)
15:             else remplazar todo X por t
16:             end if
17:         end if
18:     until No hay accion posible para E
19: end function
```

Ejemplo 5.20. *El conjunto $\{f(X, g(Y)) = f(g(Z), Z)\}$ tiene una forma resuelta, puesto que:*

$$\Rightarrow \{X = g(Z), g(Y) = Z\}$$
$$\Rightarrow \{X = g(Z), Z = g(Y)\}$$
$$\Rightarrow \{X = g(g(Y)), Z = g(Y)\}$$

Ejemplo 5.21. *El conjunto* $\{f(X, g(X), b) = f(a, g(Z), Z)\}$ *no tiene forma resuelta, puesto que:*

$$\Rightarrow \{X = a, g(X) = g(Z), b = Z\}$$
$$\Rightarrow \{X = a, g(a) = g(Z), b = Z\}$$
$$\Rightarrow \{X = a, a = Z, b = Z\}$$
$$\Rightarrow \{X = a, Z = a, b = Z\}$$
$$\Rightarrow \{X = a, Z = a, b = a\}$$
$$\Rightarrow fallo$$

Ejemplo 5.22. *El conjunto* $\{f(X, g(X)) = f(Z, Z)\}$ *no tiene forma resuelta, puesto que:*

$$\Rightarrow \{X = Z, g(X) = Z\}$$
$$\Rightarrow \{X = Z, g(Z) = Z\}$$
$$\Rightarrow \{X = Z, Z = g(Z)\}$$
$$\Rightarrow fallo$$

Este algoritmo termina y regresa una forma resuelta equivalente al conjunto de ecuaciones de su entrada; o bien regresa fallo si la forma resuelta no existe. Sin embargo, el computar $subtermino(X, t)$ hace que el algoritmo sea altamente ineficiente. Los sistemas Prolog resuelven este problema haciéndo caso omiso de la verificación de ocurrencia. El standard ISO Prolog (1995) declara que el resultado de la unificación es no decidible. Al eliminar la verificación de ocurrencia es posible que al intentar resolver $X = f(X)$ obtengamos $X = f(f(X)) \cdots = f(f(f \ldots))$. En la practica los sistemas Prolog no caen en este ciclo, pero realizan la siguiente substitución $\{X/f(\infty)\}$.

Verificación de ocurrencia

Si bien esto parece resolver el problema de eficiencia, generaliza el concepto de término, substitución y unificación al caso del infinito, no considerado en la lógica de primer orden, introduciéndo a su vez inconsistencia.

Ejemplo 5.23. *Consideren el siguiente programa definitivo:*

```
1 menor(X,s(X)).
2 foo :- menor(S(Y)), Y).
```

El programa expresa que todo número es menor que su sucesor, pero ¿Qué sucede si evaluan foo*? Pues que Prolog responderá Yes!*

La unificación también puede verse como una búsqueda en un espacio conocido como **Rejilla de generalización** (Ver figura 5.5), donde se introducen dos términos especiales llamados *top* ($\top$) y *bottom* ($\bot$). En este contexto, la unificación consiste en encontrar la junta más inferior de dos términos en el gráfico [84, 85].

Rejilla de generalización

Ejemplo 5.24. *Unificar los términos* $f(X, b)$ *y* $f(a, Y)$ *es posible porque ambos nodos convergen en* $f(a, b)$.

5.4.5 Resolución-SLD

El método de razonamiento descrito informalmente al inicio de esta sesión, puede resumirse con la siguiente regla de inferencia:

$$\frac{\forall\neg(\phi_1 \wedge \cdots \wedge \phi_{i-1} \wedge \phi_i \wedge \phi_{i+1} \wedge \cdots \wedge \phi_m) \quad \forall(\psi_0 \leftarrow \psi_1 \wedge \cdots \wedge \psi_n)}{\forall\neg(\phi_1 \wedge \cdots \wedge \phi_{i-1} \wedge \psi_1 \wedge \cdots \wedge \psi_n \wedge \phi_{i+1} \wedge \cdots \wedge \phi_m)\theta}$$

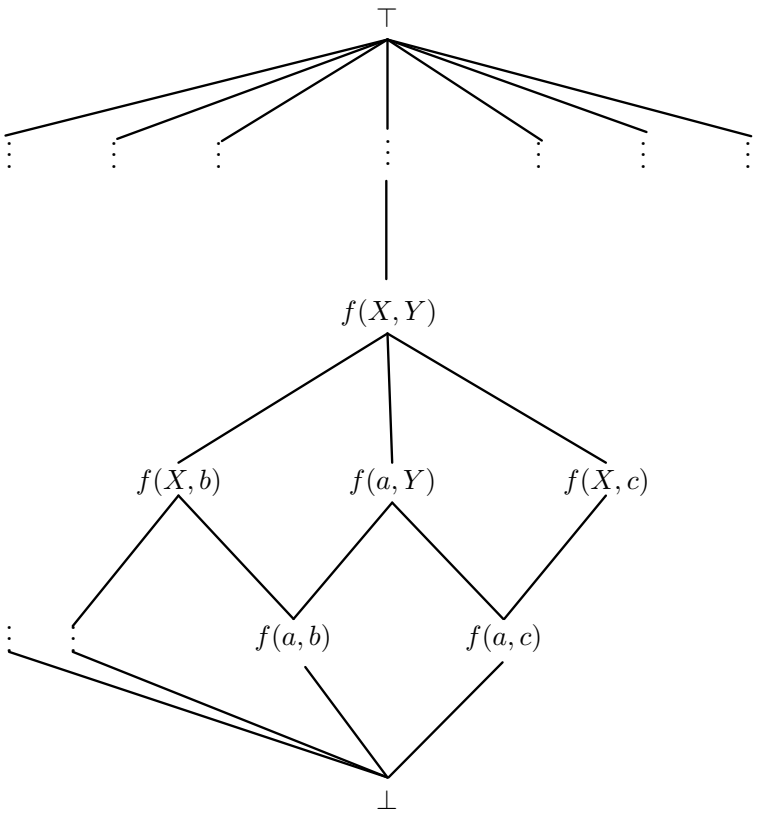


Figura 5.5: Latice de Generalización

o, de manera equivalente, usando la notación de los programas definitivos:

$$\frac{\leftarrow \phi_1, \ldots, \phi_{i-1}, \phi_i, \phi_{i+1}, \ldots, \phi_m \quad \psi_0 \leftarrow \psi_1, \ldots, \psi_n}{\leftarrow (\phi_1, \ldots, \phi_{i-1}, \psi_1, \ldots, \psi_n, \ldots, \phi_m)\theta}$$

donde:

1. $\phi_1, \ldots, \phi_m$ son fbf atómicas.
2. $\psi_0 \leftarrow \psi_1, \ldots, \psi_n$ es una cláusula definitiva en el programa Δ ($n \geq 0$).
3. $MGU(\phi_i, \psi_0) = \theta$.

La regla tiene dos premisas: una meta y una cláusula definitivas. Observen que cada una de ellas está cuantificada universalmente, por lo que el alcance de los cuantificadores es disjunto. Por otra parte, solo hay un cuantificador universal para la conclusión, por lo que se requiere que el conjunto de variables en las premisas sea disjunto. Puesto que todas las variables en las premisas están cuantificadas, es siempre posible renombrar las variables de la cláusula definitiva para cumplir con esta condición.

La meta definida puede incluir muchas fbf atómicas que unifican con la cabeza de alguna cláusula en el programa. En este caso, es deseable contar con un mecanismo determinista para seleccionar un átomo ϕ_i a unificar. Se asume una función que **selecciona** una submeta de la meta definida.

función de selección

La regla de inferencia presentada es la única necesaria para procesar programas definitivos. Esta regla es una versión de la regla de inferencia conocida como **principio de resolución**, introducido por J.A. Robinson en 1965. El principio de resolución aplica a cláusulas. Puesto que las cláusulas definitivas son más restringidas que las cláusulas, la forma de resolución presentada se conoce como resolución-SLD (resolución lineal para cláusulas definitivas con función de selección).

Principio de resolución

El punto de partida de la aplicación de esta regla de inferencia es una meta definida G_0:

$$\leftarrow \phi_1, \ldots, \phi_m \quad (m \geq 0)$$

De esta meta, una submeta ϕ_i será seleccionada, de preferencia por una función de selección. Una nueva meta G_1 se construye al seleccionar una cláusula del programa $\psi_0 \leftarrow \psi_1, \ldots, \psi_n$ $(n \geq 0)$ cuya cabeza ψ_0 unifica con ϕ_i, resultando en θ_1. G_1 tiene la forma:

$$\leftarrow (\phi_1, \ldots, \phi_{i-1}, \psi_1, \ldots, \psi_n, \ldots, \phi_m)\theta_1$$

Ahora es posible aplicar el principio de resolución a G_1 para obtener G_2, y así sucesivamente. El proceso puede terminar o no. Hay dos situaciones donde no es posible obtener G_{i+1} a partir de G_i:

1. cuando la submeta seleccionada no puede ser resuelta (no es unificable con la cabeza de una cláusula del programa).
2. cuando $G_i = \square$ (meta vacía = **f**).

Definición 5.27 (Derivación-SLD)**.** *Sea G_0 una meta definitiva, Δ un programa definitivo y $\mathcal{R}$ una función de selección. Una derivación SLD de G_0 (usando Δ y $\mathcal{R}$) es una secuencia finita o infinita de metas:*

$$G_0 \overset{\phi_0}{\rightsquigarrow} G_1 \ldots G_{n-1} \overset{\phi_{n-1}}{\rightsquigarrow} G_n$$

Para manejar de manera consistente el renombrado de variables, las variables en una cláusula ϕ_i serán renombradas poniéndoles subíndice i.

Cada derivación SLD nos lleva a una secuencias de MGUs $\theta_1, \ldots, \theta_n$. La composición

$$\theta = \begin{cases} \theta_1\theta_2 \ldots \theta_n & \text{si } n > 0 \\ \epsilon & \text{si } n = 0 \end{cases}$$

de MGUs se conoce como la **substitución computada** de la derivación.

Substitución computada

Ejemplo 5.25. *Consideren la meta definida $\leftarrow orgulloso(Z)$ y el programa discutido en la clase anterior.*

$G_0 = \leftarrow orgulloso(Z).$

$\phi_0 = orgulloso(X_0) \leftarrow padre(X_0, Y_0), recien\_nacido(Y_0).$

La unificación de $orgulloso(Z)$ y $orgulloso(X_0)$ nos da el MGU $\theta_1 = \{X_0/Z\}$. Asumamos que nuestra función de selección es tomar la submeta más a la izquierda. El primer paso de la derivación nos conduce a:

$G_1 = \leftarrow padre(Z, Y_0), recien\_nacido(Y_0).$

$\phi_1 = padre(X_1, Y_1) \leftarrow papa(X_1, Y_1).$

En el segundo paso de la resolución el MGU $\theta_2 = \{X_1/Z, Y_1/Y_0\}$ es obtenido. La derivación continua como sigue:

$G_2 = \leftarrow papa(Z, Y_0), recien\_nacido(Y_0).$

$\phi_2 = papa(juan, ana).$

$G_3 = \leftarrow recien\_nacido(ana).$

$\phi_3 = recien\_nacido(ana).$

$G_4 = \square$

la substitución computada para esta derivación es:

$$\theta_1\theta_2\theta_3\theta_4 = \{X_0/Z\}\{X_1/Z, Y_1/Y_0\}\{Z/juan, Y_0/ana\}\epsilon$$

$$= \{X_0/juan, X_1/juan, Y_1/ana, Z/juan, Y_0/ana\}$$

Las derivaciones SLD que terminan en la meta vacía ($\square$) son de especial importancia pues corresponden a refutaciones a la meta inicial (y proveen las respuestas a la meta).

Definición 5.28 (Refutación SLD). *Una derivación SLD finita:*

$$G_0 \overset{\phi_0}{\rightsquigarrow} G_1 \ldots G_{n-1} \overset{\phi_{n-1}}{\rightsquigarrow} G_n$$

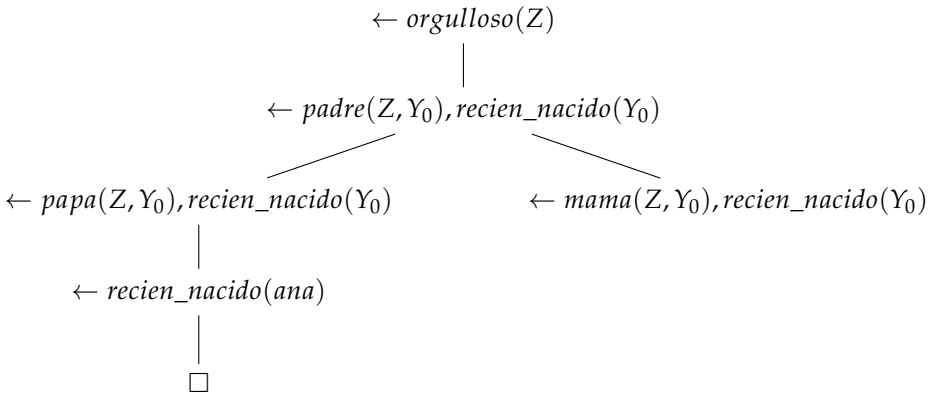
donde $G_n = \square$, se llama refutación SLD de G_0.

Definición 5.29 (Derivación fallida). *Una derivación de la meta definitiva G_0 cuyo último elemento no es la meta vacía y no puede resolverse con ninguna cláusula del programa, es llamada derivación fallida.*

Definición 5.30 (Arbol-SLD). *Sea Δ un programa definitivo, G_0 una meta definitiva, y $\mathcal{R}$ una función de selección. El árbol-SLD de G_0 (usando Δ y $\mathcal{R}$) es un árbol etiquetado, posiblemente infinito, que cumple las siguientes condiciones:*

- *La raíz del árbol está etiquetada por G_0.*
- *Si el árbol contiene un nodo etiquetado como G_i y existe una cláusula renombrada $\phi_i \in \Delta$ tal que G_{i+1} es dervidada de G_i y ϕ_i via $\mathcal{R}$, entonces el nodo etiquetado como G_i tiene un hijo etiquetado G_{i+1} El arco que conecta ambos nodos está etiquetado como ϕ_i.*

Por ejemplo:



5.4.6 Propiedades de la resolución-SLD

Definición 5.31 (Robustez). *Sea Δ un programa definitivo, $\mathcal{R}$ una función de selección, y θ una substitución de respuesta computada a partir de Δ y $\mathcal{R}$ para una meta $\leftarrow \phi_1, \ldots, \phi_m$. Entonces $\forall((\phi_1 \wedge \cdots \wedge \phi_m)\theta)$ es una consecuencia lógica del programa Δ.*

Definición 5.32 (Completez). *Sea Δ un programa definitivo, $\mathcal{R}$ una función de selección y $\leftarrow \phi_1, \ldots, \phi_m$ una meta definitiva. Si $\Delta \models \forall((\phi_1 \wedge \cdots \wedge \phi_m)\sigma)$, entonces existe una refutación de $\leftarrow \phi_1, \ldots, \phi_m$ vía $\mathcal{R}$ con una substitución de respuesta computada θ, tal que $(\phi_1 \wedge \cdots \wedge \phi_m)\sigma$ es un caso de $(\phi_1 \wedge \cdots \wedge \phi_m)\theta$.*

5.5 LECTURAS Y EJERCICIOS SUGERIDOS

El material aquí presentado está basado principalmente en los textos de Genesereth y Nilsson [43], capítulo 2; y el de Nilsson y Maluszynski [77], capítulo 1. Una lectura complementaria a estos textos son los capítulos 8 y 9 del texto de Russell y Norvig [95]. Una aproximación más computacional a la Lógica de Primer Orden, puede encontrarse en Huth y Ryan [54], capítulo 2.

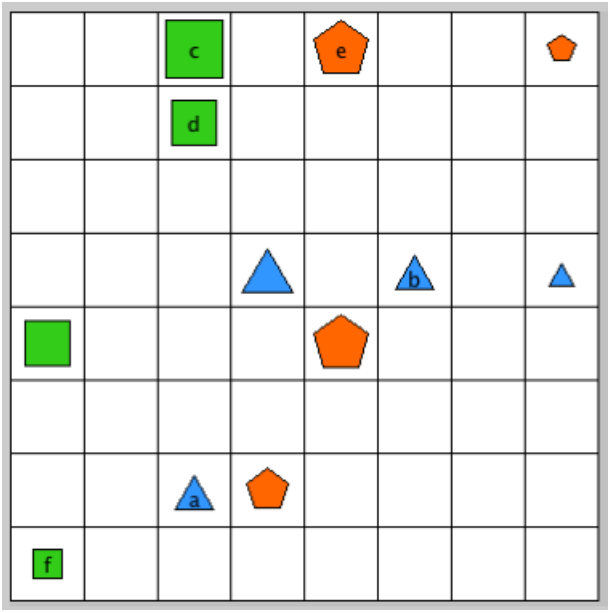


Figura 5.6: Mundo de Tarski para los ejercicios 1 y 2.

El algoritmo de unificación presentado (Ver página 87) es una versión del algoritmo propuesto por Robinson [93] en su trabajo sobre el principio de resolución. Knight [57] ofrece una perspectiva multidisciplinaria al problema de la unificación. Observen que este proceso es relevante en diferentes áreas de las Ciencias de la Computación, como la demostración de teoremas, la programación lógica, el procesamiento de lenguaje natural, la complejidad computacional y la teoría de computabilidad; de forma que un marco de referencia general sobre este tema es de agradecerse. La resolución lineal con función de selección (resolución-SL) fue introducida por Kowalski y Kuehner [59]. Su restricción a los programas definitivos (resolución-SLD) fue propuesta más tarde por el mismo Kowalski [58].

Ejercicio 5.1. *Traduzca a Lógica de primer orden el enunciado: Todo hijo de mi padre es mi hermano.*

Ejercicio 5.2. *Consideren el Mundo de Tarski [5] mostrado en la figura 5.6. Escriban una interpretación para la escena mostrada, considerando los siguientes predicados con su significado evidente:* triángulo/1, cuadrado/1, pentágono/1, mediano/1, grande/1, pequeño/1, másPequeñoQue/2, aLaIzquierdaDe/2, enLaMismaColumna/2.

Ejercicio 5.3. *En la interpretación del ejercicio anterior, cuales de las siguientes fórmulas son safisfacibles y cuales no:*

1. $triangulo(a) \wedge cuadrado(c) \wedge pentagono(e)$
2. $mediano(a) \wedge \neg grande(a) \wedge \neg pequeno(a) \wedge pequeno(f) \wedge grande(c)$
3. $masPequeno(f,a) \wedge masPequeno(a,c) \wedge aLaIzquierdaDe(f,a) \wedge aLaIzquierdaDe(e,b)$
4. $\forall X \forall Y(aLaIzquierdaDe(X,Y) \vee mismaColumna(X,Y) \vee aLaIzquierdaDe(Y,X))$
5. $\forall X \forall Y(cuadrado(X) \wedge pentagono(Y) \rightarrow aLaIzquierdaDe(X,Y))$
6. $\exists X \exists Y(triangulo(X) \wedge cuadrado(Y) \wedge mismaColumna(X,Y))$

Ejercicio 5.4. *En la misma interpretación, introduzca un predicado que exprese una relación entre tres objetos del universo de discurso. Utilice el predicado introducido en una fórmula bien formada satisfacible.*

6 CONTROL DEL RAZONAMIENTO

Hemos visto como el programador puede controlar la ejecución de un programa ordenando sus cláusulas y sus metas. Ahora revisaremos otra herramienta de control conocida como corte, para prevenir la reconsideración característica de Prolog (*backtracking*). Este operador extiende además la expresividad de lenguaje, permitiendo la definición de una forma de negación, llamada negación por fallo finito (NAF), asociada al supuesto del mundo cerrado (CWA). En lo que sigue revisaremos estos dos conceptos en detalle.

6.1 CORTE

El árbol-SLD de una meta definitiva puede tener muchas ramas que conducen al fallo de la meta y muy pocas, ó una sola rama, que conducen al éxito. Por ello, el programador podría querer incluir información de control en sus programas, para evitar que el intérprete construya ramas fallidas. Observen que esta meta-información se basa en la semántica operacional del programa, por lo que el programador debe saber como se construyen y se recorren los arboles-SLD. El predicado $!/0$ denota la operación de **corte**, y puede utilizarse como una literal en las metas definitivas. Su presencia impide la construcción de ciertos sub-arboles. *Corte*

Un intérprete de Prolog recorre los nodos de un árbol-SLD primero en profundidad. El orden de las ramas corresponde al orden textual de las cláusulas en el programa. Cuando una hoja es alcanzada, el proceso de **backtracking** es ejecutado. El proceso termina cuando no es posible hacer backtracking (todos los sub-arboles de la raíz del árbol han sido visitados). *Backtrackig*

Ejemplo 6.1. *Asumamos el siguiente programa que define que el padre de una persona es su antecesor hombre:*

$$
\begin{aligned}
padre(X,Y) &\leftarrow progenitor(X,Y), hombre(X).\\
progenitor(benjamin, antonio).&\\
progenitor(maria, antonio).&\\
progenitor(samuel, benjamin).&\\
progenitor(alicia, benjamin).&\\
hombre(benjamin).&\\
hombre(samuel).&
\end{aligned}
$$

El árbol-SLD de la meta $\leftarrow padre(X, antonio)$ *se muestra en la Figura 6.1. Bajo la función de selección implementada en Prolog, encontrará la solución* $X/benjamin$. *El intento por encontrar otra solución con* $X/maria$, *mediante el backtracking, fallará puesto que maria no satisface el predicado hombre*$/1$.

Para detallar la semántica del corte, es necesario introducir algunos conceptos auxiliares. En un árbol-SLD, cada nodo n_i corresponde a una meta G_i de una derivación-SLD y tiene un átomo seleccionado asociado α_i:

$$G_0 \overset{\alpha_0}{\rightsquigarrow} G_1 \ldots G_{n-1} \overset{\alpha_{n-1}}{\rightsquigarrow} G_n$$

Asumamos que para cierto nodo n_k, α_k no es una sub-meta de la meta inicial. Entonces α_k es un átomo β_i del cuerpo de una cláusula de la forma $\beta_0 \leftarrow$

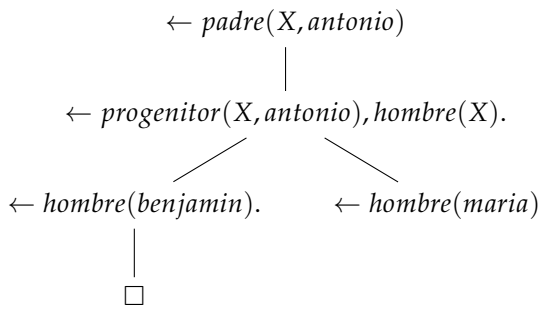


Figura 6.1: Arbol de derivación-SLD para la meta $\leftarrow padre(X, antonio)$

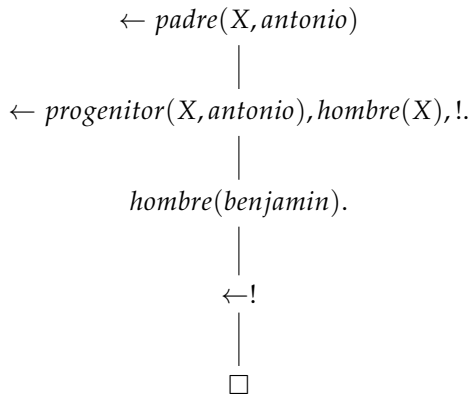


Figura 6.2: Arbol de derivación-SLD para la meta $\leftarrow padre(X, antonio)$ con las ramas fallidas podadas.

$\beta_1, \ldots, \beta_i, \ldots, \beta_n$ cuya cabeza β_0 unifica con la sub-meta seleccionada en algún nodo $n_{0<j<k}$, es decir un nodo entre la raíz del árbol y el nodo n_k. El nodo n_j se conoce como el **origen**, o padre, de α_k y se denota como $origen(\alpha_k)$. *Nodo origen*

El predicado de corte ! se procesa como un átomo ordinario situado en el cuerpo de una cláusula. Sin embargo, cuando el corte es seleccionado para computar la resolución, éste tiene éxito inmediatamente (con la substitución vacía ϵ como resultado). El nodo donde ! fue seleccionado es llamado el **nodo de corte**. Un nodo de corte puede ser visitado nuevamente durante el backtracking. En este caso, el curso normal del recorrido del árbol es alterado (por definición el recorrido continua en el nodo superior a $origen(!)$. Si el corte ocurre en la meta inicial, la ejecución simplemente termina. *Nodo de corte*

Ejemplo 6.2. *La formulación del problema padre, nos dice que a lo más existe una solución para nuestra meta. cuando la solución se encuentra, la búsqueda puede detenerse pues ninguna persona tiene más de un padre. Para forzar esta situación, el predicado de corte se agrega al final de padre/2:*

$$padre(X, Y) \leftarrow progenitor(X, Y), hombre(X), !.$$

Observen que el programa modificado en el ejemplo anterior sólo puede computar un elemento de la relación *padre*/2. El corte detendrá la búsqueda después de encontrar la primer respuesta para la meta $\leftarrow padre(X, Y)$. El origen del corte es la raíz del árbol, por lo que la búsqueda termina después de hacer backtracking al nodo de corte. La otra rama del árbol no es recorrida. El árbol-SLD del programa que incluye el corte se muestra en la figura 6.2.

Observen que la versión modificada con el corte, no puede usarse para computar más de un elemento de la relación "es padre de". El corte detendrá la búsqueda después de encontrar la primer respuesta a la meta definitiva.

A partir de la definición del corte, se sigue que los efectos del operador son:

1. Divide el cuerpo de la meta en dos partes –Después de éxito de `!/0` , no es posible hacer backtracking hacía las literales a la izquierda del corte. Sin embargo, a la derecha del corte todo funciona de manera usual.
2. Poda las ramas sin explorar directamente bajo $origen(!)$. En otras palabras, no habrá más intentos de unificar la sub-meta seleccionada de $origen(!)$ con el resto de las cláusulas del programa.

El corte es controvertido. La intención al introducir el corte, es poder controlar la ejecución de los programas, sin cambiar su significado lógico. Por tanto, la lectura lógica del corte es `true`. Operacionalmente, si el corte remueve sólo ramas fallidas del árbol-SLD, no tiene influencia en el significado lógico de un programa. Pero el corte puede remover también ramas exitosas del árbol-SLD, atentando contra la completitud de los programas definitivos, o la correctez de los programas generales.

Ejemplo 6.3. *Es bien sabido que los padres de un recién nacido están orgullosos. La proposición puede representarse con la siguiente cláusula definitiva:*

$$orgulloso(X) \leftarrow padre(X,Y), recienNacido(Y).$$

consideren las siguiente cláusulas adicionales:

$$\begin{aligned} padre(X,Y) &\leftarrow progenitor(X,Y), hombre(X). \\ progenitor(juan, maria). & \\ progenitor(juan, cristina). & \\ hombre(juan). & \\ recienNacido(cristina). & \end{aligned}$$

La respuesta a la meta $\leftarrow orgulloso(juan)$ es `true`, puesto que como describimos, juan es padre de cristina, que es un recién nacido. Ahora, si remplazamos la primera cláusula, con su versión que utiliza corte:

$$padre(X,Y) \leftarrow progenitor(X,Y), hombre(X), !.$$

Y preguntamos nuevamente a Prolog, si

$$\leftarrow orgulloso(juan).$$

la respuesta será `false`. Esto se debe a que la primer hija de juan en el programa es maria. Una vez que esta respuesta se ha encontrado, no habrá más intentos de satisfacer la meta en $origen(!)$. No se considerarán más hijos de juan en la solución computada.

El programa del ejemplo anterior se ha vuelto incompleto, algunas respuestas correctas no pueden ser computadas. Más grave aún es el caso de las metas generales, donde se puede llegar a resultados incorrectos, por ejemplo, $\leftarrow \neg orgulloso(juan)$ tendría éxito en la versión de nuestro programa que utiliza corte.

Hasta ahora hemos distinguido dos usos del corte: eliminar ramas fallidas en el árbol-SLD; y podar ramas exitosas. Eliminar ramas fallidas se considera una práctica sin riesgo, porque no altera las respuestas producidas durante la ejecución de un programa. Tales cortes se conocen como **cortes verdes**. Sin embargo, este uso del operador corte, esta ligado al uso particular de un programa. Como se ilustra en los ejemplos anteriores, para algunas metas, el operador solo eliminará ramas fallidas; pero para otras podará ramas exitosas. Cortar ramas exitosas se considera una práctica de riesgo. Por eso, tales cortes se conocen como **cortes rojos**.

Corte verde

Corte rojo

Ejemplo 6.4. *Consideremos un ejemplo de corte verde. Si en el ejemplo anterior maria es una recién nacida, agregaríamos la cláusula recienNacido(maria) a nuestro programa. Entonces la meta ← orgulloso(X) nos diría que X/juan está orgulloso. Esto es, juan tiene una doble razón para estar orgulloso. Pero a nosotros nos basta con saber sólo una vez, que orgulloso está juan. Para evitar que Prolog nos de la respuesta dos veces, definiríamos:*

$$orgulloso(X) \leftarrow padre(X,Y), recienNacido(Y), !.$$

Ejemplo 6.5. *Ahora consideren un ejemplo de corte rojo:*

$$\begin{aligned} min(X,Y,X) &\leftarrow X < Y, !. \\ min(X,Y,Y). & \end{aligned}$$

Aparentemente nuestro programa es correcto. De hecho, el programa respondería de manera correcta a metas como ← min(2,3,X) respondiendo que "Si" para X/2; y para ← min(3,2,X) respondería que "Si" para X/2. Sin embargo el programa no es correcto. Consideren la meta ← min(2,3,3) y verán que Prolog respondería "Si". La razón de esto es que la segunda cláusula dice: el menor de X e Y es siempre Y. El corte está eliminando algunas ramas fallidas, que serían útiles en la definición de min. La definición correcta, usando corte, sería:

$$\begin{aligned} min(X,Y,X) &\leftarrow X < Y, !. \\ min(X,Y,Y) &\leftarrow X \geq Y. \end{aligned}$$

6.1.1 Caso de estudio

Prolog reconsidera automáticamente si esto es necesario para satisfacer una meta. Esto es útil en el sentido que le evita al programador contender explícitamente con la reconsideración, pero si no tenemos ninguna forma de control sobre el *backtracking*, éste puede volverse la causa de programas ineficientes. Comencemos por estudiar un programa muy simple que involucra reconsideración, para identificar los puntos donde el *backtracking* es inútil y condice a la ineficiencia.

Consideren una regulación sobre el nivel de alerta de la contaminación en una ciudad. La Figura 6.3 muestra la relación entre la concentración de los contaminantes en el aire X y el estado de la alarma Y a la manera de un semáforo. La relación entre X e Y se puede establecer mediante tres reglas:

REGLA 1. Si $X < 3$ entonces $Y = verde$

REGLA 2. Si $3 \leq X$ y $X < 6$ entonces $Y = amarilla$

REGLA 3. Si $6 \leq X$ entonces $Y = roja$

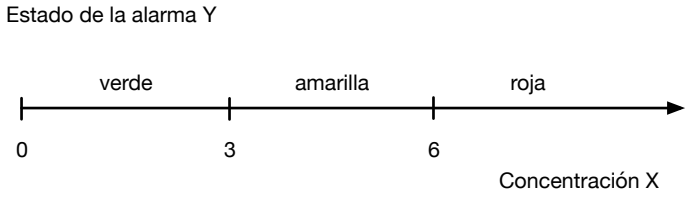


Figura 6.3: Estado de la alarma ambiental en función del nivel de contaminación.

Lo cual puede escribirse en Prolog de la siguiente manera:

```
% Concentración X de contaminantes en el aire
% f(X,Y), Y estado de la alarma

f(X,verde) :- X<3.     % Regla 1
f(X,amarilla) :- 3=< X, X<6.   % Regla 2
f(X,roja) :- 6 =< X.     % Regla 3
```

El programa asume que X será instanciada antes de la llamada a $f/2$, tal y como lo requieren los operadores de comparación utilizados en su definición. A continuación ejecutaremos dos experimentos, donde la ineficiencia de nuestra implementación se hará evidente; pero evitable con la ayuda del operador de corte.

Experimento 1

Supongamos que el nivel de contaminación en la ciudad es $X = 2$. Asumamos que el usuario del sistema no está muy al tanto de las normativas correspondientes y se pregunta si tal nivel de contaminación es inseguro y debiese generar una alarma amarilla. Para ello, podría preguntarle a Prolog si:

```
?- f(2,Y), Y=amarilla.
```

¿Cómo computa Prolog su respuesta? Al resolver la primer meta `f(2,Y)`, Y unificará con `verde`; de forma que la segunda meta toma la forma `verde=amarilla`, lo cual es falso y en consecuencia la lista completa de metas falla. Sin embargo, antes de rendirse, Prolog evaluará vía su reconsideración dos alternativas que resultan inútiles. Invoquen la meta bajo el modo de traza de Prolog para observar esto en detalle:

```
[trace]  ?- f(2,Y), Y=amarilla.
   Call: (9) f(2, _5824) ?
   Call: (10) 2<3 ?
   Exit: (10) 2<3 ?
   Exit: (9) f(2, verde) ?
   Call: (9) verde=amarilla ?
   Fail: (9) verde=amarilla ?
   Redo: (9) f(2, _5824) ?
   Call: (10) 3=<2 ?
   Fail: (10) 3=<2 ?
   Redo: (9) f(2, _5824) ?
   Call: (10) 6=<2 ?
   Fail: (10) 6=<2 ?
   Fail: (9) f(2, _5824) ?
false.
```

Las tres reglas de nuestro programa son mutuamente excluyentes, de forma que solo una de ellas tendrá éxito. Por ello, sabemos que tan pronto como una de ellas tiene éxito, no tiene caso intentar probar las otras, puesto que fallarán. Pero Prolog no sabe esto y para ello será necesario indicarle que no reconsidere en ciertos sitios del programa, en este caso, al final de las primeras dos reglas:

```
% Concentración X de contaminantes en el aire
% f(X,Y), Y estado de la alarma

f(X,verde) :-  X<3.     % Regla 1
f(X,amarilla) :-  3=< X, X<6.   % Regla 2
f(X,roja) :-  6 =< X.     % Regla 3
```

cuyo efecto en la traza del programa puede verse a continuación:

```
[trace]  ?- f(2,Y), Y=amarilla.
   Call: (9) f(2, _9422) ?
   Call: (10) 2<3 ?
   Exit: (10) 2<3 ?
   Exit: (9) f(2, verde) ?
   Call: (9) verde=amarilla ?
   Fail: (9) verde=amarilla ?
false.
```

Observen que el comportamiento de ambos programs, con y sin corte, es el mismo. La única diferencia es que el primero tardará un poco más en dar su respuesta. En estos casos se dice que el operador corte solo cambia el significado procedural del programa. Como veremos más adelante, éste no es siempre el caso.

Experimento 2

Ejecutemos otro experimento con la versión con cortes de nuestro programa. Preguntémosle la siguiente meta:

```
?- f(7,Y).
Y=roja
```

Revisemos la traza del programa:

```
[trace]  ?- f(7,Y).
   Call: (8) f(7, _10108) ?
   Call: (9) 7<3 ?
   Fail: (9) 7<3 ?
   Redo: (8) f(7, _10108) ?
   Call: (9) 3=<7 ?
   Exit: (9) 3=<7 ?
   Call: (9) 7<6 ?
   Fail: (9) 7<6 ?
   Redo: (8) f(7, _10108) ?
   Call: (9) 6=<7 ?
   Exit: (9) 6=<7 ?
   Exit: (8) f(7, roja) ?
Y = roja.
```

Las tres reglas son intentadas antes de poder computar la respuesta. Esto revela otra fuente de ineficiencia: algunos de los tests que se llevan a cabo son redundantes. Una tercer versión de nuestro programa sería como sigue:

```
% Tercer versión

f(X,verde) :- X<3, !.
f(X,amarilla) :- X<6, !.
f(_,roja).
```

Observen la traza de la meta:

```
[trace]  ?- f(7,Y).
   Call: (8) f(7, _13718) ?
   Call: (9) 7<3 ?
   Fail: (9) 7<3 ?
   Redo: (8) f(7, _13718) ?
   Call: (9) 7<6 ?
   Fail: (9) 7<6 ?
   Redo: (8) f(7, _13718) ?
   Exit: (8) f(7, roja) ?
Y = roja.
```

La respuesta es la misma, pero esta versión del programa es más eficiente que las dos anteriores. Pero, ¿Qué sucede si removemos los operadores de corte? Entonces la salida de la meta sería:

```
Y=verde;
Y=amarilla;
Y=roja;
false
```

Esto sugiere que a diferencia de nuestro segundo programa, los cortes aquí ¡cambian el resultado del programa! ¿A qué se debe la respuesta de nuestro tercer programa a las siguientes metas?

```
?- f(2,amarilla).
true
?- f(2,Y), Y=amarilla.
false
```

La traza de la primer meta explica lo sucedido:

```
[trace]  ?- f(2,amarilla).
   Call: (8) f(2, amarilla) ?
   Call: (9) 2<6 ?
```

```
   Exit: (9) 2<6 ?
   Exit: (8) f(2, amarilla) ?
true.
```

Si no tenemos cuidado con el corte, podemos cambiar la semántica de nuestros programas inadvertidamente.

Una descripción más precisa del funcionamiento del corte es como sigue: Llamaremos **meta padre** a la meta que unifica con la cabeza de la cláusula que contiene al operador corte. Cuando el corte es encontrado como una meta, tiene éxito inmediatamente, pero compromete al sistema con todas las elecciones hechas entre el momento en que la meta padre fue invocada y el momento en que el corte fue encontrado. Todas las alternativas entre el nodo padre y el corte son descartadas.

Meta padre

Ejemplo 6.6. *Consideren la siguiente cláusula:*

$$H : -B_1, B_2, \ldots, B_m, !, B_{m+2}, B_n$$

Asumamos que H unifica con la meta G, por lo que G es su meta padre. Al alcanzar el corte, la solución para las metas $B_1, \ldots B_m$ se fija, todas sus alternativas son descartadas. No así las metas $B_{m+2}, \ldots, B_n$.

6.1.2 Otros ejemplos

Implementaremos una función miembro determinista, e.g., solo computa una respuesta:

```
% single member

smember(X,[X|_]) :- !.
smember(X,[_|L]) :- smember(X,L).
```

De esta forma, su comportamiento es como sigue:

```
?- smember(X,[1,2,3]).
X=1;
false
```

A veces queremos agregar un elemento a una lista, solo en el caso de que éste no sea ya parte de ella. La siguiente función hace este trabajo:

```
% agregar sin duplicar add(+X,+L,-Y)

add(X,L,L) :- smember(X,L), !.
add(X,L,[X|L]).
```

Al igual que en el caso de *max*/3 se asume que el tercer argumento de *add*/3 no está instanciado en las llamadas. En caso contrario, su comportamiento es inesperado, por ejemplo:

```
?- add(a,[a],[a,a]).
true
```

Nuestro último ejemplo tiene que ver con clasificar objetos en categorías que cumplen ciertas condiciones y son excluyentes entre si. Por ejemplo, asumamos las siguientes relaciones entre equipos de futbol:

```
% clasificando en categorias

derrota(barcelona, real_madrid).
derrota(roma, barcelona).
derrota(cruz_azul, real_madrid).
```

La idea es clasificar a los equipos mediante una relación *clase*/2 siguiendo los siguientes criterios:

GANADOR. Es un equipo que gana todos sus encuentros.

LUCHADOR. Es un equipo que gana algunos de sus encuentros, pero pierde otros.

DEPORTISTA. Un equipo que pierde todos sus encuentros.

La definición de *clase*/2 en Prolog es como sigue:

```
% luchador es quien gana y pierde algunos partidos
% ganador es quien gana siempre
% deportista es quien pierde siempre

clase(X,luchador) :-
    derrota(X,_),
    derrota(_,X), !.

clase(X, ganador) :-
    derrota(X,_), !.

clase(X, deportista) :-
    derrota(_, X).
```

La consulta al programa es como sigue:

```
?- clase(barcelona,X).
X = luchador;
false
?- clase(barcelona, deportista).
false
```

6.2 NEGACIÓN POR FALLO FINITO

¿Cómo podemos expresar la siguiente proposición en Prolog: A Adriana le gustan todos los animales, excepto las serpientes? La primer parte de la proposición es sencilla:

```
    animal(X).
```

Pero la segunda parte es más complicada. Deberíamos expresar que si X es una serpiente, entonces es falso que X le gusta a *adriana*; y que en cualquier otro caso X le gusta a *adriana*. La traducción a Prolog es la siguiente:

```
% A Adriana le gustan todos los animales, excepto las
% serpientes

le_gusta(adriana,X) :-
    serpiente(X), !, fail.

le_gusta(adriana,X) :-
    animal(X).
```

De hecho, las dos cláusulas puede escribirse juntas de manera más compacta:

```
le_gusta(adriana,X) :-
  serpiente(X), !, fail
  ;
  animal(X).
```

Podemos usar la misma idea, para escribir un predicado $diferente/2$:

```
    X=Y, !, fail
    ;
    true.
```

Estos ejemplos sugieren que sería conveniente tener un predicado *not*/1 con la siguiente definición:

```
not(P) :-
  P,!,fail
  ;
  true.
```

Esta definición descansa enteramente en la semantica operacional de Prolog. Esto es, las sub-metas se deben resolver de izquierda a derecha, y las cláusulas se buscan en el orden en que aparecen en el texto del programa.

En lo que sigue asumiremos que *not*/1 es un predicado predefinido en Prolog definido como un operador prefijo. De hecho SWI-Prolog, define el operador $\backslash +$ para ello. De forma que los ejemplos anteriores, se pueden definir ahora de la siguiente forma:

```
1
2 le_gusta2(adriana,X) :-
3     animal(X),
4     \+ serpiente(X).
5
6 diferente2(X,Y) :-
7     \+ (X=Y).
```

6.2.1 CWA, NAF y corte

Los ejemplos de la sección anterior ilustran las ventajas y desventajas de usar el operador de corte. Las primeras incluyen:

1. Usando el operador de corte se puede mejorar la eficiencia de los programas lógicos. La idea es decirle a Prolog explícitamente –No intentes otra alternativa, pues están condenadas al fracaso.

2. Usando el operador de corte podemos definir cláusulas mutuamente exclusivas, por lo que podemos expresar reglas de la forma: Si la condición P entonces conclusión Q, y en cualquier otro caso R. Mejorando así la expresividad del lenguaje.

Las reservas con respecto al uso del operador de corte tienen que ver con que fácilmente podemos perder la correspondencia entre el significado declarativo de nuestros programas y su significado procedural. Esto es, si cambiamos el orden de las cláusulas de nuestro programa, sin usar el operador de corte, afectamos la eficiencia o las condiciones de terminación de éste, sin cambiar su significado declarativo; pero si las cláusulas usan corte, un cambio en su orden si que afecta éste significado. Veamos un ejemplo:

Ejemplo 6.7. *Si queremos expresar en Prolog* $p \Leftrightarrow (a \wedge b) \vee c$ *podemos usar el siguiente programa:*

```
1 p :- a,b.
2 p :- c.
```

Podemos cambiar el orden de las cláusulas y el significado declarativo del programa sigue siendo el mismo. Introduzcamos un corte:

```
1 p :- a, !, b.
2 p :- c.
```

El significado declarativo del programa es ahora $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$. *Pero si invertimos el orden de las cláusulas el significado declarativo se vuele* $p \Leftrightarrow c \vee (a \wedge b)$.

Como se mencionó, los casos en que el corte cambia el significado declarativo del programa, se denominan cortes rojos; en contraste con los cortes verdes que no lo hacen.

Ahora bien, la definición de la negación usando `fail` y corte, tiene un problema más serio: No se corresponde con la definición lógica de la negación. Consideren el siguiente ejemplo:

Ejemplo 6.8. *Consideren el siguiente programa de una sola cláusula:*

```
1     pintor(artur_heras).
```

Si le preguntamos al programa si `artur_heras` *es un* `pintor`*:*

```
?- pintor(artur_heras).
true
```

Lo cual es correcto pues esa respuesta se sigue del programa. Ahora, si preguntamos:

```
?- pintor(juan_gris).
false
```

Lo cual también es correcto puesto que no se sigue del programa que `juan_gris` *sea un pintor. Ahora:*

```
?- \+  pintor(juan_gris).
true
```

En este caso, el true de Prolog no significa que la meta se siga lógicamente del programa.

La última respuesta de Prolog se basa en el **supuesto del mundo cerrado** (CWA). *CWA*
i.e., un programa Prolog representa todo lo que es verdadero en el mundo que describe. De manera que aquello que no esté declarado en el programa, o sea derivable de éste lógicamente, se asume como falso; y su negación en consecuencia se asume verdadera. El CWA puede formularse como una pseudo-regla de inferencia que expresa:

$$\frac{\Delta \nvdash \alpha}{\neg\alpha} \quad (CWA)$$

Si una fbf atómica de base (sin variables) α, no puede derivarse del programa Δ siguiendo las reglas de inferencia del sistema, entonces puede derivarse $\neg\alpha$. En el caso de los sistemas correctos y completos, la condición $\Delta \nvdash \alpha$ es equivalente a $\Delta \not\models \alpha$. Como este es el caso para la resolución-SLD, la condición puede ser remplazada por $\alpha \notin M_\Delta$.

Ejemplo 6.9. *Dado el siguiente programa lógico:*

$$\begin{aligned} sobre(X,Y) &\leftarrow en(X,Y). \\ sobre(X,Y) &\leftarrow en(X,Z), sobre(Z,Y). \\ en(c,b). & \\ en(b,a). & \end{aligned}$$

la fbf $sobre(b,c)$ *no puede ser derivada por resolución-SLD a partir del programa* Δ *(vean el árbol de derivación en la Figura 6.4). En realidad* $sobre(b,c)$ *no puede ser derivada por ningún sistema correcto, puesto que no es una consecuencia lógica de* Δ*. Dada la completitud de la resolución-SLD, se sigue que* $\Delta \not\models sobre(b,c)$ *y usando la CWA inferimos que* $\neg sobre(b,c)$*.*

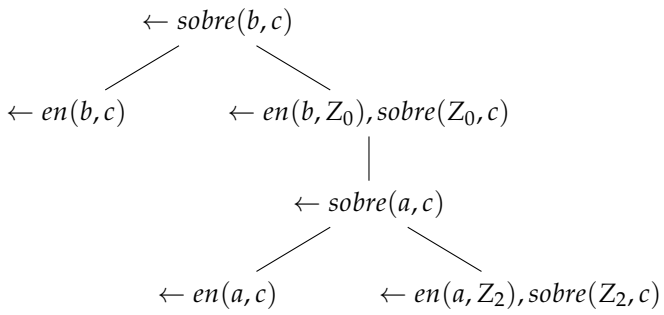


Figura 6.4: Árbol de derivación-SLD fallido

En contra de lo que podría ser nuestra primera intuición, existen problemas asociados a la CWA. Uno de ellos es que en la vida cotidiana existen muchas situaciones en donde el CWA no se puede asumir.

Otro problema, técnico en este caso, tiene que ver con que establecer que una meta no es derivable dado un programa definitivo es no decidible en el caso general. Esto es, no es posible determinar si la pseudo-regla asociada al CWA aplica o no. Una versión más débil de la suposición de mundo cerrado, se logra si asumimos que $\neg\alpha$ es derivable a partir del programa Δ si la meta $\leftarrow \alpha$ tiene un árbol-SLD finito que falla, i.e., NAF.

Es necesario contrastar la NAF con el CWA, que también puede verse como una negación por falla, pero infinita. Para ilustrar la diferencia entre los dos enfoques extendamos el programa Δ con la siguiente cláusula evidentemente verdadera $sobre(X,Y) \leftarrow sobre(X,Y)$.

Ejemplo 6.10. *El árbol-SLD de la meta $\leftarrow sobre(b,c)$ sigue sin contener refutaciones, pero ahora es infinito. Por lo tanto no podemos concluir que $\neg sobre(b,c)$ usando NAF, pero si usando CWA.*

Otro aspecto a considerar es el efecto de la negación en los cuantificadores que están bajo su alcance. Veamos un ejemplo:

Ejemplo 6.11. *Consideren el siguiente programa acerca de restaurantes:*

```
estrellas(ricard_camarena).

caro(diverxo).

razonable(X) :-
    \+ caro(X).
```

Una meta de interés sería:

```
?- estrellas(X), razonable(X).
X = ricard_camarena.
```

Ahora bien, si preguntamos aparentemente la misma meta:

```
?- razonable(X), estrellas(X).
false.
```

Usen la traza para entender este comportamiento de Prolog... y siempre prioricen calidad sobre precio :-)

Como recordarán, la diferencia entre estas dos metas que, aparentemente equivalentes, es que en el primer caso, la variable X está instanciada cuando $razonable/1$ es invocada; mientras que en el segundo caso esto no es así.

La cuestión es que la negación cambia la cuantificación de la variable bajo su alcance. La meta $caro(X)$ significa ¿Existe una X tal que $caro(X)$ es verdadera? Si ese es el caso ¿Cual X? La meta $notcaro(X)$ significa ¿Verdad que para todo X, no es el caso que $caro(X)$? ó ¿Es cierto que nadie es caro? En el ejemplo anterior, dado que $diverxo$ es caro, la respuesta es $false$. En general $notMeta$ es seguro si todas las variables en $Meta$ están instanciadas.

EL objetivo de advertir esos problemas con el operador de corte, e indirectamente con la negación, no es desalentar su uso. Por el contrario, pueden ser de gran utilidad. La cuestión es usarlos de manera adecuada, atendiendo sus singularidades.

Existen dos aproximaciones a la solución de estos problemas: ver los programas como resúmenes de programas más extensos que validan las literales negativas; o redefinir la noción de consecuencia lógica de forma que sólo algunos modelos del programa (el mínimo de Herbrand, por ejemplo) sean tomados en cuenta. En ambos casos, el efecto es descartar algunos modelos del programa que no son interesantes. Primero justificaremos la regla NAF en términos de la compleción de los programas definitivos y posteriormente, extenderemos el lenguaje de los programas definitivos para incluir en ellos literales negativas en la cabeza y cuerpo de las cláusulas.

6.3 LA COMPLECIÓN DE UN PROGRAMA

La idea que presentaremos a continuación se debe a Clark [22] y se basa en que cuando uno escribe un programa definitivo Δ, en realidad quiere expresar algo más que su conjunto de cláusulas definitivas. El programa deseado puede formalizarse como la compleción de Δ. Consideren la siguiente definición:

$$\begin{aligned} sobre(X,Y) &\leftarrow en(X,Y). \\ sobre(X,Y) &\leftarrow en(X,Z), sobre(Z,Y). \end{aligned}$$

Estas reglas especifican que un objeto está sobre un segundo objeto, si el primer objeto está encima del segundo (1) ó si el objeto está sobre otro objeto que a su vez está encima del segundo (2). Esto también puede escribirse como:

$$sobre(X,Y) \leftarrow en(X,Y) \vee (en(X,Z), sobre(Z,Y))$$

Ahora, ¿Qué sucede si remplazamos la implicación por la equivalencia lógica?

$$sobre(X,Y) \leftrightarrow en(X,Y) \vee (en(X,Z), sobre(Z,Y))$$

Está fbf expresa que X está sobre Y si y sólo si una de las condiciones es verdadera. Esto es, si ninguna de las condiciones se cumple, ¡se sigue que X no está sobre Y! Esta es la intuición seguida para justificar la negación por fallo.

Desafortunadamente, combinar cláusulas definitivas como en el ejemplo anterior, sólo es posible para cláusulas con cabezas idénticas. Por ejemplo:

$$\begin{aligned} &en(c,b). \\ &en(b,a). \end{aligned}$$

Por una simple transformación, el programa puede ser escrito como:

$$\begin{aligned} en(X_1,X_2) &\leftarrow X_1 = c, X_2 = b \\ en(X_1,X_2) &\leftarrow X_1 = b, X_2 = a \end{aligned}$$

Las cláusulas pueden combinarse en una sola fórmula, donde la implicación es remplazada por la equivalencia lógica.

$$en(X_1,X_2) \leftrightarrow (X_1 = c, X_2 = b) \vee (X_1 = b, X_2 = a)$$

La lectura lógica de esta fbf es que X_1 está en X_2 si y sólo si $X_1 = c$ y $X_2 = b$ o si $X_1 = b$ y $X_2 = a$. Esta transformación se puede realizar sobre un programa lógico definitivo Δ y el resultado se conoce como **compleción** de Δ.

Compleción

Definición 6.1 (Compleción). *Sea Δ un programa lógico definitivo. La compleción $comp(\Delta)$ de Δ es el conjunto de fórmulas obtenido a partir de las siguientes tres transformaciones:*

1. *Para cada símbolo de predicado ϕ remplazar la cláusula α de la forma:*

   $$\phi(t_1,\ldots,t_m) \leftarrow \alpha_1,\ldots,\alpha_n \quad (n \geq 0)$$

   por la fórmula:

   $$\phi(X_1,\ldots,X_m) \leftarrow \exists Y_1,\ldots,Y_i(X_1 = t_1,\ldots,X_m = t_m,\alpha_1,\ldots,\alpha_n)$$

   donde las Y_i son todas variables en α y las X_i son variables únicas que no aparecen en α.

2. *Para cada símbolo de predicado ϕ remplazar todas las fbf:*

$$\begin{aligned}\phi(X_1,\ldots,X_m) &\leftarrow \beta_1\\ &\vdots\\ \phi(X_1,\ldots,X_m) &\leftarrow \beta_j\end{aligned}$$

por la fórmula:

$$\begin{aligned}\forall X_1,\ldots,X_m(\phi(X_1,\ldots,X_m) \leftrightarrow \beta_1\vee,\ldots,\vee\beta_j \quad si \quad j>0\\ \forall X_1,\ldots,X_m(\neg\phi(X_1,\ldots,X_m)) \quad si \quad j=0\end{aligned}$$

3. *Finalmente el programa se extiende con los siguientes axiomas de igualdad libre, que definen las igualdades introducidas en el paso 1:*

$$\begin{aligned}\forall(X=X)\\ \forall(X=Y\Rightarrow Y=X)\\ \forall(X=Y\wedge Y=Z\Rightarrow X=Z)\\ \forall(X_1=Y_1\wedge\cdots\wedge X_n=Y_n\Rightarrow f(X_1,\ldots,X_n)=f(Y_1,\ldots,Y_n))\\ \forall(X_1=Y_1\wedge\cdots\wedge X_n=Y_n\Rightarrow(\phi(X_1,\ldots,X_n)\Rightarrow\phi(Y_1,\ldots,Y_n))\\ \forall(f(X_1,\ldots,X_n)=f(Y_1,\ldots,Y_n)\Rightarrow X_1=Y_1\wedge\cdots\wedge X_n=Y_n)\\ \forall(\neg f(X_1,\ldots,X_m)=g(Y_1,\ldots,Y_n))(\text{Si } f/m\neq g/n)\\ \forall(\neg X=t)(\text{Si } X \text{ es un subtermino propio de } t)\end{aligned}$$

Estas definiciones garantizan que la igualdad (=) sea una relación de equivalencia; que sea una relación congruente; y que formalice la noción de unificación. Las primeros cinco definiciones se pueden abandonar si se especifica que = representa la relación de **identidad**. *Identidad*

Ejemplo 6.12. *Consideremos la construcción de $comp(\Delta)$ tal y como se definió anteriormente. El primer paso produce:*

$$\begin{aligned}sobre(X_1,X_2) &\leftarrow \exists X,Y\ (X_1=X,X_2=Y,en(X,Y))\\ sobre(X_1,X_2) &\leftarrow \exists X,Y,Z\ (X_1=X,X_2=Y,en(Z,Y),sobre(Z,Y))\\ en(X_1,X_2) &\leftarrow (X_1=c,X_2=b)\\ en(X_1,X_2) &\leftarrow (X_1=b,X_2=a)\end{aligned}$$

dos pasos más adelante obtenemos:

$$\begin{aligned}\forall X_1,X_2(sobre(X_1,X_2 &\leftrightarrow \exists X,Y(\ldots)\wedge\exists X,Y,Z(\ldots))\\ \forall X_1,X_2(en(X_1,X_2) &\leftrightarrow (X_1=c,X_2=b)\wedge(X_1=b,X_1=a))\end{aligned}$$

y el programa se termina con las definiciones de igualdad como identidad y unificación.

La compleción $comp(\Delta)$ de un programa definitivo Δ preserva todas las literales positivas modeladas por Δ. Esto es, si $\Delta\models\alpha$ entonces $comp(\Delta)\models\alpha$. Tampoco se agrega información positiva al completar el programa: Si $comp(\Delta)\models\alpha$ entonces $\Delta\models\alpha$. Por lo tanto, al completar el programa no agregamos información positiva al mismo, solo información negativa.

Como sabemos, no es posible que una literal negativa pueda ser consecuencia lógica de un programa definitivo. Pero al substituir las implicaciones en Δ por equivalencias en $comp(\Delta)$ es posible inferir información negativa a partir del programa completado. Esta es la justificación de la regla NAF, cuyas propiedades de consistencia se deben a Clark [22]:

Teorema 6.1 (Consistencia de la NAF)**.** *Sea Δ un programa definitivo y $\leftarrow \alpha$ una meta definitiva. Si $\leftarrow \alpha$ tiene un árbol-SLD finito fallido, entonces $comp(\Delta) \models \forall(\neg\alpha)$.*

La consistencia se preserva aún si α no es de base. Por ejemplo, $\leftarrow en(a, X)$ falla de manera finita y por lo tanto, se sigue que $comp(\Delta) \models \forall(\neg en(a, X))$. La completitud de la NAF también ha sido demostrada:

Teorema 6.2 (Completitud de la NAF)**.** *Sea Δ un programa definitivo. Si $comp(\Delta) \models \forall(\neg\alpha)$ entonces existe un árbol finito fallido para la meta definitiva $\leftarrow \alpha$.*

Observen que solo enuncia la existencia de un árbol-SLD finito fallido. Como se ha mencionado, un árbol-SLD puede ser finito bajo ciertas reglas de computación e infinito bajo otras. En particular, el teorema de completitud no es válido para las reglas de computación de Prolog. La completitud funciona para una subclase de derivaciones-SLD conocidas como **justas** (*fair*), las cuales o bien son finitas o garantizan que cada átomo en la derivación (u ocurrencia de éste), es seleccionado eventualmente por las reglas de computación. Un árbol-SLD es justo si todas sus derivaciones son justas. La NAF es completa para árboles-SLD justos. Este tipo de derivaciones se pueden implementar fácilmente: selecciona la sub-meta más a la izquierda y agrega nuevas submetas al final de esta (búsqueda en amplitud). Sin embargo, pocos sistemas implementan tal estrategia por razones de eficiencia.

Derivación justa

6.4 RESOLUCIÓN SLDNF PARA PROGRAMAS DEFINITIVOS

En el capítulo 5.4 presentamos el método de resolución-SLD, utilizado para probar si una literal positiva cerrada es consencuencia lógica de un programa. En la sección anterior afirmamos que también las literales negadas pueden derivarse a partir de la terminación de programas lógicos definitivos. Combinando la resolución SLD y la negación como fallo finito (NAF), es posible generalizar la noción de meta definitiva para incluir literales positivas y negadas. Tales metas se conocen como generales.

Definición 6.2 (Meta general)**.** *Una meta general tiene la forma:*

$$\leftarrow \alpha_1, \ldots \alpha_n \quad (n \geq 0)$$

donde cada α_i es una literal positiva o negada.

La combinación de la resolución SLD y la NAF se llama resolución SLDNF.

Definición 6.3 (Resolución SLDNF para programas definitivos)**.** *Sea Δ un programa definitivo, G_0 una meta general y $\mathcal{R}$ una función de selección (también conocida como regla de computación). Una derivación SLDNF de G_0 usando $\mathcal{R}$, es una secuencia finita o infinita de metas generales:*

$$G_0 \overset{\alpha_0}{\rightsquigarrow} G_1 \ldots G_{n-1} \overset{\alpha_{n-1}}{\rightsquigarrow} G_n$$

donde $G_i \overset{\alpha_i}{\rightsquigarrow} G_{i+1}$ puede ocurrir si:

1. *la literal $\mathcal{R}$-seleccionada en G_i es positiva y G_{i+1} se deriva de G_i y α_i por un paso de resolución SLD;*
2. *la literal $\mathcal{R}$-seleccionada en G_i es negativa ($\neg\alpha$) y la meta $\leftarrow \alpha$ tiene un árbol SLD fallido y finito y G_{i+1} se obtiene a partir de G_i eliminando $\neg\alpha$ (en cuyo caso α_i, corresponde al marcador especial FF).*

Cada paso en una derivación SLDNF produce una substitución, en el caso 1 un MGU y en el caso 2, la substitución vacía ϵ.

Entonces, una literal negativa $\neg\alpha$ es demostrada si $\leftarrow \alpha$ tiene un árbol SLD finito que falla. Por dualidad, $\neg\alpha$ falla de manera finita si α es demostrada. Además de la refutación y de la derivación infinita, existen dos clases de derivaciones SLDNF completas dada una función de selección:

1. Una derivación se dice (finitamente) **fallida** si (i) la literal seleccionada es positiva y no unifica con ninguna cabeza de las cláusulas del programa, o (2) la literal seleccionada es negativa y tiene un fallo finito.
2. Una derivación se dice **plantada** (*stuck*) si la sub-meta seleccionada es de la forma $\neg\alpha$ y $\leftarrow \alpha$ tiene un fallo infinito.

Ejemplo 6.13. *Considere el siguiente programa:*

$$en(c,b)$$
$$en(b,a)$$

La meta $\leftarrow en(X,Y), \neg en(Z,X)$ *tiene una refutación-SLDNF con la substitución computada {X/c,Y/b}:*

$$\begin{aligned} G &= \leftarrow en(X,Y), \neg en(Z,X). \\ G_0 &= \leftarrow en(X,Y). \\ \alpha_0 &= en(c,b). \\ \theta_0 &= \{X/c, Y/b\} \end{aligned}$$

$$\begin{aligned} G_1 &= \neg en(Z,X)\theta_0 = \leftarrow en(Z,c) \\ \alpha_1 &= FF \\ \theta_1 &= \epsilon \\ G_2 &= \square \end{aligned}$$

$$\theta = \theta_0\theta_1 = \{X/c, Y/b\}$$

En cambio, si la función de selección hubiera computado las cláusulas de abajo hacía arriba $\alpha_0 = en(b,a)$ *la derivación hubiera sido fallida (a ustedes probarlo).*

Como es de esperarse, la resolución-SLDNF es consistente, después de todo, la resolución-SLD y la NAF son consistentes.

Teorema 6.3 (Consistencia de la resolución-SLDNF)**.** *Sea* Δ *un programa definitivo y* $\leftarrow \alpha_1, \ldots, \alpha_n$ *una meta general. Si* $\leftarrow \alpha_1, \ldots, \alpha_n$ *tiene una refutación SLDNF con una substitución computada* θ, $comp(\Delta) \models \forall(\alpha_1\theta, \ldots, \alpha_n\theta)$.

Sin embargo, la resolución-SLDNF no es completa aunque pudiéramos haber esperado lo contrario. La resolución SLDNF no es completa a pesar de que la resolución-SLD y la NAF si lo son. Un simple contra ejemplo es $\leftarrow \neg en(X,Y)$ que corresponde a la consulta "¿Hay algunos bloques X e Y, tal que X no está en Y?" Uno esperaría varias respuestas a esta consulta, por ejemplo, el bloque *a* no está encima de ningún bloque, etc.

Pero la derivación SLDNF de $\leftarrow \neg en(X,Y)$ falla porque la meta $\leftarrow en(X,Y)$ tiene éxito (puede ser demostrada). El problema es que nuestra definición de derivación fallida es demasiado conservadora. El éxito de $\leftarrow en(X,Y)$ no significa necesariamente que no halla un bloque que no esté en otro bloque, sólo que existe al menos un bloque que no está en otro.

El problema tiene su origen en que la NAF, en contraste con la resolución SLD, es sólo una prueba (*test*). Recuerden que dada la definición de la resolución SLDNF y la consistencia y completitud de la NAF, tenemos que $\neg en(X,Y)$ tiene éxito si y sólo si ($\equiv$) $en(X,Y)$ tiene asociado un árbol SLD fallido y finito; o si y sólo si $comp(\Delta) \models \forall(\neg en(X,Y))$. Por lo tanto, la meta general $\leftarrow en(X,Y)$ no debe leerse como una consulta cuantificada existencialmente, sino como una prueba universal: "Para todo bloque X e Y, ¿No está X en Y?".

Esta última consulta tiene una respuesta negativa en el modelo deseado del programa, puesto que el bloque b está en el bloque a. El problema anterior se debe a la cuantificación de las variables en la literal negativa. Si replanteamos la consulta anterior como $\leftarrow \neg en(a,b)$ entonces la resolución SLDNF alcanza una refutación puesto que $\leftarrow en(a,b)$ falla con una derivación finita.

Algunas veces se asume que la función de selección $\mathcal{R}$ permite seleccionar una literal negativa $\neg\alpha$ si la literal α no tiene variables libres o si α tiene asociada una substitución computada vacía. Estas funciones de selección se conocen como seguras (*safe*).

6.5 PROGRAMAS LÓGICOS GENERALES

Con los desarrollos anteriores estamos en posición de extender el lenguaje de los programas definitivos para incluir cláusulas que contienen literales tanto positivas como negativas en su cuerpo. Estas fbf se llaman **cláusulas generales**, y en consecuencia, un conjunto de ellas conforma un programa general. En ocasiones, a los programas generales se les conoce a veces como **programas normales**.

Cláusulas Generales

Programas Normales

Definición 6.4 (Cláusula General)**.** *Una cláusula general es una fbf de la forma* $\alpha \leftarrow \alpha_1, \ldots, \alpha_n$ *donde* α *es una fbf atómica y* $\alpha_1, \ldots, \alpha_n$ *son literales* ($n \geq 0$)*.*

Definición 6.5 (Programa General)**.** *Un programa lógico general es un conjunto finito de cláusulas generales.*

Ahora podemos extender nuestro programa del mundo de los bloques con las siguientes relaciones:

$$\begin{aligned} base(X) &\leftarrow en(Y,X), en\_la\_mesa(X). \\ en\_la\_mesa(X) &\leftarrow \neg no\_en\_la\_mesa(X). \\ no\_en\_la\_mesa(X) &\leftarrow en(X,Y). \\ en(c,b). & \\ en(b,a). & \end{aligned}$$

La primer cláusula especifica que un bloque es base si está sobre la mesa y tiene otro bloque encima. La segunda cláusula indica que cuando no es cierto que un bloque no está sobre la mesa, entonces está sobre la mesa. La tercera cláusula especifica que un bloque que está sobre otro, no está sobre la mesa.

Parece claro, pero la pregunta que deberíamos hacernos es qué tipo de sistema de prueba queremos para los programas lógicos generales y cuales serán las aproximaciones lógicas a las sutilezas, algunas ya discutidas, introducidas por este tipo de lenguajes.

Observen que aunque el lenguaje fue enriquecido, no es posible de cualquier forma que una literal negativa sea consecuencia lógica de un programa dado. La razón es la misma que para los programas definidos, la base de Herbrand de un programa Δ, B_Δ es un modelo de Δ en el que todas las literales negativas son falsas. Al igual que con los programas definidos, la pregunta es entonces como lograr inferencias negativas consistentes. Afortunadamente el concepto de compleción de programa puede aplicarse también a los programas lógicos generales.

Ejemplo 6.14. *La compleción de* $gana(X) \leftarrow mueve(X,Y), \neg gana(Y)$ *contiene la fbf:*

$$\forall X_1(gana(X_1) \leftrightarrow \exists X, Y(X_1 = X, mueve(X,Y), \neg gana(Y)))$$

Desafortunadamente, la compleción de los programas normales puede ocasionar paradojas.

Ejemplo 6.15. *Consideren la cláusula general* $p \leftarrow \neg p$*, su compleción incluye* $p \leftrightarrow \neg p$*. La inconsistencia del programa terminado se debe a que* $p/0$ *está definida en términos de su propio complemento.*

Una estrategia de programación para evitar este problema consiste en componer los programas por capas o estratos, forzando al programador a referirse a las negaciones de una relación hasta que ésta ha sido totalmente definida. Se entiende que tal definición se da en un estrato inferior a donde se presenta la negación. En la definición del **programa estratificado** usaremos Δ^p para referirnos al subconjunto de cláusulas en Δ que tienen a p como cabeza.

Programa Estratificado

Definición 6.6 (Programa Estratificado)**.** *Un programa general Δ se dice estratificado si y sólo si existe al menos una partición $\Delta_1 \cup \cdots \cup \Delta_n$ de Δ tal que :*

1. *Si $p(\ldots) \leftarrow q(\ldots), \cdots \in \Delta_i$ entonces $\Delta^q \subseteq \Delta_1 \cup \cdots \cup \Delta_i$;*
2. *Si $p(\ldots) \leftarrow \neg q(\ldots), \cdots \in \Delta_i$ entonces $\Delta^q \subseteq \Delta_1 \cup \cdots \cup \Delta_{i-1}$.*

Por ejemplo, el siguiente programa está estratificado:

Δ_2: $base(X) \leftarrow en(Y, X), en\_la\_mesa(X).$
$en\_la\_mesa(X) \leftarrow \neg no\_en\_la\_mesa(X).$
Δ_1: $no\_en\_la\_mesa(X) \leftarrow en(X, Y).$
$en(c, b).$
$en(b, a).$

Apt, Blair y Walker [2] demostraron que la completión de un programa estratificado en consistente, de forma que la situación descrita anteriormente no puede ocurrir. Sin embargo, la estratificación es solo una condición suficiente para la **consistencia**: Determinar si un programa está estratificado es decidible, pero determinar si la completión de un programa general es consistente, es indecidible. Por lo tanto, hay programas generales no estratificados cuya completión es consistente.

Consistencia

6.6 RESOLUCIÓN–SLDNF PARA PROGRAMAS GENERALES

Hemos revisado el caso de la resolución-SLDNF entre programas definitivos y metas generales. Informalmente podemos decir que la resolución-SLDNF combina la resolución-SLD con los siguientes principios:

1. $\neg\alpha$ tiene éxito ssi $\leftarrow \alpha$ tiene un árbol-SLD finito que falla.
2. $\neg\alpha$ falla finitamente ssi $\leftarrow \alpha$ tiene una refutación-SLD.

El paso de programas definitivos a programas generales, es más complicado. Para probar $\neg\alpha$, debe de existir un árbol finito fallido para $\leftarrow \alpha$. Tal árbol puede contener nuevas literales negativas, las cuales a su vez deben pueden tener éxito o fallar finitamente. Esto complica un poco la definición de la resolución-SLDNF para programas generales.

Ejemplo 6.16. *Es posible llegar a situaciones paradójicas cuando los predicados están definidos en términos de sus propios complementos. Consideren el programa no estratificado:*

$$\alpha \leftarrow \neg\alpha$$

Dada la meta inicial $\leftarrow \alpha$, se puede construir una derivación $\leftarrow \alpha \rightsquigarrow \leftarrow \neg\alpha$. La derivación puede extenderse hasta una refutación si $\leftarrow \alpha$ falla finitamente. De manera alternativa, si $\leftarrow \alpha$ tiene una refutación, entonces la derivación falla. Helas! esto es imposible pues la meta $\leftarrow \alpha$ no puede tener una refutación y fallar finitamente al mismo tiempo.

En lo que sigue, definiremos las nociones de derivación-SLDNF y árbol-SLDNF, de manera similar a la derivación-SLD y a los arboles-SLD. La idea se concreta en el

concepto de **bosque-SLDNF**: Un conjunto de árboles cuyos nodos está etiquetados con metas generales. Un **sub-bosque** se obtiene removiendo algunos nodos y sus hijos del bosque original. Dos sub-bosques B_1 y B_2 se consideran equivalentes si contieinen los mismos árboles considerando un posible renombrado de variables. B_1 es más pequeño que B_2 si es equivalente a algún sub-bosque de B_2.

Bosque-SLDNF
Sub-Bosque

Definición 6.7 (Bosque-SLDNF)**.** *Sea Δ un programa general, G_0 una meta general, y $\mathcal{R}$ una función de selección. El bosque-SLDNF de G_0 es el bosque más pequeño (considerando el posible renombrado de variables), tal que:*

1. *G_0 es la raíz del árbol.*
2. *Si G es un nodo en el bosque cuya literal seleccionada es positiva, entonces para cada cláusula α tal que G' puede ser derivada de G y α (con UMG θ), G tiene un hijo etiquetado como G'. Si no existe tal cláusula, entonces G tiene un hijo etiquetado* **FF** *(falla finita);*
3. *Si G es un nodo del bosque cuya literal seleccionada es de la forma $\neg\alpha$ (G es de la forma $\leftarrow \alpha_1, \ldots, \alpha_{i-1}, \neg\alpha, \alpha_{i+1}, \ldots, \alpha_n$), entonces:*
   - *El bosque contiene un árbol cuya raíz está etiquetada como $\leftarrow \alpha$;*
   - *Si el árbol con raíz $\leftarrow \alpha$ tiene una hoja $\square$ con la substitución computada ϵ, entonces G tiene un sólo hijo etiquetado* **FF***;*
   - *Si el árbol con raíz $\leftarrow \alpha$ es finito y tiene todas sus hojas etiquetadas* **FF***, entonces G tiene un sólo hijo (con substitución asociada ϵ) etiquetado como $\leftarrow \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n$.*

Observen que la literal negativa seleccionada $\neg\alpha$ falla sólo si $\leftarrow \alpha$ tiene una refutación con la substitución computada ϵ. Como veremos más adelante, esta condición que no era necesaria cuando definimos la resolución-SLDNF para programas definitivos, es vital para la **correctez** de la resolución en los programas generales.

Correctez

Los arboles del bosque-SLDNF son llamados **arboles-SLDNF** (completos); y la secuencia de todas las metas en una rama de un árbol-SLDNF con raíz G es llamada derivación-SLDNF completa de G (bajo un programa Δ y una función de selección $\mathcal{R}$). El árbol etiquetado por G_0 es llamado árbol principal. Un árbol con la raíz $\leftarrow \alpha$ es llamado árbol subsidiario si $\neg\alpha$ es una literal seleccionada en el bosque (el árbol principal puede ser a su vez subsidiario).

Arboles-SLDNF

Ejemplo 6.17. *Consideren el siguiente programa general estratificado Δ:*

$$
\begin{aligned}
base(X) &\leftarrow en(Y,X), en\_la\_mesa(X).\\
en\_la\_mesa(X) &\leftarrow \neg no\_en\_la\_mesa(X).\\
no\_en\_la\_mesa(X) &\leftarrow en(X,Y).\\
encima(X,Y) &\leftarrow en(X,Y).\\
encima(X,Y) &\leftarrow en(X,Z), encima(Z,Y).\\
en(c,b).&\\
en(b,a).&
\end{aligned}
$$

El bosque-SLDNF para la meta $\leftarrow base(X)$ se muestra en la Figura 6.5. El árbol principal contiene una derivación fallida y una refutación con la substitución computada $\{X/a\}$.

Las ramas de un árbol-SLDNF en un bosque-SLDNF representan todas las derivaciones-SLDNF completas de su raíz, con base en la función de selección dada. Hay cuatro clases de derivaciones-SLDNF completas:

1. Derivaciones infinitas;
2. Derivaciones finitas fallidas (terminan en **FF**);

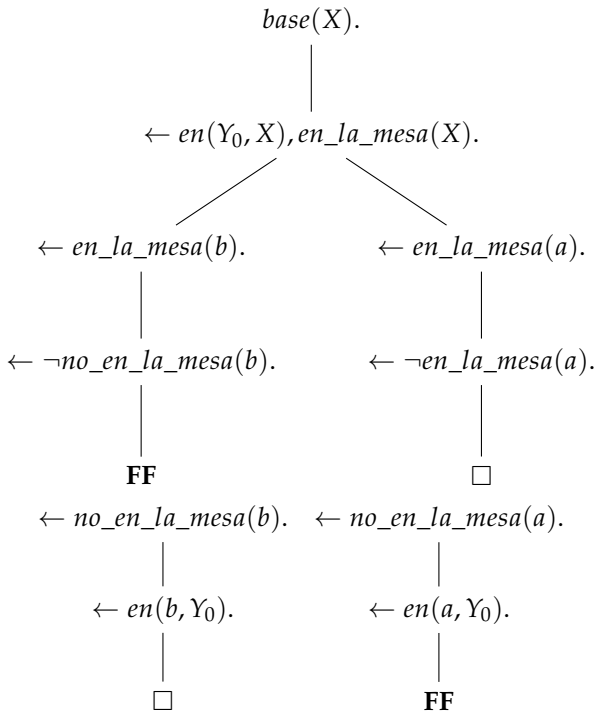


Figura 6.5: Bosque-SLDNF para la meta $\leftarrow base(X)$.

3. Refutaciones (terminan en $\square$); y
4. Derivaciones plantadas (si ninguno de los casos anteriores aplica).

Ejemplo 6.18. *Consideren el siguiente programa:*

$$\begin{aligned} termina(X) &\leftarrow \neg ciclo(X). \\ ciclo(X) &\leftarrow ciclo(X). \end{aligned}$$

El bosque-SLDNF para el ejemplo anterior se muestra en la Figura 6.6. El bosque incluye una derivación plantada para $termina(X)$ y una derivación infinita para $ciclo(X)$. Esto ilustra una de las razones por las cuales una derivación se planta: uno de sus arboles subsidiarios contiene sólo derivaciones fallidas o infinitas.

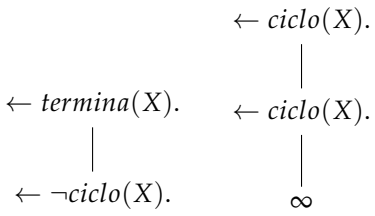


Figura 6.6: Bosque-SLDNF para la meta $\leftarrow paro(X)$.

El siguiente programa también conduce a una derivación plantada (ciclo en el cómputo de la negación):

$$\begin{aligned} paradoja(X) &\leftarrow \neg ok(X). \\ ok(X) &\leftarrow \neg paradoja(X). \end{aligned}$$

Intenten construir el bosque-SLDNF de este programa y observaran también que en este caso, la árbol principal es a su vez un árbol subsidiario.

La última razón para que una derivación quede plantada es ilustrada por el siguiente programa:

$$\begin{array}{rcl} tope(X) & \leftarrow & \neg bloqueado(X). \\ bloqueado(X) & \leftarrow & en(Y, X). \\ en(a, b). & & \end{array}$$

Es evidente que $tope(a)$ debería poder derivarse de este programa. Sin embargo, el árbol-SLDNF de la meta $\leftarrow tope(X)$ no contiene refutaciones. De hecho, esta meta se planta aún cuando $\leftarrow bloqueado(X)$ tiene una refutación. La razón de esto es que $\leftarrow bloqueado(X)$ no tiene ninguna derivación que termine con una substitución computada vacía. A la meta $\leftarrow \neg tope(X)$, Prolog no responde b, sino que ¡no todos los bloques están en el tope de la pila! De manera que la implementación de la resolución-SLDNF en la mayoría de los Prolog no es robusta, aunque nuestra definición si lo es.

Teorema 6.4 (Correctez de la resolución-SLDNF)**.** *Sea Δ un programa general y $\leftarrow \alpha_1, \ldots, \alpha_n$ una meta general. Entonces:*

- *Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene una substitución de respuesta computada θ, entonces $comp(\Delta) \models \forall(\alpha_1\theta \wedge \cdots \wedge \alpha_n\theta)$.*
- *Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene un árbol-SLDNF finito que falla, entonces $comp(\Delta) \models \forall(\neg(\alpha_1 \wedge \cdots \wedge \alpha_n))$.*

La definición de bosque-SLDNF no debe verse como una implementación de la resolución-SLDNF, sólo representa el espacio ideal de computación donde la correctez puede ser garantizada. Nada se ha dicho en cuanto al orden en que el bosque debe construirse. Al igual que en los demás casos, Prolog sigue una estrategia primero en profundidad: Ante una meta $\neg\alpha$, Prolog suspende la construcción del árbol de resolución en espera de la respuesta a la meta $\leftarrow \alpha$. El hecho de que Prolog no verifique si la substitución de respuesta de una refutación fue ϵ, al igual que la ausencia de chequeo de ocurrencias, contribuyen a que Prolog no sea robusto.

6.7 LECTURAS Y EJERCICIOS SUGERIDOS

La discusión presentada aquí sobre corte y negación por fallo finito en Prolog, se basa en el libro de Bratko [18]. Los aspectos generales más técnicos sobre estos temas se basan en el libro de Nilsson y Maluszynski [77]. Apt y Bol [3] nos ofrecen una revisión de la negación en el contexto de la programación lógica.

7 SISTEMAS EXPERTOS

Un **sistema experto** es un programa que se comporta como un experto humano el algún dominio específico, usando conocimiento sobre ese dominio en particular. Es de esperar, que el programa sea capaz de explicar sus decisiones y su manera de razonar. También se espera que el programa sea capaz de contender con la incertidumbre y falta de información inherentes a esta tarea. En lo que sigue, nos basaremos en el texto de Bratko [18] para desarrollar los conceptos asociados a un sistema experto.

Sistema Experto

Algunas **aplicaciones** basada en sistemas expertos incluyen tareas como el diagnóstico médico, la localización de fallas en equipos y la interpretación de datos cuantitativos. Puesto que el éxito de estos sistemas depende en gran medida de su conocimiento sobre estos dominios, también se les conoce como **sistemas basados en el conocimiento** o *KBS*, por sus siglas en inglés. Aunque no todos los sistemas basados en el conocimiento ofrecen explicaciones sobre sus decisiones, ésta es una característica esencial de los sistemas expertos, necesaria principalmente en dominios inciertos como el diagnóstico médico, para garantizar la confianza del usuario en las recomendaciones del sistema; o para detectar un fallo flagrante en su razonamiento.

Aplicaciones

Sistemas basados en el conocimiento

La **incertidumbre** y la incompletez son inherentes a los sistemas expertos. La información sobre el problema a resolver puede ser incompleta o ni fiable. Las relaciones entre los objetos del dominio pueden ser aproximadas.

Incertidumbre

Ejemplo 7.1. *En el dominio médico, puede ser el caso de que **no estemos seguros** si un síntoma se ha presentado en un paciente, o que la medida de un dato es absolutamente correcta.*

Ejemplo 7.2. *El hecho de que ciertos fármacos **pueden** presentar reacciones adversas secundarias, pero éste no **suele** ser el caso. Este tipo de conocimiento requiere un manejo explícito de la incertidumbre.*

Para implementar un sistema experto, normalmente debemos considerar las siguientes funciones:

SOLUCIÓN DEL PROBLEMA. Una función capaz de usar conocimiento sobre el dominio específico del sistema, incluyendo el manejo de incertidumbre.

INTERFAZ DEL USUARIO. Una función que permita la interacción entre el usuario y el sistema experto, incluyendo la capacidad de explicar las decisiones y el razonamiento del sistema.

Cada una de estas funciones es extremadamente complicada, y puede depender del dominio de aplicación y de requerimientos prácticos específicos. Los problemas que pueden surgir en el diseño e implementación de un sistema experto tienen que ver con elegir una adecuada representación del conocimiento y los métodos de razonamiento asociados a tal representación. En lo que sigue, desarrollaremos un marco básico que podrá ser refinado posteriormente, conforme a nuestras necesidades. Es conveniente dividir el sistema experto en tres **módulos**, como se muestra en la figura 7.1:

Módulos de un sistema experto

BASE DE CONOCIMIENTOS. Incluye todo el conocimiento específico sobre un dominio de aplicación: Hechos, reglas y/o restricciones que describen las relaciones en el dominio. Puede incluir también métodos, heurísticas e ideas para resolver los problemas en ese dominio particular.

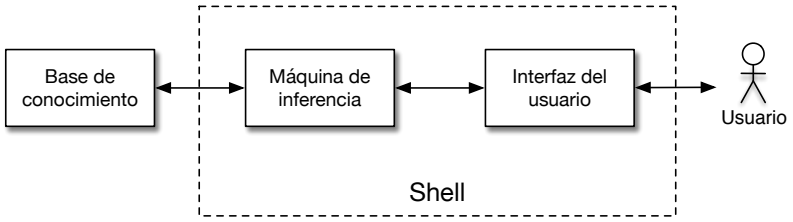


Figura 7.1: La estructura de un sistema experto.

MÁQUINA DE INFERENCIA. Incluye todos los procedimientos para usar activamente la base de conocimientos.

INTERFAZ DEL USUARIO. Se encarga de la comunicación entre el usuario y el sistema experto, debe proveer la información suficiente para que el usuario entienda el funcionamiento de la máquina de inferencia a lo largo del proceso.

También es conveniente ver la máquina de inferencia y la interfaz del usuario como un módulo independiente de cualquier base de conocimientos, lo que se conoce como un **shell** de sistema experto, o simplemente *shell*. Este esquema separa el conocimiento de los algoritmos que lo usan. Tal división es adecuada por los siguientes razones: La base de conocimientos depende claramente del cada dominio de aplicación específico, mientras que el *shell* es, al menos en principio, independiente de dicho dominio. De forma que, una forma racional de desarrollar sistemas expertos consiste en diseñar un solo *shell* que pueda usarse universalmente. Para ello las bases de conocimientos deberán apegarse al formalismo o representación que el *shell* puede manejar. En la práctica, este principio de diseño es difícil de mantener. Al menos que los dominios de aplicación sean muy similares entre si, lo normal es que el *shell* requiera de adecuaciones al cambiar de dominio. Sin embargo, aún en esos casos, la separación propuesta mantiene las ventajas de la modularidad.

Shell

En lo que sigue, aprenderemos a representar conocimiento usando reglas en formato si-entonces, conocidas como reglas de producción; así como los mecanismos de inferencia básicos sobre este formato: Los razonamientos por encadenamiento hacía adelante y hacía atrás. Estudiaremos las mejoras a estas reglas que se introducen al considerar la incertidumbre, las redes semánticas y la representación de conocimientos basada en marcos.

7.1 REGLAS DE PRODUCCIÓN

En principio, cualquier formalismo consistente con el que podamos expresar conocimiento acerca del dominio de un problema, puede ser considerado para su uso en un sistema experto. Sin embargo, el uso de reglas si-entonces, también llamadas **reglas de producción**, es tradicionalmente el formalismo más popular para representar conocimiento en un sistema experto. En principio, tales reglas son enunciados condicionales, pero pueden tener varias **interpretaciones**, por ejemplo:

Reglas de producción

Interpretaciones posibles

- Si condición P entonces conclusión C.
- Si situación S entonces acción A.
- Si las condiciones C_1 y C_2 son el caso, entonces la condición C no lo es.

De forma que, las reglas de producción son una forma natural de representar conocimiento, que además poseen las siguientes **características deseables**:

Características deseables

MODULARIDAD. Cada regla define una pequeña, relativamente independiente, pieza de información.

AGREGACIÓN. Es posible agregar nuevas reglas al sistema, de forma relativamente independiente al resto de sus reglas.

FLEXIBILIDAD. Como una consecuencia de la modularidad, las reglas del sistema pueden modificarse con relativa independencia de las otras reglas.

TRANSPARENCIA. Facilitan la explicación de las decisiones tomadas por el sistema experto. En particular, es posible automatizar la respuesta a preguntad del tipo ¿Cómo se llegó a esta conclusión? y ¿Porqué estás interesado en tal información?

Conocimiento categórico vs probabilístico

Las reglas de producción a menudo definen relaciones lógicas entre conceptos del dominio de un problema. Las relaciones puramente lógicas pueden caracterizarse como **conocimiento categórico**, en el sentido de que son siempre absolutamente verdaderas. En algunos dominios, como el médico, el **conocimiento probabilístico** prevalece. En este caso, las relaciones no son siempre verdaderas y se establecen con base en su regularidad empírica, usando grados de certidumbre. Las reglas de producción deben ajustarse para contender con enunciados del tipo: Si condición A entonces conclusión C con un grado de certeza F.

Ejemplo 7.3. *Una regla con conocimiento no categórico en el dominio médico. Proviene del sistema experto desarrollado para el libro de Negrete-Martínez, González-Pérez y Guerra-Hernández [73] sobre la pericia artificial y su implementación incremental en Lisp. Aquí la regla se muestra en un pseudo-código más cercano a lo que haremos en Prolog:*

```
Regla 037:
IF
paciente(hipertermia,si), paciente(tos,si), paciente(insufResp,si)
;
paciente(estertoresAlveolares,si), paciente(condensacionPulmonar,si)
THEN
paciente(neumonia,[si,80]).
```

en este caso, la conclusión de la regla incluye un factor de certidumbre, no estrictamente probabilistico, que expresa que tan confiable es tal conclusión. Veremos más al respecto al abordar la incertidumbre.

Por lo general, si se quiere desarrollar un sistema experto, será necesario consultar a un experto humano en el dominio del problema y estudiar el dominio uno mismo. Al proceso de extraer conocimiento de los expertos y la literatura de un dominio dado, para acomodarlo a un formalismo de representación se le conoce como ingeniería del conocimiento [1].

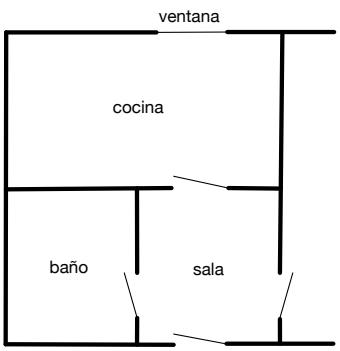


Figura 7.2: El plano de una casa donde debemos detectar fugas de agua.

1 En inglés suele usarse el intraducible término de *knowledge elicitation*.

El proceso de ingeniería del conocimiento, como tal, queda fuera de los alcances de este curso, pero jugaremos con un dominio sencillo en nuestros experimentos. La figura 7.2 muestra el plano de una casa donde deberemos localizar fugas de agua.

Luego de consultar con nuestro plomero favorito y consultar en internet los manuales de plomería para *dummies*, llegamos a la conclusión de que la fuga puede presentarse en el baño o en la cocina. En cualquier caso, la fuga causa que haya agua en el piso de la sala. Observen que asumimos que la fuga solo se da en un sitio, no en ambos al mismo tiempo. Esto se puede representar como una **red de inferencia**, tal y como se muestra en la figura 7.3.

Red de inferencia

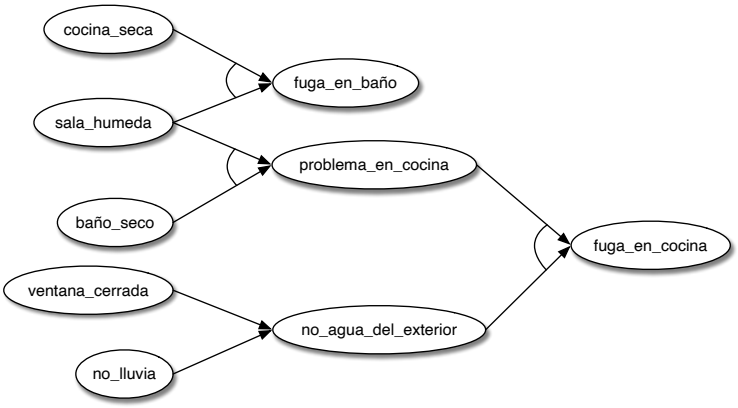


Figura 7.3: La red de inferencia de nuestro conocimiento sobre fugas.

Los nodos en la red de inferencia representan proposiciones y las ligas, reglas en la base de conocimientos. Los arcos que conectan ligas, indican la naturaleza conjuntiva entre las correspondiente proposiciones.

Ejemplo 7.4. *De acuerdo con lo anterior, una regla de nuestro sistema experto debería ser:*

```
IF baño_seco AND sala_humeda THEN problema_en_cocina.
```

A la red de inferencia representada de esta forma, también se le conoce como **grafo AND/OR**.

Grafo AND/OR

7.2 RAZONAMIENTO CON REGLAS DE PRODUCCIÓN

Una vez que el conocimiento se ha representado en algún formalismo, necesitamos de un procedimiento de razonamiento para extraer conclusiones de éste. Hay dos formas básicas de razonamiento con reglas de producción:

- Razonamiento hacía atrás.
- Razonamiento hacía adelante.

7.2.1 Razonamiento hacía atrás

Una vez que tenemos nuestra red de inferencias, el razonamiento hacía atrás funciona de la siguiente manera. Partimos de una hipótesis, por ejemplo, que hay una fuga en la cocina; y razonamos hacía atrás sobre la red de inferencias. Para resolver esta hipótesis necesitamos que no agua en el exterior y problema en la cocina sean verdaderas. La primera es el caso si no llueve o la ventana está cerrada, etc. El razonamiento se conoce como **hacía atrás**, porque forma una cadena de las hipótesis (problema en la cocina) hacía las evidencias (ventana cerrada).

Razonamiento hacía atrás

Tal procedimiento es muy fácil de implementar en Prolog, de hecho, ¡es la forma en que este lenguaje razona! La manera directa de implementar nuestra red de inferencias es usando las reglas provistas por Prolog:

```
% Base de conocimientos para fuga en casa

% Conocimiento previo

fuga_en_bano :-
    sala_seca,
    cocina_seca.

problema_en_cocina :-
    sala_humeda,
    bano_seco.

no_agua_del_exterior :-
    ventana_cerrada
    ;
    no_lluvia.

fuga_en_cocina :-
    problema_en_cocina,
    no_agua_del_exterior.

% Evidencia

sala_humeda.
bano_seco.
ventana_cerrada.
```

Observen que la evidencia se representa como hechos, mientras que el conocimiento previo hace uso de reglas Prolog.

Nuestra hipótesis puede ahora verificarse mediante la siguiente consulta:

```
?- fuga_en_cocina.
true
```

Sin embargo, usar las reglas de Prolog directamente tiene algunas desventajas:

1. La sintaxis de estas reglas no es la más adecuada para alguien que no conoce Prolog. Este es el caso de nuestros expertos humanos, que deben especificar nuevas reglas, leer las existentes y posiblemente modificarlas, sin saber Prolog.
2. La base de conocimientos es sintácticamente indistinguible del resto del sistema experto. Esto, como se mencionó, no es deseable.

Como vimos en el tutorial de Prolog, es muy sencillo definir una sintaxis diferenciada para las reglas de producción de nuestro sistema experto, haciendo uso de la definición de operadores. De forma que un interprete para razonamiento hacía atrás basado en reglas de producción luciría así:

```
% Un interprete de encadenamiento hacía atrás para reglas if-then.

:-  op( 800, fx, if).
:-  op( 700, xfx, then).
:-  op( 300, xfy, or).
:-  op( 200, xfy, and).

is_true( P) :-
   fact( P).

is_true( P) :-
   if Cond then P,        % Una regla relevante,
   is_true( Cond).        % cuya condición Cond es verdadera.

is_true( P1 and P2) :-
   is_true( P1),
   is_true( P2).

is_true( P1 or P2) :-
   is_true( P1)
   ;
   is_true( P2).
```

Hemos introducido los operadores infijos `then`, `or` y `and`, junto con un operador prefijo `if` para representar nuestras reglas de producción de una manera diferenciada de las reglas de Prolog. Por ello es necesario establecer la semántica de nuestras reglas, a través del preciado `is_true`. Observen que los operadores definidos también se usan en las reglas semánticas.

Adecuando el formato de las reglas anteriores, nuestra base de conocimientos quedaría implementado de la siguiente forma:

```
%%% base de conocimiento para fugas, en fomato de reglas if-then

%%% Conocimiento previo

if sala_humeda and cocina_seca
then fuga_en_bano.

if sala_humeda and bano_seco
then problema_en_cocina.

if ventana_cerrada or no_lluvia
then no_agua_del_exterior.

if problema_en_cocina and no_agua_del_exterior
then fuga_en_cocina.

%%% Evidencias

fact(sala_humeda).        % Cambiar a sala\_seca para que falle la meta
fact(bano_seco).
fact(ventana_cerrada).    % Comentar para probar explicaciones porque
```

Observen el uso del predicado `fact` para representar los hechos del sistema experto. La hipótesis de fuga en la cocina puede verificarse de la siguiente forma:

```
?- is_true(fuga_en_cocina).
true
```

Desventajas

Una **desventaja** del procedimiento de inferencia definido es que el usuario debe incluir en la base de conocimientos, toda la evidencia con la que cuenta en forma de hechos, antes de poder iniciar el proceso de razonamiento. Lo ideal sería que las evidencias fueran provistas por el usuario conforme se van necesitando, de manera interactiva. Esto lo resolveremos más adelante.

7.3 RAZONAMIENTO HACÍA ADELANTE

En el razonamiento hacía atrás partíamos de las hipótesis para ir hacía la evidencia. Algunas veces resulta más natural razonar en la dirección opuesta: A partir de nuestro conocimiento previo y la evidencia disponible, explorar que conclusiones podemos obtener. Por ejemplo, una vez que hemos confirmado que la sala está mojada y que el baño está seco, podríamos concluir que hay un problema en la cocina.

Programar un procedimiento de razonamiento hacía adelante en Prolog, sigue siendo sencillo, si bien no trivial como es el caso del razonamiento hacía atrás. Nuestro interprete es el siguiente:

```
% Un intérprete simple para el razonamiento hacia adelante

forward :-
    new_derived_fact(P),    % Se deriva un nuevo hecho.
    !,
    write('Nuevo hecho derivado: '), write(P), nl,
    assert(fact(P)),
    forward   % Buscar más hechos nuevos.
    ;
    write('No se derivaron más hechos.').   % Terminar, no más hechos derivados.
```

```
11
12 new_derived_fact(Concl) :-
13     if Cond then Concl,    % Una regla
14     \+ fact(Concl),        % cuya conclusión no es un hecho
15     composed_fact(Cond).   % Su condición es verdadera?
16
17 composed_fact(Cond) :-
18     fact(Cond).            % Un hecho simple
19
20 composed_fact(Cond1 and Cond2) :-
21     composed_fact(Cond1),
22     composed_fact(Cond2). % Ambos operandos verdaderos.
23
24 composed_fact(Cond1 or Cond2) :-
25     composed_fact(Cond1)
26     ;
27     composed_fact(Cond2). % Al menos un operando verdadero.
```

Nuestra base de conocimiento seguirá siendo la usada en el caso del razonamiento hacía atrás. Algo de modularidad hemos conseguido. Por simplicidad, asumimos que las reglas en esa base de conocimiento no tienen variables. El interprete parte de la evidencia conocida, especificada mediante el predicado `fact` en la base de conocimientos, deriva los hechos que se pueden inferir mediante las reglas de producción y los agrega a la base de conocimiento, mediante el predicado `assert` (lo cual nos lleva a definir `fact` como dinámico). La ejecución de este procedimiento de razonamiento es como sigue:

```
1 ?- forward.
2 Nuevo hecho derivado: problema_en_cocina
3 Nuevo hecho derivado: no_agua_del_exterior
4 Nuevo hecho derivado: fuga_en_cocina
5 No se derivaron más hechos.
6 true.
```

Una interfaz para cargar los modulos definidos hasta ahora, se implementa como sigue:

```
1 % Cargar el sistema basado en conocimiento.
2
3 :- dynamic(fact/1).
4
5 :- [encadenamientoAtras].
6 :- [encadenamientoAdelante].
7 :- [kbFugasIfThen].
```

7.4 COMPARANDO LOS RAZONAMIENTOS

Las reglas de producción, representadas en nuestra red de inferencias (Fig. 7.3), forman cadenas que van de la izquierda a la derecha. Los elementos en el lazo izquierdo de estas cadenas son información de entrada, mientras que los del lado derecho son información derivada:

$$\text{Información de Entrada} \quad \rightarrow \cdots \rightarrow \quad \text{Información Derivada}$$

Estos dos tipos de información reciben varios nombres, dependiendo del contexto en el cual son usados. La información de entrada puede llamarse **datos**, por ejemplo, en el caso de información de entrada que requiere mediciones. La información derivada puede llamarse **metas**, o hipótesis, o diagnóstico o explicación. De forma que las cadenas de inferencia conectan varios tipos de información como:

Datos

Metas

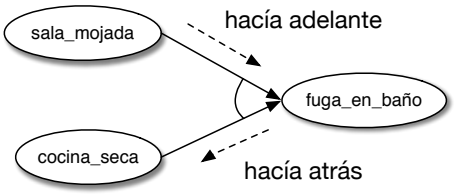


Figura 7.4: Un razonamiento en ambos sentidos disparado por la evidencia de que la sala está mojada.

Datos	$\rightarrow \cdots \rightarrow$	Metas
Evidencia	$\rightarrow \cdots \rightarrow$	Hipótesis
Observaciones	$\rightarrow \cdots \rightarrow$	Diagnósticos
Descubrimientos	$\rightarrow \cdots \rightarrow$	Explicaciones
Manifestaciones	$\rightarrow \cdots \rightarrow$	Causas

Ambos razonamientos involucran búsqueda, pero difieren en su dirección. El razonamiento hacía atrás busca de las metas a los datos, mientras que el razonamiento hacía adelante lo hace en el sentido inverso. Como el primero inicia con una meta, se dice que es un proceso **dirigido por metas**. El razonamiento hacía adelante, puede verse como un procesamiento **dirigido por datos**.

Razonamiento dirigido por metas y por datos

Una pregunta evidente es ¿Qué tipo de razonamiento es mejor? La respuesta depende del problema. Si lo que queremos es verificar si una hipótesis determinada es verdadera, entonces el razonamiento hacía atrás resulta más natural. Si es el caso que tenemos numerosas hipótesis y ninguna razón para comenzar con una de ellas en particular, entonces el encadenamiento hacía adelante es más adecuado. En general, el razonamiento hacía adelante es más adecuado en situaciones de monitoreo, donde los datos se adquieren continuamente.

La forma de la red de inferencias puede ayudar a decidir que tipo de razonamiento usar. Si solo hay unos pocos nodos de datos (el flanco izquierdo de la gráfica) y muchos nodos meta (el flanco derecho), entonces el encadenamiento hacía adelante sería el más apropiado; y viceversa, el encadenamiento hacía atrás es más apropiado en el caso contrario.

Las tareas de un experto son normalmente más intrincadas y requieren de una combinación de encadenamientos en ambas direcciones. En medicina, por ejemplo, normalmente algunas observaciones iniciales disparan razonamientos del médico hacía adelante, para generar una hipótesis de diagnóstico inicial. Ésta debe ser confirmada o rechazada con base en evidencia adicional, la cual se obtiene mediante procesos de razonamiento hacía atrás. En nuestro ejemplo, observen que la evidencia de que la sala está mojada podría disparar un razonamiento como el mostrado en la figura 7.4.

7.5 GENERANDO EXPLICACIONES

Existen formas estándar de generar explicaciones en los sistemas basados en reglas de producción. Los tipos de explicación computables son respuestas a las preguntas de tipo ¿Cómo? y ¿Porqué?

7.5.1 Explicaciones tipo ¿Cómo?

Cuando nuestro sistema experto genera una respuesta, el usuario puede preguntar –¿Cómo obtuviste esa respuesta? La explicación consiste en la traza de como se derivo la respuesta en cuestión.

Ejemplo 7.5. *Supongamos que la última respuesta computada por nuestro sistema es que hay una fuga en la cocina. La explicación sería como sigue, por que:*

1. *Hay un problema en la cocina, lo cual fue concluído a partir de que la sala está mojada y el baño seco.*
2. *El agua no provino del exterior, lo cual fue concluído a partir de que la ventana estaba cerrada.*

Árbol de derivación

Tal explicación es de hecho el **árbol de derivación** de la meta, es decir, el árbol-SLD (Def. 5.30) que Prolog construye cada vez que le planteamos una meta. Definamos `<=` como un operador infijo, para poder representar el árbol de prueba de una proposición P de la siguiente manera:

1. Si P es un hecho, entonces el árbol de prueba es P mismo.
2. Si P fue derivado usando una regla: if *Cond* then P, el árbol de prueba es P `<=` *PruebaCond*; donde *PruebaCond* es a su vez, el árbol de prueba de *Cond*.
3. Sean P_1 y P_2 proposiciones cuyos árboles de prueba son A_1 y A_2, entonces el árbol de prueba de P_1 `and` P_2 es A_1 `and` A_2. El caso de `or`, se resuelve análogamente.

Esta definición se puede implementar en Prolog, refinando el predicado `is_true/1` con un segundo argumento que unifique con el árbol de prueba:

```
% is\_true(P,A) A es el árbol de prueba de la meta proposicional P

:- op(800, xfx, <=).

is_true(P,P) :-
    fact(P).

is_true(P, P <= CondA) :-
    if Cond then P,
    is_true(Cond,CondA).

is_true(P1 and P2, A1 and A2) :-
    is_true(P1,A1),
    is_true(P2,A2).

is_true(P1 or P2, A1 or A2) :-
    is_true(P1,A1)
    ;
    is_true(P2,A2).
```

Si incluimos este módulo en nuestra versión 2 de la interfaz, tendremos:

```
% Cargar el sistema basado en conocimiento.

:- dynamic(fact/1).

:- [encadenamientoAtras].
:- [encadenamientoAdelante].
:- [explicacionesComo].
:- [kbFugasIfThen].
```

Y su uso, se ilustra con la siguiente consulta:

```
?- [loadSystemV2].
true.
```

```
?- is_true(fuga_en_cocina,A).
A =  (fuga_en_cocina<=(problema_en_cocina<=sala_humeda and
bano_seco)and(no_agua_del_exterior<=ventana_cerrada or _66))
```

Por supuesto, podemos mejorar la manera en que el árbol de derivación se despliega para el usuario. Para ello, definiremos un predicado `show_tree/1`:

```
% show\_tree(A): Imprime el árbol de derivación A.

show_tree(A) :-
    show_tree(A,0).

% show\_tree(A,N): Imprime el árbol de derivación A,
% al nivel de indentación N.

show_tree(A,N) :-
    var(A), tab(N), writeln('cualquier_cosa').

show_tree(A1 <= A2,N) :- !,
    tab(N), write(A1), writeln(' <='),
    show_tree(A2,N).

show_tree(A1 and A2,N) :- !,
    N1 is N + 2,
    show_tree(A1,N1), tab(N1), writeln('and'),
    show_tree(A2,N1).

show_tree(A1 or A2,N) :- !,
    N1 is N + 2,
    show_tree(A1,N1), tab(N1), writeln('or'),
    show_tree(A2,N1).

show_tree(A,N) :-
    tab(N),writeln(A).
```

Ahora podemos hacer la siguiente consulta:

```
?- [loadSystemV2].
true.

?- is_true(fuga_en_cocina,A), show_tree(A).
fuga_en_cocina <=
  problema_en_cocina <=
    sala_humeda
    and
    bano_seco
  and
  no_agua_del_exterior <=
    ventana_cerrada
    or
    cualquier_cosa
A =  (fuga_en_cocina<=(problema_en_cocina<=sala_humeda and
bano_seco)and(no_agua_del_exterior<=ventana_cerrada or _12250))
```

7.5.2 Explicaciones tipo ¿Porqué?

Interacción

La explicación sobre porqué un dato es relevante, a diferencia de las del tipo ¿Cómo?, se necesitan en tiempo de ejecución, no al final de una consulta. Esto require que el usuario pueda **interactuar** con la máquina de inferencia del sistema experto, através de la interfaz.

Una forma de introducir interacción en nuestro sistema, es que éste pregunte al usuario por las evidencias que va necesitando en su proceso de razonamiento. En ese momento, el usuario podrá preguntar, porqué tal información es necesaria. Será necesario especificar que proposiciones pueden ser preguntadas al usuario, mediante el predicado `askable/1`.

Ejemplo 7.6. *Se puede agregar a nuestra base de conocimientos, las siguientes líneas:*

```
%%% Hechos que se pueden preguntar la usuario

askable(no_lluvia).
askable(ventana_cerrada).
```

Si hemos agregado *askable*(P) a nuestra base de conocimientos, cada vez que la meta P se presente, el sistema experto preguntará si P es el caso, con tres respuesta posibles: `si`, `no`, y `porque`. La última de ellas, indica que el usuario está interesado en saber porqué tal información es necesaria. Esto es particularmente necesario en situaciones donde el usuario no sabe la respuesta, y establecer si P es o no el caso, requiere de cierto esfuerzo.

En caso de que el usuario responda `si` al sistema, la proposición P es agregada a la memoria de trabajo, usando `assert(fact(P))`. Debemos agregar también una cláusula al estilo de `already_asked(P)` a la memoria de trabajo, para evitar preguntar de nuevo por P cuando esto ya no es necesario. La respuesta a `porque` debería ser una cadena de reglas que conectan la evidencia P con la meta original que se le ha planteado al sistema. Veamos la implementación de este mecanismo en Prolog:

```
:- dynamic already_asked/1.

% iis_true(P,A): P es el caso con la explicación A.
% Versión interactiva con el usuario.

iis_true(P,A) :-
    explore(P,A,[]).

% explore(P,A,T): A es una explicación de porque P es verdadera, H
% es una cadena de reglas que liga P con las metas anteriores.

explore(P, P, _) :-
    fact(P).

explore(P1 and P2, A1 and A2, T) :-
    explore(P1, A1, T),
    explore(P2, A2, T).

explore(P1 or P2, A1 or A2, T) :-
    explore(P1, A1, T)
    ;
    explore(P2, A2, T).

explore(P, P <= ACond, T) :-
    if Cond then P,
    explore(Cond, ACond, [if Cond then P | T]).

explore(P,A,T) :-
    askable(P),
    \+ fact(P),
    \+ already_asked(P),
    ask_user(P, A, T).

% ask_user(P,A,T): Pregunta al usuario si P es el caso,
% generando las explicaciones A y T.

ask_user(P, A, T) :-
    nl, write('Es cierto que: '), write(P),
    writeln(' Conteste si/no/porque:'),
    read(Resp),
    process_answer(Resp,P,A,T).

process_answer(si,P, P <= preguntado,_) :-
    asserta(fact(P)),
    asserta(already_asked(P)).

process_answer(no,P,_,_) :-
    asserta(already_asked(P)),
    fail.

```

```
process_answer(porque,P,A,T) :-
    display_rule_chain(T,0), nl,
    ask_user(P,A,T).

% display\_rule\_chain(R,N): Despliega las reglas R,
% identando a nivel N.

display_rule_chain([],_).

display_rule_chain([if Cond then P | Reglas], N) :-
    nl, tab(N), write('Para explorar si '),
    write(P), write(' es el caso, usando la regla '),
    nl, tab(N), write(if Cond then P),
    N1 is N + 2,
    display_rule_chain(Reglas,N1).
```

que agregaremos a nuestra tercera versión de la interfaz. Observen que el predicado principal se llama ahora `iis_true/2`, para indicar que es la versión interactiva del anterior `is_true/2` [2] De esta forma, nuestra interfaz `loadSystemV3.pl` quedaría como:

```
% Cargar el sistema basado en conocimiento.

:- dynamic(fact/1).

:- [encadenamientoAtras].
:- [encadenamientoAdelante].
:- [explicacionesComo].
:- [explicacionesPorque].
:- [kbFugasIfThen].
```

Una consulta al sistema, sería como sigue. Antes de ejecutar la consulta, verifique que ha comentado la línea correspondiente a `fact(ventana_cerrada` en la base de conocimientos original, para que el sistema experto deba preguntar por este hecho o por `no_lluvia`. El usuario puede pedir explicaciones al respecto en ambos casos.

```
?- [loadSystemV3].
true.

?- iis_true(fuga_en_cocina,A), show_tree(A).

Es cierto que: ventana_cerrada Conteste si/no/porque:
|: porque.

Para explorar si no_agua_del_exterior es el caso, usando la regla
if ventana_cerrada or no_lluvia then no_agua_del_exterior
  Para explorar si fuga_en_cocina es el caso, usando la regla
  if problema_en_cocina and no_agua_del_exterior then fuga_en_cocina

Es cierto que: ventana_cerrada Conteste si/no/porque:
|: si.

fuga_en_cocina <=
  problema_en_cocina <=
    sala_humeda
    and
    bano_seco
  and
  no_agua_del_exterior <=
    ventana_cerrada <=
    preguntado
    or
    cualquier_cosa

A =  (fuga_en_cocina<=(problema_en_cocina<=sala_humeda and bano_seco)
and(no_agua_del_exterior<=(ventana_cerrada<=preguntado)or _6410))
```

2 De otra forma hay un conflicto de nombres, que Prolog resuelve definiendo nuevamente el predicado en cuestión a partir del último archivo cargado.

Obviamente el módulo de explicaciones del tipo ¿Porqué? solo funciona con reglas proposicionales. Si usásemos variables, habría que preguntar también por los valores de estas. De cualquier forma, nuestro sistema ilustra los principios que queríamos presentar sobre las explicaciones en los sistemas expertos.

7.6 INCERTIDUMBRE

Hasta ahora, hemos asumida que la representación de conocimiento de nuestro problema es categórica, en decir, las respuestas a todas nuestras preguntas son verdaderas o falsas. Como datos, las reglas también han sido concebidas de manera categórica. Sin embargo, mucha de la experticia humana no es categórica. Cuando un experto busca la solución a un problema dado, en muchas ocaciones se asumen muchos hechos, que si bien generalmente son ciertos, suelen presentar excepciones. Tanto los datos como las reglas asociadas a un dominio de problema, pueden ser parcialmente certeros. Tal incertidumbre se puede modelar si asignamos alguna calificación, otra que no sea verdadero o falso, a los hechos reconocidos. Estas calificaciones pueden expresarse como descriptores, como verdadero, altamente probable, probable, poco probable, imposible. Alternativamente, el grado de certeza pueda expresarse como un número real en algún intervalo, por ejemplo, entre 0 y 1, o entre -5 y +5. Tales valores reciben diversos nombres como **grados de certeza**, o grados de creencia, o probabilidad subjetiva. La mejor manera de expresar incertidumbre es usando probabilidades, debido a sus fundamentos matemáticos. Sin embargo, razonar correctamente usando la probabilidad es más demandante que los esquemas *ad hoc* de incertidumbre, que aquí revisaremos.

Grados de certeza

En esta sección extenderemos nuestra representación basada en reglas de producción con un **esquema de incertidumbre** simple, que aproxima las probabilidades de manera muy rudimentaria. Cada proposición P será asociada a un número FC entre 0 y 1, que representará su grado de certeza. Usaremos un par $P : FC$ para esta representación. La notación adoptada para las nuevas reglas será: `if` *Cond* `then` $P : FC$.

Proposiciones y reglas con certeza

En toda representación que maneje incertidumbre, es necesario especificar la forma de combinar la certeza de los hechos y las reglas en el sistema.

Ejemplo 7.7. *Sean dos proposiciones, P_1 y P_2, con valores asociados de certeza $c(P_1)$ y $c(P_2)$ respectivamente ¿Cual es la certeza $c(P_1$ `and` $P_2)$?*

La misma pregunta aplica para el resto de los operadores en nuestro lenguaje de reglas de producción. En nuestro caso, el esquema de combinación será el siguiente

$$c(P_1 \textit{ and } P_2) = \text{mín}(c(P_1), c(P_2)) \tag{7.1}$$

$$c(P_1 \textit{ or } P_2) = \text{máx}(c(P_1), c(P_2)) \tag{7.2}$$

En el caso de una regla `if` P_1 `then` $P_2 : FC$, entonces:

$$c(P_2) = c(P_1) \times FC \tag{7.3}$$

Por simplicidad asumiremos que las reglas tienen implicaciones únicas. Si ese no fuese el caso, las reglas en cuestión pueden escribirse con ayuda del operador `or`, para satisfacer esta restricción de formato. Nuestra nueva base de conocimientos, llamada `kbFugasIncert.pl` incluye:

```
1 %%% base de conocimiento para fugas, en fomato de reglas if-then
2
3 %%% Conocimiento previo con certidumbre
4
5 if sala_humeda and cocina_seca
6 then fuga_en_bano.
7
8 if sala_humeda and bano_seco
```

```
then problema_en_cocina:0.9.

if ventana_cerrada or no_lluvia
then no_agua_del_exterior.

if problema_en_cocina and no_agua_del_exterior
then fuga_en_cocina.

% Evidencia con certidumbre

given(sala_humeda, 1).     % verdadero
given(bano_seco, 1).
given(cocina_seca, 0).     % falso
given(no_lluvia, 0.8).     % muy probable
given(ventana_cerrada, 0).
```

Observen que la regla de la línea 9, o mejor dicho, su conclusión, ha sido relativizada con un grado de certeza de 0.9. La máquina de inferencia basada en las ecuaciones 7.1-7.3, es como sigue:

```
% Interprete basado en reglas con incertidumbre

% cert(P,C): C es el grado de certeza de la proposición P

cert(P,C) :-
    given(P,C).

cert(P1 and P2, C) :-
    cert(P1,C1),
    cert(P2,C2),
    min(C1,C2,C).

cert(P1 or P2, C) :-
    cert(P1,C1),
    cert(P2,C2),
    max(C1,C2,C).

cert(P, C) :-
    if Cond then P:C1,
    cert(Cond,C2),
    C is C1 * C2.

cert(P, C) :-
    if Cond then P,
    cert(Cond,C1),
    C is C1.

% max y min

max(X,Y,X) :-  X>=Y,!.
max(_,Y,Y).

min(X,Y,X) :-  X=<Y,!.
min(_,Y,Y).
```

Una nueva interfaz versión 4 carga los archivos correspondientes:

```
% Cargar el sistema basado en conocimiento.

:-  dynamic(fact/1).

:-  [encadenamientoAtras].
:-  [encadenamientoAdelante].
:-  [incertidumbre].
:-  [kbFugasIncert].
```

La consulta al sistema es como sigue:

```
?- cert(fuga_en_cocina,FC).
FC = 0.8
```

Dificultades con la incertidumbre

Esquemas como el aquí propuesto son fácilmente criticables por no seguir los axiomas de la probabilidad formal.

Ejemplo 7.8. *Asumamos que el factor de certeza de a es 0.5 y el de b es 0. Entonces, el factor de certeza de a* ***or*** *b es 0.5. Ahora, si en factor de certeza de b se incrementa a 0.5, esto no tiene efecto en el factor de certeza de la disyunción, que sigue siendo 0.5; lo cual resulta, al menos, contraintuitivo.*

Por otra parte, se ha argumentado que la teoría de probabilidad, aunque matemáticamente bien fundada, no es ni práctica, ni apropiada en estos casos, por las siguientes razones:

- Los expertos humanos parecen tener problemas para razonar basados realmente en la teoría de probabilidad. La verosimilitud que usan, no se corresponde con la noción de probabilidad definida matemáticamente.
- El procesamiento de información con base en la teoría de probabilidad, parece requerir de información que no siempre está disponible; o asumir simplificaciones que no siempre se justifican.

Estas objections aplican a esquemas simplificados como el aquí propuesto. En realidad, la opinión unánime es a favor del uso de enfoques híbridos que aprovechen la teoría de probabilidad, pero para ello hacen falta máquinas de inferencia más elaboradas, por ejemplo, basadas en redes bayesianas.

7.7 RAZONAMIENTO BAYESIANO

Redes de creencias

Las redes bayesianas, también llamadas **redes de creencias**, proveen una manera de usar el cálculo de probabilidades para el manejo de la incertidumbre en nuestras representaciones de conocimiento. Estas redes proveen mecanismos eficientes para manejar las dependencias probabilísticas, mientras explotan las independencias; resultando en un a representación natural de la **causalidad**. En lo que sigue, revisaremos como se usan las dependencias e independencias entre variables en una red bayesiana e implementaremos un método para computar las **probabilidades condicionales** en los modelos de redes bayesianas.

Causalidad

Probabilidades condicionales

7.7.1 Probabilidades, creencias y redes Bayesianas

¿Cómo podemos manejar la incertidumbre correctamente, con principios bien fundamentados y pragmatismo? Como comentamos en el capítulo precedente sobre sistemas expertos (Sección 7.6), conseguir pragmatismo y rigor matemático es una meta difícil de alcanzar, pero las redes Bayesianas ofrecen una buena solución en esa dirección.

En el resto del capítulo asumiremos que el medio ambiente puede representarse por medio de un vector de **variables** que pueden tomar valores aleatoriamente de un dominio, es decir, un conjunto de valores posibles. En los ejemplos que introduciremos, asumiremos que nuestras variables tienen un **dominio booleano**, es decir, podrán tener como valor falso o verdadero.

Variables

Dominio booleano

Ejemplo 7.9. *`ladron` y `alarma` son dos variables de este tipo. La variable `alarma` es verdadera cuando la alarma suena, y la variable `ladron` es verdadera cuando un ladrón ha entrado en casa. En cualquier otro caso, ambas variables son falsas.*

De esta forma, el estado del medio ambiente en un momento dado, se puede especificar completamente computando el valor de las variables de su representación

en ese momento. Observen que nuestro supuesto de variables booleanas no constituye una limitación significativa. De hecho, al terminar el capítulo será evidente como contender dominios multi-valor.

Cuando las variables son booleanas, es normal pensar en ellas como si representasen **eventos**.

Eventos

Ejemplo 7.10. *Cuando la variable* `alarma` *se vuelve verdadera, el evento alarma sonando sucedió.*

Un agente, natural o artificial, normalmente no está completamente seguro de cuando estos eventos son verdaderos o falsos. Normalmente, el agente razona acerca de la **probabilidad** de que la variable en cuestión sea verdadera. Las probabilidades en este caso, son usadas como una medida del grado de creencia. Este grado depende de que tanto el agente conoce su medio ambiente. De forma que tales creencias se conocen tambien como **probabilidades subjetivas**, donde subjetivo no quiere decir arbitrario, al menos no en el sentido de que no estén gobernadas por la teoría de la probabilidad.

Probabilidad

Probabilidades subjetivas

Necesitaremos un poco de notación. Sean X e Y dos proposiciones, entonces, como de costumbre:

- $X \wedge Y$ es la conjunción de X e Y.
- $X \vee Y$ es la disyunción de X e Y.
- $\neg X$ es la negación de X.

La expresión $p(X)$ denota la probabilidad de que la proposición X sea verdadera. La expresión $p(X|Y)$ denota la probabilidad condicional de que la proposición X sea verdadera, dado que la proposición Y lo es.

Un **meta** típica es este contexto toma esta forma: Dado que los valores de ciertas variables han sido observados ¿Cuales son las probabilidades de que otras variables de interés tomen cierto valor específico? O, dado que ciertos eventos han sido observados ¿Cual es la probabilidad de que sucedan otros eventos de interés?

Metas

Ejemplo 7.11. *Si observamos que la alarma suena ¿Cual es la probabilidad de que un ladrón haya entrado en la casa?*

La mayor dificultad para resolver estas metas está en manejar la dependencia entre las variables del problema. Sea el caso que nuestro problema contempla n variables booleanas; entonces, necesitamos $2^n - 1$ números para definir la distribución de probabilidad completa entre los 2^n estados posibles del medio ambiente! Esto no es sólo caro computacionalmente, sino imposible, debido a que toda esa información no suele estar a la disposición del agente.

Afortunadamente, suele ser el caso que no es necesario contar con todas esas probabilidades. La distribución de probabilidades completa, no asume nada acerca de la independencia entre variables. Afortunadamente, algunas cosas son independientes de otras. Las redes bayesianas proveen una forma elegante para declarar la dependencia e independencia entre variables. La Figura 7.5, muestra una red bayesiana acerca de ladrones y sistemas de alarma. La idea es que el sensor se dispare si un ladrón ha entrado en casa, activando la alarma y llamando a la policía. Pero, un relámpago fuerte puede también puede activar el sensor, con las complicaciones correspondientes. Esta red bayesiana puede responder a preguntas del tipo: Supongamos que hace buen tiempo y escuchamos la alarma. Dados estos dos hechos ¿Cual es la probabilidad de que un ladrón haya entrado en casa?

La estructura de esta red bayesiana indica algunas dependencias probabilísticas, y también algunas independencias. Nos dice, por ejemplo, que `ladron` no es dependiente de `relampago`. Sin embargo, si se llega a saber que `alarma` es verdadera, entonces bajo las condiciones dadas, `ladron` y `relampago` ya no son independientes.

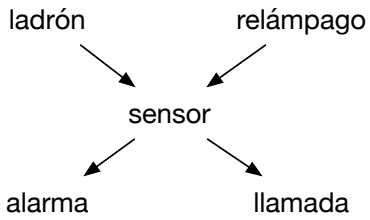


Figura 7.5: Una red bayesiana para detectar ladrones en casa.

Ejemplo 7.12. *En nuestro pequeño ejemplo, es evidente que las ligas en la red bayesiana indican causalidad. El* `ladron` *es causa de que el* `sensor` *se dispare. El* `sensor` *por su parte, es la causa de que la* `alarma` *se encienda. De forma que la estructura de la red permite razonamientos como el siguiente: Si* `alarma` *es verdadera, entonces un* `ladron` *pudo entrar en casa, puesto que esa es una de las causas de que la alarma suene. Si entonces nos enteramos de que hay una tormenta, la presencia del ladrón debería volverse menos probable. La alarma se explica por otra causa, un posible relámpago.*

En el ejemplo anterior, el razonamiento es de diagnóstico y predictivo al mismo tiempo. Al sonar la alarma, podemos diagnosticar que la causa posible es la presencia de un ladrón. Al enterarnos de la tormenta, podemos predecir que la causa real fue un relámpago.

Inferencias probabilísticas
Descendiente
Independencia

Formalicemos ahora el significado de las ligas en una red bayesiana ¿Qué clase de **inferencias probabilísticas** podemos hacer dada una red? Primero definiremos que un nodo Z es `descendiente` de X si hay un camino siguiendo las ligas desde X hasta Z. Ahora, supongamos que $Y_1, Y_2, \ldots$ son padres de X en una red bayesiana. Por definición, la red implica la siguiente relación de **independencia** probabilística: X es independiente de los nodos que no son sus descendientes, dados sus padres. De forma que, para computar la probabilidad de X, basta con tomar en cuenta los descendientes de X y sus padres $(Y_1, Y_2, \ldots)$. El efecto de las demás variables en X es acumulado através de sus padres.

Ejemplo 7.13. *Supongamos que sabemos que* `sensor` *es verdadera, y que estamos interesados en la probabilidad de* `alarma`*. Todos los demás nodos en la red no son descendientes de* `alarma`*, y dado que el valor del único padre de* `alarma` *es conocido, su probabilidad no depende de ninguna otra variable. Formalmente:*

$$p(alarma \mid ladron \wedge relampago \wedge sensor \wedge llamada) = p(alarma \mid sensor)$$

Aunque las ligas en una red bayesiana pueden interpretarse naturalmente como relaciones causales, debe observarse que éste no es necesariamente el caso.

Causa raíz

Para completar la representación de un modelo probabilístico con una red bayesiana debemos, además de especificar la estructura de la red, definir algunas probabilidades de la siguiente manera: Para los nodos que no tienen padres, conocidos como **causas raíz**, es necesario especificar sus probabilidades *a priori*. Para los demás nodos debemos especificar su probabilidad condicional, tomando en cuenta el estado de sus padres, es decir $p(X \mid$ padres de $X)$. La red bayesiana de nuestro ejemplo se puede especificar en Prolog de la siguiente manera:

```
1  %%% Red bayesiana alarma-ladron
2
3  % Estructura de la Red
4
5  padre( ladron, sensor).    % el ladrón tiende a disparar el sensor
6  padre( relampago, sensor). % un relámpago fuerte también
7  padre( sensor, alarma).
8  padre( sensor, llamada).
9
10 % Probabilidades
11
```

```
p( ladron, 0.001). % Probabilidad a priori
p( relampago, 0.02).
p( sensor, [ ladron, relampago], 0.9). % Probabilidad condicionada
p( sensor, [ ladron, not relampago], 0.9).
p( sensor, [ not ladron, relampago], 0.1).
p( sensor, [ not ladron, not relampago], 0.001).
p( alarma, [ sensor], 0.95).
p( alarma, [ not sensor], 0.001).
p( llamada, [ sensor], 0.9).
p( llamada, [ not sensor], 0.0).
```

Observen que hay dos causas raíz en la red: `ladron` y `relampago`, por lo que es necesario especificar sus probabilidades *a priori*. El nodo `sensor` tiene dos padres, `ladron` y `relampago`, por lo que sus probabilidades condicionadas, dado el estado de sus padres, forman una tabla de cuatro entradas (2^n, donde n es el número de padres). Las variables `alarma` y `llamada` sólo tienen un padre, por lo que sus tablas de probabilidad condicional tienen dos entradas. Con ello, hemos tenido necesidad de representar 10 probabilidades; pero sin información sobre la independencia entre variables, hubiésemos tenido que representar $2^5 - 1 = 31$ probabilidades. La estructura de la red y las probabilidades especificadas nos han ahorrado 21 probabilidades.

7.7.2 Cálculo de probabilidades

Recordemos algunas fórmulas del cálculo de probabilidades, que nos serán de utilidad para razonar con las redes bayesianas. Sean X e Y dos proposiciones, entonces:

- $p(\neg X) = 1 - p(X)$
- $p(X \wedge Y) = p(X)p(Y)$
- $p(X \vee Y) = p(X) + p(Y) - p(X \wedge Y)$
- $p(X) = p(X \wedge Y) + (X \wedge \neg Y) = p(Y)p(X \mid Y) + p(\neg Y)p(X \mid \neg Y)$

Las proposiciones X es Y se dice **independientes** si $p(X \mid Y) = p(X)$ y $p(Y \mid X) = p(Y)$. Esto es, si Y no afecta mi creencia sobre X y viceversa. Si X e Y son independientes, entonces:

- $p(X \wedge Y) = p(X)p(Y)$

Variables disjuntas

Se dice que las proposiciones X e Y son **disjuntas** si no pueden ser verdaderas al mismo tiempo, en cuyo caso: $p(X \wedge Y) = 0$ y $p(X \mid Y) = 0$ y $p(Y \mid X) = 0$.

Sean $X_1, \ldots, X_n$ proposiciones, entonces:

$$p(X_1 \wedge \cdots \wedge X_n) = p(X_1)p(X_2 \mid X_1)p(X_3 \mid X_1 \wedge X_2) \ldots p(X_n \mid X_1 \wedge \cdots \wedge X_{n-1})$$

Si todas las variables son independientes, esto se reduce a:

$$p(X_1 \wedge \cdots \wedge X_n) = p(X_1)p(X_2)p(X_3) \ldots p(X_n)$$

Teorema de Bayes

Finalmente, necesitaremos el **teorema de Bayes**:

$$p(X \mid Y) = p(X)\frac{p(Y \mid X)}{p(Y)}$$

que se sigue de la regla de la conjunción definida previamente. El teorema es útil para razonar sobre causas y efectos. Consideremos que un ladrón es una causa de que la alarma se encienda, es natural razonar en términos de que proporción de ladrones disparan alarmas, es decir $p(alarma \mid ladron)$. Pero cuando oímos la alarma, estamos interesados en saber la probabilidad de su causa, es decir $p(ladron \mid alarma)$. Aquí es donde entra el teorema de Bayes en nuestra ayuda:

$$p(ladron \mid alarma) = p(ladron)\frac{p(alarma \mid ladron)}{p(alarma)}$$

Una variante del teorema de Bayes, toma en cuenta el **conocimiento previo** B. Esto nos permite razonar acerca de la probabilidad de una hipótesis H, dada la evidencia E, bajo el supuesto del conocimiento previo B:

Conocimiento previo

$$p(H \mid E \wedge B) = p(H \mid B)\frac{p(E \mid H \wedge B)}{p(E \mid B)}$$

7.7.3 Implementación

En esta sección implementaremos un programa que compute las probabilidades condicionales de una red bayesiana. Dada una red, queremos que este intérprete responda a preguntas del estilo de: ¿Cual es la probabilidad de una proposición dada, asumiendo que otras proposiciones son el caso?

Ejemplo 7.14. *Algunas preguntas al intérprete de razonamiento Bayesiano, podrían ser:*

- $p(ladron \mid alarma) = ?$
- $p(ladron \wedge relampago) = ?$
- $p(ladron \mid alarma \wedge \neg relampago) = ?$
- $p(alarma \wedge \neg llamada \mid ladron) = ?$

El intérprete computará la respuesta a cualquiera de estas metas, aplicando recursivamente las siguientes reglas:

1. Probabilidad de una conjunción:

$$p(X_1 \wedge X_2 \mid Cond) = p(X_1 \mid Cond) \times p(X_2 \mid X_1 \wedge Cond)$$

2. Probabilidad de un evento cierto:

$$p(X \mid Y_1 \wedge \cdots \wedge X \wedge \ldots) = 1$$

3. Probabilidad de un evento imposible:

$$p(X \mid Y_1 \wedge \cdots \wedge \neg X \wedge \ldots) = 0$$

4. Probabilidad de una negación:

$$p(\neg X \mid Cond) = 1 - p(X \mid Cond)$$

5. Si la condición involucra un descendiente de X, usamos el teorema de Bayes. Si Y es un descendiente de X en la red, entonces:

$$p(X \mid Y \wedge Cond) = p(X \mid Cond)\frac{p(Y \mid X \wedge Cond)}{p(Y \mid Cond)}$$

6. Si la condición no involucra a ningún descendiente de X, puede haber dos casos:
   a) Si X no tiene padres, entonces $p(X \mid Cond) = p(X)$, donde $p(X)$ está especificada en la red.
   b) Si X tiene padres, entonces:

$$p(X \mid Cond) = \sum_{S \in estadosPosibles(Padres)} p(X \mid S)p(S \mid Cond)$$

Ejemplo 7.15. *¿Cual es la probabilidad de un ladrón dado que sonó la alarma?* $p(ladron \mid alarma) =$ *?. Primero, por la regla 5:*

$$p(ladron \mid alarma) = p(ladron)\frac{p(alarma \mid ladron)}{p(alarma)}$$

y por la regla 6:

$$p(alarma \mid ladron) = p(alarma)p(sensor \mid ladron) + p(alarma \mid \neg sensor \mid ladron)$$

y por la misma regla 6:

$$\begin{aligned} p(sensor \mid ladron) \quad = \quad & p(sensor \mid ladron \wedge relampago)p(ladron \wedge relampago \mid ladron) + \\ & p(sensor \mid \neg ladron \wedge relampago)p(\neg ladron \wedge relampago \mid ladron) + \\ & p(sensor \mid ladron \wedge \neg relampago)p(ladron \wedge \neg relampago \mid ladron) + \\ & p(sensor \mid \neg ladron \wedge \neg relampago)p(\neg ladron \wedge \neg relampago \mid ladron) \end{aligned}$$

Aplicando las reglas 1,2,3 y 4 como corresponde, esto se reduce a:

$$\begin{aligned} p(sensor \mid ladron) \quad &= \quad 0{,}9 \times 0{,}02 + 0 + 0{,}9 \times 0{,}98 + 0 = 0{,}9 \\ p(alarma \mid ladron) \quad &= \quad 0{,}95 \times 0{,}9 + 0{,}001 \times (1 - 0{,}9) = 0{,}8551 \end{aligned}$$

Usando las reglas 1,4 y 6 obtenemos:

$$p(alarma) = 0{,}00467929$$

Finalmente

$$p(ladron \mid alarma) = 0{,}001 \times 0{,}8551 / 0{,}00467929 = 0{,}182741$$

El mecanismo de inferencia se implementa como sigue:

```
% Motor de inferencia para Red Bayesiana
:-  op(  900, fy, not).

prob( [X | Xs], Cond, P) :-  !,   % Prob de la conjunción
  prob( X, Cond, Px),
  prob( Xs, [X | Cond], PRest),
  P is Px * PRest.

prob( [], _, 1) :-  !.            % Conjunción vacía

prob( X, Cond, 1) :-
  miembro( X, Cond), !.            % Cond implica X

prob( X, Cond, 0) :-
  miembro( not X, Cond), !.         % Cond implica X es falsa

prob( not X, Cond, P) :-  !,      % Negación
  prob( X, Cond, P0),
  P is 1 - P0.

% Usa la regla de Bayes si Cond0 incluye un descendiente de X

prob( X, Cond0, P) :-
  borrar( Y, Cond0, Cond),
  pred( X, Y), !,           % Y es un descendiente de X
  prob( X, Cond, Px),
  prob( Y, [X | Cond], PyDadoX),
  prob( Y, Cond, Py),
  P is Px * PyDadoX / Py.         % Asumiendo Py > 0

% Casos donde Cond no involuchra a un descendiente

prob( X, _, P) :-
```

```
  p( X, P), !.                    % X una causa raíz - prob dada

prob( X, Cond, P) :-  !,
  findall( (Condi,Pi), p(X,Condi,Pi), CPlist),  % Conds padres
  suma_probs( CPlist, Cond, P).

% suma\_probs( CondsProbs, Cond, SumaPond)
%   CondsProbs es una lista de conds y sus probs
%   SumaPond suma de probs de Conds dada0 Cond

suma_probs( [], _, 0).

suma_probs( [ (Cond1,P1) | CondsProbs], Cond, P) :-
  prob( Cond1, Cond, PC1),
  suma_probs( CondsProbs, Cond, PResto),
  P is P1 * PC1 + PResto.

% pred(var1,var2). var1 es predecesora de var2 en la red.

pred( X, not Y) :- !,        % Y negada
  pred( X, Y).

pred( X, Y) :-
  padre( X, Y).

pred( X, Z) :-
  padre( X, Y),
  pred( Y, Z).

% miembro(X,L). X es miembro de L.

miembro( X, [X | _]).

miembro( X, [_ | L]) :-
  miembro( X, L).
```

De forma que las siguientes consultas son posibles:

```
?- prob(ladron,[alarma],P).
P = 0.182741321476588.

?- prob(alarma,[],P).
P = 0.00467929198.

?- prob(ladron,[llamada],P).
P = 0.23213705371651422.

?- prob(ladron,[llamada,relampago],P).
P = 0.008928571428571428.

?- prob(ladron,[llamada,not relampago],P).
P = 0.47393364928909953.
```

Aunque nuestra implementación es corta y concisa, es poco eficiente. No al grado de ser un problema, para redes pequeñas como las del ejemplo, peros si para redes más grandes. El problema es, a grandes rasgos, que la complejidad el algoritmo crece exponencialmente con respecto al número de padres de un nodo en la red. Esto se debe a la sumatoria sobre todos los estados posibles de los padres de un nodo.

7.8 REDES SEMÁNTICAS Y MARCOS

Representaciones estructuradas

En esta sección estudiaremos otros dos formalismos de representación de conocimiento, ampliamente usados en Inteligencia Artificial. Ambos difieren de las reglas de producción, en el hecho de que están diseñados para representar, de manera **estructurada**, grandes cantidades de hechos. Este conjunto de hechos es normalmente

compactado: Se obvian algunos hechos, cuando estos pueden ser inferidos. Ambos formalismos, redes semánticas y marcos, utilizan la **herencia**, de manera similar a como se utiliza en los lenguajes de programación orientados a objetos. *Herencia*

Ambos formalismos pueden implementarse fácilmente en Prolog, básicamente adoptando de manera disciplinada, una forma particular de estilo y organización de nuestros programas lógicos.

7.8.1 Redes semánticas

Una red semántica está compuesta por entidades y relaciones entre ellas. Normalmente, una red semántica se representa como un grafo. Existen varios tipos de redes, que siguen diferentes convenciones, pero normalmente, los **nodos** de la red representan entidades; mientras que las relaciones se denotan por medio de **ligas** etiquetadas. La figura 7.6 muestra un ejemplo de red semántica. *Nodos* *Ligas*

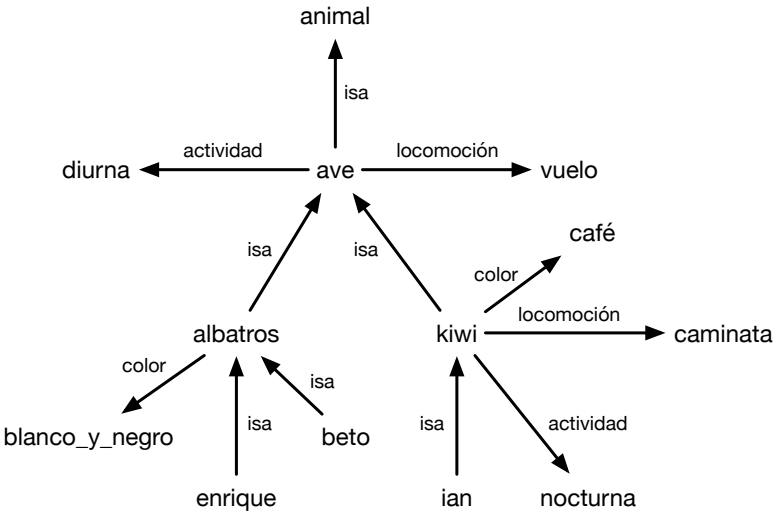


Figura 7.6: Una red semántica.

La relación especial `isa` denota la relación "es un". La red del ejemplo representa los siguientes hechos:

- Un ave es un tipo de animal.
- El vuelo es el medio de locomoción de las aves.
- Un albatros es un ave.
- Enrique y Beto son albatros.
- Un kiwi es un ave.
- La caminata es el medio de locomoción de un kiwi.
- etc.

Observen que algunas veces `isa` relaciona una clase con una super-clase, por ejemplo, `ave` con `animal`; y otras relaciona un miembro con su clase, por ejemplo: `enrique` con `albatros`. La red se puede implementar directamente en Prolog:

```
1 % Red semántica
2
3 isa(ave,animal).
4 isa(albatros,ave).
5 isa(kiwi,ave).
6 isa(enrique,albatros).
```

```
isa(beto,albatros).
isa(ian,kiwi).

actividad(ave,diurna).
actividad(kiwi,nocturna).

color(albatros,blanco_y_negro).
color(kiwi,cafe).

locomocion(ave,vuelo).
locomocion(kiwi,caminata).
```

Para poder inferir propiedades haciendo uso del mecanismo de herencia, definimos un predicado `fact/1`, que explore las propiedades de la super clase de una entidad, cuando ésta no pueda establecerse directamente. El predicado es como sigue:

```
% Interprete Redes Semánticas

fact(Hecho) :-
    Hecho,!.

fact(Hecho) :-
    Hecho =.. [Rel,Arg1,Arg2],
    isa(Arg1,SuperClaseArg1),
    SuperHecho =.. [Rel,SuperClaseArg1,Arg2],
    fact(SuperHecho).
```

Finalmente, definimos una interfaz `seRedSemantica.pl` para llamar nuestro método de inferencia basado en herencia y la base de conocimiento basada en redes semánticas:

```
% Sistema Experto basado en Frames

:- [inferenciaHerencia].
:- [kbRedSemantica].
```

Una consulta a este sistemas, es como sigue:

```
?- [seRedSemantica].
true.

?- fact(locomocion(ian,Loc)).
Loc = caminata.

?- fact(locomocion(enrique,Loc)).
Loc = vuelo.
```

7.8.2 Marcos

La representación de conocimiento basada en marcos, puede verse como un predecesor de la programación orientada a objetos. En este tipo de representación, los hechos se agrupan en objetos. Por objeto, entendemos una entidad física concreta o abstracta. Un **marco** (*frame*) es una estructura de datos, cuyos componentes se llaman **ranuras** (*slots*). Las ranuras pueden guardar información de diferentes tipos, por ejemplo:

Marco
Ranuras

- Valores.
- Referencias a otros marcos.
- Procedimientos para computar valores.

Es posible dejar vacía una ranura, en cuyo caso, ésta obtendrá su valor por medio de una inferencia. Como en el caso de las redes semánticas, la inferencia incluye herencia: Cuando un marco representa una clase de objetos, como `albatros`; y otra

representa una super clase, como ave, el marco clase hereda los valores del marco super clase.

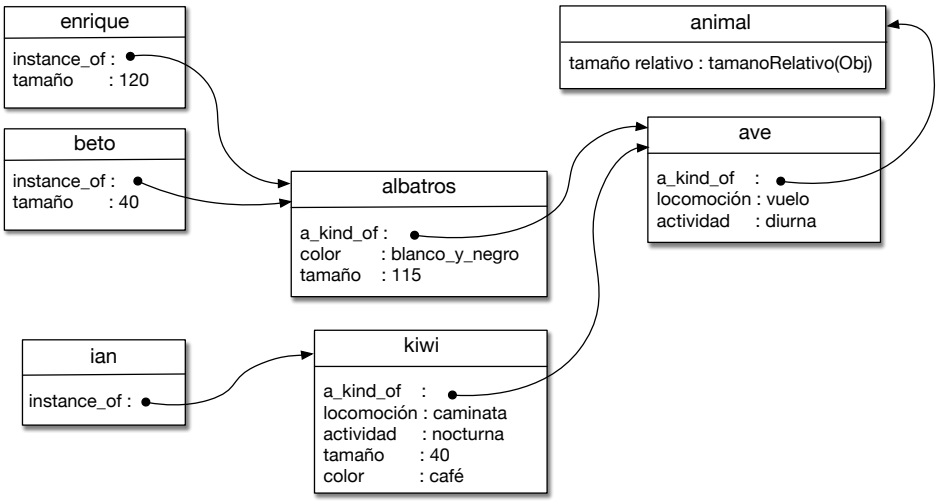


Figura 7.7: Una base de conocimiento basada en marcos. Los rectángulos son marcos y sus componentes son ranuras. que pueden contener valores, ligas a otros marcos y funciones.

Gráficamente, nuestra base de conocimientos basada en marcos, se muestra en la figura 7.7. Su implementación en Prolog es la siguiente:

```
% Base de conocimiento basada en Marcos
% Los hechos se representan como marco(ranura,valor).
% El valor puede ser un procedimiento para su cálculo.

% Marco ave: un ave prototípica

ave(a_kind_of,animal).
ave(locomocion,vuelo).
ave(actividad,diurna).

% Marco albatros: un ave prototípica con hechos extra:
% es blanco y negro; y mide 115cm

albatros(a_kind_of,ave).
albatros(color,blanco_y_negro).
albatros(tamano,115).

% Marco kiwi: un ave prototípica con hechos extra:
% camina, es nocturno, café y mide unos 40cm

kiwi(a_kind_of,ave).
kiwi(locomocion,caminata).
kiwi(actividad,nocturna).
kiwi(tamano,40).
kiwi(color,cafe).

% Marco enrique: es un albatros

enrique(instance_of,albatros).
enrique(tamano,120).

% Marco beto: es un albatros bebé

beto(instance_of,albatros).
beto(tamano,40).

% Marco ian: es un kiwi.

ian(instance_of,kiwi).

```

```
% Marco animal: su tamaño relativo se calcula ejecutando un
% procedimiento

animal(tamanoRelativo,
       execute(tamanoRelativo(Obj,Val), Obj, Val)).

% tamanoRelativo(Obj,TamRel): El tamaño relativo TamRel de Obj

tamanoRelativo(Obj,TamRel) :-
    value(Obj,tamano,TamObj),
    value(Obj,instance_of,ClaseObj),
    value(ClaseObj,tamano,TamClase),
    TamRel is (TamObj/TamClase) * 100.
```

Para usar esta base de conocimientos, es necesario definir un procedimiento para consultar los valores de las ranuras. Tal procedimiento se define como `value/3` en el motor de inferencia correspondiente:

```
% Motor de inferencia para Marcos

value(Frame,Slot,Value) :-
    Query =.. [Frame,Slot,Value],
    Query, !. % valor encontrado directamente.

value(Frame,Slot,Value) :-
    parent(Frame,ParentFrame), % Un marco más general
    value(ParentFrame,Slot,Value).


parent(Frame,ParentFrame) :-
(    Query =.. [Frame, a_kind_of, ParentFrame]
    ;
    Query =.. [Frame, instance_of, ParentFrame]
)    ,
    Query.
```

Si la relación `value(Frame,Slot,Value)` se satisface, debe ser el caso que la relación $Frame(Slot, Value)$ esté incluída en nuestra base de conocimientos. De otra forma, el valor de esa ranura deberá obtenerse recurriendo a la herencia. El predicado `parent/2` busca si hay un padre del marco que recibe como primer argument. Este predicado se satisface si el marco en cuestión es una sub clase o un caso de otro marco. Esto es suficiente para consultas del tipo:

```
?- value(enrique,actividad,Act).
Act = diurna

?- value(kiwi,actividad,Act).
Act = nocturna.
```

Consideremos ahora un caso más complicado de inferencia, cuando el valor de una ranura se obtiene computando una función. Por ejemplo, el marco `animal` tiene la ranura correspondiente a tamaño relativo, que se computa con la función `tamanoRelativo/2`. Su computo es un radio, expresado como porcentaje, entre el tamaño particular de un espécimen y el de la especie en general. Por ejemplo, el tamaño de relativo de `beto` sería $(40 \div 115) \times 100 = 34{,}78\,\%$. Para ello, es necesario utilizar la herencia y detectar cuando un valor se puede computar directamente o mediante el llamado a una función. El motor de inferencia para esto es como sigue:

```
% Motor inferencial para marcos V2: Incluye funciones

value(Frame,Slot,Value) :-
    value(Frame, Frame, Slot, Value).

value(Frame, SuperFrame, Slot, Value) :-
    Query =.. [SuperFrame, Slot, ValAux],
    Query,
    process(ValAux, Frame, Value), !.

```

```
value(Frame, SuperFrame, Slot, Value) :-
    parent(SuperFrame, ParentSuperFrame),
    value(Frame, ParentSuperFrame, Slot, Value).

parent(Frame,ParentFrame) :-
(    Query =.. [Frame, a_kind_of, ParentFrame]
    ;
    Query =.. [Frame, instance_of, ParentFrame]
)    ,
    Query.

process(execute(Proc, Frame, Value), Frame, Value) :- !,
    Proc.

process(Value,_,Value).  % un valor, no un procedimiento.
```

Ahora podemos realizar las siguientes consultas:

```
?- value(enrique,tamanoRelativo,Tam).
Tam = 104.34782608695652

?- value(beto,tamanoRelativo,Tam).
Tam = 34.78260869565217

?- value(ian,tamanoRelativo,Tam).
Tam = 100

?- value(beto,color,C).
C = blanco_y_negro.
```

La base de conocimientos basada en marcos y su motor de inferencia, incluyendo funciones, se puede llamar con el *script* `seMarcos.pl`:

```
% Sistema Experto basado en marcos.

%:- [inferenciaMarcos].
:- [inferenciaMarcosFunc].
:- [kbMarcos].
```

si se desea usar la versión sin funciones, basta cambiar los comentarios en el *script*. Evidentemente, el predicado `value/3` puede usarse en las reglas de producción definidas al principio de este capítulo.

7.9 LECTURAS Y EJERCICIOS SUGERIDOS

El contenido de este capítulo está basado en el libro de Bratko [18] (cap. 15), haciendo énfasis en el uso de la programación lógica para implementar los mecanismos usados en los sistemas expertos: bases de conocimiento, motores de inferencia, heurísticas, e interfaz. El libro de Negrete-Martínez, González-Pérez y Guerra-Hernández [73] hace una revisión en el mismo sentido, pero basada en la programación funcional. La revisión de los componentes de un sistema experto, es más profunda en este texto. Un ejercicio interesante, sería programar los mecanismos propuestos en el "Pericia Artificial" usando Prolog. El libro provee además una revisión exhaustiva de las herramientas y aplicaciones de los sistemas expertos. Los ejercicios de este capítulo se complementan con el uso de `clips` [3] un *shell* de sistemas expertos, producido por la NASA. Los capítulos 9 y 10 de la revisión de técnicas avanzadas de Prolog de Covington, Nute y Avellino [27], presentan otra implementación de un `shell` de sistema experto y su manejador de incertidumbre. Un texto obligado en este tema es la revisión de los experimentos entorno a MYCIN, propuesto por Buchanan y Shortliffe [20]. Erman et al. [37] discuten la integración del conocimiento y el manejo de la incertidumbre en Hersay-II, un sistema para el

3 http://clipsrules.sourceforge.net

entendimiento del discurso hablado. Ligeza [65] ofrece una revisión exhaustiva de los fundamentos lógicos de los sistemas basados en reglas.

Nuestro colega Sucar [104], Premio Nacional de Ciencia 2016, nos ofrece una introducción generalista a los modelos gráficos probabilísticos. Bolstad [8] nos presenta una introducción a la estadística bayesiana, donde el énfasis es más matemático y menos computacional. Un complemento ideal para este capítulo es el artículo de Poole [87], donde se aborda la integración de la lógica y la teoría de decisión bayesiana, desde una perspectiva basada en la representación del conocimiento. Recientemente, Pearl, Glymour y Jewell [81] nos ofrecen un libro sobre la estadística bayesiana, con énfasis en la inferencia causal.

Ejercicios

Ejercicio 7.1. *Implemente un mecanismo de razonamiento como el sugerido en la figura 7.4.*

Ejercicio 7.2. *Implemente los ejemplos de este capítulo usando* `clips`*.*

Ejercicio 7.3. *Use las técnicas bayesianas aquí presentadas, como alternativa al sistema experto que contiende con incertidumbre mediante factores de certeza.*

8 PLANEACIÓN

La planeación [40, 1, 48] es un tema de interés tradicional en la IA, que involucra razonar acerca de los efectos de las acciones y la secuencia en que éstas se aplican para lograr un efecto acumulativo deseado. En esta sesión desarrollaremos planificadores simples para ilustrar los principios de la planeación.

La Figura 8.1 muestra una tarea de planeación muy utilizada en IA, el mundo de los bloques. De hecho, ya hemos utilizado este escenario al hablar acerca de la lógica de primer orden. En este capítulo, el problema se usará para introducir una representación de conocimiento que nos permita razonar explícitamente acerca de los efectos de las acciones que nuestro programa puede ejecutar.

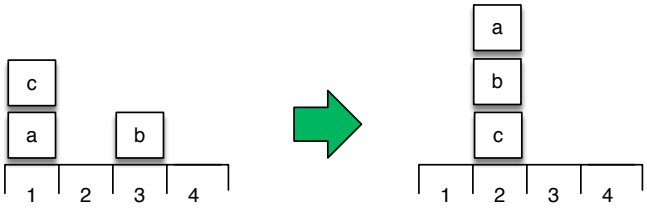


Figura 8.1: *El mundo de bloques, revisitado.*

Aunque el mundo de los bloques puede verse como un caso de los criticados mundos sintéticos usados en IA, debe observarse que encontrar un plan óptimo en estos ambientes es un problema NP-duro [51]. Además, la configuración original del problema, puede redefinirse para incluir aquellas características deseables del medio ambiente, que no están presentes en su formulación original. Por ejemplo: no determinismo, dinamismo, observación parcial, etc. La sección 2.3, discute que características pueden estar presentes al considerar un medio ambiente.

Si coincidimos con Patrick Winston [1], en que el sujeto de estudio de la IA son los algoritmos construidos a partir de restricciones expuestas por representaciones que dan soporte a modelos del razonamiento, la percepción y la acción; este capítulo introduce en nuestro curso, los componentes de acción y, de manera muy limitada, el de percepción.

Veamos un ejemplo de como las técnicas de representación y razonamiento basados en programas definitivos, pueden ser usadas para hacer algo que parece planeación, pero no lo es. ¿Cómo puedo escribir un programa que nos diga como salir del laberinto de la figura 8.2? Adoptando el *modus operandi* seguido hasta ahora, tal cuestión implica decidir qué representación del laberinto debo adoptar, para explotar alguno de los métodos de razonamiento que hemos visto.

El laberinto puede representarse como un grafo de conectividad, que establece que posiciones son contiguas. Usaremos los predicados `conecta/2` y `conectado/2` para ello. El segundo, simplemente establece que el primero es una relación simétrica:

```
1 %%% conectado/2
2 %%%   computa si la Pos1 esta conectada con la Pos2
3
4 conectado(Pos1, Pos2) :-  conecta(Pos1, Pos2).
5 conectado(Pos1, Pos2) :-  conecta(Pos2, Pos1).
6
7 %%% Datos del laberinto (adyacencia)
```

1 La definición que sigue se puede encontrar en el curso de Winston sobre Introducción a la IA en MIT: https://ai6034.mit.edu/wiki/index.php?title=Main_Page

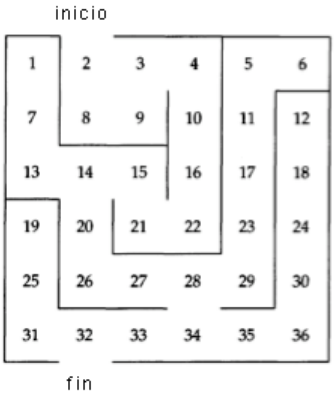


Figura 8.2: Un laberinto.

```
8
9  conecta(inicio,2).
10 conecta(1,7).
11 conecta(2,8).
12 conecta(2,3). %%% agrega varias soluciones.
13 conecta(3,4).
14 conecta(3,9).
```

Observen una de las ventajas de la representación en primer orden adoptada: `conectado/2` reduce el tamaño de nuestra base de conocimientos a la mitad. Si la representación fuese proposicional, la simetría de `conecta/2` tendría que expresarse por extensión, es decir, incluyendo en la base de conocimientos todas las relaciones de `conecta/2` en ambos sentidos. La relación `conectado/2` reduce media base de conocimiento en una regla (línea 5).

El caso es que ahora sabemos que `inicio` está conectado con 2, y 2 con 3, etc. Por lo que podemos hacer una búsqueda recursiva para encontrar un camino que me lleve de `inicio` a `fin` en el laberinto representado como un grafo de conectividad. Dada mi posición actual, podré moverme a una posición que satisface `conectado/2` y, para evitar ciclos infinitos, que no haya sido visitada anteriormente:

```
1  %%% sol/0
2  %%%  computa los caminos de solución para el laberinto.
3
4  sol :-  camino([inicio], Sol), write(Sol).
5
6  %%% camino/2
7  %%%   si el camino llego al fin, regresa el camino de fin a inicio.
8  %%%   en caso contrario busca ir a una posición que no se haya visitado
9  %%%   anteriormente (\+ miembro)
10
11 camino([fin|RestoDelCamino], [fin|RestoDelCamino]).
12 camino([PosActual|RestoDelCamino], Sol) :-
13   conectado(PosActual,PosSiguiente),
14   \+ miembro(PosSiguiente, RestoDelCamino),
15   camino([PosSiguiente,PosActual|RestoDelCamino], Sol).
```

Como suele ser el caso, yo como programador solo estoy definiendo lo que es un camino solución en el laberinto: Una secuencia de pasos conectados de `inicio` a `fin`, sin volver sobre mis pasos (línea 14). La solución se encuentra vía la aplicación de la regla de resolución-SLD de Prolog, es decir, por una búsqueda en profundidad del camino solución, con reconsideración. La relación `miembro/2` está definida en el programa; si no quieren definir su versión de ella, pueden usar `member/2`, que está definida en todos los Prolog. La consulta a este programa es como sigue:

```
1 ?- sol.
2 [fin,32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,inicio]
3 true ;
```

```
[fin,32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,2,inicio]
true ;
false.
```

Aunque los dos caminos encontrados por nuestro programa, pueden verse como secuencias de decisiones que nos sacan del laberinto, dificilmente pueden verse como un plan ¿Porqué?

Un proceso de planeación suele verse como un razonamiento explícito acerca de los efectos de las acciones y la secuencia en que estas se aplican para lograr un efecto acumulativo dado [17]. Ghallab, Nau y Traverso [48] define la planeación como un proceso deliberativo abstracto y explícito, que elige y organiza acciones anticipando sus efectos; la meta es alcanzar una serie de objetivos predefinidos de la mejor manera posible. En este caso, ni las acciones, ni las metas son consideradas explícitamente. Tampoco hay una consideración sobre el mejor plan posible. En lo que sigue, definiremos procesos basados en búsqueda que si toman en cuenta estos factores, y por lo tanto, planean.

El capítulo está organizado de la siguiente manera: Primero se aborda un formalismo para definir las acciones con las que se construirán los planes y las metas del proceso. Posteriormente, abordaremos una estrategia medios-fines para la construcción de planes, que iremos refinando a lo largo del capítulo. La primer mejora a nuestro planificador será la consideración de metas protegidas. Luego explotaremos la búsqueda en amplitud para generar planes más cortos. Posteriormente, abordaremos la regresión basada en metas para guiar la búsqueda hacia atrás que lleva a cabo el planeador. Todos los planificadores abordados hasta ahora, son ciegos, es decir, no tienen información sobre el costo o beneficio de las acciones a considerar en el plan. La siguiente mejora es introducir esta información via una búsqueda primero el mejor. Terminaremos con algunas consideraciones sobre el uso de variables en los planes y la planeación no lineal.

8.1 ESTADOS, ACCIONES Y METAS

Para mayor claridad, asumiremos que el mundo de los bloques, nuestro medio ambiente, es observable y determinista. Esto es, las acciones de nuestro programa son los únicos factores de cambio; y todo cambio puede ser **percibido** por el programa. Como se ha mencionado, estas restricciones podrán relajarse posteriormente, de ser necesario.

Percepción

Ahora bien, una acción generalmente tiene efectos locales, en el sentido que no afecta todo el medio ambiente. Una buena representación de las acciones debería tomar esto en cuenta. Para facilitar razonar acerca de estos efectos focalizados de las acciones, un **estado** del medio ambiente será representado como una lista de relaciones válidas en un momento dado. Esta lista estado solo incluirá aquellas relaciones que son relevantes [2] para nuestro problema de planificación.

Estado

Ejemplo 8.1. *Para el caso del mundo de los bloques, las relaciones a considerar son en/2 y libre/1, con su semántica intuitiva. El estado del mundo de cubos de la izquierda, en la Figura 8.1, sería:*

```
[ libre(2), libre(4), libre(c), libre(b), en(a,1), en(b,3), en(c,a) ]
```

Cada **acción** posible se define entonces en términos de las condiciones que deben observarse en el estado actual, para que ésta pueda ser ejecutada; y de sus efectos esperados. Específicamente:

Acción

- **Condiciones.** Una lista de las condiciones que debe satisfacerse, para que la acción pueda ejecutarse.

2 Qué es relevante, es la cuestión en el centro del llamado *frame problem*, identificado inicialmente por McCarthy y Hayes [71]. Aunque el problema fue resuelto originalmente por Reiter, Scherl y Levesque [97], discuten ésta y otras soluciones.

- **Agregar.** Una lista de observaciones que, se espera, ocurran después de ejecutarse la acción.
- **Borrar.** Una lista de observaciones que, se espera, dejen de ser verdaderas después de ejecutarse la acción.

Ejemplo 8.2. *En el dominio del mundo de los bloques, la única acción posible será mover/3. La definición completa de esta acción es como sigue:*

```
%%% Acción mover en mundo de bloques

precond(mover(Bloque,De,A),
        [libre(Bloque),libre(A),en(Bloque,De)]) :-
    bloque(Bloque),
    objeto(A),
    A\==Bloque,
    objeto(De),
    De\==A,
    Bloque\==De.

agregar(mover(Bloque,De,A),[en(Bloque,A),libre(De)]).

borrar(mover(Bloque,De,A),[en(Bloque,De),libre(A)]).
```

De manera que para poder mover un *Bloque* de la posición *De* a la posición *A*, es necesario que el *Bloque* y la posición *A* estén libres; y que el bloque *Bloque* esté en la posición *De*. El resto de *precond*/2 establece otro tipo de restricciones: que *Bloque* sea un bloque, y *A* y *De* sean objetos en el universo de discurso; que *A* sea diferente de *Bloque*; que se debe mover el bloque a una nueva posición (*A* es diferente de *De*); y no mover el bloque de si mismo (*Bloque* es diferente de *De*). Las definiciones de *agregar*/2 y *borrar*/2 completan la especificación de *mover*/3.

A diferencia de la búsqueda ciega para encontrar un camino solución en el laberinto de la introducción al capítulo, los procesos de planeación suelen requerir de información sobre el dominio de aplicación. A esta información se le suele conocer como **ontología** del problema, es decir, lo que sé del dominio sobre el cual quiero elaborar planes. En el caso del mundo de los bloques la ontología define lo que es un `objeto/1`, un `bloque/1` y un `lugar/1` en la mesa:

Ontología

Ejemplo 8.3. *La ontología del mundo de los bloques:*

```
%%% Ontología

objeto(X) :-  lugar(X); bloque(X).

bloque(a).
bloque(b).
bloque(c).

lugar(1).
lugar(2).
lugar(3).
lugar(4).
```

Nuestras **metas** serán representadas como una lista de observaciones que deberían darse luego de ejecutar el plan. Con estas definiciones es posible establecer el estado inicial y metas en el mundo de los bloques:

Meta

Ejemplo 8.4. *El escenario mostrado en la Figura 8.1 y la meta de colocar el bloque a en el bloque b, pueden representarse con las siguientes relaciones:*

```
estado([libre(2),libre(4),libre(b),libre(c),en(a,1),en(b,3),en(c,a)]).

metas([en(a,b)]).
```

Observen que la definición de las acciones, establece también el espacio de planes posibles, conocido como **espacio de planeación**. Ahora veremos como a partir de esta representación, es posible derivar los planes mediante un procedimiento conocido como análisis medios-fines.

Espacio de planeación

8.2 ANÁLISIS MEDIOS-FINES

Sean el estado y las metas iniciales los definidos en el ejemplo 8.4. El trabajo de un planeador consiste entonces en encontrar una secuencia de acciones que satisfagan las metas iniciales. Un planeador típico razonaría de la siguiente forma:

1. Encontrar una acción que satisfaga `en(a,b)`. Para ello usamos la relación *agregar*/2, encontrando que tal acción es de la forma `mover(a,De,b)`. Esta acción deberá formar parte de nuestro plan, pero no podemos ejecutarla inmediatamente dado nuestro estado inicial.

2. Hacer posible la ejecución de la acción `mover(a,De,b)`. Para ello usamos la relación *precond*/2 encontrando que, las condiciones para poder ejecutar esta acción son:

```
[ despejado(a), despejado(b), en(a,De) ]
```

   En el estado inicial tenemos que `despejado(b)` y que `en(a,De)` para *De*/1; pero `despejado(a)` no se satisface, así que el planeador se concentra en esta fórmula como su nueva meta.

3. Volvemos a buscar en la relación *agregar*/2 para encontrar una acción que satisfaga `despejado(a)`. Tal acción tiene la forma `mover(Bloque,a,A)`. La condición para ejecutar esta acción es:

```
[ despejado(Bloque), despeado(A), en(Bloque,a) ]
```

   la cual se satisface en nuestro estado inicial para *Boque*/*c* y *A*/2. De forma que `mover(c,a,2)` puede ejecutarse en ese estado, modificando el estado del problema de la siguiente manera:

   - Eliminamos del estado inicial las relaciones que la acción borra.
   - Incluímos las relaciones que la acción agrega al estado inicial del problema.

   esto produce la lista:

```
[ despejado(a), despejado(b), despejado(c), despejado(4), en(a,1),
  en(b,3), en(c,2) ]
```

4. Ahora podemos ejecutar la acción `mover(a,1,b)`, con lo que la meta inicial se satisface. El plan encontrado es:

```
[ mover(c,a,2), mover(a,1,b) ]
```

Este estilo de razonamiento se conoce como **análisis medios-fines**. Observen que el ejemplo planteado el plan se encuentra directamente, sin necesidad de reconsiderar (no hicimos *backtracking* en ningún momento). Esto ilustra como el proceso de razonar sobre el efecto de las acciones y las metas guían la planeación en una dirección adecuada. Desafortunadamente, no siempre se puede evitar la reconsideración. De hecho, ocurre lo contrario: la explosión combinatoria y la búsqueda son típicas en la planeación.

Análisis medios-fines

El principio general de planeación por análisis medios-fines se ilustra en la figura 8.3. Puede describirse como sigue: Para resolver una lista de *Metas* a partir de un estado inicial Edo_0, y alcanzar un estado final Edo_f, hacer lo siguiente: Si todas las *Metas* son verdaderas en Edo_0, entonces $Edo_f = Edo_0$. En cualquier otro caso:

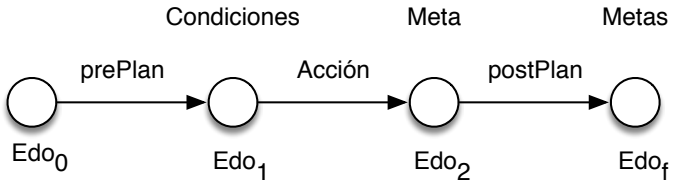


Figura 8.3: Análisis medios-fines

1. Seleccionar una *Meta* no solucionada en *Metas*.
2. Encontrar una *Accion* que agregue *Meta* al estado actual.
3. Hacer posible la ejecución de *Accion*, resolviendo su *Condicion* para obtener el estado intermedio Edo_1.
4. Aplicar la *Accion* en el estado Edo_1, para obtener el estado intermedio Edo_2 donde *Meta* se cumple.
5. Resolver *Metas* en el estado Edo_2 para llegar a Edo_f.

El código del planeador medios fines es como sigue:

```
%%% Planeador medios fines

plan(Edo,Metas,[],Edo) :-
  metas_satisfechas(Edo,Metas).

plan(Edo,Metas,Plan,EdoFinal)  :-
    append(PrePlan,[Accion|PostPlan],Plan),
    seleccionar(Edo,Metas,Meta),
    lograr(Accion,Meta),
    precond(Accion,Condicion),
    plan(Edo,Condicion,PrePlan,EdoInter1),
    aplicar(EdoInter1,Accion,EdoInter2),
    plan(EdoInter2,Metas,PostPlan,EdoFinal).

metas_satisfechas(_,[]).

metas_satisfechas(Edo,[Meta|Metas]) :-
    member(Meta,Edo),
    metas_satisfechas(Edo,Metas).

seleccionar(Edo,Metas,Meta)  :-
    member(Meta,Metas),
    not(member(Meta,Edo)).

lograr(Accion,Meta)  :-
    agregar(Accion,Metas),
    member(Meta,Metas).

aplicar(Edo,Accion,NewEdo)  :-
    borrar(Accion,ListaBorrar),
    borrar_todos(Edo,ListaBorrar,Edo1),!,
    agregar(Accion,ListaAgregar),
    append(ListaAgregar,Edo1,NewEdo).

% Borra de [X|L1] todos los elementos que aparecen en L2
% dando como resultado Diff.

borrar_todos([],_,[]).

borrar_todos([X|L1],L2,Diff)  :-
    member(X,L2),!,
    borrar_todos(L1,L2,Diff).

borrar_todos([X|L1],L2,[X|Diff])  :-
```

Como suele ser el caso, la implementación en Prolog del análisis medios fines ($plan/4$), es prácticamente una traducción directa de la descripción del proceso en castellano. El resto del código implementa los predicados auxiliares necesarios. Para invocar al planeador, ejecutamos en Prolog la siguiente meta:

```
?- estado(Edo0), metas(M), plan(Edo0,M,P,EdoF).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
M = [en(a, b)],
P = [mover(c, a, 2), mover(a, 1, b)],
EdoF = [en(a, b), libre(1), en(c, 2), libre(a), libre(4), libre(c), en(b, 3)]
```

8.3 METAS PROTEGIDAS

Consideren ahora la siguiente consulta a $plan/4$:

```
?- estado(Edo0), plan(Edo0,[en(a,b),en(b,c)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [ mover(b, 3, c), mover(b, c, 3), mover(c, a, 2), mover(a, 1, b),
       mover(a, b, 1), mover(b, 3, c), mover(a, 1, b) ]
```

Aunque el plan resultante cumple con su cometido, no es precisamente eficiente, ni elegante. De hecho, existe un plan de tres movimientos para lograr las metas de este caso. Esto se debe a que el mundo de los bloques es más complejo de lo que parece, debido a la **combinatoria**. En este problema, el planeador tiene acceso a más opciones entre diferentes acciones, que tienen sentido bajo el análisis medios-fines. Más opciones significa mayor complejidad combinatoria.

Combinatoria

Regresemos al ejemplo, lo que sucede es que el planeador persigue diferentes metas en diferentes etapas de la construcción del plan. Por ejemplo:

Acción	Objetivo medios-fines
$mover(b,3,c)$	satisfacer $en(b,c)$
$mover(b,c,3)$	satisfacer $clear(c)$ y ejecutar siguiente acción
$mover(c,a,2)$	satisfacer $clear(a)$ y $mover(a,1,b)$
$mover(a,1,b)$	satisfacer $on(a,b)$
$mover(a,b,1)$	satisfacer $clear(b)$ y $mover(b,3,c)$
$mover(b,3,c)$	satisfacer $en(b,c)$ otra vez
$mover(a,1,b)$	satisfacer $en(a,b)$ otra vez

Lo que esta tabla muestra es que a veces el planeador destruye metas que ya había satisfecho. El planeador logra fácilmente satisfacer una de las dos metas planteadas, `en(b,c)` pero la destruye al buscar como satisfacer la otra meta `en(a,b)`. Lo peor es que esta forma desorganizada de seleccionar las metas, puede incluso llevar al fracaso en la búsqueda del plan, como en el siguiente ejemplo:

```
?- estado(E), plan(E,[libre(2), libre(3)], P, _).
ERROR: Out of local stack
```

Hagan un trace de esta corrida, para saber porque la meta falla.

Una manera de evitar este comportamiento en nuestro planeador, es mantener una lista de **metas protegidas**, de forma que las acciones que destruyen estas metas no puedan ser seleccionadas al planear. El planeador medios-fines con metas protegidas se implementa de la siguiente manera:

Metas protegidas

```
%%% Medios fines con metas protegidas

plan_metas_protegidas(EdoInicial,Metas,Plan,EdoFinal):-
    plan_mp(EdoInicial,Metas,[],Plan,EdoFinal).

plan_mp(Edo,Metas,_,[],Edo) :-
```

```
7       metas_satisfechas(Edo,Metas).
8
9   plan_mp(Edo,Metas,Protegido,Plan,EdoFinal) :-
10      append(PrePlan,[Accion|PostPlan],Plan),
11      seleccionar(Edo,Metas,Meta),
12      lograr(Accion,Meta),
13      precond(Accion,Condicion),
14      preservar(Accion,Protegido),
15      plan_mp(Edo,Condicion,Protegido,PrePlan,EdoInter1),
16      aplicar(EdoInter1,Accion,EdoInter2),
17      plan_mp(EdoInter2,Metas,[Meta|Protegido],PostPlan,EdoFinal).
18
19  preservar(Accion,Metas) :-
20      borrar(Accion,ListaBorrar),
21      not( (member(Meta,ListaBorrar),
22      member(Meta,Metas))).
```

Basicamente hemos agregado una restricción al momento de elegir la acción que satisface una *Meta* dada. Además de ser ejecutable, como en el caso del análisis medios-fines original, debe preservar las metas protegidas (línea 14). Una acción preserva una *Meta* data, si no hay intersección entre su lista *borrar*/2 y las metas de la planeación.

De forma que si ejecutamos la consulta:

```
1  ?- estado(Edo0), plan_metas_protegidas(Edo0,[libre(2), libre(3)], P, _).
2  Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
3  P = [mover(b, 3, 2), mover(b, 2, 4)]
```

obtenemos una solución, aunque ésta sigue sin ser la mejor. Un sólo movimiento `mover(b,3,4)` hubiese sido suficiente para cumplir con nuestras metas. Los planes **innecesariamente largos** son resultado de la estrategia de búsqueda usada por nuestro planeador.

Planes innecesariamente largos

8.4 PROCEDIMIENTOS PRIMERO EN AMPLITUD

Los planeadores implementados hasta ahora, usan esencialmente una estrategia de búsqueda primero en profundidad, pero no por completo. Para poder estudiar lo que está pasando, debemos poner atención al orden en que se generan los planes candidatos. La meta

```
1    append(PrePlan, [Accion|PostPlan], Plan)
```

es central en este aspecto. La variable *Plan* todavía no está instanciada cuando esta meta es alcanzada. El predicado *append*/3 genera al reconsiderar, candidatos alternativos para *PrePlan* en el siguiente orden:

```
1  PrePlan = [];
2  PrePlan = [_];
3  PrePlan = [_,_];
4  PrePlan  = [_,_,_];
5  ...
```

Los candidatos cortos para *PrePlan* son los primeros en ser considerados. *PrePlan* establece las condiciones para que *Accion* pueda ejecutarse. Esto permite encontrar una acción cuyas condiciones pueden satisfacerse por un plan tan corto como sea posible, a la manera de una búsqueda primero en amplitud. Por otra parte, la lista de candidatos para el *PostPlan* está totalmente no instanciada, y por tanto su longitud es ilimitada. Por lo tanto, la **estrategia de búsqueda** resultante es globalmente primero en profundidad, y localmente primero en amplitud. Se trata de una búsqueda primero en profundidad con respecto al encadenamiento hacía adelante de las acciones que se agregan al plan emergente, donde cada acción es validada por un *PrePlan* cuya búsqueda se lleva a cabo primero en amplitud.

Estrategias de búsqueda

Una forma de minimizar la longitud de los planes es forzar al planeador, en su parte de búsqueda en amplitud, de forma que los planes cortos sean considerados antes que los largos. Podemos imponer esta estrategia embebiendo nuestro planificador en un procedimiento que genere planes candidatos ordenados por tamaño creciente. Por ejemplo:

```
%%% Planificador primero en amplitud

plan_primero_amplitud(Edo,Metas,Plan,EdoFinal) :-
    candidato(Plan),
    plan(Edo,Metas,Plan,EdoFinal).

candidato([]).

candidato([_|Resto]) :-
    candidato(Resto).
```

De forma que la siguiente consulta es posible:

```
?- estado(Edo0), plan_primero_amplitud(Edo0,[libre(2),libre(3)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(b, 3, 4)]
```

El mismo efecto puede lograrse de manera más elegante, insertando el generador de planes directamente en el procedimiento *plan*/4 de forma que:

```

%%% Planficador primero en amplitud elegante.

plan_metas_protegidas_amplitud(EdoInicial,Metas,Plan,EdoFinal):-
    plan_mp_amplitud(EdoInicial,Metas,[],Plan,EdoFinal).

plan_mp_amplitud(Edo,Metas,_,[],Edo) :-
    metas_satisfechas(Edo,Metas).

plan_mp_amplitud(Edo,Metas,Protegido,Plan,EdoFinal) :-
    append(Plan,_,_),
    append(PrePlan, [Accion|PostPlan],Plan),
    seleccionar(Edo,Metas,Meta),
    lograr(Accion,Meta),
    precond(Accion,Condicion),
    preservar(Accion,Protegido),
    plan_mp_amplitud(Edo,Condicion,Protegido,PrePlan,EdoInter1),
    aplicar(EdoInter1,Accion,EdoInter2),
```

La consulta es equivalente a la anterior:

```
?- estado(Edo0), plan_metas_protegidas_amplitud(Edo0,[libre(2),libre(3)],P,_).
Edo0 = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(b, 3, 4)]
```

Este resultado es óptimo, sin embargo la meta:

```
?- estado(E), plan_metas_protegidas_amplitud(E,[en(a,b), en(b,c)], P, _).
E = [libre(2), libre(4), libre(b), libre(c), en(a, 1), en(b, 3), en(c, a)],
P = [mover(c, a, 2), mover(b, 3, a), mover(b, a, c), mover(a, 1, b)]
```

sigue siendo problemática. Este resultado se obtiene con y sin protección de metas siguiendo la estrategia primero en amplitud. El segundo movimiento del plan parece superfluo y aparentemente no tiene sentido. Investiguemos por qué se le incluye en el plan y por qué aún en el caso de la búsqueda primero en amplitud, el plan resultante no es el óptimo.

Dos preguntas son interesantes en este problema: ¿Qué razones encuentra el planeador para construir este curioso plan? y ¿Por qué el planeador no encuentra el plan óptimo e incluye la acción `mover(b,3,a)`. Atendamos la primer pregunta. La última acción `mover(a,1,b)` atiende la meta `en(a,b)`. Los tres primeros movimientos están al servicio de cumplir las condiciones de esta acción, en particular la condición `libre(a)`. El tercer movimiento despeja a y una condición de este movimiento es

`en(b,a)`. Esto se cumple gracias al curioso segundo movimiento `mover(b,3,a)`. Esto ilustra la clase de exóticos planes que pueden emerger durante un razonamiento medios-fines.

Con respecto a la segunda pregunta, ¿Por qué después de `mover(c,a,2)`, el planeador no considera inmediatamente `mover(b,3,c)`, lo que conduce a un plan óptimo? La razón es que el planeador estaba trabajando en la meta `en(a,b)` todo el tiempo. La acción que nos interesa es totalmente superflua para esta meta, y por lo tanto no es considerada. La cuarta acción logra `en(a,b)` y ¡por pura suerte `en(b,c)`! Este último resultado no es una decisión planeada de nuestro sistema.

Incompletez medios-fines

De lo anterior se sigue, que el procedimiento medios-fines, tal y como lo hemos implementado es **incompleto**, no sugiere todas las acciones relevantes para el proceso de planificación. Esto se debe a la localidad con que se computan las soluciones. Solo se sugerirán acciones relevantes para la meta actual del sistema. La solución al problema está en este enunciado: Se debe permitir la interacción entre metas en el proceso de planificación.

Antes de pasar al tema de la interacción entre metas, consideren que al introducir la estrategia primero en amplitud para buscar planes más cortos, hemos elevando considerablemente el tiempo de computación necesario para hallar una solución. Usen *time*/1 para evaluar el impacto en términos de inferencias necesarias para alcanzar la meta de construir la torre (de 6 a 192K inferencias en mi caso).

8.5 REGRESIÓN DE METAS

Supongan que estamos interesados en una lista de *Metas* que se deben cumplir en cierto estado E_f. Sea E_0 el estado anterior a E_f y A la acción considerada para ejecutarse en E_0 ¿Qué $Metas_0$ deben considerarse en E_0 para que *Metas* se cumpla en E_f? La figura 8.4 ilustra esto.

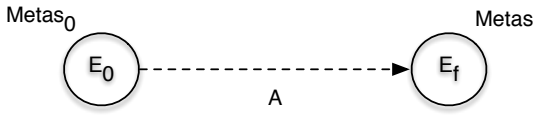


Figura 8.4: Planeación por regresión de metas.

$Metas_0$ debe tener las siguientes propiedades:

1. La acción A debe ser posible en E_0, por lo que $Metas_0$ debe incluir las condiciones de ejecución de A.
2. Para cada meta $M \in Metas$, se cumple que:
   - La acción A agrega M; ó
   - $M \in Metas_0$ y A no borra M.

Regresión de metas

El cómputo para determinar $Metas_0$ a partir de *Metas* y la acción A se conoce como **regresión de metas**. Por supuesto, sólo estamos interesados en aquellas acciones que agregan alguna meta M a *Metas*. Las relaciones entre los conjuntos que definen A y el de *Metas* se ilustran en la figura 8.5

El mecanismo de regresión de metas puede usarse como planeador de la siguiente manera. Para satisfacer una lista de *Metas* a partir de un estado inicial E_0, se procede como sigue: Si *Metas* se cumple en E_0, entonces el plan vacío es suficiente; en cualquier otro caso, seleccionar una meta $M \in Metas$ y una acción A que agregue M; entonces computar la regresión de *Metas* vía A obteniendo así $NuevasMetas$ y buscar un plan para satisfacer $NuevasMetas$ desde E_0.

El procedimiento puede mejorarse si observamos que algunas combinaciones de metas son imposibles. Por ejemplo $en(a,b)$ y $libre(b)$ no pueden satisfacerse al mismo tiempo. Esto se puede formular vía la relación:

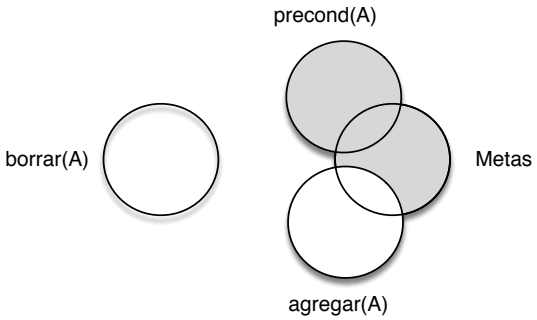


Figura 8.5: Relaciones entre los conjuntos que definen la acción A y las *Metas*. El área sombreada representa las $Metas_0$ resultado de la regresión. Observen que la intersección entre *Metas* y la lista borrar de A debe ser vacía.

```
imposible(Meta,Metas).
```

que indica que la *Meta* es imposible en combinación con las *Metas*. Para el caso del mundo de los bloques la incompatibilidad entre las metas se define como:

```
%%% metas incompatibles

imposible(en(X,X),_).

imposible(en(X,Y), Metas) :-
  member(despejado(Y),Metas)
  ;
  member(en(X,Y1),Metas), Y1 \== Y
  ;
  member(en(X1,Y),Metas), X1 \== X.

imposible(despejado(X),Metas) :-
  member(en(_,X),Metas).
```

El resto del planeador es como sigue:

```
%%% Planeador medios fines con regresión de metas

plan(Estado, Metas, []) :-
  satisfecho(Estado, Metas).

plan(Estado, Metas, Plan) :-
  append( PrePlan, [Accion], Plan),
  seleccionar( Estado, Metas, Meta),
  lograr(Accion, Meta),
  precond(Accion, Condicion),
  preservar(Accion, Metas),
  regresion(Metas, Accion, MetasReg),
  plan(Estado, MetasReg, PrePlan).

satisfecho(Estado, Metas) :-
  borrar_todos(Metas,Estado,[]).

seleccionar(_, Metas, Meta) :-
  member( Meta, Metas).

lograr( Accion, Meta) :-
  agregar( Accion, Metas),
  member( Meta, Metas).

borrar_todos( [], _, []).

borrar_todos( [X | L1], L2, Diff) :-
  member( X, L2), !,
  borrar_todos( L1, L2, Diff).

borrar_todos( [X | L1], L2, [X | Diff]) :-
  borrar_todos( L1, L2, Diff).
```

```
33
34 preservar(Accion,Metas) :-
35   borrar(Accion,ListaBorrar),
36   not( (member(Meta,ListaBorrar),
37         member(Meta,Metas))).
38
39 regresion(Metas, Accion, MetasReg) :-
40   agregar(Accion, NuevasRels),
41   borrar_todos(Metas, NuevasRels, RestoMetas),
42   precond(Accion, Condicion),
43   agregarNuevo(Condicion,RestoMetas,MetasReg).
44
45 agregarNuevo([],L,L).
46
47 agregarNuevo([Meta|_],Metas,_) :-
48   imposible(Meta,Metas),
49   !,
50   fail.
51
52 agregarNuevo([X|L1],L2,L3) :-
53   member(X,L2), !,
54   agregarNuevo(L1,L2,L3).
55
56 agregarNuevo([X|L1],L2,[X|L3]) :-
57   agregarNuevo(L1,L2,L3).
```

8.6 MEDIOS FINES CON BÚSQUEDA PRIMERO EL MEJOR

Los planeadores construídos hasta ahora hacen uso de estrategias de búsqueda básicas: primero en profundidad, primero en amplitud, o una combinación de ambas. Estas estrategias son totalmente desinformadas, en el sentido que no pueden usar información del dominio del problema para guiar su selección entre alternativas posibles. En consecuencia, estos planeadores son sumamente ineficientes, salvo en casos muy especiales. Existen diversas maneras de introducir una guía **heurística**, basada en el dominio del problema, en nuestros planeadores. Algunos lugares donde esto puede hacerse son:

Heurísticas

- En la relación $seleccionar(Estado, Metas, Meta)$ que decide el orden en que las metas serán procesadas. Por ejemplo, una guía en el mundo de los bloques es que las torres deben estar bien cimentadas, de forma que la relación $en/2$ más arriba de la torre, debería resolverse al último (o primero en el planeador por regresión, que soluciona el plan en orden inverso). Otra guía es que las metas que ya se cumplen en el medio ambiente, deberían postergarse.
- En la relación $lograr(Accion, Meta)$ que decide que acción alternativa será intentada para lograr una meta dada. Observen que nuestro planeador también genera alternativas al procesar $precond/2$. Por ejemplo, algunas acciones son "mejores" porque satisfacen más de una meta simultáneamente. También, con base en la experiencia, podemos saber que cierta condición es más fácil de satisfacer que otras.
- Decisiones acerca de qué conjunto de regresión de metas debe considerarse a continuación. Por ejemplo, seguir trabajando en el que parezca más fácil de resolver, buscando así el plan más corto.

Esta última idea muestra como podemos imponer una estrategia primero el mejor en nuestro planeador. Esto implica computar un estimado heurístico de la dificultad de los diversos conjuntos de regresión de metas alternativos, para expandir el más promisorio.

Recuerden que para usar este tipo de estrategia es necesario especificar:

1. Una relación $s/3$ entre nodos del espacio de búsqueda: $s(Nodo_1, Nodo_2, Costo)$.
2. Los nodos meta en el espacio: $meta(Nodo)$.
3. Una función heurística de la forma $h(Nodo, Hestimado)$.
4. El nodo inicial de la búsqueda.

Una forma de definir estos requisitos es asumir que los conjuntos de regresión de metas son nodos en el espacio de búsqueda. Esto es, en el espacio de búsqueda hará una liga entre $Metas_1$ y $Metas_2$ si existe una acción A tal que:

1. A agrega alguna $meta \in Metas_1$.
2. A no destruye ninguna $meta \in Metas_1$
3. $Metas_2$ es el resultado de la regresión de $Metas_1$ a través de A, tal y como definimos en nuestro planeador anterior: $regresion(Metas_1, A, Metas_2)$.

Por simplicidad, asumiremos que todas las acciones tienen el mismo costo, y en consecuencia asignaremos $Costo = 1$ en todas las ligas del espacio de búsqueda. Por lo que la relación $s/3$ se define como:

```
s(Metas1,Metas2) :-
  member(Meta,Metas1),
  lograr(Accion,Meta),
  precond(Accion,Cond),
  preservar(Accion,Metas1),
  regresion(Metas1,Accion,Metas2).
```

Cualquier conjunto de metas que sea verdadero en la situación inicial de un plan, es un nodo meta en el espacio de búsqueda. El nodo inicial de la búsqueda es la lista de metas que el plan debe lograr.

Aunque la representación anterior tiene todos los elementos requeridos, tiene un pequeño defecto. Esto se debe a que nuestra búsqueda primero el mejor encuentra un camino solución como una secuencia de estados y no incluye acciones entre los estados. Por ejemplo, la secuencia de estados (listas de metas) para logra $en(a, b)$ en el estado inicial que hemos estado usando es:

```
[ [despejado(c), despejado(2), en(c,a), despejado(b), en(a,1)]
  [despejado(a), despejado(b), en(a,1)]
  [en(a,b)] ]
```

El primer estado es verdadero por la situación inicial, el segundo es resultado de la acción $mover(c, a, 2)$ y el tercero es resultado de la acción $mover(a, 1, b)$.

Observen que la búsqueda primero el mejor regresa el camino solución en el orden inverso. En nuestro caso es una ventaja, porque los planes son construidos en la regresión hacía atrás, así que al final obtendremos la secuencia de acciones en el orden correcto. Sin embargo, es raro no tener mención explícita a las acciones en el plan, aunque puedan reconstruirse de las diferencias entre listas de metas. Podemos incluir las acciones en el camino solución fácilmente, basta con agregar a cada estado la acción que se sigue de él. De forma que los nodos del espacio de búsqueda tendrán la forma:

```
Metas -> Acción
```

Su implementación detallada es la siguiente:

```
:-  op(300,xfy, ->).

s(Metas -> AccSiguiente,MetasNuevas -> Accion, 1) :-
  member(Meta,Metas),
  lograr(Accion,Meta),
  precond(Accion,Cond),
  preservar(Accion,Metas),
```

```
  regresion(Metas,Accion,MetasNuevas).

meta(Metas -> Accion) :-
  inicio(Estado),
  satisfecho(Estado,Metas).

h(Metas -> Accion,H) :-
  inicio(Estado),
  borrar_todos(Metas,Estado,Insatisfecho),
  length(Instatisfecho,H).

inicio([en(a,1),en(b,3),en(c,a),despejado(b),despejado(c),
        despejado(2),despejado(4)]).
```

Ahora podemos usar nuestro viejo buscador primero el mejor:

```
primeroMejor(Inicio,Solucion) :-
  expandir([],hoja(Inicio,0/0),9999,_,si,Solucion).


%%% expandir(Camino,Arbol,Umbral,Arbol1,Solucionado,Solucion)
%%%    Camino es el recorrido entre Inicio y el nodo en Arbol
%%%    Arbol1 es Arbol expandido bajo el Umbral
%%%    Si la meta se encuentra, Solucion guarda el camino solución
%%%    y  Solucionado = si

%  Caso 1: la hoja con Nodo es una meta, construye una solución

expandir(Camino,hoja(Nodo,_),_,_,si,[Nodo|Camino]) :-
  meta(Nodo).

%  Caso 2: una hoja con f-valor menor o igual al Umbral
%  Generar succesores de Nodo y expandirlos bajo el Umbral

expandir(Camino,hoja(Nodo,F/G),Umbral,Arbol1,Solucionado,Sol) :-
  F  =<  Umbral,
(  bagof( M/C,(s(Nodo,M,C),not(member(M,Camino))),Succ),
   !,     % Nodo tiene sucesores
   listaSuccs(G,Succ,As), % Encontras subárboles As
   mejorF(As,F1),   % f-value of best successor
   expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
      Solucionado,Sol)
  ;
   Solucionado = nunca  % Nodo no tiene sucesores
)  .

%  Caso 3: Nodo interno con f-valor menor al Umbral
%  Expandir el subárbol más promisorio con cuyo
%  resultado, continuar/7 decidirá como proceder

expandir(Camino,arbol(Nodo,F/G,[A|As]),Umbral,Arbol1,
   Solucionado,Sol)  :-
  F  =<  Umbral,
  mejorF(As,MejorF), min(Umbral,MejorF,Umbral1),
  expandir([Nodo|Camino],A,Umbral1,A1,Solucionado1,Sol),
  continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,
      Solucionado1,Solucionado,Sol).

%  Caso 4: Nodo interno con subárboles vacío
%  Punto muerto, el problema nunca ser
á resuelto

expandir(_,arbol(_,_,[]),_,_,nunca,_) :-  !.

%  Caso 5:  f-valor mayor que el Umbral
%  Arbol no debe crecer

expandir(_,Arbol,Umbral,Arbol,no,_)  :-
  f(Arbol,F), F > Umbral.

%%% continuar(Camino,Arbol,Umbral,NuevoArbol,SubarbolSolucionado,
```

```
%%%          ArbolSolucionado,Solucion)

% Caso 1: el subartol y el arbol están solucionados
% la solución está en Sol

continuar(_,_,_,_,si,si,Sol).

continuar(Camino,arbol(Nodo,F/G,[A1|As]),Umbral,Arbol1,no,
    Solucionado,Sol)  :-
  insertarArbol(A1,As,NAs),
  mejorF(NAs,F1),
  expandir(Camino,arbol(Nodo,F1/G,NAs),Umbral,Arbol1,
     Solucionado,Sol).

continuar(Camino,arbol(Nodo,F/G,[_|As]),Umbral,Arbol1,nunca,
    Solucionado,Sol)  :-
  mejorF(As,F1),
  expandir(Camino,arbol(Nodo,F1/G,As),Umbral,Arbol1,
     Solucionado,Sol).

%%% listaSuccs(G0,[Nodo1/Costo1, ...], [hoja(MejorNodo,MejorF/G), ...])
%%%   hace una lista de árboles sucesores ordendados por F-valor

listaSuccs(_,[],[]).

listaSuccs(G0,[Nodo/C|NCs],As) :-
  G is G0 + C,
  h(Nodo,H),     % Heuristic term h(N)
  F is G + H,
  listaSuccs(G0,NCs,As1),
  insertarArbol(hoja(Nodo,F/G),As1,As).

%%% Inserta A en una lista de arboles As preservando el orden por f-valor

insertarArbol(A,As,[A|As]) :-
  f(A,F), mejorF(As,F1),
  F =< F1, !.

insertarArbol(A,[A1|As], [A1|As1])  :-
  insertarArbol(A,As,As1).


%%% Extraer f-valores

f(hoja(_,F/_),F).    % f-valor de una hoja
f(arbol(_,F/_,_),F).     % f-valor de un árbol

mejorF([A|_],F) :-  f( A, F).
mejorF([], 9999).

min(X,Y,X) :-  X  =<  Y, !.
min(_,Y,Y).
```

De forma que podemos procesar el plan con la siguiente llamada:

```
?- primeroMejor([en(a,b), en(b,c)] -> stop, Plan).
Plan = [[despejado(2), en(c, a), despejado(c), en(b, 3),
         despejado(b), en(a, 1)]->mover(c, a, 2),
        [despejado(c), en(b, 3), despejado(a), despejado(b),
         en(a, 1)]->mover(b, 3, c),
        [despejado(a), despejado(b), en(a, 1), en(b, c)]
         ->mover(a, 1, b),
        [en(a, b), en(b, c)]->stop]
```

La acción nula *stop* es necesaria pues todos los nodos deben incluir una acción. Aunque la heurística usada es simple, el programa debe ser más rápido que las versiones anteriores. Eso si, el precio a pagar es una mayor utilización de la memoria, debido a que debemos mantener el conjunto de alternativas competitivas.

8.7 VARIABLES Y PLANES NO LINEALES

A manera de comentario final, consideraremos dos casos que pueden mejorar la eficiencia de los planificadores construidos en esta sesión. El primer caso consiste en permitir que las acciones y las metas contengan variables no instanciadas; el segundo caso es considerar que los planes no son lineales.

8.7.1 Acciones y metas no instanciadas

Las variables que ocurren en nuestros planeadores están siempre instanciadas. Esto se logra, por ejemplo en la relación *precond*/2 cuyo cuerpo incluye la meta *bloque*(*Bloque*) entre otras. Este tipo de meta hace que *Bloque* siempre está instanciada. Esto puede llevar a la generación de numerosos movimientos alternativos irrelevantes. Por ejemplo, cuando al planeador se le plantea como meta *despejar*(*a*), éste utiliza *lograr*/2 para generar movimientos que satisfagan *libre*(*a*):

```
mover(De,a,A)
```

Entonces se computan las condiciones necesarias para ejecutar esta acción:

```
precond(mover(De,a,A)),Cond)
```

Lo cual fuerza, al reconsiderar, varias instanciaciones alternativas para *De* y *A*:

```
mover(b,a,1)
mover(b,a,2)
mover(b,a,3)
mover(b,a,4)
mover(b,a,c)
mover(b,a,1)
mover(b,a,2)
```

Para hacer más eficiente este paso del planeador, es posible permitir variables no instanciadas en las metas. Para el ejemplo del mundo de los bloques, las condiciones de *mover* serían definidas como:

```
precond(mover(Bloque,De,A),
        [libre(Bloque),libre(A),en(Bloque,De)]).
```

Si reconsideramos con esta nueva definición la situación inicial, la lista de condiciones computadas sería:

```
[libre(Bloque),libre(A),en(Bloque,A)]
```

Observen que esta lista de metas puede ser satisfecha inmediatamente en la situación inicial de nuestro ejemplo si *Bloque*/*c* y *A*/2. Esta mejora en eficiencia se logra postergando la decisión de cómo instanciar las variables, al momento en que ya se cuenta con más información para ello.

Este ejemplo ilustra el poder de la representación con variables, pero el precio a pagar es una mayor complejidad. Para empezar, nuestro intento por definir *precond* para *mover*/3 es erróneo, pues permite movimientos como *mover*(*c*, *a*, *c*), que da como resultado que !el bloque *c* está en el bloque *c*! Esto podría arreglarse si especificáramos que *De* y *A* deben ser diferentes:

```
precond(mover(Bloque,De,A),
        [despejado(Bloque),despejado(A),en(Bloque,De),
         diferente(Bloque,A), diferente(De,A),
         diferente(Bloque,De)]).
```

donde *diferente*/2 significa que los dos argumentos no denotan al mismo objeto Prolog. Una condición como estas, no depende del estado del problema, de forma que no puede volverse verdadero mediante acción alguna, pero debe verificarse evaluando el predicado correspondiente. Una manera de manejar estas cuasi-metas es agregar al predicado *metas_logradas*/2 la siguiente cláusula:

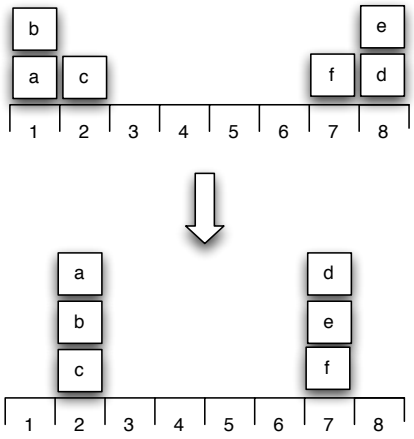


Figura 8.6: Una tarea de planeación con dos planes independientes

```
metas_logradas(Estado,[Meta|Metas]) :-
   satisface(Meta),
   metas_logradas(Estado,Metas).
```

De forma que debemos definir también:

```
satisface(diferente(X,Y)).
```

Tal relación tiene éxito si X y Y no se corresponden. Si X y Y son lo mismo, la condición es falsa. Este caso debería tratarse con imposible, pues la condición deberá seguir siendo falsa, sin importar las acciones que serán adoptadas en el plan. En otro caso, estamos ante falta de información y $satisface$ se debería postergar.

8.7.2 Planes no lineales

Un problema con nuestro planeador es que considera todos los posibles ordenes de las acciones, aún cuando las acciones son completamente independientes. Consideren el problema ilustrado en la figura 8.6, donde la meta es construir dos pilas de bloques que están de antemano bien separados. Las dos pilas puede construirse independientemente con los siguientes planes:

```
Plan1 = [mover(b,a,c), mover(a,1,b)]
Plan2 = [mover(e,d,f), mover(d,8,e)]
```

El punto importante aquí es que estos planes no interaccionan entre ellos, de forma que el orden de las acciones sólo es relevante dentro de cada plan. Tampoco importa si se ejecuta primero $Plan1$ o $Plan2$ y es incluso posible ejecutarlos de manera alternada, por ejemplo:

```
[mover(b,a,c), mover(e,d,f), mover(d,8,e), mover(a,1,b)]
```

Sin embargo, nuestro planeador considerará las 24 permutaciones posibles de las cuatro acciones, aunque existan solo 4 alternativas: 2 permutaciones para cada uno de los planes. El problema se debe a que el planeador insiste en el orden total de las acciones en el plan. Una mejora se lograría si, en los casos donde el orden no es importante, la precedencia entre las acciones se mantiene indefinida. Entonces nuestros planes serán conjuntos de acciones parcialmente ordenadas. Los planeadores que aceptan este tipo de representación se conocen como planeadores **no lineales**.

Consideremos nuevamente el ejemplo de la figura 8.6. Analizando las metas $en(a,b)$ y $en(b,c)$ el planeador no lineal concluye que las siguientes dos acciones son necesarias en el plan:

```
M1 = mover(a,X,b)
```

```
M2 = mover(b,Y,c)
```

No hay otra forma de resolver ambas metas, pero el orden de estas acciones es aún indeterminado. Ahora consideren las condiciones de ambas acciones. La condición de $mover(a, X, b)$ incluye $libre(a)$, la cual no se satisface en la situación inicial, por lo que necesitamos una acción de la forma:

```
M3 = mover(Bloque,a,A).
```

que precede a $M1$. Ahora tenemos una restricción en el orden de las acciones:

```
antes(M3,M1)
```

Ahora revisamos si $M3$ y $M1$ pueden ser el mismo movimiento. Como este no es el caso, el plan tendrá que incluir tres movimientos. Ahora el planeador debe preguntarse si hay una permutación de $[M1, M2, M3]$ tal que $M3$ preceda a $M1$, tal que la permutación es ejecutable en el estado inicial del problema y las metas se cumplen en el estado resultante. Dadas las restricciones de orden anteriores tres permutaciones de seis, cumplen con los requisitos:

```
[M3,M1,M2]
[M3,M2,M1]
[M2,M3,M1]
```

Y de estas permutaciones, solo la del medio cumple con el requisito de ser ejecutable bajo la sustitución $Bloque/c, A/2, X/1, Y/3$. Como se puede intuir, la complejidad computacional no puede ser evitada del todo por un planeador no lineal, pero puede ser aliviada considerablemente.

8.8 LECTURAS Y EJERCICIOS SUGERIDOS

La planeación es un problema muy estudiado en Inteligencia Artificial. Los planificadores aquí construídos fueron introducidos por Bratko [17] en el capítulo dedicado al tema. Una revisión exhaustiva y reciente de este problema puede encontrarse en el libro de Ghallab, Nau y Traverso [48]. Una lectura más tradicional se puede encontrar en el compendio de artículos revisado por Allen, Hendler y Tate [1]. La descripción de las acciones en estos planificadores es muy cercana a la propuesta de STRIPS, originalmente presentada por Fikes y Nilsson [40]. El mundo de los bloques ha sido ampliamente discutido en este contexto, un interesante estudio sobre su complejidad y la fuente de esta fue realizado por Ghallab, Nau y Traverso [48].

Ejercicios

Ejercicio 8.1. *Pruebe estos planificadores en otros dominios de aplicación, por ejemplo, el proceso de inscripción a la universidad. Describe las acciones en los términos adecuados y las metas del proceso.*

Ejercicio 8.2. *¿Cual de los planificadores resulta de interés para su tema de investigación? Justifique la respuesta.*

Ejercicio 8.3. *Lea el artículo de Ghallab, Nau y Traverso [48]. Explique el concepto de* deadlock *y analice si alguno de los planificadores presentados pueden contender con él.*

9 REPRESENTACIONES BDI

Las Lógicas BDI nos permiten razonar acerca de nuestros agentes racionales, permitiendo especificar y verificar el comportamiento de estos sistemas. Sus componentes modales Intencionales, temporales y de acción, son lo suficientemente **expresivos** como para abordar creencias, deseos, intenciones y la toma de decisión en esos términos. Desafortunadamente, la **complejidad computacional** de su teoría de prueba es elevada, aunque no mayor que el de su componente temporal; y la evidencia clara de su completitud fue tardía [88].

Expresividad y Costo de las Lógicas BDI

A pesar de ello, se propusieron una serie de **arquitecturas BDI** como implementaciones de este modelo de agencia, siendo PRS [44, 47, 46, 55, 62], y su exitosa revisión conocida como dMARS [45], el caso más representativo. En estas implementaciones es común encontrarse con simplificaciones difíciles de justificar, por ejemplo, las actitudes proposicionales no son operadores modales, sino estructuras de datos; Se usan meta-planes o programas definidos por el usuario para acelerar el cómputo llevado a cabo; etc. Si consideramos además la complejidad del código asociado a estos sistemas, el resultado es una aparente falta de rigor teórico; o por lo menos una evidente **divergencia** entre el estudio formal de los agentes BDI y su implementación.

Arquitecturas BDI

Fosa teórico-práctica

Los primeros intentos para resolver este problema, consistieron en proponer una **arquitectura BDI abstracta**, como una idealización que permitiera seguir investigando las propiedades teóricas de estos agentes, atendiendo al mismo tiempo los aspectos prácticos de su implementación. Un resultado sobresaliente es la especificación de dMARS realizada por d'Inverno et al. [33]. Para ello, se optó por modelar esta arquitectura, usando el lenguaje de especificación Z [66] [1], que combina descripciones textuales del sistema especificado con esquemas que hacen uso de la teoría de conjuntos, la lógica de primer orden y el cálculo lambda. Una visión completa sobre el uso de Z para el modelado de agentes puede revisarse en el libro de d'Inverno y Luck [35]. Sin embargo, debido a su nivel de abstracción, no fue posible establecer una **correspondencia** entre la teoría del modelo, la teoría de prueba y la arquitectura abstracta propuesta. No obstante, estas herramientas son interesantes, en tanto que especificaciones precisas, muy cercanas a la implementación [2].

Arquitecturas abstractas

Teoría de Correspondencia

Siendo uno de los personajes centrales en el desarrollo de las Lógicas y las Arquitecturas BDI, Rao [89] propuso entonces una ruta alternativa hacía una solución para cerrar la fosa teórico-práctica: Partir del análisis de una implementación BDI existente, dMARS, para formalizar su semántica operacional. El resultado es $AgentSpeak(L)$, un lenguaje de programación basado en una lógica restringida de primer orden con eventos y acciones. El comportamiento de un agente está dado por su programa escrito en este lenguaje. Las creencias, deseos e intenciones no son expresiones modales, pero pueden verse como tales desde una postura Intencional:

$AgentSpeak(L)$

- El modelo del agente mismo, del ambiente en que se encuentra y de otros agentes, constituyen las **creencias** del agente.

Creencias

- Los estados a los que un agente quiere llegar, con base en sus estímulos internos y externos, puede verse como los **deseos** del agente.

Deseos

1 Z se estandarizó, obteniendo una especificación ISO, en 2002. El portal Community Z Tools (CZT) ofrece todo lo necesario para trabajar con este lenguaje de especificación: http://czt.sourceforge.net/index.html

2 De hecho, para los interesados, d'Inverno y Luck [34] han modelado en Z el formalismo que nos ocupa en este capítulo, $AgentSpeak(L)$.

- Y los planes que el agente adopta para satisfacer su estímulos, pueden verse como sus **Intenciones**. *Intenciones*

De esta forma, $AgentSpeak(L)$ puede es un lenguaje de especificación ejecutable, cuyo comportamiento es similar a las Arquitecturas BDI en sus detalles fundamentales, acercando así los aspecto teóricos de la agencia BDI, a los detalles de su implementación. Aunque esta aproximación es más cercana a la **Programación Orientada a Agentes** (AOP) propuesta por Shoham [98], debe resaltarse que se ha abandonado toda aproximación modal a la Intencionalidad. *AOP*

En el resto del capítulo describiremos formalmente la sintaxis y la semántica operacional de $AgentSpeak(L)$, tal y como se ha implementado en *Jason* [12], el conocido intérprete de este lenguaje AOP implementado en Java. Posteriormente, propondremos una aproximación modal a la Intencionalidad de los programas $AgentSpeak(L)$, con base en su semántica operacional. Tal aproximación [50] nos permitirá razonar acerca de los agentes implementados en este lenguaje de programación. *Organización* *Jason* $CTL_{AgentSpeak(L)}$

9.1 SINTAXIS

El **alfabeto** de $AgentSpeak(L)$ asume un conjunto ilimitado de símbolos de variables (Var), funciones ($Func$), predicados ($Pred$) y acciones ($Actn$). Los símbolos de las constantes, como se verá más adelante, están incluidos en los de las funciones ($Const \subseteq Func$). También forman parte del alfabeto los conectivos lógicos usuales $\{\neg, \wedge\}$; El símbolo de negación fuerte $\sim$; y algunos símbolos especiales, que incluyen $\{!, ?, +, -, :, ;, \leftarrow, : -, \top\}$. *Alfabeto*

Las **variables** se denotan mediante cadenas de caracteres que inician con una mayúscula. El símbolo $\top$ denota elementos vacíos en el lenguaje, como una secuencia vacía. Se adoptan las definiciones estándar para término, átomo y literal, pero los átomos pueden adornarse para indicar su origen –La percepción del agente (`percept`), su propia base de creencias (`self`), u otro agente (id), tal y como lo ilustra la figura9.1): *Variables*

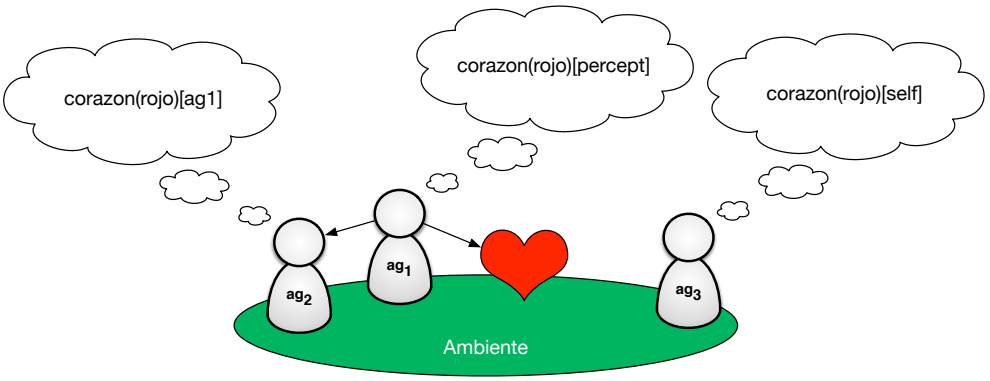


Figura 9.1: Las fuentes de un átomo en $AgentSpeak(L)$ pueden ser la percepción(ag_1), la comunicación (ag_2) o el razonamiento del propio agente (ag_3).

9.1.1 Términos, átomos y literales

Definición 9.1 (Término). *El conjunto de **términos** en $AgentSpeak(L)$ incluye las variables, las constantes y los términos compuestos:* *Término*

$$t \quad ::= \quad v \mid c \mid f(t_1, \ldots, t_n) \quad (n \geq 1) \tag{9.1}$$

donde $v \in Var$ es una variable; $c \in Const$, es una constante; y $f \in Func$ es una función de aridad $n \geq 1$, aplicada a $t_{1 \leq i \leq n}$ términos. $Const \subseteq Func$ es un conjunto de funciones de aridad cero.

Definición 9.2 (Átomo). *El conjunto de átomos en $AgentSpeak(L)$ incluye las proposiciones y los predicados, con anotaciones o sin ellas:* *Átomos*

$$\begin{aligned} at &::= p \mid p(t_1, \ldots, t_n) \quad (n \geq 1) && (9.2)\\ &\mid at[s_1, \ldots, s_m] \quad (m \geq 1) && (9.3)\\ s &::= \texttt{percept} \mid \texttt{self} \mid id && (9.4) \end{aligned}$$

donde $p \in Pred$ de aridad $n \geq 1$ es un predicado, aplicado a $t_{1 \leq i \leq n}$ términos; y si es de aridad cero, es una proposición atómica. Los átomos se adornan con al menos una anotación s, que indica su fuente de origen.

En la versión de $AgentSpeak(L)$ adoptada por *Jason*, en realidad todos los átomos tienen al menos una anotación, su fuente de origen (s). Las versiones de at no anotadas se incluye para tener compatibilidad con la versión original del lenguaje.

Observen que los términos compuestos y los átomos tienen la misma apariencia sintáctica, pero son de naturaleza diferente. Un término compuesto, al igual que los simples, denota un elemento del Universo de Discurso ($\mathcal{U}$), ya sea porque es una función del estilo $f : \mathcal{U}^n \mapsto \mathcal{U}$; o bien porque es una estructura de datos del estilo $f : \mathcal{U}^n$. Los átomos denotan relaciones entre elementos del universo de discurso, son predicados del estilo $p : \mathcal{U}^n \mapsto Bool$.

Ejemplo 9.1. *Si asumimos que $\mathcal{U} = \mathbb{N}$, la función $suma(4,3)$ es un término que denota al 7; y la función $cons(1, cons(2, [])) = [1,2]$ también es un término que denota a la lista $[1,2]$. El predicado $mayorQue(4,3)$ es un átomo que mapea a verdadero; y el predicado $vacio([1,2])$ también es un átomo que mapea a falso.*

Definición 9.3 (Literal). *Un átomo o su negación fuerte [3] son una literal. La negación se conoce como literal negativa y el átomo no negado como literal positiva:* *Literales*

$$lit ::= at \mid \sim at \quad (9.5)$$

9.1.2 Fórmulas bien formadas y creencias

Con las definiciones anteriores podemos abordar el concepto de fórmula lógica bien formada (fbf):

Definición 9.4 (Fórmula lógica). *Las fórmulas lógicas bien formadas en $AgentSpeak(L)$ incluyen los átomos, su negación y su conjunción:* *Fórmulas lógicas*

$$fbf ::= lit \mid \neg fbf \mid fbf \wedge fbf \quad (9.6)$$

donde lit es una literal; $\neg$ denota la negación débil y $\wedge$ la conjunción.

La versión original de $AgentSpeak(L)$ no consideraba la negación fuerte, que fue introducida por *Jason*. Si esta sintaxis require compatibilidad con la versión original del lenguaje, hay que substituir las literales (lit) de la definición, por átomos (at). De esa forma, solo la negación lógica débil forma parte de las fórmulas lógicas bien formadas.

Por otra parte, *Jason* provee un operador explícito de disyunción || como parte de la las fórmulas lógicas bien formadas. En esta formalización del lenguaje hemos incluido únicamente la conjunción y la negación, asumiendo que los demás operadores lógicos pueden definirse en esos términos. Por ejemplo, $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$.

9.1.3 Programa de agente

Definición 9.5 (Programa de agente). *En $AgentSpeak(L)$, un **programa de agente** es un conjunto de creencias (bs), planes (ps) y metas (gs):* *Programa de agente*

$$ag \quad ::= \quad bs \quad ps \quad gs \tag{9.7}$$

Definición 9.6 (Creencias). *Una **creencia** en $AgentSpeak(L)$ es una literal de base o una regla:* *Creencias*

$$b \quad ::= \quad lit \mid rule \tag{9.8}$$

$$rule \quad ::= \quad lit_1 \text{ :- } fbf \tag{9.9}$$

$$bs \quad ::= \quad b_1, \ldots, b_n \quad (n \geq 0) \tag{9.10}$$

tal que lit no tiene variables sin instanciar como argumentos.

La versión original de $AgentSpeak(L)$ no consideraba el uso de reglas, si se requiere compatibilidad con esa versión del lenguaje hay que eliminar la ecuación 9.9 y la referencia a *rule* en la ecuación 9.8 de la definición anterior. De igual manera, si se requiere considerar solamente la negación débil, hay que substituir literales (*lit*) por átomos (*at*).

Aunque las creencias de un agente se asemejan a los hechos y reglas de la **programación lógica**, hay que recordar que los átomos de $AgentSpeak(L)$ están anotados. *Diferencias con Prolog*

Definición 9.7 (Planes). *Un **plan** en $AgentSpeak(L)$ es una estructura compuesta por un evento disparador (te), un contexto (ct) y un cuerpo (h):* *Planes*

$$p \quad ::= \quad te : ct \leftarrow h. \tag{9.11}$$

$$ps \quad ::= \quad p_1, \ldots, p_n \quad (n \geq 1) \tag{9.12}$$

Definición 9.8 (Evento disparador). *El **evento disparador** de un plan en $AgentSpeak(L)$ consiste en agregar, o eliminar, una creencia o una meta:* *Evento disparador*

$$te \quad ::= \quad +b \mid -b \mid +g \mid -g \tag{9.13}$$

Definición 9.9 (Contexto). *El **contexto** de un plan en $AgentSpeak(L)$ puede ser vacío o una fórmula lógica bien formada:* *Contexto*

$$ct \quad ::= \quad \top \mid fbf \tag{9.14}$$

Definición 9.10 (Cuerpo). *El **cuerpo** de un plan en $AgentSpeak(L)$ es una secuencia, posiblemente vacía, de acciones, actualizaciones de creencias y/o metas:* *Cuerpo*

$$h \quad ::= \quad h1; \top \mid \top \tag{9.15}$$

$$h1 \quad ::= \quad a \mid u \mid g \mid h1; h1 \tag{9.16}$$

Definición 9.11 (Acciones). *Las **acciones** en $AgentSpeak(L)$ son llamadas a procedimientos:* *Acciones*

$$a \quad ::= A(t_1, \ldots, t_n) \quad (n \geq 0) \tag{9.17}$$

donde $A \in Actn$ es una acción aplicada a $t_{1 \leq i \leq n}$ términos.

Definición 9.12. *Las **actualizaciones de creencias** en $AgentSpeak(L)$ consisten en agregar una nueva creencia o eliminar aquellas que unifican con una literal dada:* *Actualizaciones*

$$u \quad ::= \quad +b \mid -lit \tag{9.18}$$

Definición 9.13 (Metas). *Las **metas** en $AgentSpeak(L)$ pueden ser alcanzables (!) o verificables (?):* *Metas*

$$g \quad ::= \quad !lit \mid ?lit \tag{9.19}$$

$$gs \quad ::= \quad g_1, \ldots, g_n \quad (n \geq 0) \tag{9.20}$$

La compatibilidad con la versión orignal de $AgentSpeak(L)$, requiere substituir las referencia a literal (*lit*) por átomo (*at*). El cuadro 9.1 resume la gramática de $AgentSpeak(L)$ aquí desarrollada. El ejemplo 9.2, muestra un programa $AgentSpeak(L)$ completo. *Gramática*

3 En contraposición con la negación tradicional o débil, denotada por $\neg$.

ag	::=	$bs \quad ps \quad gs$	
bs	::=	$b_1, \ldots, b_n$	$(n \geq 0)$
b	::=	$lit \mid rule$	$ground(lit)$
lit	::=	$at \mid \sim at$	
at	::=	$p \mid p(t_1, \ldots, t_n)$	$(p \in Pred, n \geq 1)$
	$\mid$	$at[s_1, \ldots, s_m]$	$(m > 1)$
s	::=	`percept` $\mid$ `self` $\mid id$	
$rule$	::=	lit :- fbf	
fbf	::=	$lit \mid \neg fbf \mid fbf \wedge fbf$	
t	::=	$v \mid c \mid f(t_1, \ldots, t_n)$	$(v \in Var, c \in Const, f \in Func, n \geq 1)$
ps	::=	$p_1, \ldots, p_n$	$(n \geq 1)$
p	::=	$te : ct \leftarrow h$	
te	::=	$+lit \mid -lit \mid +g \mid -g$	
ct	::=	$fbf \mid \top$	
h	::=	$h_1; \top \mid \top$	
h_1	::=	$a \mid u \mid g \mid h_1; h_1$	
a	::=	$ac(t_1, \ldots, t_n)$	$(ac \in Actn, n \geq 0)$
u	::=	$+b \mid -lit$	
gs	::=	$g_1, \cdots, g_n$	$(n \geq 0)$
g	::=	$!lit \mid ?lit$	

Cuadro 9.1: Gramática BNF de $AgentSpeak(L)$ para el lenguaje AOP *Jason*. Adaptada y extendida de Bordini, Hübner y Wooldridge [12]

9.1.4 Conjuntos, Pilas y Colas

Algunas definiciones que usaremos más adelante incluyen operaciones sobre conjuntos, pilas y colas. Para los conjuntos usaremos la notación y operaciones estándar; mientras que para pilas y colas haremos uso de las siguientes definiciones:

Definición 9.14 (Pila). *La expresión $p = [e_1 \ddagger e_2 \ddagger \ldots \ddagger e_n]$ denota una **pila** p de tamaño n. La pila vacía se denota como $\emptyset$. Se definen las siguientes operaciones sobre las pilas y sus elementos:* *Pila*

- $top(p) = e_1$.
- $pop(p) = e_1$ *y* $p = [e_2 \ddagger \ldots \ddagger e_n]$.
- $push(e, p) = [e \ddagger e_1 \ddagger e_2 \ddagger e_n]$.

Definición 9.15 (Cola). *La expresión $c = \{e_1 \ddagger e_2 \ddagger \ldots \ddagger e_n\}$ denota una **cola** c de tamaño n. La cola vacía se denota como $\emptyset$. Se definen las siguientes operaciones sobre las colas y sus elementos:* *Cola*

- $fst(c) = e_1$.
- $pop(c) = e_1$ *y* $c = \{e_2 \ddagger \ldots \ddagger e_n\}$.
- $push(e, c) = \{e_1 \ddagger e_2 \ddagger \ldots \ddagger e_n \ddagger e\}$.

Observen que $push/2$ y $pop/2$ son operaciones destructivas, las pilas y las colas son modificadas como resultado de su ejecución. Las operaciones $top/1$ y $fst/1$ no lo son.

9.2 SEMÁNTICA

La ejecución de un programa de agente $AgentSpeak(L)$ está determinado por una **se-** *Semántica operacional*

Ejemplo 9.2. *El siguiente código $AgentSpeak(L)$, tal y como se escribiría en su intérprete Jason [12]:*

```
// Agente cap04/src/asl/beto.asl

/* creencias iniciales y reglas */
factorial(1,1).
factorial(Num,Fact) :-
    Fact = Num * FactAux & factorial(Num-1,FactAux).

/* metas iniciales */
!start.

/* planes */
+!start : val(X) <-
    ?factorial(X,F);
    .print("El factorial de ", X, " es ",F,").
```

es un programa válido y ejecutable que computa el factorial de 5:

```
[beto] El factorial de 5 es 120.
```

Observen que el operador de conjunción se denota por &. La aritmética está predefinida, lo mismo que la acción `.print`.

mántica operacional, al estilo de las propuestas por Plotkin [86]. Estas semánticas se basan en un sistema de transiciones entre configuraciones de un programa. Una configuración puede verse como el estado de un agente en un momento dado:

Definición 9.16 (Sistema de transiciones)**.** *Un **sistema de transiciones** es una estructura $\langle \Gamma, \rightarrow \rangle$, donde Γ es un conjunto de elementos γ, conocidos como **configuraciones**, y $\rightarrow$ es una relación binaria sobre Γ, llamada **transición**.*

Sistema de transiciones

9.2.1 Configuraciones

En el caso de $AgentSpeak(L)$, una **configuración** $\gamma = \langle ag, C, M, T, s \rangle$ está compuesta por:

Configuración

- Un **programa** del agente $ag = \langle bs, ps, gs \rangle$ formado por las creencias bs, los planes ps y las metas gs del agente, tal y como se definen en la gramática del lenguaje (Ecuación 9.7) y se ilustra en el ejemplo 9.2.

  Programa del agente

- Una **circunstancia** del agente C es una tupla $\langle I, E, A \rangle$ donde: I es el conjunto de **intenciones** $\{i_1, i_2, \ldots, i_n\}$, tal que cada $i \in I$ es una pila de planes parcialmente instanciados; E es una cola de **eventos** $\{\langle te_1, i_1 \rangle, \langle te_2, i_2 \rangle, \ldots, \langle te_n, i_n \rangle\}$, tal que cada te es un evento disparador y cada i es una intención. Si $i = \top$ se dice que el evento es externo; en caso contrario el evento es interno, es decir, generado por una intención previa; y A es el conjunto de **acciones** a ser ejecutadas por el agente en el ambiente.

  Circunstancia *Intenciones* *Eventos* *Acciones*

- M es una tupla $\langle In, Out, SI \rangle$ donde In es el **buzón** del agente, Out es una lista de mensajes a ser enviados, e SI es un registro de intenciones suspendidas [4], es decir, aquellas que esperan un mensaje de respuesta para reanudarse. Los detalles sobre la comunicación se abordarán en la sección 9.3.

  Buzón

- T es una tupla de **registros** $\langle R, Ap, \iota, \epsilon, \rho \rangle$ donde R es el conjunto de **planes relevantes** dado cierto evento; $Ap \subseteq R$ es el conjunto de **planes aplicables**, aquellos que el agente cree poder ejecutar; ι, ϵ y ρ son, respectivamente, la intención, el evento, y el plan actualmente considerados en el razonamiento del agente.

  Registros

4 También se usará para guardar las intenciones suspendidas en espera de que una acción termine de ejecutarse.

- $s \in \{SelEv, RelPl, AppPl, SelAppl, SelInt, AddIM, ExecInt, ClrInt, ProcMsg\}$ denota una **etiqueta** para la configuración. *Etiqueta*

La relación de transición $\rightarrow$ se definen en términos de un conjunto de **reglas semánticas** con la forma: *Reglas semánticas*

$$(\textbf{regla}) \quad \frac{cond}{\gamma \rightarrow \gamma'}$$

donde la configuración γ puede transformarse en una nueva configuración γ', si las condiciones expresadas en *cond* se cumplen. Para definir las reglas semánticas, adoptaremos la siguiente notación:

- Para hacer referencia al componente E de una circunstancia C, escribimos C_E. De manera similar accedemos a los demás componentes de una configuración o del programa de agente. Por ejemplo, las creencias de un agente se denotan como ag_{bs}.
- Para indicar que no hay ninguna intención siendo considerada en la ejecución del agente, se emplea $T_i = \top$. De forma similar para T_ϵ, T_ρ y demás registros de una configuración.
- Se usa $i[p]$ para denotar que p es el plan en el tope de la intención i; también se puede usar $p = top(i)$.
- Si asumimos que p es un plan de la forma $te : ct \leftarrow h$, entonces:
  - $TrEv(p) = te$ denota el evento disparador de p.
  - $Ct(p) = ct$ denota el contexto de p.
  - $Head(p) = te : ct$ denota la cabeza de p.
  - $Body(p) = h$ denota el cuerpo de p.

Finalmente, se definen las siguientes funciones de selección de eventos $S_E = pop(C_E)$, planes aplicables $S_{Ap} = pop(T_{Ap})$ e intenciones $S_I = pop(C_I)$. Observen que estas funciones de selección son **destructivas**, en el sentido que el elemento seleccionado es eliminado de la pila, cola o conjunto donde se encontraba. *Selección destructiva*

9.2.2 Consecuencia lógica, planes relevantes y aplicables

Antes de definir las reglas semánticas que componen la relación de transición de la semántica operacional de $AgentSpeak(L)$, es necesario introducir algunas definiciones auxiliares cercanas a la programación lógica, por estar asociadas a los conceptos de unificador más general y consecuencia lógica. Estos conceptos se usarán para computar conjuntos planes relevantes, planes aplicables y la respuesta a las metas verificables del agente.

Unificador más general

El problema de **unificación** se define así [57]: Dados dos términos, ¿Existe una substitución que los haga sintácticamente iguales? *Unificación*

Ejemplo 9.3. *Consideren los siguientes términos, $factorial(Num, Fact)$ y $factorial(5, F)$, si substituimos Num por 5 y $Fact$ por F, conseguimos que ambos términos sean idénticos sintácticamente. En cambio, el término $factorial(1, 1)$ no se puede unificar con $factorial(5, F)$. No hay manera de substituir 1 por 5 para lograr que ambos términos sean idénticos.*

Formalicemos primero el concepto de substitución:

Definición 9.17 (Substitución). *Una **substitución** es un conjunto finito de la forma:* *Substitución*

$$\theta = \{t_1/v_1, \ldots, t_n/v_n\}, \quad (n \geq 0)$$

*donde las v_i son variables únicas y cada t_i es un término que no incluye a v_i. La forma t_i/v_i se conoce como **ligadura** [5] de v_i y expresa que el término t_i substituye a la variable v_i. La substitución vacía se denota por ϵ, y se conoce también como **substitución identidad**.* *Ligadura* *Substitución identidad*

Ejemplo 9.4. *Los siguientes conjuntos son substituciones válidas:*

- $\theta = \{f(Y)/X, Z/Y\}$
- $\theta' = \{a/X, b/Y, Y/Z\}$

En cambio, las siguientes no lo son:

- $\theta = \{f(X)/X, Z/Y\}$
- $\theta' = \{a/X, b/Y, Y/X\}$

Las substituciones pueden verse también como funciones con dominio y rango en el conjunto de términos del lenguaje, vía su aplicación:

Definición 9.18 (Aplicación de substitución). *La **aplicación** de la substitución θ al término t se denota $t\theta$ y resulta en un nuevo término donde todas las ligaduras de la substitución se han llevado a cabo en t de forma paralela.* *Aplicación*

Ejemplo 9.5. *Sea la substitución $\theta = \{5/Num, F/Fact\}$ y el término $t = factorial(Num, Fact)$, $t\theta = factorial(5, F)$. En tanto que, $t\epsilon = factorial(Num, Fact)$, de ahí el nombre de substitución identidad.*

Las substituciones pueden ser sujeto de la operación de composición para crear nuevas substituciones:

Definición 9.19 (Composición). *La **composición** de dos substituciones, $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ y $\theta' = \{t'_1/v'_1, \ldots, t'_m/v'_m\}$, es a su vez una substitución, definida de la siguiente manera:* *Composición*

$$\theta \circ \theta' = \{t_1\theta'/v_1, \ldots, t_n\theta'/v_n\} \cup t'_i/v'_i \in \theta' \mid v'_i \text{ no ocurre en } \theta$$

Eliminando todas las ligaduras de la forma t/t.

Ejemplo 9.6. *Sean $\theta = \{g(X, Y)/W\}$ y $\theta' = \{a/X, b/Y, c/Z\}$ dos substituciones, la composición $\theta \circ \theta' = \{g(a, b)/W, a/X, b/Y, c/Z\}$.*

Con los elementos anteriores podemos formalizar el concepto de unificador y unificador más general:

Definición 9.20 (Unificador más general). *Sean t_1 y t_2 términos. Se dice que la substitución θ es un **unificador** de esos términos si $t_1\theta = t_2\theta$. Una substitución θ se dice más general que una substitución σ, si y sólo si existe una substitución τ tal que $\sigma = \theta \circ \tau$. Un unificador θ se dice el **unificador más general** (umg) de dos términos, si y sólo si es más general que cualquier otro unificador entre esos términos.* *Unificador* *Unificador más general*

Ejemplo 9.7. *Consideren los términos $f(X)$ y $f(g(Y))$, el umg de ellos es $\theta = \{g(Y)/X\}$, pero existen otros muchos unificadores, por ejemplo $\{g(a)/X, a/Y\}$. Intuitivamente, el umg de dos términos es el más simple de todos sus unificadores. Observen que $f(X)\theta = f(g(Y))\theta = f(g(Y))$.*

Definir un algoritmo para computar el *umg* no está dentro de los objetivos de este curso. Para los interesados, mi curso de Programación para la Inteligencia Artificial [6], incluye un algoritmo para ello (Secciones 6.3 y 6.4 del curso, en el archivo ia2-notas-06). Para los más interesados, Knight [57] nos ofrece una revisión del problema de unificación en las ciencias de la computación y diversos algoritmos para su cómputo.

En lo que sigue, asumiremos que el *umg* está definido, como de hecho lo está en *Jason*. Ahora bien, los términos en este lenguaje AOP están **anotados** y esto debe tomarse en cuenta al computar la unificación. En general, dos términos $t_1[a_1, \ldots, a_n]$ y $t_2[a'_2, \ldots, a'_m]$ unifican, si y sólo si (i) existe un umg θ tal que $t_1\theta = t_2\theta$ y (ii) $\{a_1, \ldots, a_n\} \subseteq \{a'_1, \ldots, a'_m\}$. Esto es, si las fuentes de t_1 están incluidas en las de t_2.

Unificación con anotaciones

Ejemplo 9.8. *Consideremos algunos ejemplos del umg en Jason:*

- $umg(p(c)[s_1], p(c)[s_1, s_2]) = \top$ *debido a que no hay variables ligadas y las anotaciones del primer término están incluidas en las del segundo;*
- $umg(p(c)[a_1], p(c)[a_2])$ *falla debido a que la anotaciones del primer término no están incluidas en las anotaciones del segundo.*
- $umg(p(X)[a_2], p(c)[a_1, a_2, a_3]) = \{c/X\}$. *Ahora el unificador incluye una ligadura.*

Consecuencia lógica

La relación de consecuencia lógica estable que una fórmula bien formada se sigue de las creencias del agente:

Definición 9.21 (Consecuencia Lógica)**.** *La expresión $ag_{bs} \models fbf$ denota que la fórmula bien formada fbf es **consecuencia lógica** de las creencias del agente ag. Siguiendo la definición sintáctica de una fbf (Ecuación 9.6), tenemos que:*

Consecuencia lógica

- $ag_{bs} \models lit[s_1, \ldots, s_n]$, *si y sólo si, existe una literal* $lit'[s'_1, \ldots s'_m] \in ag_{bs}$, *tal que* $lit\theta = lit'$. θ *se conoce como unificador de respuesta.*
- $ag_{bs} \models \neg fbf$, *si y sólo si,* $ag_{bs} \not\models fbf$, *tal que* $\theta = \epsilon$.
- $ag_{bs} \models fbf_1 \wedge fbf_2\ \theta$, *si y sólo si,* $ag_{bs} \models fbf_1\ \theta_1$ *y* $ag_{bs} \models fbf_2\ \theta_2$. *El unificador de respuesta* $\theta = \theta_1 \circ \theta_2$.

Si consideramos a las reglas como parte de las creencias, debemos agregar el siguiente caso:

- $ag_{bs} \models lit[s_1, \ldots, s_n]$, si y sólo si, en las creencias del agente ag, existe una regla $lit'[s'_1, \ldots, s'_n] \text{ :- } fbf$ tal que: (i) $lit\theta' = lit'\theta'$ y (ii) $ag_{bs} \models fbf$ con substitución de respuesta θ''. $\theta = \theta' \circ \theta''$.

Ejemplo 9.9. $p(X)[a_1]$ *se sigue de* $ag_{bs} = \{p(t)[a_1, a_2]\}$; *pero* $p(X)[a_1, a_2]$ *no se sigue de* $ag_{bs} = \{p(t)[a_1]\}$.

La distribución de Jason incluye información complementaría sobre la unificación y las anotaciones [7]. Finalmente, definiremos una función auxiliar para obtener la substitución de respuesta cuando una fbf es consecuencia lógica de las creencias del agente:

Definición 9.22 (Substitución de Respuesta)**.** *La **substitución de respuesta** para una fbf, es un unificador θ tal que la aplicación de θ a la fbf, se sigue de las creencias del agente. Formalmente:*

Substitución de Respuesta

$$SubstResp(fbf) \quad = \quad \theta \mid ag_{bs} \models fbf\theta \tag{9.21}$$

Planes relevantes y aplicables

Con las definiciones anteriores podemos computar los conjuntos de planes relevantes y aplicables para un evento dado. Intuitivamente, los planes relevantes son aquellos que son útiles para contender con el evento; mientras que los aplicables son aquellos planes relevantes que pueden ejecutarse en ese momento, de acuerdo a las creencias del agente.

Definición 9.23 (Planes relevantes). *El conjunto de los **planes relevantes** para un evento $\langle te, i \rangle \in C_E$, está compuesto por los planes definidos para contender con tal evento. Formalmente, aquellos planes cuyo evento disparador unifica con te:* *Planes relevantes*

$$PRels(te) = \{p\theta \mid p \in ag_{ps} \wedge \theta = umg(te, TrEv(p))\} \tag{9.22}$$

Definición 9.24 (Planes aplicables). *El conjunto de los **planes aplicables** es un subconjunto de los planes relevantes que el agente cree poder ejecutar, es decir, aquellos cuyo contexto es consecuencia lógica de las creencias del agente. Formalmente:* *Planes aplicables*

$$PApls(PRels) = \{p\theta \mid p \in PRels \wedge ag_{bs} \models Ct(p)\theta\} \tag{9.23}$$

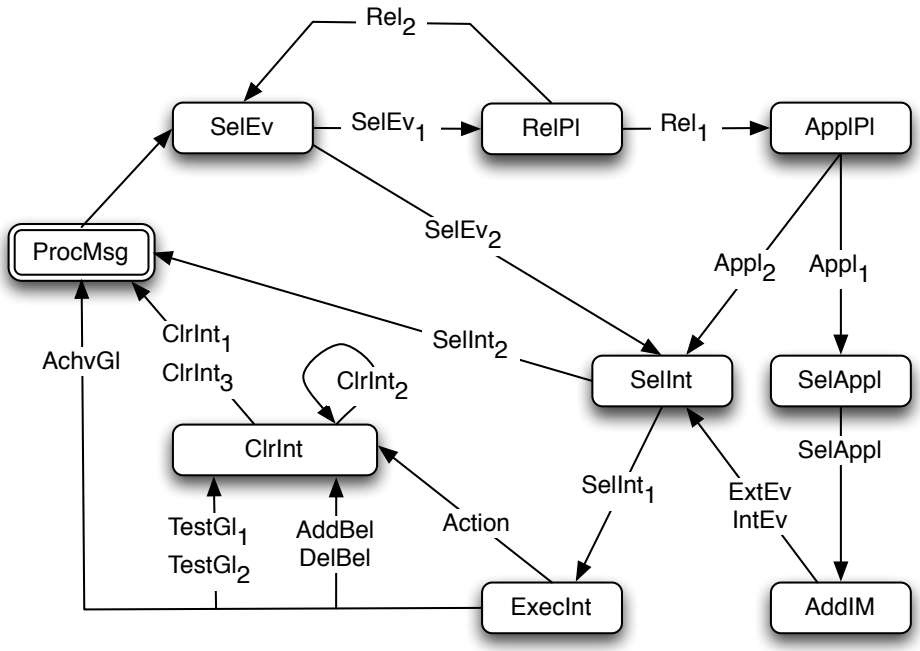


Figura 9.2: El ciclo de razonamiento de Jason. Adaptado de Guerra-Hernández, Castro-Manzano y El-Fallah-Seghrouchni [50].

9.2.3 Reglas semánticas

La relación de transición entre configuraciones de $AgentSpeak(L)$ se define mediante un conjunto de reglas semánticas que inducen el **ciclo de razonamiento** mostrado en la figura 9.2. Las etiquetas de los nodos denotan la configuración correspondiente. Los arcos están etiquetados con los identificadores de las reglas de transición que definiremos en esta sección. *Ciclo de razonamiento*

El estado inicial del ciclo de razonamiento es *ProcMsg*, una configuración donde se actualizan los eventos que el agente recibe vía su percepción y la comunicación. Conceptualmente, el ciclo de razonamiento puede dividirse en dos fases, con respecto a las intenciones:

- Formación. Comprende los caminos entre *ProcMsg* y *AddIM*.
- Ejecución. Comprende los caminos entre *SelInt* y *ProcMsg*.

5 En algunos textos se usa la notación invertida $v_i \backslash t_i$, o el mapeo $t_i \mapsto v_i$.
6 http://www.uv.mx/personal/aguerra/pia/
7 /doc/tech/annotations.pdf

De forma que un agente está continuamente tratando de formar nuevas intenciones, en respuesta a los eventos percibidos, y ejecutando las formadas con anterioridad.

Los eventos percibidos son agregados a la cola C_E. Las reglas que llevan a cabo la **selección de eventos** para su procesamiento, son como sigue. Si la cola de eventos no está vacía, un evento es seleccionado y agregado al registro correspondiente de la configuración, que pasa de seleccionar eventos a computar planes relevantes:

Selección de evento

$$(\mathbf{SelEv_1}) \quad \frac{C_E \neq \emptyset, S_E = \langle te, i \rangle}{\langle ag, C, M, T, SelEv \rangle \rightarrow \langle ag, C', M, T', RelPl \rangle} \quad \text{t.q. } T'_\epsilon = \langle te, i \rangle \tag{9.24}$$

Recuerden que $S_E = pop(C_E)$, elimina el evento seleccionado de C_E, de forma que la circunstancia del agente cambia, al igual que el registro de evento seleccionado. Todas las funciones de selección tienen un comportamiento análogo a éste. En caso de que la cola de eventos esté vacía, simplemente se procede a iniciar la segunda fase del ciclo de razonamiento, seleccionando una intención para su ejecución:

$$(\mathbf{SelEv_2}) \quad \frac{C_E = \emptyset}{\langle ag, C, M, T, SelEv \rangle \rightarrow \langle ag, C, M, T, SelInt \rangle} \tag{9.25}$$

Los **planes relevantes** se computan a partir del evento seleccionado. El conjunto de planes relevantes computados se guarda en el registro correspondiente, y se procede a computar quien de ellos es aplicable:

Planes relevantes

$$(\mathbf{Rel_1}) \quad \frac{T_\epsilon = \langle te, i \rangle, PRels(te) \neq \emptyset}{\langle ag, C, M, T, RelPl \rangle \rightarrow \langle ag, C, M, T', AppPl \rangle} \quad \text{t.q. } T'_R = PRels(te) \tag{9.26}$$

Si no existen planes relevantes, el ciclo de razonamiento regresa a seleccionar otro evento:

$$(\mathbf{Rel_2}) \quad \frac{T_\epsilon = \langle te, i \rangle, PRels(te) = \emptyset}{\langle ag, C, M, T, RelPl \rangle \rightarrow \langle ag, C, M, T, SelEv \rangle} \tag{9.27}$$

El caso de los **planes aplicables** es parecido. Si hay planes aplicables, se actualiza el registro correspondiente y se procede a seleccionar uno de ellos para formar una intención:

Planes aplicables

$$(\mathbf{Appl_1}) \quad \frac{PApls(T_R) \neq \emptyset}{\langle ag, C, M, T, ApplPl \rangle \rightarrow \langle ag, C, M, T', SelAppl \rangle} \quad \text{t.q. } T'_{Ap} = PApls(T_R) \tag{9.28}$$

Puesto que al computar si un plan p es aplicable, generamos un unificador de respuesta θ, el registro T_{Ap} almacena conjuntos de pares (p, θ). Si no hay planes aplicables, se procede a seleccionar una intención para su ejecución:

$$(\mathbf{Appl_2}) \quad \frac{PApls(T_R) = \emptyset}{\langle ag, C, M, T, ApplPl \rangle \rightarrow \langle ag, C, M, T, SelInt \rangle} \tag{9.29}$$

La siguiente regla **selecciona** un plan aplicable:

Selección de plan aplicable

$$(\textbf{SelAppl}) \quad \frac{S_{Ap} = (p, \theta)}{\langle ag, C, M, T, SelAppl\rangle \rightarrow \langle ag, C, M, T', AddIM\rangle} \quad \text{t.q. } T'_\rho = (p, \theta) \tag{9.30}$$

AgentSpeak(L) distingue dos tipos de eventos: Internos y externos. Los primeros tienen su origen en una intención; mientras que los segundos lo tienen en una percepción. Esto influye la manera como se **crea** una nueva intención. Los eventos externos crean una intención totalmente nueva con el plan aplicable seleccionado:

Creación de una Intención

$$(\textbf{ExtEv}) \quad \frac{T_\epsilon = \langle te, \top\rangle, T_\rho = (p, \theta)}{\langle ag, C, M, T, AddIM\rangle \rightarrow \langle ag, C', M, T, SelInt\rangle} \quad \text{t.q. } C'_I = C_I \cup \{[p\theta]\} \tag{9.31}$$

Los eventos internos, dada su naturaleza, apilan el plan relevante y aplicable seleccionado, en la Intención previa que generó el evento en cuestión:

$$(\textbf{IntEv}) \quad \frac{T_\epsilon = \langle te, i\rangle, T_\rho = (p, \theta)}{\langle ag, C, M, T, AddIM\rangle \rightarrow \langle ag, C', M, T, SelInt\rangle} \quad \text{t.q. } C'_I = C_I \cup \{i[p\theta]\} \tag{9.32}$$

Observen que en este caso, la intención i debe agregarse nuevamente en C_I con p en su tope. Esto tiene que ver con el concepto de **intención suspendida**, del que hablaremos al revisar las reglas para resolver las metas alcanzables. La forma exacta en que una Intención se suspende se verá más adelante, al definir las reglas de ejecución de una Intención.

Intención suspendida

Las reglas para **seleccionar** una intención a ser ejecutada son como sigue. Si hay Intenciones en la circunstancia del agente, se selecciona una para guardarla en el registro correspondiente:

Selección Intención

$$(\textbf{SelInt}_\textbf{1}) \quad \frac{C_I \neq \emptyset, S_I = i}{\langle ag, C, M, T, SelInt\rangle \rightarrow \langle ag, C, M, T', ExecInt\rangle} \quad \text{t.q. } T'_\iota = i \tag{9.33}$$

Si no hay Intenciones en la circunstancia del agente, se vuele al inicio del ciclo de razonamiento para actualizar los eventos percibidos por el agente:

$$(\textbf{SelInt}_\textbf{2}) \quad \frac{C_I = \emptyset}{\langle ag, C, M, T, SelInt\rangle \rightarrow \langle ag, C, M, T, ProcMsg\rangle} \tag{9.34}$$

El grupo de reglas que describiremos a continuación, expresa el efecto de la ejecución de los planes. El plan siendo ejecutado es siempre aquel que se encuentra en el tope de la intención que ha sido previamente seleccionada. Todas las reglas en este grupo terminan descartando i, por lo que otra intención puede ser seleccionada posteriormente. Las reglas se ejecutan dependiendo del componente del cuerpo del plan que se ha seleccionado.

La siguiente regla aplica cuando el primer elemento del cuerpo del plan que está en el tope de la Intención seleccionada, es una **acción**. En ese caso la acción es almacenada en el registro correspondiente:

Ejecución de una acción

$$(\textbf{Action}) \quad \frac{T_\iota = i[head \leftarrow a; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T', ClrInt\rangle} \tag{9.35}$$
$$\text{t.q. } C'_A = C_A \cup \{a\}, T'_\iota = i[head \leftarrow h], \quad C'_I = (C_I \backslash \{T_\iota\}) \cup \{T'_\iota\}$$

La siguiente regla aplica cuando el primer elemento del cuerpo del plan que está en el tope de la Intención seleccionada, es una **meta alcanzable**:

Ejecución de una meta alcanzable

$$(\textbf{AchvGl}) \quad \frac{T_\iota = i[head \leftarrow !at; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle} \tag{9.36}$$
$$\text{t.q. } C'_E = C_E \cup \{\langle +!at, T_\iota\rangle\}, \quad C'_I = C_I \setminus \{T_\iota\}$$

Observe como la Intención que generó el evento interno es removida del conjunto de Intenciones C_I. Esto implementa la **suspención** de una Intención. Sólo cuando el curso de acción que se haya definido para procesar el evento generado haya concluído, se puede continuar con la ejecución de la intención que había sido suspendida [8].

Intención suspendida

Las **metas verificables**, se procesan mediante las siguientes dos reglas:

Ejecución meta verificable

$$(\textbf{TestGl}_\mathbf{1}) \quad \frac{T_\iota = i[head \leftarrow ?at; h], Test(ag_{bs}, at) = at\theta}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T, ClrInt\rangle} \tag{9.37}$$
$$\text{t.q. } T'_\iota = i[(head \leftarrow h)\theta], \quad C'_I = (C_I \backslash \{T_\iota\}) \cup \{T'_\iota\}$$

$$(\textbf{TestGl}_\mathbf{2}) \quad \frac{T_\iota = i[head \leftarrow ?at; h], Test(ag_{bs}, at) = \bot}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M, T, ClrInt\rangle} \tag{9.38}$$
$$\text{t.q. } C'_E = C_E \cup \{\langle -?at, T_\iota\rangle\}, \quad C'_I = C_I \backslash \{T_\iota\}$$

Observen que si la consecuencia lógica, computada por $Test/2$, **falla**; la intención en si no falla, sino que genera un evento ($TestGl_2$) para que sea procesado por un plan posteriormente. Esto introduce otra diferencia con la Programación Lógica, pensada para permitir consultas más elaboradas que las que *Test* puede computar.

Fallo meta verificable

Al igual que en dMARS [33] y a diferencia de $AgentSpeak(L)$, los agentes en Jason pueden **agregar o eliminar creencias** durante la ejecución de sus planes. Las siguientes reglas se encargan de ello:

Actualización creencias

$$(\textbf{AddBel}) \quad \frac{T_\iota = i[head \leftarrow +b; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag', C', M, T', ClrInt\rangle} \tag{9.39}$$
$$\text{t.q. } ag'_{bs} = ag_{bs} + b_{[self]}, \quad C'_E = C_E \cup \{\langle +b_{[self]}, \top\rangle\}, \quad T'_\iota = i[head \leftarrow h], \quad C'_I = (C_I \backslash \{T_\iota\}) \cup \{T'_\iota\}$$

8 En Jason, el operador !! evita la suspensión de las intenciones, creando una intención nueva, como si se tratase de una respuesta a un evento externo.

$$(\textbf{DelBel}) \quad \frac{T_\iota = i[head \leftarrow -at; h]}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag', C', M, T', ClrInt\rangle}$$

$$\begin{aligned} \text{t.q. } ag'_{bs} &= ag_{bs} - at_{[self]}, \\ C'_E &= C_E \cup \{\langle -at_{[self]}, \top\rangle\}, \\ T'_\iota &= i[head \leftarrow h], \\ C'_I &= (C_I \backslash \{T_\iota\}) \cup \{T'_\iota\} \end{aligned} \tag{9.40}$$

Observen que aunque los eventos generados por estas operaciones tienen como origen una intención, estos son tratados como si fuesen **eventos externos**. Esto es, la intención que genero estos eventos no se suspende. *Excepción*

Para concluir con la semántica operacional de Jason se definen tres reglas más, las llamadas *clearing house rules*. $ClearInt_1$ simplemente remueve una intención del conjunto de intenciones de un agente cuando no hay más que hacer al respecto, es decir: Ya no quedan más acciones, metas o actualizaciones que ejecutar en el cuerpo del plan único en la intención. $ClearInt_3$ se encarga del caso trivial donde la limpieza no debe realizarse aún, así que el ciclo de razonamiento pasa a la configuración $ProcMsg$. La regla $ClearInt_2$ contiende con el caso donde el plan en el tope de la intención tiene un cuerpo vacío, pero la intención tiene más planes. En este caso el ciclo de razonamiento regresa a $ClrInt$ porque pueden ser necesarias más labores de limpieza. *Limpieza*

$$(\textbf{ClrInt}_\textbf{1}) \quad \frac{T_\iota = [head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C', M, T, ProcMsg\rangle} \tag{9.41}$$

$$\text{t.q. } C'_I = C_I \setminus \{T_\iota\}$$

$$(\textbf{ClrInt}_\textbf{2}) \quad \frac{T_\iota = i[head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C', M, T, ClrInt\rangle} \tag{9.42}$$

$$\begin{aligned} \text{t.q. } C'_I = (C_I \setminus \{T_\iota\}) \ &\cup \{k[(head' \leftarrow h)\theta]\} \\ \text{si } i &= k[head' \leftarrow g; h] \\ \text{y } g\theta &= TrEv(head) \end{aligned}$$

$$(\textbf{ClrInt}_\textbf{3}) \quad \frac{T_\iota \neq [head \leftarrow \top] \wedge T_\iota \neq i[head \leftarrow \top]}{\langle ag, C, M, T, ClrInt\rangle \rightarrow \langle ag, C, M, T, ProcMsg\rangle} \tag{9.43}$$

9.3 ACTOS DE HABLA

Vieira et al. [108] han propuesto una semántica extendida de $AgentSpeak(L)$ para implementar un lenguaje de comunicación basado en Actos de habla, para Jason. Debido a una literatura más abundante, pero sobre todo por razones históricas, se optó por adoptar KQML en esta tarea. La semántica de KQML fue propuesta por Labrou y Finin [61], con base en los trabajos de Perrault y Allen [83] y Cohen y Levesque [26]. La idea central consiste en especificar operadores tipo STRIPS que definen las las **condiciones** preparatorias, posteriores y de satisfacción de cada performativa. *KQML* *Condiciones KQML*

Ejemplo 9.10. *Para tell tendríamos:*

- *Condiciones preparatorias:*

  - $Pre(S) : bel(S, X) \wedge know(S, want(R, know(R, bel(S, X))))$
  - $Pre(R) : intend(R, know(R, bel(S, X)))$
- *Condiciones posteriores:*
  - $Pos(S) : know(S, know(R, bel(S, X)))$
  - $Pos(R) : know(R, bel(S, X))$
- *Satisfacción:*
  - $know(R, bel(S, X))$

donde las condiciones preparatorias se leen así: el emisor del mensaje S cree X y sabe que el receptor del mensaje R quiere saber si ese es el caso; mientras que R intenta saber si ese es el caso, si S cree X. El resto de las condiciones puede leerse de manera similar.

No es de extrañar que la implementación y verificación de tal implementación sean difíciles de lograr. La abstracción Intencional oculta el nivel de diseño en tal especificación. Afortunadamente, en el caso de $AgentSpeak(L)$ extendido, podemos recurrir a las anotaciones para definir una semántica de los actos de habla. En lo que sigue, asumiremos la sintaxis y la semántica operacional definidas previamente y las extenderemos como sigue.

9.3.1 Sintaxis

Asumiremos la existencia de una acción especial predefinida para enviar mensajes: `.send(Id,Ilf,Cnt)`, donde Id es la identidad del destinatario, ilf es la fuerza ilocutoria del mensaje y Cnt su contenido.

9.3.2 Semántica

Mensajes

En realidad, los **mensajes** tienen como soporte una tupla $\langle mid, dest, fil, cnt \rangle$, donde:

- mid es un **identificador** único del mensaje, generado automáticamente.
- $dest$ es el nombre del agente **destinatario**.
- $fil \in \{Tell, Untell, Achieve, Unachieve, TellHow, UntellHow, AskOne, AskAll\}$ es la **fuerza ilocutoria** del mensaje.
- cnt es el **contenido** del mismo. Dependiendo de la fuerza performativa del mensaje, su contenido puede ser: una fórmula atómica (at), o un conjunto de fórmulas atómicas (ATs), o una creencia (b), es decir una literal de base, o un conjunto de creencias (Bs), o un conjunto de planes (PLs).

Buzones

Toda configuración de la semántica operacional provee un mecanismo **asíncrono** para el envío y recepción de mensajes con su fuerza performativa. Recuerden que la **circunstancia** del un agente incluye la estructura M, un conjunto de **buzones** que incluye uno de de **entrada** M_{In}, uno de **salida** M_{Out} y uno de **intenciones suspendidas** M_{SI}. En realidad, el buzón de intenciones suspendidas guarda pares de la forma $\langle mid, i \rangle$, donde mid es el identificador del mensaje que causó que la intención i fuera suspendida en espera de la respuesta solicitada. Los mensajes se almacenan en un buzón y uno de ellos es procesado cada vez que el agente inicia un ciclo de razonamiento.

Solicitud de información

Cuando un agente envía una **solicitud de información**, se ha optado por suspender la intención asociada hasta recibir la respuesta solicitada. Esto garantiza que el agente tenga la información solicitada antes de continuar ejecutando el cuerpo del plan que incluye la solicitud de información. Se asume que la información recibida pasa a formar parte de las **creencias** del agente, por lo que una meta verificable es necesaria para poderla usar posteriormente.

A continuación definimos dos reglas semánticas para ejecutar la acción interna `.send`. Estas reglas pueden verse como especializaciones de la regla *Action* que ejecuta acciones en los planes de los agentes y tienen prioridad sobre ésta. La primer regla contiende con solicitudes de información y por tanto, suspende intenciones:

Envío de mensajes

$$(\textbf{ExecActSndAsk}) \quad \frac{\begin{array}{c} T_\iota = i[head \leftarrow .send(id, ilf, cnt); h], \\ ilf \in \{AskOne, AskAll, AskHow\} \end{array}}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M', T, ProcMsg\rangle} \tag{9.44}$$

$$\begin{array}{rl} \text{Donde:} & M'_{Out} = M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\ & M'_{SI} = M_{SI} \cup \{(mid, i[head \leftarrow h])\} \\ & C'_I = C_I \backslash \{T_\iota\} \end{array}$$

Cuando la fuerza performativa es de otro tipo, simplemente se forma el mensaje en el buzón de salida; pero el siguiente paso en el razonamiento en ese caso es *ClrInt*, en lugar de pasar directamente a *ProcMsg*:

$$(\textbf{ExecActSnd}) \quad \frac{\begin{array}{c} T_\iota = i[head \leftarrow .send(id, ilf, cnt); h], \\ ilf \notin \{AskOne, AskAll, AskHow\} \end{array}}{\langle ag, C, M, T, ExecInt\rangle \rightarrow \langle ag, C', M', T, ClrInt\rangle} \tag{9.45}$$

$$\begin{array}{rl} \text{Donde:} & M'_{Out} = M_{Out} \cup \{\langle mid, id, ilf, cnt\rangle\} \\ & C'_I = (C_I \backslash \{T_\iota\}) \cup \{i[head \leftarrow h]\} \end{array}$$

Siempre que un mensaje nuevo es enviado, asumimos que el sistema crea un nuevo identificador *mid* único. Más adelante veremos que al contestar a un mensaje, el mismo *mid* se mantiene en la respuesta obtenida, de forma que un agente pueda saber que cierto mensaje le llegó en respuesta a un mensaje suyo.

Necesitamos una nueva función de **selección** $\mathcal{S}_M$ que opere sobre los buzones. Analogamente a las otras funciones de selección en Jason, la función regresa el primer mensaje en el buzón que recibe como parámetro. Necesitamos también una función booleana $SocAcc(id, ilf, at)$ donde ilf es la fuerza ilocutoria del mensaje del agente id con contenido proposicional at, que determina si el mensaje es **socialmente aceptable** en un contexto dado. Estas funciones pueden ser reprogramadas al igual que las otras funciones de selección del lenguaje.

Función de selección

Socialmente aceptable

A continuación definimos la regla asociada a la recepción de un mensaje cuya fuerza performativa es *Tell*, como en un informe no solicitado:

Tell

$$(\textbf{Tell}) \quad \frac{\begin{array}{c} \mathcal{S}_M(M_{In}) = \langle mid, id, Tell, Bs\rangle, \\ \forall i\ (mid, i) \notin M_{SI} \\ SocAcc(id, Tell, Bs) \end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle} \tag{9.46}$$

$$\begin{array}{rl} \text{Donde:} & M'_{In} = M_{In} \backslash \{\langle mid, id, Tell, Bs\rangle\} \\ & \forall b\ b \in Bs : ag'_{bs} = ag_{bs} + b[id] \\ & C_{E'} = C_E \cup \{\langle +b[id], \top\rangle\} \end{array}$$

La otra posibilidad es recibir el informe con fuerza performativa *Tell*, en respuesta a una solicitud previa (existe una intención suspendida asociada al *mid*):

$$(\textbf{TellRepl})\ \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, Tell, Bs\rangle,\\ \exists i\ (mid, i) \in M_{SI},\\ SocAcc(id, Tell, Bs)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle}$$

$$\begin{array}{rl}\text{Donde: } & M'_{In} = M_{In}\backslash\{\langle mid, id, Tell, Bs\rangle\}\\ & M'_{SI} = M_{SI}\backslash\{(mid, i)\}\\ & C'_I = C_I \cup \{i\}\\ & \forall b\ b \in Bs : ag'_{bs} = ag_{bs} + b[id]\\ & C_{E'} = C_E \cup \{\langle +b[id], \top\rangle\}\end{array} \tag{9.47}$$

La regla asociada a recibir un mensaje con fuerza performativa *Untell* que no responde a una solicitud previa, es: *Untell*

$$(\textbf{Untell})\ \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, Untell, ATs\rangle,\\ \forall i\ (mid, i) \notin M_{SI},\\ SocAcc(id, Tell, ATs)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle}$$

$$\begin{array}{rl}\text{Donde: } & M'_{In} = M_{In}\backslash\{\langle mid, id, Tell, ATs\rangle\}\\ & \forall b.\ b \in \{at\theta\ |\theta \in Test(ag_{bs}, at) \wedge at \in ATs\} :\\ & \quad ag'_{bs} = ag_{bs} - b[id]\\ & C_{E'} = C_E \cup \{\langle -b[id], \top\rangle\}\end{array} \tag{9.48}$$

Observen que el contenido de estos mensajes es una fórmula atómica por lo que puede incluir variables para las cuales será necesario computar sus posibles valores mediante unificación (*Test*). En el caso que el emisor del mensaje sea la única fuente de la creencias, esta es removida de las creencias del receptor. En otro caso, solo la fuente en la etiqueta es eliminada. En caso de que el mensaje se de en respuesta a una solicitud previa, la regla que aplica es la siguiente:

$$(\textbf{UntellRepl})\ \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, Untell, ATs\rangle\\ \exists i\ (mid, i) \in M_{SI}\\ SocAcc(id, Untell, ATs)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle}$$

$$\begin{array}{rl}\text{Donde: } & M'_{In} = M_{In}\backslash\{\langle mid, id, Tell, ATs\rangle\}\\ & M'_{SI} = M_{SI}\backslash\{(mid, i)\}\\ & C'_I = C_I \cup \{i\}\\ & \forall b.\ b \in \{at\theta\ |\theta \in Test(ag_{bs}, at) \wedge at \in ATs\} :\\ & \quad ag'_{bs} = ag_{bs} - b[id]\\ & C_{E'} = C_E \cup \{\langle -b[id], \top\rangle\}\end{array} \tag{9.49}$$

En el contexto social adecuado, es posible que un agente reciba un mensaje pidiéndole satisfacer una meta, por ejemplo, cuando el emisor tiene poder sobre el receptor. La regla que regula la recepción de este tipo de mensajes es:

$$(\textbf{Achieve}) \quad \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, Achieve, at\rangle \\ SocAcc(id, Achieve, at)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag, C', M', T, SelEv\rangle} \tag{9.50}$$

$$\begin{array}{rl}\text{Donde:} & M'_{In} = M_{In} \backslash \{\langle mid, id, Achieve, at\rangle\} \\ & C'_E = C_E \cup \{+!at, \top\}\end{array}$$

Observen que la recepción de un mensaje socialmente aceptable con fuerza ilocutoria *Achieve* crea una nueva pila en las intenciones, al igual que lo hacen los eventos externos. Esto puede usarse como un mecanismo de focus de atención. Para los mensajes de fuerza ilocutoria *Unachieve* se usa la acción interna estándar `.drop_desire`.

Una extensión interesante de Jason a los actos de habla tradicionales, es la consideración del concepto de *know-how* [102], lo cual implica que la librería de planes de un agente no es estática –puede agregar planes que otro agente le comunica y eliminar planes a petición de otro agente. La recepción de mensajes con fuerza ilocutoria *TellHow* se regula por la siguiente regla:

$$(\textbf{TellHow}) \quad \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, TellHow, PLs\rangle \\ \exists i\ (mid, i) \in M_{SI} \\ SocAcc(id, TellHow, PLs)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle} \tag{9.51}$$

$$\begin{array}{rl}\text{Donde:} & M'_{In} = M_{In} \backslash \{\langle mid, id, TellHow, at\rangle\} \\ & M'_{SI} = M_{SI} \backslash \{(mid, i)\} \\ & C'_I = C_I \cup \{i\} \\ & ag'_{ps} = ag_{ps} \cup PLs\end{array}$$

La regla para los mensajes con fuerza ilocutoria *UntellHow* es muy similar:

$$(\textbf{UntellHow}) \quad \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, UntellHow, PLs\rangle \\ SocAcc(id, UntellHow, PLs)\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag', C', M', T, SelEv\rangle} \tag{9.52}$$

$$\begin{array}{rl}\text{Donde:} & M'_{In} = M_{In} \backslash \{\langle mid, id, UntellHow, at\rangle\} \\ & M'_{SI} = M_{SI} \backslash \{(mid, i)\} \\ & ag'_{ps} = ag_{ps} - PLs\end{array}$$

Para terminar de ilustrar la semántica de los actos de habla, consideremos la requisición de información mediante un mensaje con fuera ilocutoria *AskOne*:

$$(\textbf{AskOne}) \quad \frac{\begin{array}{c}\mathcal{S}_M(M_{In}) = \langle mid, id, AskOne, \{at\}\rangle \\ SocAcc(id, AskOne, \{at\})\end{array}}{\langle ag, C, M, T, ProcMsg\rangle \rightarrow \langle ag, C, M', T, SelEv\rangle}$$

$$\begin{array}{rl}\text{Donde:} & M'_{In} = M_{In} \backslash \{\langle mid, id, AskOne, at\rangle\} \\ & M'_{Out} = \begin{cases} M_{Out} \cup \{\langle mid, id, Tell, AT\rangle\}, & \\ AT = \{at\theta\}, \theta \in Test(ag_{bs}, at) & \text{Si } Test(ag_{bs}, at) \neq \{\} \\ M_{Out} \cup \{\langle mid, id, Untell, \{at\}\rangle\} & \text{En otro caso} \end{cases}\end{array}$$

(9.53)

La respuesta a un mensaje con fuera ilocutoria *AskAll*, a diferencia de *AskOne* regresa todas las creencias que unifica con el átomo solicitado, mientras que éste último solo regresa el átomo en cuestión o un mensaje de fuerza ilocutoria *Untell* con el átomo como contenido.

9.4 $CTL_{AgentSpeak(L)}$

La semántica operacional de $AgentSpeak(L)$ provee una especificación clara, precisa y computable del ciclo de razonamiento de los agentes BDI, pero ha perdido en el camino precisamente los operadores Intencionales de creencia, deseo e intención. Es por ello que propusimos $CTL_{AgentSpeak(L)}$ [49, 50], una lógica con operadores Intencionales y temporales, cuya semántica se basa en la semántica operacional de $AgentSpeak(L)$. *$CTL_{AgentSpeak(L)}$*

9.4.1 Sintaxis

Las fbfs de $CTL_{AgentSpeak(L)}$ incluyen expresiones intencionales y temporales:

- Si $at \in AgentSpeak(L)$, entonces at, $\mathsf{BEL}(at)$, $\mathsf{DES}(at)$, $\mathsf{INTEND}(at)$ son fbf **intencionales** de $CTL_{AgentSpeak(L)}$. *Fbfs Intencionales*
- Las fbf de **estado** son: *Fbfs de estado*
  - s1 Toda fbf intencional es una fbf de estado.
  - s2 Si ϕ y ψ son fbf de estado, $\phi \wedge \psi$ y $\neg\phi$ lo son.
  - s3 Si ϕ es una fbf de camino, entonces $\mathsf{E}\phi$ es una fbf de estado.
- Las fbf de **camino** son: *Fbfs de camino*
  - p1 Toda fbf de estado es una fbf de camino.
  - p2 Si ϕ y ψ son fbf de camino, $\neg\phi$, $\phi \wedge \psi$, $\bigcirc\phi$, $\Diamond\phi$ y $\phi \,\mathsf{U}\, \psi$ son fbf de camino.

De forma que podemos expresar enunciado como que intentaré en todo futuro posible eventualmente ir a Paris: $\mathsf{INTEND}(\mathsf{A}\Diamond ir(paris))$, tal y como lo hacíamos con las lógicas BDI_{CTL}.

9.4.2 Semántica de los operadores BDI

Para definir la semántica de los operadores intencionales necesitaremos una función auxiliar, que dada una intención nos regrese el conjunto de fbf atómicas que son sujeto de una **meta alcanzable**: *Metas alcanzables*

$$
\begin{aligned}
agls(\top) &= \{\}, \\
agls(i[p]) &= \begin{cases} \{at\} \cup agls(i) & \text{if } p = +!at : ct \leftarrow h, \\ agls(i) & \text{otherwise} \end{cases}
\end{aligned}
$$

Observen que se trata de una función recursiva. Las metas alcanzables de una intención vacía son el conjunto vacío. Si la intención no es vacía y el evento disparador del plan en su tope es una meta alcanzable, entonces se guarda la fbf atómica correspondiente para agregarla a las fbf que se colectarán en el resto de la intención.

Definición 9.25 (Semántica Intencional)**.** *La semántica de los operadores BDI es como sigue:*

$$\begin{aligned}
\mathsf{BEL}_{\langle ag,C\rangle}(\phi) &\equiv ag_{bs} \models \phi \\
\mathsf{INTEND}_{\langle ag,C\rangle}(\phi) &\equiv \phi \in \bigcup_{i \in C_I} agls(i) \vee \bigcup_{\langle te,i\rangle \in C_E} agls(i) \\
\mathsf{DES}_{\langle ag,C\rangle}(\phi) &\equiv \langle +!\phi, i\rangle \in C_E \vee \mathsf{INTEND}(\phi)
\end{aligned}$$

Si el agente ag y su circunstancia C son explícitos, escribimos simplmenete $\mathsf{BEL}(\phi)$, $\mathsf{DES}(\phi)$ e $\mathsf{INTEND}(\phi)$. De forma que, se dice que un agente ag **cree** la fbf atómica ϕ, si ésta es consecuencia lógica de sus creencias. Se dice que un agente **intenta** ϕ, si ésta fbf atómica es sujeto de una meta alcanzable en las intenciones, tanto activas como suspendidas, del agente. Se dice que un agente **desea** la fbf ϕ, si ésta fbf atómica es intendada o existe un evento donde la fbf es sujeto de una meta alcanzable.

Creer
Intentar
Desear

9.4.3 Semántica de los operadores temporales

Como suele ser el caso, la semántica de los operadores temporales requiere de un modelo basada en una estrcutura de Kripke.

Definición 9.26 (Estructura de Kripke $AgentSpeak(L)$)**.** *Sea $K = \langle S, R, V\rangle$ es una **estructura de Kripke** definida sobre el sistema de transición (Γ), donde:*

- *S es un conjunto de configuraciones $\langle ag, C, M, T, s\rangle$.*
- *$R \subseteq S^2$ es una relación serial t.q. para todo $(c_i, c_j) \in R$, $(c_i, c_j) \in \Gamma$.*
- *V es una función de evaluación sobre los operadores intencionales y temporales, por ej., $V_{\mathsf{BEL}}(c, \phi) \equiv \mathsf{BEL}_{\langle ag,C\rangle}(\phi)$ en la configuración $c = \langle ag, C, M, T, s\rangle$, etc.*

Le expresión $x = c_0, \ldots, c_n$ denota un **camino**, una secuencia de configuraciones t.q. para todo $c_i \in S$, $(c_i, c_{i+1}) \in R$. La expresión x^i denota el sufijo del camino x a partir de la configuración c_i.

Definición 9.27 (Semántica temporal)**.** *La semántica de las fbf de estado es como sigue:*

$$\begin{aligned}
K, c_i \models \mathsf{BEL}(\phi) &\Leftrightarrow \phi \in V_{\mathsf{BEL}}(c_i, \phi) \\
K, c_i \models \mathsf{INTEND}(\phi) &\Leftrightarrow \phi \in V_{\mathsf{INTEND}}(c_i, \phi) \\
K, c_i \models \mathsf{DES}(\phi) &\Leftrightarrow \phi \in V_{\mathsf{DES}}(c_i, \phi) \\
K, c_i \models \mathsf{E}\phi &\Leftrightarrow \exists x^i \exists c_{j \geq i} \in x^i \text{ t.q. } K, c_j \models \phi \\
K, c_i \models \mathsf{A}\phi &\Leftrightarrow \forall x^i \exists c_{j \geq i} \in x^i \text{ t.q. } K, c_j \models \phi
\end{aligned}$$

La semántica de las fbf de camino es como sigue:

$$\begin{aligned}
K, c_i \models \phi &\Leftrightarrow K, x^i \models \phi, \text{cuando } \phi \text{ es una fbf de estado} \\
K, c_i \models \bigcirc\phi &\Leftrightarrow K, x^{i+1} \models \phi \\
K, c_i \models \Diamond\phi &\Leftrightarrow \exists c_{j \geq i} \in x^i \text{ t.q. } K, c_j \models \phi \\
K, c_i \models \phi \,\mathsf{U}\, \psi &\Leftrightarrow (\exists c_{k \geq i} \text{ t.q. } K, x^k \models \psi \wedge \\
&\qquad \forall c_{i \leq j < k} \quad K, x^j \models \phi) \vee (\exists c_{j \geq i} \quad K, x^j \models \phi).
\end{aligned}$$

Observen que la semántica del operador "hasta" (U) corresponde a su versión débil (ψ puede no ocurrir nunca). Las definiciones de corrida, satisfacción y válidez son las comunes.

Definición 9.28 (Corrida)**.** *Dada una configuración inicial c_0, la **corrida** K^0_Γ denota una secuencia de configuraciones $c_0, c_1, \ldots$ t.q. $\forall i \geq 1, c_i = \Gamma(c_{i-1})$.*

Corrida

Definición 9.29 (Satisfacción)**.** *Dada una corrida K^0_Γ, se dice que la propiedad $\phi \in CTL_{AgentSpeak(L)}$ se **satisface** si $\forall i \geq 0, K^0_\Gamma, c_i \models \phi$.*

Satisfacción

Definición 9.30 (Validez)**.** *Una propiedad $\phi \in CTL_{AgentSpeak(L)}$ es **válida**, si $K^0_\Gamma, c_0 \models \phi$ para toda configuración inicial c_0.*

Validez

9.4.4 Propiedades BDI de $AgentSpeak(L)$

Bordini y Moreira [13] utilizaron el segmento BDI de $CTL_{AgentSpeak(L)}$ para investigar sus propiedades con respecto a la tesis de asimetría, encontrando que $AgentSpeak(L)$ no es equivalente a ninguna de las lógicas estudiadas por Rao y Georgeff. El cuadro 9.2 muestra que axiomas satisface $AgentSpeak(L)$, comparándo el resultado con otras lógicas BDI.

	AT1	AT2	AT3	AT4	AT5	AT6	AT7	AT8	AT9
Realismo fuerte	✓		✓	✓		✓	✓		✓
Realismo	✓	✓		✓	✓		✓	✓	
Realismo débil	✓	✓	✓	✓	✓	✓	✓	✓	✓
$AgentSpeak(L)$		✓	✓	✓		✓		✓	✓

Cuadro 9.2: La tesis de asimetría en $AgentSpeak(L)$ y otras Lógicas BDI.

Nuestro interés [49, 50] era saber que tipo de estrategia de compromiso siguen los agentes $AgentSpeak(L)$ que implementa Jason. Por tanto, primero se abordó el siguiente problema: ¿Es válido el axioma de posposición finita?

$$\mathsf{INTEND}(\phi) \rightarrow \mathsf{A}\Diamond(\neg\mathsf{INTEND}(\phi))$$

Prueba: Asumamos $K, c_0 \models \mathsf{INTEND}(\phi)$, entonces dada la definición de INTEND, existe un plan $p \in C_I \cup C_E$ con la cabeza $+!\phi$ en c_0. p es finito por definición. Siguiendo $ClrInt_X$ el plan es abandonado por éxito o fracaso. □

Los agentes $AgentSpeak(L)$ **no satisfacen** la propiedad de compromiso ciego:

$$\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}\Diamond\mathsf{BEL}(\phi)$$

Prueba: Asumamos una configuración inicial c_0, donde $ag = \langle\{b(t_1)\}, \{+b(t_1) : \top \leftarrow p(t_2).\ \ +!p(t_2) : \top \leftarrow +b(t_3).\}$. Se obtendrá una configuración c' donde $C_I = \{[+!p(t_2) : \top \leftarrow\ \ +b(t_3).]\}$ y $C_E = \{\}$ t.q. $K, c' \models \mathsf{INTEND}(p(t_2))$ y $K, c' \not\models \mathsf{BEL}(p(t_2))$. En la siguiente configuración c'', $K, c'' \models \neg\mathsf{INTEND}(p(t_2))$ y $K, c'' \not\models \mathsf{BEL}(p(t_2))$. □

Un agente $AgentSpeak(L)$ **satisface** una forma limitada del compromiso racional: Gu

$$\mathsf{INTEND}(\mathsf{A}\Diamond\phi) \rightarrow \mathsf{A}(\mathsf{INTEND}(\mathsf{A}\Diamond\phi)\ \mathsf{U}\ \neg\mathsf{BEL}(\mathsf{E}\Diamond\phi))$$

Prueba: Siguiendo la demostración de posposición finita, en los casos de fallo el agente satisface eventualmente $\bigcirc\neg\mathsf{INTEND}(\phi)$ debido a Rel_2 –Para un evento $\langle te, i[+!\phi : c \leftarrow h.]\rangle$ no hubo planes relevantes y la intención asociada es abandonada. □

Se trata de una forma limitada de compromiso racional porque no hay representación explícita de la razón de abandono, aunque ésta se podría aprender.

9.5 LECTURAS Y EJERCICIOS SUGERIDOS

Rao [89] introduce $AgentSpeak(L)$ y su semántica operacional, la primera parte de este capítulo se basa en este artículo. Bordini, Hübner y Wooldridge [12] es el libro de texto oficial sobre *Jason* y la programación de SMA con este lenguaje. La semántica operacional presentada en la segunda parte de este capítulo se basa en este libro. Plotkin [86] ofrece una introducción a las semánticas operacionales estructurales y su pertinencia. d'Inverno et al. [33] nos ofrecen una especificación de dMARS, el predecesor de $AgentSpeak(L)$ donde pueden revisarse las principales ideas detras de la programación BDI. Aunque esta especificación no está basada en una semántica operacional, es de utilidad para comprender las extensiones provistas por *Jason* y otras posibles extensiones a implementar.

Ejercicios

Ejercicio 9.1. *En el ejemplo 9.2 identifique con precisión los elementos del programa, de ser posible, argumentando con las reglas sintácticas correspondientes. Por ejemplo:* `factorial(1,1)` *es una creencia, de acuerdo a la ecuación 9.8, etc.*

Ejercicio 9.2. *En el ejemplo 9.4 explique las razones por las cuales las substituciones son válidas o no.*

Ejercicio 9.3. *Desarrolle paso a paso la composición de substituciones del ejemplo 9.6.*

10 JASON

Jason [11, 9, 12, 10] es un intérprete programado en Java, para una versión extendida de *AgentSpeak*(*L*). Como tal, implementa la semántica operacional del lenguaje y provee una plataforma para el desarrollo de SMA, con muchas opciones configurables por el usuario. Se trata de un código abierto, distribuido bajo una licencia GNU LGPL. Sus extensiones incluyen comunicación basada en actos de habla [108], herramientas para simulación social [14] y un sistema de módulos [80], entre otras.

En este capítulo introduciremos Jason y ejemplificaremos su uso. La organización del capítulo es como sigue: Primero abordaremos la instalación de Jason y sus ambientes de desarrollo: El estándar basado en jEdit; y el basado en Eclipse, gracias a un plug-in desarrollado por Zatelli. Posteriormente, abordamos Jason *à la* Prolog, es decir, haciendo uso de las creencias y las metas verificables que configuran un sistema muy similar a los sistemas de programación lógica. Se revisará el uso de hechos, reglas y metas verificables; la aritmética; las representaciones en primer orden; las anotaciones; y la negación fuerte y débil. A continuación, revisaremos el concepto de los módulos en Jason, puesto que ahora todo programa de agente está modularizado; y terminaremos el capítulo abordando la definición de acciones internas.

10.1 INSTALACIÓN

Jason puede descargarse gratuitamente desde su página en internet [1], donde además encontrarán una descripción del lenguaje, documentación, ejemplos, proyectos asociados etc. Tambien encontrarán una liga a la página del libro **Programando Sistemas Multi-Agentes en AgentSpeak(L) usando Jason** [2], el mejor soporte para este lenguaje de programación orientado a agentes.

Libro

10.1.1 Distribución en sourceforge

La **distribución** de Jason se muestra en la Figura 10.1. Lo importante es que el ejecutable, en este caso `scripts/jason` esté en algún sitio accesible.

Distribución

Ejemplo 10.1. *El folder completo de la distribución,* `jason-2`*, puede colocarse en el folder de* `Aplicaciones` *en MacOS.*

Observen que el código fuente está disponible en `src` y que los ejemplos y demos del sitio web del lenguaje están includos en los folders `examples` y `demos`, respectivamente. La documentación en `doc` incluye algunos artículos relevantes y la descripción del API de Jason. La carpeta `scripts` tiene todos los scripts para ejecutar `Jason` desde consola. Como mencioné, es importante por lo tanto que esté en la ruta de acceso del sistema. Desde la versión 2.0 del lenguaje, no hay una aplicación disponible en la interfaz gráfica de los diferentes sistemas operativos donde `Jason` puede ejecutarse. En lugar de ello, deben usarse estos scripts. Como se menciona en el `README` es necesario tener instalado java 1.7 o superior (y de preferencia `gradle`).

1 http://jason.sourceforge.net/wp/
2 http://jason.sourceforge.net/jBook/jBook/Home.html

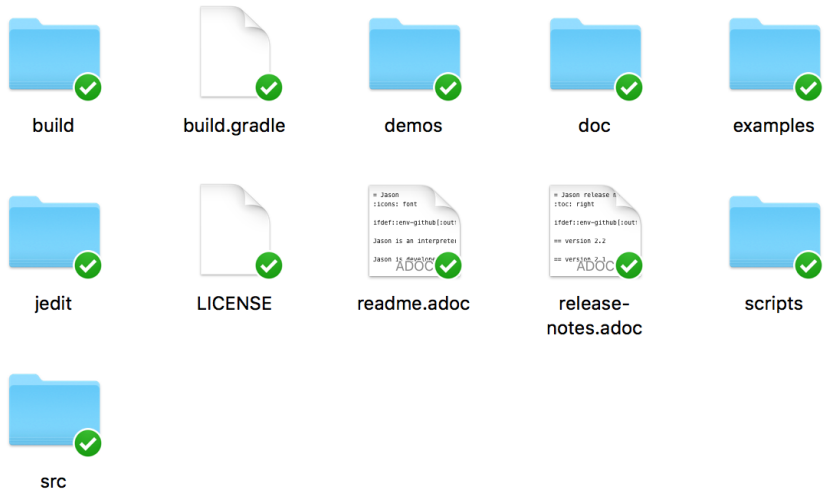


Figura 10.1: La carpeta principal de la distribución de Jason.

10.1.2 Distribución en github

Si bien las versiones del repositorio oficial en `sourcforge` se actualizan cuando se libera una versión mayor del lenguaje, es posible instalar los avances intermedios si basamos nuestra instalación usando el repositorio para desarrolladores disponible en `github` [3] y compilándolo con ayuda de `gradle`:

```
1 > git clone https://github.com/jason-lang/jason.git
2 > cd jason
3 > gradle config
```

Otras tareas para `gradle` se describen en el Cuadro 10.1. Se deduce que `config` hace en realidad tres cosas: Compila las fuentes para generar los archivos `jar` correspondientes; configura el archivo de propiedades de `jason` y coloca todos los `jar` en la carpeta `bulid/libs`; y solicita al usuario su autorización para configurar las variables `JASON_HOME` y `PATH` adecuadamente [4].

Acción	Descripción
`jar`	Genera un nuevo `jason.jar`
`doc`	Genera javadoc y transforma asciidoc en html.
`eclipse`	Genera una configuración para proyectos eclipse.
`config`	Ejecuta las tres acciones anteriores.
`clean`	Borra los archivos generados.
`release`	Produce un zip en `build/distributions`.

Cuadro 10.1: Acciones Gradle para instalar Jason.

10.2 AMBIENTES DE DESARROLLO

`Jason` cuenta con dos ambientes de desarrollo, el oficial basado en `jEdit` –Un editor de código multi lenguaje, implementado en java; y en `eclipse`, con todas las ventajas para `java` que esto conlleva.

3 https://github.com/jason-lang/jason.git
4 Efectivamente, es mejor usar `Jason` en un ambiente UNIX-like.

10.2.1 jEdit

Si abren una consola y ejecutan `jason-ide` tendrán acceso a la ventana principal de un ambiente de desarrollo basado en **jEdit** (Figura 10.2). Al estar implementado en `java`, este IDE no responde exactamente igual que las aplicaciones MacOS nativas, p. ej., su menú está en la ventana de la aplicación y no en la barra de menues traticional.

jEdit

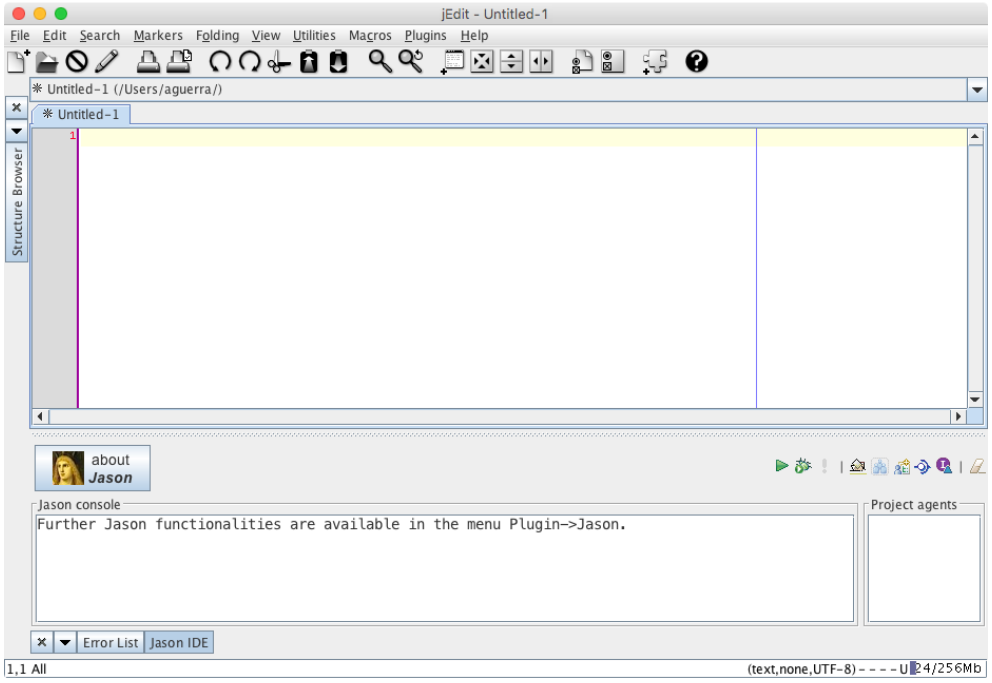


Figura 10.2: La ventana principal del IDE de Jason, basado en jEdit.

En este ambiente pueden definir y editar los componentes del sistema; ver el listado de agentes en el mismo; así como depurar y ejecutar el sistema definido. Es importante señalarle a Jason que versión de Java utilizaremos. Para ello el menú `Plugins:Plugins options` (Ver Figura 10.3) permite configurar éste y otros parámetros de Jason, como que versión de ant se usará para compilar ó las librerías que serán usadas con el sistema (jade, cartago, etc.). Observen que en mi caso, estoy usando la distribución `github` de `jason`; `java` 9; y la versión de `ant` que viene incluída en la distribución de `jason`, al igual que `jade`.

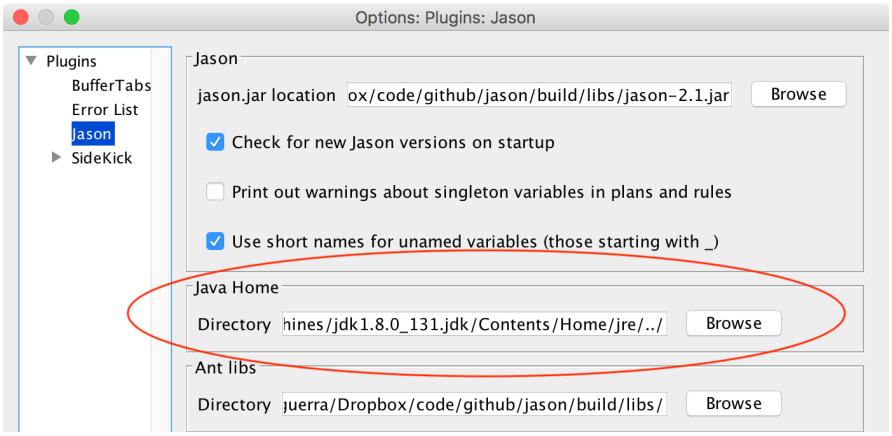


Figura 10.3: Configuración de Jason y Java en jEdit.

10.2.2 Eclipse

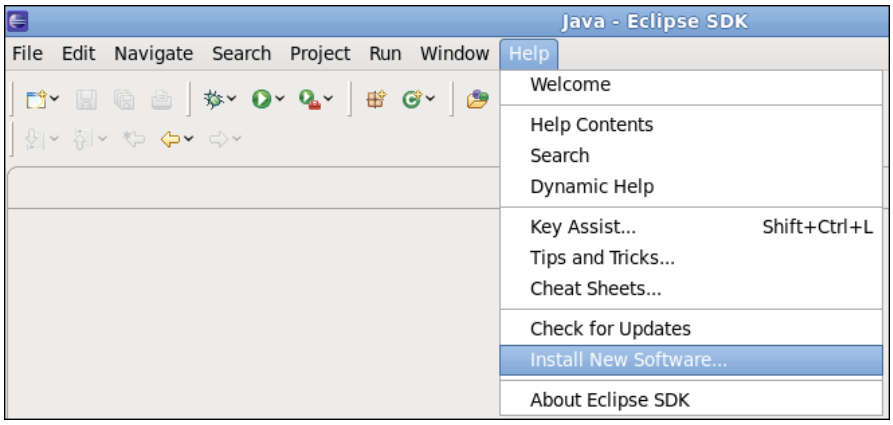


Figura 10.4: La ventana para instalar nuevo software en Eclipse.

Plug-in Eclipse

Tambien es posible instalar un *plug-in* [5] en `eclipse` [6] para trabajar con `Jason`. Para ello, desde `eclipse`, vaya al menu `Ayuda/Instalar Nuevo Software`, lo cual abrirá la ventana que se muestra en la Figura 10.4. Dar clic en el botón de agregar **repositorio** (*Add...*), para proveer la siguiente información:

Repositorio

- Nombre: jasonide
- Localidad: http://jason.sourceforge.net/eclipseplugin/juno/

Acepte la licencia del *plug-in* (la misma que Jason) y el ambiente estará instalado.

Diferencias

La organización de los proyectos en `jEdit` y `eclipse` **difieren** ligeramente. La diferencia más evidente es que, mientras que en jEdit las fuentes de nuestros programas están en el directorio raíz del proyecto, en Eclipse están en la carpeta `src`, que a su vez contiene dos carpetas `java` y `asl`. En este caso, es necesario especificar el lugar donde están los programas de agente, mediante la siguiente instrucción en el archivo principal del proyecto (el que tiene la extensión `mas2j`): `aslSrcPath: "src/asl"`.

10.3 DEFINIENDO UN SMA

Proyecto nuevo
Infraestructura

En el ambiente de desarrollo basado en `jEdit`, el botón  abre una ventana para definir un **proyecto nuevo**. La Figura 10.5 muestra las entrada para crear un SMA, al que llamaremos `saludos`. La **infraestructura** centralizada define un SMA donde todos los agentes estarán ejecutándose en la misma computadora. Es posible distribuir a los agentes en una red usando como infraestructura `Jade` [6].

El resultado de esta operación es un folder en la ruta indicada como directorio, p. ej. `/Users/aguerra/saludos`. Este directorio contendrá el archivo `saludos.mas2j` con la definición del SMA:

```
1  /* Jason Project */
2
3  MAS saludos {
4    infrastructure: Centralised
5    agents:
6  }
```

Añadir agentes

Observen que no hay agentes en este SMA. El botón  nos permite **añadir**

5 http://jason.sourceforge.net/eclipseplugin/juno/
6 https://www.eclipse.org

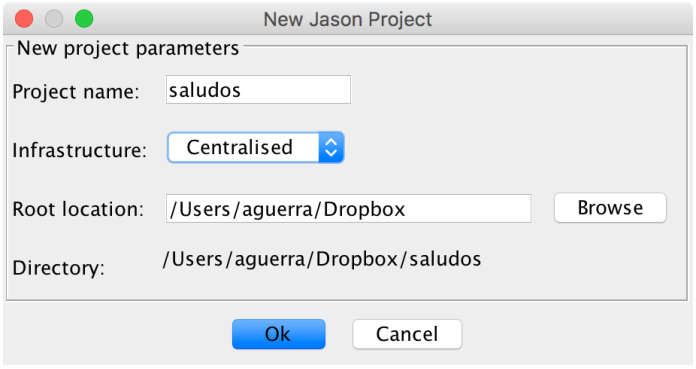


Figura 10.5: Definiendo un nuevo Sistema Multiagente llamado `saludos`.

agentes. Añadiremos dos agentes al sistema, `enrique` y `beto`. La Figura 10.6 muestra la ventana para la definición del agente `enrique`, se procede de igual manera para `beto`.

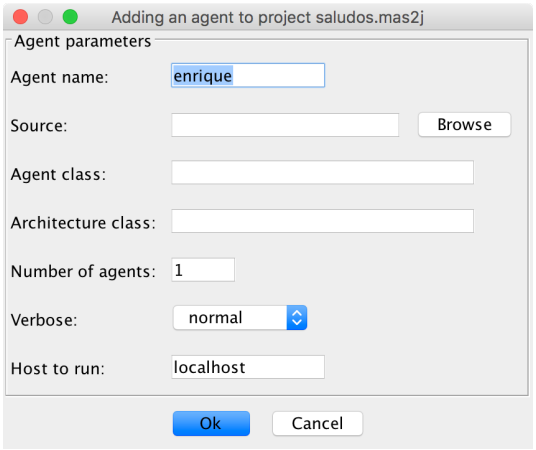


Figura 10.6: Añadiendo al agente enrique al SMA.

Clase y Arquitectura de agentes

Aunque no usaremos las opciones disponibles de momento, observen que es posible definir una **clase** y una **arquitectura** para los agentes que añadimos al sistema. También es posible clonar un agente el número de veces que queramos. Como el sistema que definimos usa la infraestructura centralizada, nuestros agentes corren en el servidor local. También es posible variar la cantidad de información que un agente despliega al ser ejecutado, variando el nivel de `verbose`. Después de agregar a ambos agentes, nuestro archivo `saludos.mas2j` debería lucir así:

```
1 /* Jason Project */
2
3 MAS saludos {
4     infrastructure: Centralised
5     agents:
6         enrique;
7         beto;
8 }
```

Además, en el folder el proyecto encontraremos los archivos fuente de `enrique.asl` y `beto.asl`. Primero, modificaremos al agente `enrique`, de forma que su archivo fuente sea como sigue:

```
1   // Agent enrique in project saludos.mas2j
2
3   /* Initial beliefs and rules */
4
5   /* Initial goals */
```

```
6
7   !start.
8
9   /* Plans */
10
11  +!start : true <- .send(beto,tell,hola).
```

Luego eliminaremos de `beto` todo lo que no sea comentarios. Observen que no hemos definido creencias para ninguno de los dos agentes.

El botón  lanza el **inspector de mentes**, que nos permite ejecutar el SMA paso a paso; y observar el estado mental de los agentes. Como es de esperar, dadas las definiciones previas, `enrique` saluda a `beto` y éste último registra en sus creencias el saludo por la performativa `tell` del mensaje en cuestión. Eventualmente el inspector de mentes debería mostrar esta información para `beto`, tal y como se muestra en la Figura 10.7.

Inspector de mentes

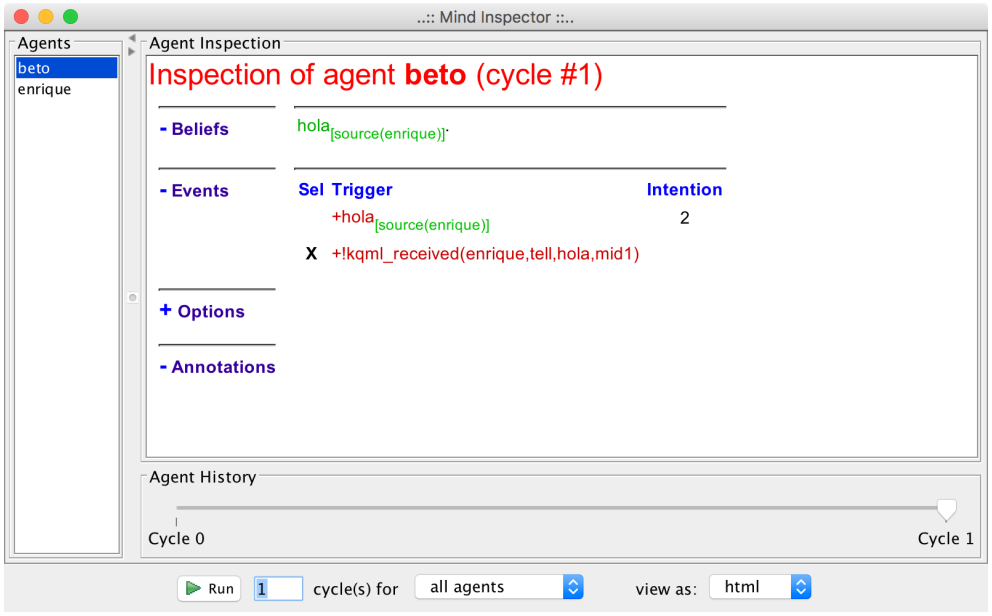


Figura 10.7: El estado mental del agente `beto` al recibir el mensaje de `enrique`

Podemos hacer que `beto` sea más receptivo, modificando su programa para reaccionar al mensaje de `enrique`.

```
1   // Agent beto in project saludos.mas2j
2
3   /* Initial beliefs and rules */
4
5   /* Initial goals */
6
7   /* Plans */
8
9   +hola[source(Ag)] <- .print("Recibí un saludo de ", Ag).
```

Es más, podemos hacer de `beto` un agente educado:

```
1  // Agent beto in project saludos.mas2j
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6
7  /* Plans */
8
9  +hola[source(Ag)] <-
10   .print("Recibí un saludo de ", Ag);
11   .send(Ag,tell,hola).
```

Lo mismo para `enrique`:

```
// Agent enrique in project saludos.mas2j

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start : true <- .send(beto,tell,hola).

+hola[source(Ag)] <-
  .print("Recibí un saludo de ", Ag);
  .send(Ag,tell,hola).
```

¿Cual será la salida en consola al ejecutar este SMA? Aunque parezca extraño, los agentes no entran en un ciclo de `saludos`, debido a que percepción y actualización de creencias no son equivalentes. De forma que la salida en consola al ejecutar este SMA será como se muestra en la Figura 10.8.

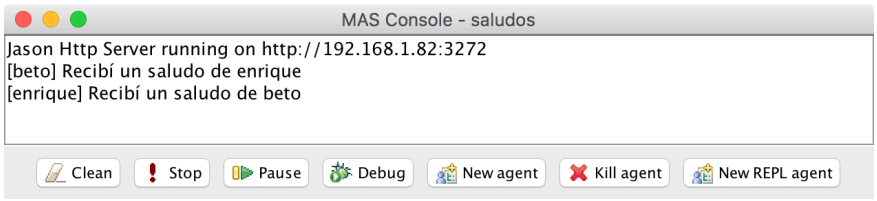


Figura 10.8: La consola de `beto` durante la ejecución del SMA.

La percepción, en este caso, está asociada a la recepción del mensaje con performativa `tell`; mientras que la actualización de creencias tiene que ver con qué hace cada agente con el contenido del mensaje. Como `hola` ya está en las creencias de ambos agentes luego del primer mensaje recibido, en los mensajes posteriores no se agrega nada al estado mental de los agentes y, por tanto, el evnto `+hola` nunca se produce, de ahí que no se impriman más mensajes en la consola.

Es posible utilizar `Jade` como infraestuctura, modificando el archivo de definición del sistema `saludos.mas2j` de la siguiente manera:

```
/* Jason Project */

MAS saludos {
  infrastructure: Jade
  agents:
  enrique;
  beto;
}
```

esto permite ejecutar un agente `sniffer` (Ver Figura 10.9) para observar la comunicación entre `enrique` y `beto`; además de que eventualmente nos permitiría distribuir los agents en una red de cómputo y otras facilidades. Para que el `sniffer` se ejecute automáticamente al lanzar el sistema, es necesario indicarlo en las preferencias del *plug-in* de Jason en el ambiente de desarrollo basado en jEdit.

10.4 IMPLEMENTACIÓN DE MEDIOS AMBIENTES

Los agentes están situados en su medio ambiente y los lenguajes de programación orientados a agentes deberían proveer una noción explícita de éste. Aunque esto no pareciera ser mandatorio en el caso de los agentes puramente comunicativos, como `enrique` y `beto`, observen que los actos de habla buscan ajustar el medio ambiente

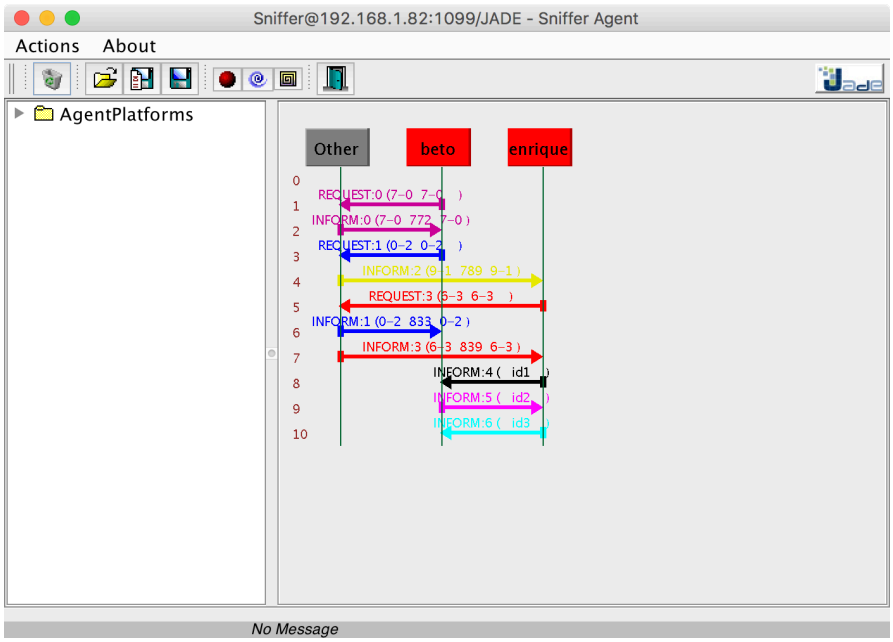


Figura 10.9: Un agente `sniffer` monitoreando comunicaciones.

a los estados Intencionales del agente; y que las creencias son representaciones del agente ajustadas al medio ambiente.

En el caso de agentes situados en medios ambientes reales, aunque la simulación no es mandatoria, tiene algunas ventajas a saber: Los agentes y los SMA son sistemas distribuidos de un alto grado de complejidad. Aunque existen herramientas formales para la verificación de estos sistemas, la validación mediante simulación sigue siendo una práctica muy extendida.

Arquitectura de agente

En todo caso, simulado o real, el acceso de Jason al medio ambiente se define a través de Java. La **arquitectura general** de un agente incluye los métodos Java que definen la interacción con el ambiente, como se muestra en diagrama de secuencia UML de la Figura 10.10.

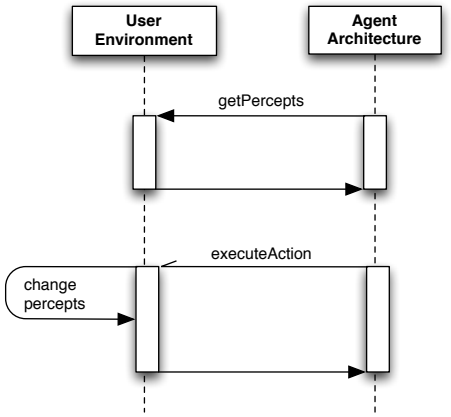


Figura 10.10: Interacción entre la implementación del ambiente y la arquitectura del agente.

Focus de atención
Actualización de creencias

La arquitectura de un agente utiliza el método `getPercepts` para obtener las percepciones del ambiente simulado. Estas pueden verse como propiedades del ambiente accesibles al agente, de forma que este método establece un mecanismo de **focus** de atención. A partir de esta información el agente **actualiza** sus creencias, normalmente cuando su ciclo de razonamiento está en el estado *ProcMsg*. Observen que la percepción y la actualización de creencias son dos procesos diferentes.

Ahora bien, cuando el agente ejecuta una acción, la arquitectura solicita al ambiente la ejecución de la acción y suspende la intención asociada hasta que el ambiente provee **retroalimentación** sobre la ejecución de la acción, normalmente, que la acción ha sido ejecutada. La verificación de si los efectos esperados de la acción se cumplieron o no, está asociada normalmente a la percepción y no a esta retroalimentación.

Retroalimentación

Observen que el ciclo del razonamiento del agente continua mientras la intención asociada a la acción ejecutada está **suspendida**. Esto tiene un efecto similar a si el método `executeAction` fuese invocado de forma asíncrona. Si el ambiente está siendo ejecutado en otra máquina, el lapso de esta suspensión puede ser considerable.

Intenciones suspendidas

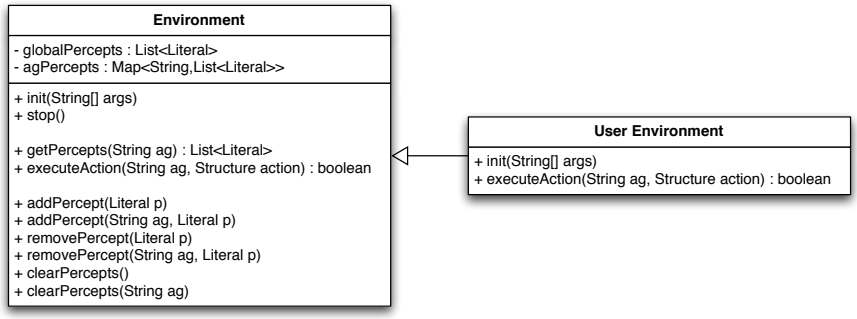


Figura 10.11: Implementación de un ambiente extendiendo la clase `Environment`.

Para implementar un ambiente en Jason, el programador normalmente extiende la clase `Environment` y redefine (usando *override*) los métodos `executeAction`, `init` y `stop`. La Figura 10.11 muestra un diagrama de clase mostrando esta relación. Una implementación de la clase ambiente extendida suele tener la estructura mostrada en el Cuadro 10.2.

Clase Environment

```
1  import jason.asSyntax.*;
2  import jason.environment.*;
3
4  public class <EnvironmentName> extends Environment {
5    // Los miembros de la clase...
6
7    @Override
8    public void init(String[] args) {
9      // Qué hacer al iniciar la ejecución...
10   }
11
12   @Override
13   public boolean executeAction(String ag, Structure act) {
14     // Efectos de las acciones...
15   }
16
17   @Override
18   public void stop() {
19     // Qué hacer al detener el sistema...
20   }
21 }
```

Cuadro 10.2: Implementación del ambiente del usuario.

El Cuadro 10.3 resume los métodos de Java que pueden usarse para programar un ambiente Jason. Solo objetos de la clase `Literal`, que es parte del paquete `jason` pueden agregarse a las listas de percepciones mantenidas por la clase `Environment`. En esta parte no debería considerarse agregar anotaciones a las literales, pues todas son anotadas automáticamente con `source(percept)`.

Clase Literal

La mayor parte del código relacionado con la implementación de ambientes está en el método `executeAction`, que debe declararse tal y como se muestra en el

Ejecución de una acción

Método	Semántica
`addPercept(L)`	Agrega la literal L a la lista global de percepciones.
`addPercept(A,L)`	Agrega la literal L a las percepciones del agente A.
`removePercept(L)`	Remueve la literal L de la lista global de percepciones
`removePercept(A,L)`	Remueve la literal L de las percepciones del agente A.
`clearPercepts()`	Borra las percepciones de la lista global.
`clearPercepts(A)`	Borra las percepciones del agente A.

Cuadro 10.3: Métodos Java para programar ambientes Jason.

Cuadro 10.2. Siempre que un agente trata de ejecutar una acción en el ambiente, el nombre del agente y una estructura representando la acción solicitada son enviadas a este método como parámetros. El código en `executeAction` suele verificar la estructura *à la Prolog* que representa la acción y el agente que intenta ejecutar la acción. Luego, para cada combinación acción/agente que sea relevante, el código hace lo necesario en el **modelo** del ambiente. Normalmente esto incluye cambiar ciertas percepciones. Observen que la ejecución de una acción es booleana y regresa falso si la solicitud de ejecución al ambiente fallo. Un plan falla si alguna de sus acciones falla al ser ejecutada.

Recuerden que la percepción y la actualización de creencias no son procesos equivalentes. Esta posible confusión es causa de algunos errores al implementar ambientes y su interacción con los agentes mediante las clases y métodos definidos en Jason. Se suele esperar que los agentes mantengan en su estado mental las percepciones aún cuando estás solo estén presentes durante un ciclo de razonamiento. Esto es falso. Si un agente necesita recordar percepciones pasadas que ya no se dan en el ambiente, es necesario crear **notas mentales** al percibir la propiedad en cuestión a través de sus planes. Las notas mentales se recuerdan hasta que explícitamente son olvidadas. Las creencias asociadas a una percepción son eliminadas en cuanto la percepción se deja de observar en el ambiente. También es posible que una percepción desaparezca como efecto de la ejecución de una acción, antes de que el agente pueda formar una creencia acerca de ella. Aunque consideren que el proceso de actualización de creencias genera eventos asociados a agregar y borrar creencias.

Notas mentales

Veamos todo esto con un ejemplo sencillo. Vamos a crear un nuevo SMA llamado `piros` y agregar un agente llamado `piro`. El código del agente será como sigue:

```
1  // Agent piro in project piros.mas2j
2
3  /* Initial beliefs and rules */
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : true <- incendiar.
12
13 +fuego <- .print("Fuego! Corran").
```

El agente `piro` es un buen piromano que incendia su ambiente y, eso si, una vez que percibe fuego, avisa que hay que iniciar la huída. La creencia `fuego` deberá ser agregada por el ambiente, en respuesta a la acción externa `incendiar`.

El botón  nos permite definir un ambiente para el SMA que estamos definiendo. Al darle clic una ventana nos pedirá el nombre del ambiente. La única consideración aquí es que el ambiente es una clase de Java, y por tanto, su nombre debe

iniciar con mayúscula. En nuestro caso el ambiente se llamará `PirosAmb`. Modifique el código del ambiente como sigue:

```
// Environment code for project piros.mas2j

import jason.asSyntax.*;
import jason.environment.*;
import java.util.logging.*;

public class PirosAmb extends Environment {

  private Logger logger = Logger.getLogger("piros.mas2j."+PirosAmb.class.getName());

  @Override
  public void init(String[] args) {
    super.init(args);
  }

  @Override
  public boolean executeAction(String agName, Structure action) {
    if (action.getFunctor().equals("incendiar")) {
      addPercept(Literal.parseLiteral("fuego"));
      return true;
    } else {
      logger.info("executing: "+action+", but not implemented!");
      return false;
    }
  }

  @Override
  public void stop() {
    super.stop();
  }
}
```

en este caso, la inicialización y cierre del ambiente se heredan de la clase `Environment`. Solo estamos redefiniendo el método `executeAction` para agregar la percepción `fuego` en respuesta a la acción `incendiar`. Observen el uso de `addPercept` para implementar la percepción del fuego. Al ejecutar el SMA, la salida en consola es la siguiente:

```
[piro] Fuego! Corran
```

10.5 JASON À LA PROLOG

Programación Lógica

De cierta forma, las creencias de Jason y las metas verificables se comportan de manera muy similar a un sistema de **Programación Lógica**, digamos Prolog [103, 17, 25]. Para ilustrar esto vamos a crear un nuevo proyecto en `eclipse`, llamado `creencias`, con una infraestructura centralizada y sin un medio ambiente asociado. Si todo va bien, el navegador de Jason, debe mostar el proyecto que incluye al agente `sample_agent`.

10.5.1 Hechos, reglas y metas verificables

Modifiquemos este agente para incluir creencias sobre su familia (clásico ejemplo de Prolog):

```
// Agent agent1 in project creencias

/* Initial beliefs and rules */

progenitor(carmelo,alejandro).
progenitor(carmen,alejandro).
progenitor(carmelo,laura).
```

```
progenitor(carmen,laura).
progenitor(laura,rafael).
progenitor(isidro,rafael).

/* Initial goals */

!start.

/* Plans */

+!start <-
  ?progenitor(laura,rafael);
  .print("Laura es progenitor de Rafael");
  ?progenitor(carmelo,Y);
  .print("Caramelo es progenitor de ", Y, ".");
  ?progenitor(X,rafael);
  .print(X," es un progenitor de Rafael").
```

Preguntas

A diferencia de Prolog, es el agente y no el usuario quien hace las `preguntas` en este caso. Para ello se define el plan de este agente. Su primera acción verifica si un hecho es verdadero (que Laura es progenitor de Rafael); y luego se hacen dos preguntas más para saber de quién es progenitor Carmelo y quién es progenitor de Rafael. La acción interna `.print`, imprime mensajes en consola. En este caso, la salida del programa sería la siguiente:

```
[agent1] Laura es progenitor de Rafael
[agent1] Caramelo es progenitor de alejandro.
[agent1] laura es un progenitor de rafael
```

Fallo

Si una pregunta **falla**, el plan falla y la intención asociada también. Por ejemplo, si agregamos la meta verificable `?madre(laura,rafael)` al final del plan del agente, tendremos un fallo, ya que tal meta no puede ser resuelta:

```
[agent1] Laura es progenitor de Rafael
[agent1] Caramelo es progenitor de alejandro.
[agent1] laura es un progenitor de rafael
[agent1] No failure event was generated for +!start[code(madre(laura,
rafael)),code_line(25),code_src(".../creencias/src/asl/sample_agent.asl"),
error(test_goal_failed), error_msg("Failed to test '?madre(laura,rafael)'"),
source(self)]
```

Observen el uso de las etiquetas para registrar el fallo: La meta `+!start` falló, debido a que una meta verificable `?madre(laura,rafael)` ha fallado. Hay varios términos en las etiquetas del evento de fallo que nos puede ser de utilidad en estos casos:

- *code*(C) nos indica que C fue el elemento del programa que causó el fallo.
- *code_src*(Asl) nos indica que Asl es el programa de agente que falló.
- *code_line*(L) nos indica que el error se produjo en la línea L.
- *error*(X) nos indica que el error X se produjo.
- *error_msg*(Msg) nos indice que Msg es el mensaje que será desplegado en consola para señalar el error.+

Planes de fallo

Esta información puede ser usada para agregando los planes relevantes ($-!\alpha$), para **contender con el error**:

```
// Agent agent2 in project creencias

/* Initial beliefs and rules */

progenitor(carmelo,alejandro).
progenitor(carmen,alejandro).
progenitor(carmelo,laura).
progenitor(carmen,laura).
```

```
progenitor(laura,rafael).
progenitor(isidro,laura).

/* Initial goals */

!start.

/* Plans */

+!start <-
  ?progenitor(laura,rafael);
  .print("Laura es progenitor de Rafael");
  ?progenitor(carmelo,Y);
  .print("Caramelo es progenitor de ", Y, ".");
  ?progenitor(X,rafael);
  .print(X," es un progenitor de Rafael");
  ?madre(laura,rafael).

-!start[error(Error)] <-
  .print("El plan +!start falló por el error ", Error).
```

con lo que el error es procesado adecuadamente:

```
[agent1] Laura es progenitor de Rafael
[agent1] Carmelo es progenitor de alejandro.
[agent1] laura es un progenitor de rafael
[agent1] El plan +!start falló por el error test_goal_failed
```

En realidad, en este caso querríamos agregar conocimiento al agente para contender con la meta problemática, en lugar de procesar el fallo una vez que éste ha sucedido. Agregar conocimiento, significa agregar creencias al agente, incluyendo reglas:

```
// Agent agent3 in project creencias

/* Initial beliefs and rules */

progenitor(carmelo,alejandro).
progenitor(carmen,alejandro).
progenitor(carmelo,laura).
progenitor(carmen,laura).
progenitor(laura,rafael).
progenitor(isidro,laura).

mujer(laura).
mujer(carmen).
hombre(carmelo).
hombre(alejandro).
hombre(isidro).

madre(X,Y) :-  mujer(X) & progenitor(X,Y).
padre(X,Y) :-  hombre(X) & progenitor(X,Y).

/* Initial goals */

!start.

/* Plans */

+!start <-
  ?progenitor(laura,rafael);
  .print("Laura es progenitor de Rafael");
  ?progenitor(carmelo,Y);
  .print("Caramelo es progenitor de ", Y, ".");
  ?progenitor(X,rafael);
  .print(X," es un progenitor de Rafael");
  ?madre(laura,rafael);
  .print("Laura es madre de Rafael");
  ?madre(Z,alejandro);
  .print(Z, " es madre de Alejandro").

```

```
-!start[error(Error)] <-
  .print("El plan +!start falló por el error ", Error).
```

La salida es la siguiente:

```
[agente3] Laura es progenitor de Rafael
[agente3] Caramelo es progenitor de alejandro.
[agente3] laura es un progenitor de Rafael
[agente3] Laura es madre de Rafael
[agente3] carmen es madre de Alejandro
```

Por supuesto que las reglas pueden ser recursivas, por ejemplo:

```
// Agent agent4 in project creencias

/* Initial beliefs and rules */

progenitor(carmelo,alejandro).
progenitor(carmen,alejandro).
progenitor(carmelo,laura).
progenitor(carmen,laura).
progenitor(laura,rafael).
progenitor(isidro,rafael).

mujer(laura).
mujer(carmen).
hombre(carmelo).
hombre(alejandro).
hombre(isidro).

madre(X,Y) :-  mujer(X) & progenitor(X,Y).
padre(X,Y) :-  hombre(X) & progenitor(X,Y).

ancestro(X,Y) :-  progenitor(X,Y).
ancestro(X,Y) :-  progenitor(X,Z) & progenitor(Z,Y).

/* Initial goals */

!start.

/* Plans */

+!start <-
  ?ancestro(carmelo,rafael);
  .print("Carmelo es un ancestro de Rafael");
  ?ancestro(X,rafael);
  .print(X, " es un ancestro de Rafael");
  .findall(Xs, ancestro(Xs,rafael),L);
  .print("Los ancestros de Rafael son ",L).
```

Con la siguiente salida:

```
[agente4] Carmelo es un ancestro de Rafael
[agente4] laura es un ancestro de Rafael
[agente4] Los ancestros de Rafael son [laura,isidro,carmelo,carmen]
```

Observen la acción interna `.findall`, que se usa al igual que en Prolog, para colectar todas las respuestas posibles a una meta dada. La acción interna `.setof` hace lo mismo, pero sin incluir soluciones repetidas, construyendo el conjunto solución de manera incremental. El primer argumento de estas acciones es un patrón que representa la forma en que los resultados serán recolectados. En este caso, como solo colectamos la variable `X`, solo los nombres de quien sea ancestro serán colectados. Si substituímos `X` por `ancestro(X)`, obtendríamos una lista de estos.

```
[agente4] Los ancestros de Rafael son [ancestro(laura),ancestro(isidro),
ancestro(carmelo),ancestro(carmen)]
```

El segundo argumento de estas acciones es la meta a resolver. Su tercer argumento es una **lista**, donde los resultados son recolectados.

Listas

10.5.2 Listas

Las listas se representan igual que en Prolog. La lista **vacía** puede denotarse por `[]` y la lista que tiene una **cabeza** `X` y una **cola** `[Xs]` de denota como `[X|Xs]`. Veamos un ejemplo de búsqueda en una lista.

Lista vacía
Cabeza
Cola

```
// Agent agente5 in project creencias

/* Initial beliefs and rules */

busqueda(X,[X|_]).
busqueda(X,[Y|Ys]) :- busqueda(X,Ys).

/* Initial goals */

!start.

/* Plans */

+!start : true <-
  Lista = [1,2,3,4,5];
  ?busqueda(3,Lista);
  .print("3 es miembro de la lista ",Lista);
  .findall(X,busqueda(X,Lista),L);
  .print("Los miembros de la Lista son ",L).
```

Cuya salida en consola es:

```
[agente5] 3 es miembro de la lista [1,2,3,4,5]
[agente5] Los miembros de la Lista son [1,2,3,4,5]
```

Agreguemos la regla *elimina*/3 que elimina un elemento de una lista y regresa la lista modificada:

```
// Agent agente6 in project creencias

/* Initial beliefs and rules */

busqueda(X,[X|_]).
busqueda(X,[Y|Ys]) :- busqueda(X,Ys).

elimina(X,[X|Xs],Xs).
elimina(X,[Y|Ys],[Y|Zs]) :- elimina(X,Ys,Zs).

/* Initial goals */

!start.

/* Plans */

+!start : true <-
  Lista = [1,2,3,4,5];
  .print("La lista original es ",Lista);
  ?elimina(3,Lista,Resultado);
  .print("Eliminar 3 de la lista resulta en ",Resultado).
```

Cuya salida es:

```
[agente6] La lista original es [1,2,3,4,5]
[agente6] Eliminar 3 de la lista resulta en [1,2,4,5]
```

Muchas de estos predicados se pueden implementar como **acciones internas** predefinidas y definidas por el usuario en Java. El cuadro 10.4 resume las acciones predefinidas para el manejo de listas y conjuntos.

Acciones internas

El siguiente agente prueba muchas de las acciones para listas:

```
// Agent agente7 in project creencias

/* Initial beliefs and rules */

```

Acción interna	Descripción
$.member(X, Xs)$	X es miembro de Xs.
$.length(X, L)$	La longitud de X es L.
$.empty(X)$	X es una lista vacía.
$.concat(L_1, \ldots, L_n)$	Concatena todas las listas en L_n.
$.delete(X, L, R)$	Elimina X de L resultando la lista R.
$.reverse(L, R)$	La lista R es el reverso de L.
$.shuffle(L, R)$	R es la lista L revuelta.
$.nth(N, L, R)$	R es en N-ésimo elemento de la lista L.
$.max(L, R)$	R es el máximo elemento de la lista L.
$.min(L, R)$	R es el mínimo elemento de la lista L.
$.sort(L, R)$	R es la lista resultante de ordenar L.
$.list(L)$	Verifica si L es una lista.
$.suffix(R, L)$	R es un sufijo de la lista L.
$.prefix(R, L)$	R es un prefijo de la lista L.
$.sublist(R, L)$	R es una sub-lista de la lista L.
$.difference(L_1, L_2, R)$	R es la diferencia entre L_1 y L_2.
$.intersection(L_1, L_2, R)$	R es la intersección de L_1 y L_2.
$.union(L_1, L_2, R)$	R es la unión de L_1 y L_2.

Cuadro 10.4: Acciones internas predefinidas para listas y conjuntos.

```
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : true <-
12   Lista1 = [1,2,3,4,5];
13   Lista2 = [a,b,c,d,e];
14   .print("La lista 1 es ",Lista1);
15   .print("La lista 2 es ",Lista2);
16   .member(X,Lista1);
17   .print(X, " es miembro de la lista 1");
18   .length(Lista1,Long);
19   .print("La longitud de la lista 1 es ",Long);
20   .concat(Lista1,Lista2,L3);
21   .print("Pegar la lista 1 y 2 nos da ",L3);
22   .delete(X,Lista1,L4);
23   .print("Borrar ",X," de la lista 1, nos da ",L4," Ooops!");
24   .delete(c,Lista2,L5);
25   .print("Borrar c de la lista 2 no es problema ",L5);
26   .shuffle(Lista1,L6);
27   .print("Revolver la lista 1 produce ",L6);
28   .reverse(Lista2,L7);
29   .print("Invertir la lista 2 ",L7);
30   .nth(Long-1,Lista1,Last);
31   .print("El último elemento de la lista 1 es ",Last);
32   .max(Lista1,MaxL1);
33   .print("El máximo elemento en la lista 1 es ",MaxL1);
34   .min(Lista2,MinL2);
35   .print("El mínimo elemento de la lista 2 es ",MinL2);
36   .sort(L6,L8);
37   .print("Ordenar la lista 1 revuelta resulta en ",L8).
```

Cuya salida se muestra a continuación:

```
1 [agente7] La lista 1 es [1,2,3,4,5]
```

```
[agente7] La lista 2 es [a,b,c,d,e]
[agente7] 1 es miembro de la lista 1
[agente7] La longitud de la lista 1 es 5
[agente7] Pegar la lista 1 y 2 nos da [1,2,3,4,5,a,b,c,d,e]
[agente7] Borrar 1 de la lista 1, nos da [1,3,4,5] Ooops!
[agente7] Borrar c de la lista 2 no es problema [a,b,d,e]
[agente7] Revolver la lista 1 produce [3,5,4,1,2]
[agente7] Invertir la lista 2 [e,d,c,b,a]
[agente7] El último elemento de la lista 1 es 5
[agente7] El máximo elemento en la lista 1 es 5
[agente7] El mínimo elemento de la lista 2 es a
[agente7] Ordenar la lista 1 revuelta resulta en [1,2,3,4,5]
```

Cláusulas vs Acciones Internas

Eficiencia

Comunicación Semántica

Algunas **observaciones** son pertinentes. Como su nombre lo indica, estos constructores no son creencias del agente, como si lo son las reglas y los hechos ejemplificados anteriormente. Las acciones internas son operaciones implementadas en Java, que no afectan el medio ambiente del agente. En principio, deberían ser más **eficientes** que sus contrapartes implementadas à la Prolog, pero como veremos luego, no son explotables al usar **actos de habla**.

Además, al no ser cláusulas, la **semántica** de estas operaciones no se sigue de la Programación Lógica, sino de su implementación en Java (La cual está muy bien documentada en el caso de las acciones internas predefinidas por Jason). Consideren *.delete* como ejemplo: El primer argumento de esta operación puede ser un término, una cadena de texto, o un número; y su comportamiento depende del tipo de argumento recibido de forma poco afortunada: Si queremos borrar las ocurrencias de 1 en una lista de números, esta acción no nos sirve, pues en realidad borrará el segundo elemento de la lista al ser su primer argumento un número (Ver salida anterior). El siguiente agente define una cláusula *del* que borra todas las ocurrencias de un término, número o no, en una lista.

```
// Agent agente8 in project creencias

/* Initial beliefs and rules */

del(_,[],[]).
del(X,[X|L1],L2) :-  del(X,L1,L2).
del(X,[H|L1],[H|L2]) :-  X\==H & del(X,L1,L2).

/* Initial goals */

!start.

/* Plans */

+!start : true <-
  Lista = [1,2,3,2,4,2,5];
  .delete(1,Lista,R1);
  .print("Eliminar 1 de la lista ",Lista," resulta en ",R1);
  ?del(2,Lista,R2);
  .print("Eliminar 2 de la lista ",Lista," resulta en ",R2).
```

Su salida en consola es:

```
[agente8] Eliminar 2 de la lista [1,2,3,2,4,2,5] resulta en [1,3,4,5]
```

10.5.3 Artimética

Jason provee una serie de operadores aritméticos predefinidos (Ver cuadro 10.5) como acciones internas. Al igual que con las acciones para listas, algunos de ellos se pueden reprogramar à la Prolog, si su semántica no es la esperada.

El siguiente agente hace uso de algunas funciones aritméticas:

```
// Agent agente9 in project creencias

/* Initial beliefs and rules */
```

$math.abs(N)$	$math.acos(N)$	$math.asin(N)$	$math.atan(N)$
$math.average(L)$	$math.cell(N)$	$math.cos(N)$	$.count(B)$
$math.e$	$math.floor(N)$	$.lenght(L)$	$math.log(N)$
$math.max(N_1, N_2)$	$math.min(N_1, N_2)$	$math.pi$	$math.random(N)$
$math.round(N)$	$math.sin(N)$	$math.sqrt(N)$	$math.std\_dev(L)$
$math.sum(L)$	$math.tan(N)$	$system.time$	

Cuadro 10.5: Acciones internas artiméticas.

```
4
5  /* Initial goals */
6
7  !start.
8
9  /* Plans */
10
11 +!start : true <-
12   Lista1 = [1,2,3,4,5];
13   .print("La lista 1 es ",Lista1);
14   .print("La longitud de la lista 1 ", .length(Lista1));
15   .print("La sumatoria de la lista 1 es ", math.sum(Lista1));
16   .print("El promedio de la lista 1 es ", math.average(Lista1)).
```

Su salida en consola es:

```
1 [agente9] La lista 1 es [1,2,3,4,5]
2 [agente9] La longitud de la lista 1 es 5
3 [agente9] La sumatoria de la lista 1 es 15
4 [agente9] El promedio de la lista 1 es 3
```

10.5.4 Otras estructuras

El siguiente agente incluye una serie de cláusulas para trabajar con árboles binarios.

```
1  // Agent agente10 in project creencias
2
3  /* Initial beliefs and rules */
4
5  insertaArbol(X,vacio,arbol(X,vacio,vacio)).
6
7  insertaArbol(X,arbol(X,A1,A2),arbol(X,A1,A2)).
8
9  insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1N,A2)) :-
10   X<Y & insertaArbol(X,A1,A1N).
11
12 insertaArbol(X,arbol(Y,A1,A2),arbol(Y,A1,A2N)) :-
13   X>Y & insertaArbol(X,A2,A2N).
14
15 creaArbol([],A,A).
16
17 creaArbol([X|Xs],AAux,A) :-
18   insertaArbol(X,AAux,A2) &
19   creaArbol(Xs,A2,A).
20
21 lista2arbol(Xs,A) :-  creaArbol(Xs,vacio,A).
22
23 nodos(vacio,[]).
24
25 nodos(arbol(X,A1,A2),Xs) :-
26   nodos(A1,Xs1) &
27   nodos(A2,Xs2) &
28   .concat(Xs1,[X|Xs2],Xs).
29
30 ordenaLista(L1,L2) :-
```

```
31    lista2arbol(L1,A) &
32    nodos(A,L2).
33
34  /* Initial goals */
35
36  !start.
37
38  /* Initial plans */
39
40  +!start <-
41    Lista1 = [5,3,4,1,2];
42    ?lista2arbol(Lista1,Arbol1);
43    .print("La lista 1 es ", Lista1);
44    .print("El árbol creado de la lista es ",Arbol1);
45    ?nodos(Arbol1,Nodos1);
46    .print("Cuyos nodos en orden son ",Nodos1).
```

Su salida en consola es la siguiente:

```
1  [agente10] La lista 1 es [5,3,4,1,2]
2  [agente10] El árbol creado de la lista es arbol(5,arbol(3,arbol(1,
3  vacio,arbol(2,vacio,vacio)),arbol(4,vacio,vacio)),vacio)
4  [agente10] Cuyos nodos en orden son [1,2,3,4,5]
```

10.5.5 Anotaciones

Anotación

Todas las creencias de Jason tienen al menos una **anotación** asociada, su fuente. En el inspector de mentes del sistema (Ver figura 10.12), podrán ver que todas las creencias usadas hasta ahora están adornadas con la etiqueta `source(self)`, que significa que se trata de creencias añadidas por el mismo agente. De hecho, como los agentes comunican y perciben creencias, etiquetar la fuente de éstas era una necesidad. Posteriormente, se generalizó el uso de las etiquetas para incluir meta-información asociada a las literales de Jason.

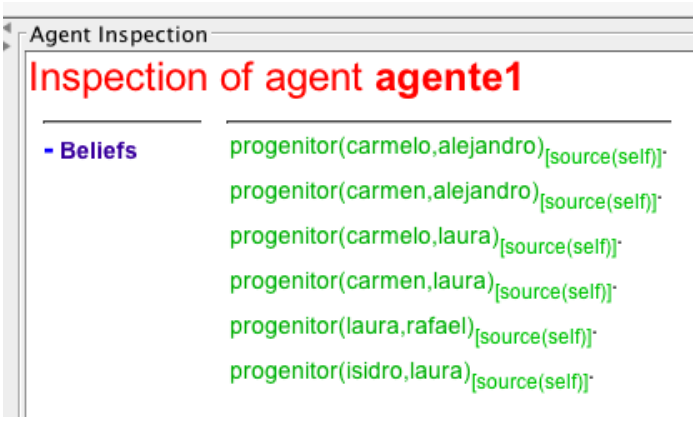


Figura 10.12: Las creencias de un agente siempre están anotadas, al menos con source(self).

Las anotaciones no cambian el poder expresivo del lenguaje de programación, pero mejoran su legibilidad. Su sintaxis es la de una lista de términos (cuidado, no de literales). Por ejemplo:

```
1  p(t)[source(self),costo(10),prioritario]
```

puede representar que la literal $p(t)$ ha sido agregada a las creencias por el agente mismo, tiene un costo de 10 unidades y se trata de algo prioritario. Observen que todo ello es meta-información sobre la creencia.

Aunque la sintaxis de las anotaciones se corresponde con la de una lista de términos, en realidad su semántica es la de un conjunto y así es como son consideradas por Jason.

Unificación con anotaciones

El uso de las anotaciones introduce una restricción al computar el unificador más general entre dos **literales**. L_1 unifica con L_2 si y sólo si las anotaciones de L_1 son un subconjunto de las de L_2. Por ejemplo

Unificador más general

Unificación entre literales

```
p(t) = p(t)[a1];          // Unifica
p(t)[a1] = p(t);          // No unifica
p(t)[a2] = p(t)[a1,a2,a3] // Unifica
```

Como las anotaciones son listas que representan conjuntos, la notación de **acceso a listas** para cabeza y cola pueden usarse con ellas:

Acceso a listas

```
p(t)[a2|As] = p(t)[a1,a2,a3]    // As unifica con [a1,a3]
p(t)[a1,a2,a3] = p(t)[a1,a4|As] // As unifica con [a2,a3]
```

La unificación con **variables** es un poco más complicada, ya que debe considerar los diversos casos de unificación para $X[As] = Y[Bs]$; y si las variables en cuestión son de base o no. Cuando X e Y son de base:

Unificación con variables

```
X     = p[Cs]  // unifica X con p[Cs]
Y     = p[Ds]  // unifica Y con p[Ds]
X[As] = Y[Bs]  // unifica si (Cs ∪ As) ⊂ (Ds ∪ Bs)
```

El siguiente ejemplo ilustra este caso:

```
X = p[a1,a2];
Y = p[a1,a3];
X[a4] = Y[a2,a4,a5]; // unifica
```

donde $Cs = \{a_1, a_2\}$, $Ds = \{a_1, a_3\}$, $As = \{a_4\}$ y $Bs = \{a_2, a_4, a_5\}$. De lo que se sigue que $Cs \cup As = \{a_1, a_2, a_4\}$ y $Ds \cup Bs = \{a_1, a_2, a_3, a_4, a_5\}$ y por tanto, se satisface la restricción definida previamente.

Cuando solo X es de base, la unificación se resuelve de la siguiente forma:

```
X     = p[Cs]
X[As] = Y[Bs]  // unifica si (Cs ∪ As) ⊂ Bs
               // e Y unifica con p
```

Cuando solo Y es de base, la unificación se resuelve de la siguiente forma:

```
Y     = p[Ds]
X[As] = Y[Bs]  // unifica si As ⊂ (Ds ∪ Bs)
               // y X unifica con p
```

10.5.6 Negación fuerte y débil

A diferencia de Prolog, donde el **principio del mundo cerrado** (*Closed World Assumption, CWA*) se adopta automáticamente, Jason puede contender también con una representación fuerte de la **negación**. Recuerden que el *CWA* expresa que todo lo que no se sabe cierto, o no es derivable de lo que se sabe cierto siguiendo las reglas del programa, es falso. En este sentido, Jason provee el operador `not`, donde la negación de una fórmula es cierta, si el intérprete falla al derivar dicha fórmula.

CWA

Negación

El operador de **negación fuerte** es utilizado para representar que el agente explícitamente cree que cierta fórmula no es el caso. La semántica de las negaciones, cuando se aplican a literales, se muestra en el cuadro 10.6.

Negación fuerte

El siguiente ejemplo ilustra todo lo que conocemos de las creencias. El agente cree que la *caja*1 es *roja*, pero según *beto* la *caja*1 es verde. Para complicar más la historia, según *enrique* la *caja*1 no es verde. La meta principal del agente es reportar de que color es la caja.

```
// Agent agente11 in project creencias

/* Initial beliefs and rules */

```

Sintaxis	Semántica
l	El agente cree que l es verdadera
$\sim l$	El agente cree que l es falsa
not l	El agente no cree que l es verdadera
not $\sim l$	El agente no cree que l es falsa.

Cuadro 10.6: Semántica de la negación de literales.

```
5  color(caja1,verde)[source(beto)].
6  ~color(caja1,verde)[source(enrique)]. // azul no causa contradicción
7  color(caja1,rojo). // verde hace que enrique sea el mentiroso
8
9  colorSegunYo(Caja,Color) :-
10   color(Caja,Color)[source(Src)] &
11   (Src == self | Src == percept).
12
13 descr(Ag,mentiroso) :-
14   mentiroso(Ag)[cert(C1)] &
15   daltonico(Ag)[cert(C2)] &
16   C1 > C2.
17 descr(Ag,daltonico) :-  daltonico(Ag).
18 descr(Ag,confiable).
19
20 /* Initial goals */
21
22 !start.
23
24 /* Plans */
25
26 @contradiccion
27 +!start : color(caja1,Color) & ~color(caja1,Color)[source(S2)] <-
28   .print("Contradicción detectada");
29   ?color(caja1,Color1)[source(S1)];
30   .print("La caja1 es de color ",Color1,", según ",S1);
31   ?colorSegunYo(caja1,Color2);
32   .print("Aparentemente el color de la caja1 es ",Color2,", según yo");
33   if (Color1 \== Color2) {
34     +mentiroso(S1)[cert(0.7)]; // Invertir y beto será mentiroso
35     +daltonico(S1)[cert(0.3)];
36   } else {
37     +mentiroso(S2)[cert(0.3)];
38     +daltonico(S2)[cert(0.7)];
39   };
40   ?descr(S1,Des1);
41   .print(S1, " es un agente ", Des1);
42   ?descr(S2,Des2);
43   .print(S2, " es un agente ", Des2).
44
45 @sinContradiccion
46 +!start <-
47   .print("No hay contradicciones detectadas");
48   ?colorSegunYo(caja1,Color);
49   .print("La caja1 es de color ",Color,", según yo").
```

Hay dos planes para contender con la meta principal del agente. El primero detecta contradicciones y el segundo no. En el segundo plan, el agene se pregunta por el color de la caja desde su propia perspectiva (la fuente es `self` o `percept`) y reporta el color encontrado.

Cuando la contradicción es detectada el agente confronta la situación. Reporta el color según su perspectiva y ajusta cuentas con los otros agentes. Si hay otro agente reportando un color diferente, nuestro agente creerá que tal agente es mentiroso o daltónico, con cierto grado de certidumbre. En caso contrario, hay un tercer agente causando la contradicción y éste es el mentiroso/daltónico. La salida en consola para este caso es:

```
[agente11] Contradicción detectada
[agente11] La caja1 es de color verde, segun beto
[agente11] Aparentemente el color de la caja1 es rojo, según yo
[agente11] beto es un agente daltonico
[agente11] enrique es un agente confiable
```

Si cambiamos la información sobre el color de la *caja*1 provista por *enrique* a *azul* (línea 6), tendremos que ya no hay contradicción detectable y la salida del programa es la siguiente:

```
 [agente11] No hay contradicciones detectadas
 [agente11] La caja1 es de color rojo, según yo
```

En cambio si nuestro agente creeyera que la *caja*1 es de color *verde* (línea 7), entonces el daltónico resultaría *enrique*:

```
[agente11] Contradicción detectada
[agente11] La caja1 es de color verde, segun beto
[agente11] Aparentemente el color de la caja1 es verde, según yo
[agente11] beto es un agente confiable
[agente11] enrique es un agente daltonico
```

Si se invierten los grados de certeza (líneas 34 y 35), resultará que *beto* es *mentiroso* en lugar de *daltonico*.

```
[agente11] Contradicción detectada
[agente11] La caja1 es de color verde, según beto
[agente11] Aparentemente el color de la caja1 es roja, según yo
[agente11] beto es un agente mentiroso
[agente11] enrique es un agente confiable
```

10.6 ACCIONES INTERNAS

Es posible definir acciones internas personalizadas, similares a las que hemos introducido en la sección anterior, por ejemplo `math.abs`, etc. Como el ejemplo sugiere, las acciones deben organizarse en **librerías**, que son paquetes de Java; mientras que las acciones propiamente dichas, son clases de Java que implementan la interfaz `InternalAction`. Jason provee una implementación por defecto de esta interfaz, conocida como `DefaultInternalAction`. Las acciones internas se denotan como `librería.acción`.

Librerías

Vamos a crear un SMA centralizado con un solo agente:

```
MAS distancia {

  infrastructure: Centralised

  agents:
    beto agent;

  aslSourcePath:
    "src/asl";
}
```

En donde el agente `agent1` haga uso de una acción interna para calcular la distancia euclidiana entre dos puntos:

```
// Agent in project distancia

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

```

```
+!start : true <-
  ia.distancia(10,10,20,50,D);
  .println("La distancia euclidiana entre (10,10) y (20,50) es ",D).
```

El código del agente `agente1` ilustra claramente el uso que queremos dar a la acción interna `distancia`. Los ambientes de desarrollo de Jason, permiten agregar acciones internas al sistema. Detalles más, detalles menos, lo importante es decirle al ambiente de desarrollo en que paquete será incluída la acción interna. La implementación de la acción es como sigue (generada, al igual que el SMA en el ambiente de desarrollo basado en eclipse):

```
1  // Internal action code for project distancia
2
3  package ia;
4
5  import jason.*;
6  import jason.asSemantics.*;
7  import jason.asSyntax.*;
8
9  /**
10  * @author aguerra
11  * ia.distancia: Computa la distancia euclidiana entre dos puntos.
12  */
13
14 public class distancia extends DefaultInternalAction {
15
16   private static final long serialVersionUID = 1L;
17
18   @Override
19     public Object execute(TransitionSystem ts, Unifier un, Term[] args)
20     throws Exception {
21         ts.getAg().getLogger().info("executing internal action 'distancia'");
22         try{
23           NumberTerm x1 = (NumberTerm) args[0];
24           NumberTerm y1 = (NumberTerm) args[1];
25           NumberTerm x2 = (NumberTerm) args[2];
26           NumberTerm y2 = (NumberTerm) args[3];
27
28           double distance = Math.abs(x1.solve()-x2.solve()) +
29                     Math.abs(y1.solve()-y2.solve());
30
31           NumberTerm result = new NumberTermImpl(distance);
32           return un.unifies(result,args[4]);
33         } catch (ArrayIndexOutOfBoundsException e) {
34           throw new JasonException("La acción interna 'distancia'"+
35               "no ha recibido cinco argumentos!");
36           } catch (ClassCastException e) {
37             throw new JasonException("La acción interna 'distancia'"+
38                 "ha recibido argumentos no numéricos!");
39             } catch (Exception e) {
40               throw new JasonException("Error en 'distancia'");
41             }
42   }
43 }
```

El diagrama de clases de esta acción se muestra en la Figura 10.13. Su lectura es como sigue: La acción interna `distancia` en el paquete `ia`, extiende la clase `DefaultInternalAction` que implementa la interfaz `InternalAction`.

La definición de la acción se hace en el paquete `ia` (línea 3). Es necesario importar la sintaxis y la semántica de Jason para poder procesar los argumentos de la acción que son términos numéricos (líneas 22-25).

Como los argumentos de `distancia` son términos numéricos, es posible usar el método `solve` para computarlos, de manera que es posible evaluar algo como `ia.distancia(1/2,1/2,3/4,3/4,D)`. El parámetro de salida de esta acción es el último, por lo que el valor computado `distance`, debe ser convertido a un término numérico (línea 30), antes de unificarlo con el argumento `args[4]`.

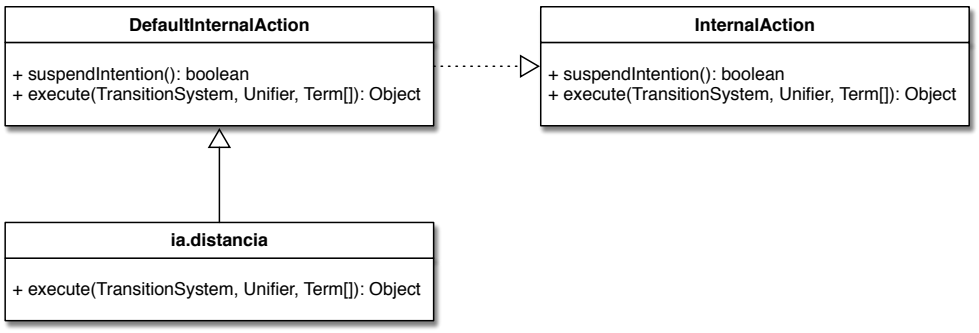


Figura 10.13: El diagrama de clases de la acción interna distancia.

La definición de `ia.distancia` también ilustra como procesar dos errores comunes: pasarle el número incorrecto de parámetros y pasarle parámetros no numéricos (líneas 32-39).

Si todo va bien, la salida en consola es la siguiente:

```
1  [agent1] executing internal action 'ia.distancia'
2  [agent1] La distancia euclidiana entre (10,10) y (20,50) es 50
```

10.7 MÓDULOS

10.8 ACTOS DE HABLA

La semántica operacional descrita en el capítulo anterior se implementa como una librería de planes que todos los agentes cargan al ser creados. La librería se encuentra en el directorio `/src/main/resources/asl/kqmlPlans.asl l` de la distribución de *Jason*. Los planes que contienden con enunciados performativos *tell* son:

```
11 /* ---- tell performatives ---- */
12
13 @kqmlReceivedTellStructure
14 +!kqml_received(Sender, tell, NS::Content, _)
15    :  .literal(Content) &
16       .ground(Content) &
17       not .list(Content) &
18       .add_nested_source(Content, Sender, CA)
19    <- ++NS::CA.   // add with new focus (as external event)
20 @kqmlReceivedTellList
21 +!kqml_received(Sender, tell, Content, _)
22    :  .list(Content)
23    <- !add_all_kqml_received(Sender,Content).
24
25 @kqmlReceivedTellList1
26 +!add_all_kqml_received(_,[]).
27
28 @kqmlReceivedTellList2
29 +!add_all_kqml_received(Sender,[NS::H|T])
30    :  .literal(H) &
31       .ground(H)
32    <- .add_nested_source(H, Sender, CA);
33       ++NS::CA;
34       !add_all_kqml_received(Sender,T).
35
36 @kqmlReceivedTellList3
37 +!add_all_kqml_received(Sender,[_|T])
38    <- !add_all_kqml_received(Sender,T).
39
40 @kqmlReceivedUnTell
41 +!kqml_received(Sender, untell, NS::Content, _)
```

```
42      <- .add_nested_source(Content, Sender, CA);
43         --NS::CA.
```

Observen que el primer caso a considerar es cuando el contenido (*Content*) del mensaje es una creencia (se trata de una sola literal de base, como verifican las acciones internas en el contexto del plan). En ese caso, se agrega la fuente al contenido con la acción interna `.add_nested_source` y se agrega la creencia con su nueva anotación a las creencias del agente. El segundo caso a considerar es cuando se recibe una lista de creencias que serán procesadas recursivamente. El último caso contiende con los enunciados performativos *untell*: se procede a anotar la creencia con su fuente y una vez etiquetada de esa forma, a eliminarla de las creencias del agente.

Los planes que contienden con enunciados performativos *achieve* son:

```
46  /* ---- achieve performatives ---- */
47
48  @kqmlReceivedAchieve
49  +!kqml_received(Sender, achieve, NS::Content, _)
50      : not .list(Content) & .add_nested_source(Content, Sender, CA)
51     <- !!NS::CA.
52  @kqmlReceivedAchieveList
53  +!kqml_received(Sender, achieve, Content, _)
54      : .list(Content)
55     <- !add_all_kqml_achieve(Sender,Content).
56
57
58  @kqmlReceivedAchieveList1
59  +!add_all_kqml_achieve(_,[]).
60
61  @kqmlReceivedAchieveList2
62  +!add_all_kqml_achieve(Sender,[NS::H|T])
63     <- .add_nested_source(H, Sender, CA);
64        !!NS::CA;
65        !add_all_kqml_achieve(Sender,T).
66
67
68  @kqmlReceivedUnAchieve[atomic]
69  +!kqml_received(_, unachieve, NS::Content, _)
70     <- .drop_desire(NS::Content).
```

Los planes que contienden con enunciados performativos *ask* son:

```
87  /* ---- ask performatives ---- */
88
89  @kqmlReceivedAskOne1
90  +!kqml_received(Sender, askOne, NS::Content, MsgId)
91      : NS::Content
92     <- .send(Sender, tell, NS::Content, MsgId).
93
94  @kqmlReceivedAskOne1b
95  +!kqml_received(Sender, askOne, NS::Content, MsgId)
96     <- .add_nested_source(Content, Sender, CA);
97        ?NS::CA;
98        // remove source annot from CA
99        CA  =.. [_,F,Ts,_];
100       CA2 =.. [_,F,Ts,[]];
101       .send(Sender, tell, NS::CA2, MsgId).
```

Los planes que contienden con enunciados performativos *knowHow* son:

```
90  -!kqml_received(Sender, askOne, NS::Content, MsgId)
91     <- .send(Sender, untell, NS::Content, MsgId).
92
93  @kqmlReceivedAskAll2
94  +!kqml_received(Sender, askAll, NS::Content, MsgId)
95     <- .findall(NS::Content, NS::Content, List);
96        .send(Sender, tell, List, MsgId).
97
98
99  /* ---- know-how performatives ---- */
100
```

```
// In tellHow, content must be a string representation
// of the plan (or a list of such strings)

@kqmlReceivedTellHow
+!kqml_received(Sender, tellHow, Content, _)
   <- .add_plan(Content, Sender).

// In untellHow, content must be a plan's
// label (or a list of labels)
@kqmlReceivedUnTellHow
```

Finalmente, se incluye un plan para contender con los errores generales de comunicación:

```
   <- .remove_plan(Content, Sender).

// In askHow, content must be a string representing
// the triggering event
@kqmlReceivedAskHow
```

10.8.1 Caso de estudio

Definiremos un SMA de dos agentes, *enrique* y *beto* que se comunican utilizando Actos de habla y las acciones internas provistas por Jason. El archivo principal es como sigue:

```
/*
Demo de comunicación

Un agente (enrique) se comunica con otro (beto) usando actos de
habla implementados en KQML y la acción interna .send
*/

MAS comunicacion {

    infrastructure: Centralised
    agents:
        enrique [beliefs="receptor(beto)"];
        beto [verbose=1]; // verbose=2 para ver más detalles

  aslSourcePath: "src/asl";
}
```

Este archivo `mas2j` no tiene novedades, salvo que las creencias de *ale* han sido inicializadas con la opción `beliefs`, de forma que *ale* crea que el receptor es *ana*. Esto se utilizará, como veremos a continuación, para dirigir los mensajes. Por otra parte, se puede incrementar el nivel de detalle de la salida en consolola con la opción `verbose`. De los dos agentes, *beto* responderá a los mensajes de *enrique*, como tiene un código más corto, lo revisaremos primero:

```
// Agente beto en el proyecto comunicacion.mas2j

vl(1).
vl(2).

/* El siguiente plan se dispara cuando se recibe un mensaje
   tell. El plan agrega una creencia cuya fuente es el agente
   emisor del tell, enrique en este caso. */
+vl(X)[source(Ag)]
   :  Ag \== self
   <- .print("Recibió un tell ",vl(X)," de ", Ag).

/* Igual que el caso anterior pero con una performativa achieve
   en lugar de tell. */
+!ir(X,Y)[source(Ag)] : true
   <- .println("Recibió un achieve ",ir(X,Y)," de ", Ag).

/* Cuando bob pregunta t2(X), la respuesta no está en mis
```

```
   creencias. Por tanto el evento "+?t2(X)" se crea y es
   manejado por el siguiente plan. */
+?t2(X) : vl(Y) <- X = 10 + Y.

/* El siguiente plan es usado para reconfigurar la respuesta a
   un mensaje con performativa askOne. El plan solo es usado
   si el contenido de askOne es "nombreComp". Se puede usar
   un evento de tipo +? para esto, se trata solo de un ejemplo
   de sobrecarga de directivas de comunicación KQML. */
+!kqml_received(Sender, askOne, nombreCompl, ReplyWith) : true
   <- .send(Sender,tell,"Beto Guerra", ReplyWith). // respuesta
```

Lo único a resaltar es el uso de los planes para reconfigurar los actos de habla. Recuerden que la acción interna `.send` se encarga del transporte de los mensajes, pero que estos se ven traducidos a eventos para ser procesados. En este ejemplo, estamos especializando el uso de la performativa `askOne` para que la solicitud no sea procesada por *enrique* como una meta verificable. La variable *ReplyWith* unifica como el identificador *mid* tanto en la solicitud, como en la respuesta.

La definición del agente *enrique* es más larga, así que la analizaremos por partes. Primero, éste agente tiene una sola meta alcanzable $!inicio$ que se resuelve intercambiando mensajes con *beto*. Recuerden que la creencia $receptor(beto)$ fue inicializada vía el archivo principal `mas2j`. Veamos los primeros casos:

```
// Agente enrique en el proyecto comunicacion.mas2j

!inicio.

+!inicio : receptor(A) // Esta creencia viene del mas2j
   <- .println("Enviando tell vl(10)");
      .send(A, tell, vl(10));

      .println("Enviando achieve ir(10,2)");
      .send(A, achieve, ir(10,2));
```

Iremos intercalando la salida en consola para explicar la interacción entre estos dos agentes. Los mensajes de salida correspondientes a los dos Actos de habla anteriores son:

```
[ale] Enviando tell vl(10)
[ale] Enviando achieve ir(10,2)
```

Además de la salida, el estado mental del agente *beto* está cambiando. En este caso se ha agregado $v1(10)_{[source(enrique)]}$ a sus creencias. También ha añadido $\langle +!ir(10,2)_{[source(enrique)]}, \top \rangle$ a su cola de eventos. Continuemos con el resto de los mensajes en el plan inicial del agente *enrique*:

```
      .println("Enviando solicitud síncrona ");
      .send(A, askOne, vl(X), vl(X));
      .println("La respuesta a la solicitud es: ", X, " (debe ser 10)");

      .println("Enviando solicitud asíncrona ");
      .send(A, askOne, vl(_)); // como es asíncrona no tiene 4o argumento
      // la respuesta se recibe vía un evento  +vl(X)

      .println("Preguntando algo que Ana no cree, pero puede responder con +? ");
      .send(A, askOne, t2(_), Ans2);
      .println("La respuesta a la solicitud es: ", Ans2, " (debe ser t2(20))");

      .println("Preguntando por algo que ",A,"  no sabe.");
      .send(A, askOne, t1(_), Ans1);
      .println("La respuesta es: ", Ans1, " (debe ser false)");

      .println("Solicitando valores con askall");
      .send(A, askAll, vl(Y), List1);
      .println("La respuesta es: ", List1, " (debe ser [vl(10),vl(1),vl(2)])");

      .println("Solicitando un askall de t1(X).");
      .send(A, askAll, t1(Y), List2);
```

```
34        .println("La respuesta es: ", List2, " (debe ser []).");
```

Estos actos de habla hacen diferentes solicitudes a *ana* sobre sus creencias. Algunas solicitudes pueden ser resueltas como metas verificables; otras por medio de planes; y otras no pueden responderse. La salida en consola para estos mensajes es como sigue:

```
[ale] Enviando solicitud síncrona
[ana] Recibió un tell vl(10) de ale
[ale] La respuesta a la solicitud es: 10 (debe ser 10)
[ale] Enviando solicitud asíncrona
[ana] Recibió un achieve ir(10,2) de ale
[ale] Preguntando algo que Ana no cree, pero puede responder con +?
[ale] Valor recibido 10 de ana
[ale] La respuesta a la solicitud es: t2(20)[source(ana)] (debe ser t2(20))
[ale] Preguntando por algo que ana  no sabe.
[ale] La respuesta es: false (debe ser false)
[ale] Solicitando valores con askall
[ale] La respuesta es: [vl(10)[source(ana)],vl(1)[source(ana)],vl(2)[source(ana)]] (debe ser [vl(10),vl(1),vl(2)])
[ale] Solicitando un askall de t1(X).
[ale] La respuesta es: [] (debe ser []).
```

Observen que la salida incluye algunas de las respuetas de *beto* a los mensajes iniciales de *enrique*. Por ejemplo, la segunda línea es una respuesta al evento generado por la primer solicitud de *enrique*. El retardo en la respuesta se debe a que *beto* debe procesar el evento, seleccionando un plan aplicable, formando la intención correspondiente y ejecutándola. Lo mismo sucede para la respuseta a la solicitud $!ir(20,2)$ por parte de *enrique*.

Observen también el comportamiento diferente entre las solicitudes síncronas y asíncronas. Las primeras instancian la respuesta en el mensaje mismo; las segundas generan eventos.

Finalmente, el código del agente *enrique* se completa como sigue:

```
36        .println("Preguntado el nombre completo de Beto.");
37        .send(A, askOne, nombreCompl, FN);
38        .println("El nombre completo de ",A," es ",FN);
39
40        // Preguntare a Ana el plan para ir a algún sitio
41        .send(A, askHow, {+!ir(_,_)[source(_)]});
42        .wait(500); // esperar la respuesta 500 ms
43        .print("Planes recibidos:");
44        .list_plans( {+!ir(_,_)[source(_)]} );
45        .print;
46
47        // Otra implementación (no agrega el plan automáticamente a la
48        // librería de planes)
49        .send(A, askHow, {+!ir(_,_)[source(_)]}, ListOfPlans);
50        .print("Planes recibidos: ", ListOfPlans);
51
52        // Envíandole a beto un plan para !hello
53        .plan_label(Plan,hp); // obtiene un Plan a partir de su etiqueta (hp)
54        .println("Enviando un tellhow de: ",Plan);
55        .send(A,tellHow,Plan);
56
57        .println("Pidiéndole a ",A," satisfacer !hola(ale).");
58        .send(A,achieve, hola(ale));
59        .wait(2000);
60
61        .println("Pidiéndole a ",A," satisfacer -!hola(ale).");
62        .send(A,unachieve, hola(ale));
63
64        // Enviar un untellHow a beto
65        .send(A,untellHow,hp).
66
67
68  +vl(X)[source(A)]
69     <- .print("Valor recibido ",X," de ",A).
70
```

```
71 @hp // El plan que será enviado a beto
72 +!hola(Quien)
73    <- .println("Hola ",Quien);
74       .wait(100);
75       !hola(Quien).
```

Esta parte del código incluye ejemplos más elaborados del uso de los Actos de habla en Jason. Primero, la línea 37 explota la especialización de *askOne* que *beto* implementa para poder contestar directamente su nombre. El resto del las solicitudes tienen que ver con intercambio de planes y la solicitud de iniciar una meta alcanzable y dejar de hacerlo. La salida en consola es como sigue:

```
1  [ale] Preguntado el nombre completo de Beto.
2  [ale] El nombre completo de ana es Beto Guerra
3  [ale] Planes recibidos:
4  [ale] @l__4[source(ana)] +!ir(_41X,_42Y)[source(_40Ag)] <- .println("Recibió un achieve ",ir(_41X,_42Y)," de ",_40Ag)
5  [ale]
6  [ale] Planes recibidos: [{ @l__4 +!ir(_44X,_45Y)[source(_43Ag)] <- .println("Recibió un achieve ",ir(_44X,_45Y)," de
7  [ale] Enviando un tellhow de: { @hp +!hola(_43Quien) <- .println("Hola ",_43Quien); .wait(100); !hola(_43Quien) }
8  [ale] Pidiéndole a ana satisfacer !hola(enrique).
9  [ana] Hola enrique
10 ...
11 [ana] Hola enrique
12 [ale] Pidiéndole a ana satisfacer -!hola(enrique).
```

10.9 LECTURAS Y EJERCICIOS SUGERIDOS

Definitivamente, la mejor referencia para Jason es el libro de Bordini, Hübner y Wooldridge [12], en el cual se basa la presentación sobre las creencias de este capítulo. La introducción a Jason se basa en los mini tutoriales incluídos en su distribución [7]. Como se mencionó, la distribución de Jason también incluye una serie de ejemplos y demos que cubren todos los aspectos a revisar de este lenguaje de programación orientado a agentes.

Existen muchas referencias a la Programación Lógica y Prolog que pueden usarse para profundizar este paradigma y explotarlo mejor en Jason. Clocksin y Melish [25] nos ofrecen una introducción concisa y breve a Prolog. Bratko [17] escribió un clásico en IA, que ejemplifica el uso de Prolog en problemas propios del área. Este texto es mucho más extenso que el anterior, pero pude consultarse por temas de interés, por ejemplo búsquedas, heurísticas, sistemas expertos, planeación, aprendizaje, etc. El texto propuesto por Sterling y Shapiro [103] forma parte de una excelente serie dedicada a la Programación Lógica del MIT Press, va en el mismo estilo que el texto anterior. Algunos de los programas $AgentSpeak(L)$ de este capítulo, son adaptaciones de algunos programas lógicos de mi curso de Programación para la IA [8].

La referencia obligada para entender los problemas de la negación y el principio del mundo cerrado, es el artículo de Reiter [91], aunque esa presentación se da en el contexto de la Programación Lógica y las Bases de Datos.

El capítulo 3 del libro compilado por Weiß [110] ofrece una introducción a la comunicación entre agentes. Labrou, Finin y Peng [60]; y Dignum y Greaves [32] ofrecen una revisión de diversos trabajos en comunicación entre agentes.

Ejercicios sugeridos

Ejercicio 10.1. *Modifique que agente piro para que el mismo apague el fuego que ha iniciado... con cierta probabilidad de éxito.*

7 /doc/mini-tutorial
8 http://www.uv.mx/personal/aguerra/pia

Ejercicio 10.2. *Agregue otro agente al proyecto piros, que sea sensible a las acciones de piro.*

Ejercicio 10.3. *¿Cómo se puede lograr que* `enrique` *y* `beto` *se saluden por siempre?*

Ejercicio 10.4. *Convierta el proyecto* `cleaning-robots` *de la distribución de Jason, en un proyecto compatible con el* plug-in *de Eclipse.*

Ejercicio 10.5. *Modifique al agente* `r1` *del proyecto* `cleaning-robots`*, para que una vez que tenga la basura, avise al agente* `r2`*.*

Ejercicio 10.6. *Ilustre con ejemplos los casos de unificación entre variables anotadas, cuando una de ellas es de base y la otra no.*

Ejercicio 10.7. *En el caso del agente* `11`*, cambie el orden de las creencias asociadas a beto, enrique y self como fuentes. ¿Qué sucede con la salida del programa?*

Ejercicio 10.8. *En el caso del agente* `11`*, invierta el orden de los planes ¿Qué sucede en la salida del programa?*

Ejercicio 10.9. *Revisen este material identificando alguna fuerza ilocutoria no definida en Jason, por ejemplo* `promise` *¿Qué tan complicado es agregarla a Jason?*

Ejercicio 10.10. *Intenten correr el ejemplo de ana y ale en una arquitectura distribuida bajo Jade.*

BIBLIOGRAFÍA

[1] JF Allen, JA Hendler y A Tate. *Readings in planning*. Morgan Kaufmann Publishers, 1990 (vid. págs. 140, 157).

[2] KR Apt, HA Blair y A Walker. "Towards a theory of declarative knowledge". En: (1988), págs. 89-148 (vid. pág. 109).

[3] KR Apt y RN Bol. "Logic programming and negation: A survey". En: *The Journal of Logic Programming* 1994.19 (1994), págs. 9-71 (vid. pág. 112).

[4] Aristóteles. "Prior Analytics No 391". En: *Loeb Classical Library*. Cambridge, MA, USA: Harvard University Press, 1960 (vid. pág. 10).

[5] D Barker-Plummer et al. *Tarski's World, Revised and Expanded*. Vol. 169. Lecture Notes in Computer Science/Logic/Philosophy. Stanford, CA, USA: CSLI Publications, 2008 (vid. págs. 64, 92).

[6] F Bellifemine, G Caire y D Greenwood. *Developing Multi-Agent Systems with JADE*. England: John Wiley & Sons, Ltd, 2007 (vid. pág. 183).

[7] M Ben-Ari. *Mathematical Logic for Computer Science*. Third. London, UK: Springer-Verlag, 2012 (vid. pág. 65).

[8] WM Bolstad. *Introduction to Bayesian Statistics*. Hoboken, NJ, USA: Wiley-Interscience, 2004 (vid. pág. 139).

[9] RH Bordini y JF Hübner. "BDI agent programming in agentspeak using Jason". En: *Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI), London, UK, 27-29 June, 2005, Revised Selected and Invited Papers*. Ed. por F Toni y P Torroni. Vol. 3900. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2006, págs. 143-164 (vid. pág. 180).

[10] RH Bordini, JF Hübner y DM Tralamazza. "Using Jason to implement a team of gold miners". En: *CLIMA VII. Computational Logic in Multi-Agent Systems*. Ed. por K Inoue, K Satoh y F Toni. Vol. 4371. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer-Verlag, 2007, págs. 304-313 (vid. pág. 180).

[11] RH Bordini, JF Hübner y R Vieira. "Multi-Agent Programming: Languages, Platforms and Applications". En: ed. por RH Bordini et al. Springer-Verlag, 2005. Cap. Jason and the Golden Fleece of Agent-Oriented Programming (vid. pág. 180).

[12] RH Bordini, JF Hübner y M Wooldridge. *Programming Multi-Agent Systems in Agent-Speak using Jason*. John Wiley & Sons Ltd, 2007 (vid. págs. 159, 162, 163, 178, 180, 208).

[13] RH Bordini y ÁF Moreira. "Proving BDI properties of agent-oriented programming languages". En: *Annals of Mathematics and Artificial Intelligence* 42 (2004), págs. 197-226 (vid. pág. 178).

[14] RH Bordini et al. "The MAS-SOC Approach to Multi-agent Based Simulation". En: *RASTA 2002*. Ed. por G Lindermann y et al. Vol. 2934. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer-Verlag, 2004, págs. 70-91 (vid. pág. 180).

[15] R Brachman y H Levesque. *Knowledge representation and reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2004 (vid. págs. 1, 3, 8).

[16] RJ Brachman. "The basics of knowledge representation and reasoning". En: *Bell Labs Technical Journal* 67.1 (1988), págs. 7-24 (vid. pág. 8).

[17] I Bratko. *Prolog programming for Artificial Intelligence*. Addison-Wesley, 2001 (vid. págs. 67, 142, 157, 190, 208).

[18] I Bratko. *Prolog programming for Artificial Intelligence*. Fourth. Essex, England: Pearson, 2012 (vid. págs. 8, 112, 113, 138).

[19] RA Brooks. *Cambrian Intelligence: the Early History of the New AI*. Cambridge, MA, USA: The MIT Press, 1999 (vid. págs. 15, 22).

[20] BG Buchanan y EH Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. The Addison-Wesley Series in Artificial Intelligence. Reading, MA, USA: Addison-Wesley, 1984 (vid. pág. 138).

[21] A Church. "A Note on the Entscheidungsproblem". En: *Journal of Symbolic Logic* 1 (1936), págs. 40-41 (vid. pág. 82).

[22] K Clark. "Negations as Failure". En: *Logic and Databases*. Ed. por H Gallaire y J Minker. New York, USA: Plenum Press, 1978, págs. 293-322 (vid. págs. 104, 105).

[23] WF Clocksin. *Clause and effect: Prolog programming for the working programmer*. Berlin, Germany: Springer, 1997 (vid. pág. 8).

[24] WF Clocksin y CS Melish. *Programación en Prolog*. Ciencia Informática. Barcelona, España: Editorial Gustavo Gili, S.A., 1987 (vid. pág. 8).

[25] WF Clocksin y CS Melish. *Programming in Prolog, using the ISO standard*. Berlin-Germany: Springer-Verlag, 2003 (vid. págs. 67, 190, 208).

[26] PR Cohen y HJ Levesque. "Speech Acts and Rationality". En: *ACL*. 1985, págs. 49-60 (vid. pág. 171).

[27] MA Covington, D Nute y A Avellino. *Prolog programming in depth*. Prentice Hall International, 1997 (vid. pág. 138).

[28] A Covrigaru y R Lindsay. "Deterministic Autonomous Systems". En: *AI Magazine* Fall (1991), págs. 110-117 (vid. págs. 14, 25, 26).

[29] D van Dalen. *Logic and Structure*. Fifth. Berlin Germany: Springer-Verlag, 2013 (vid. págs. 45, 65).

[30] R Davis, H Shrobe y P Szolovits. "What is a knowledge representation?" En: *AI magazine* 14.1 (1993), pág. 17 (vid. págs. 1, 8).

[31] D Dennett. *The Intentional Stance*. Cambridge, MA, USA: The MIT Press, 1987 (vid. pág. 4).

[32] F Dignum y M Greaves. *Issues in Agent Communication*. Vol. 1916. Lecture Notes in Computer Science. Springer Verlag, 2000 (vid. pág. 208).

[33] M d'Inverno et al. "A formal specification of dMARS". En: *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*. Ed. por M Singh, A Rao y M Wooldridge. Vol. 1365. Lecture Notes in Artificial Intelligence. Berlin-Germany: Springer Verlag, 1998, págs. 155-176 (vid. págs. 158, 170, 178).

[34] M d'Inverno y M Luck. "Engineering AgentSpeak(L): A Formal Computational Model". En: *Journal of Logic and Computation* 8.3 (1998), págs. 233-260 (vid. pág. 158).

[35] M d'Inverno y M Luck. *Understanding Agent Systems*. Second. Berlin, Germany New York: Springer, 2004 (vid. pág. 158).

[36] A Doxiadis. *El tío Petros y la conjetura de Goldbach*. Madrid, España: Ediciones B, 2000 (vid. pág. 30).

[37] LD Erman et al. "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty". En: *ACM Computing Surveys (CSUR)* 12.2 (1980), págs. 213-253 (vid. pág. 138).

[38] O Etzioni. "Intelligence without Robots". En: *AI Magazine* 14.4 (1993) (vid. pág. 15).

[39] R Fagin et al. *Reasoning about Knowledge*. Cambridge, MA, USA: The MIT Press, 1995 (vid. pág. 20).

[40] RE Fikes y NJ Nilsson. "STRIPS: a new approach to the application of theorem proving to problem solving". En: *Artificial Intelligence* 2 (1971), págs. 189-208 (vid. págs. 140, 157).

[41] L Foner. *What's an agent, anyway? A sociological case study*. Inf. téc. Agents Memo 93-01. Cambridge, MA, USA: MIT Media Lab, 1993 (vid. pág. 13).

[42] S Franklin y A Graesser. "Is it an agent, or just a program?" En: *Intelligent Agents III*. Ed. por JP Muller, M Wooldridge y NR Jennings. Lecture Notes in Artificial Intelligence 1193. Berlin, Germany: Springer-Verlag, 1997, págs. 21-36 (vid. pág. 10).

[43] MR Genesereth y NJ Nilsson. *Logical Foundations for Artificial Intelligence*. Palo Alto, CA, USA: Morgan Kauffman Publishers, Inc., 1987 (vid. págs. 8, 65, 68, 91).

[44] MP Georgeff y AL Lansky. "Procedural Knowledge". En: *Proceedings of the IEEE* 74.10 (1986), págs. 1383-1398 (vid. pág. 158).

[45] MP Georgeff y AS Rao. "A profile of the Australian Artificial Intelligence Institute". En: *IEEE Expert* 11.6 (1996), págs. 89-92 (vid. pág. 158).

[46] M Georgeff y F Ingrad. "Decision-making in an Embedded Reasoning System". En: *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*. Detroit, MI., USA, 1989, págs. 972-978 (vid. pág. 158).

[47] M Georgeff y A Lansky. "Reactive reasoning and planning". En: *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*. Seattle, WA., USA, 1987, págs. 667-682 (vid. pág. 158).

[48] M Ghallab, D Nau y P Traverso. *Automated planning: theory and practice*. San Franciso, CA, USA: Morgan Kaufmann, 2004 (vid. págs. 140, 142, 157).

[49] A Guerra-Hernández, JM Castro-Manzano y A El-Fallah-Seghrouchni. "Toward an AgentSpeak(L) Theory of Commitment and Intentional Learning". En: *MICAI 2008*. Ed. por A Gelbuch y EF Morales. Vol. 5317. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer-Verlag, 2008, págs. 848-858 (vid. págs. 176, 178).

[50] A Guerra-Hernández, JM Castro-Manzano y A El-Fallah-Seghrouchni. "CTL AgentSpeak(L): a Specification Language for Agent Programs". En: *Journal of Algorithms* 64 (2009), págs. 31-40 (vid. págs. 159, 167, 176, 178).

[51] N Gupta y DS Nau. "On the complexity of blocks world planning". En: *Artificial Intelligence* 56.2-3 (1992), págs. 223 -254 (vid. pág. 140).

[52] F van Harmelen, V Lifschitz y B Porter. *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. Amsterdam, The Netherlands: Elsevier, 2008 (vid. pág. 8).

[53] M Huns y M Singh, eds. *Readings in Agents*. San Mateo, CA, USA: Morgan Kauffman Publisher, 1998 (vid. pág. 27).

[54] M Huth y M Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, UK: Cambridge University Press, 2004 (vid. págs. 28, 45, 65, 91).

[55] FF Ingrand y MP Georgeff. "An Architecture for Real-Time Reasoning and System Control". En: *IEEE Expert* 7.6 (1992), págs. 34-44 (vid. pág. 158).

[56] D Kayser. *La représentation des connaissances*. Collection Informatique. Paris, France: Editions Hermes, 1997 (vid. págs. 1, 8).

[57] K Knight. "Unification: a multidisciplinary survey". En: *ACM Comput. Surv.* 21.1 (1989), págs. 93-124. ISSN: 0360-0300. DOI: http://doi.acm.org/10.1145/62029.62030 (vid. págs. 92, 164, 165).

[58] RA Kowalski. "Predicate Logic as a Programming Language". En: *Information Processing*. Ed. por JL Rosenfeld. North-Netherlands, 1974, págs. 569-574 (vid. pág. 92).

[59] RA Kowalski y D Kuehner. "Linear Resolution with Selection Function". En: *Artificial Intelligence* 2.3/4 (1971), págs. 227-260 (vid. págs. 81, 92).

[60] Y Labrou, T Finin e Y Peng. "The current landscape of Agent Communication Languages". En: *Intelligent Systems, IEEE Computer Society* 14.2 (1999) (vid. pág. 208).

[61] Y Labrou y TW Finin. "A Semantics Approach for KQML - A General Purpose Communication Language for Software Agents". En: *CIKM '94 Proceedings of the third international conference on Information and knowledge management*. New York, NY, USA: ACM, 1994, págs. 447-455 (vid. pág. 171).

[62] J Lee et al. "UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications". En: *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*. Houston, Texas: American Institute of Aeronautics y Astronautics, 1994, págs. 842-849 (vid. pág. 158).

[63] HJ Levesque. *Thinking as Computation*. Cambridge, MA, USA: The MIT Press, 2012 (vid. pág. 8).

[64] D Levitin. *This is Your Brain on Music: The Science of a Human Obsesion*. New York, NY, USA: A Plume Book, 2006 (vid. pág. 21).

[65] A Ligeza. *Logical foundations for rule-based systems*. Vol. 11. Springer, 2006 (vid. pág. 139).

[66] D Lightfoot. *Formal Specification Using Z*. Macmillan Computer Science Series. London, UK: The Macmillan Press LTD, 1991 (vid. pág. 158).

[67] P Maes. *How to Do the Right Thing*. Inf. téc. Cambridge, MA, USA: MIT Media Lab, 1989 (vid. pág. 13).

[68] E Mares. "Relevance Logic". En: *The Stanford Encyclopedia of Philosophy (Summer 2012 Edition)*, (2012). Ed. por EN Zalta. URL: http://plato.stanford.edu/archives/sum2012/entries/logic-relevance/ (vid. pág. 41).

[69] J McCarthy. "Programs with common sense". En: *Proceedings of the Symposium on the Mechanization of Thought Processes*. Teddington, England, 1959 (vid. pág. 25).

[70] J McCarthy. *Ascribing Mental Qualities to Machines*. Inf. téc. Stanford, CA, USA: Computer Science Department, Stanford University, 1979 (vid. pág. 4).

[71] J McCarthy y PJ Hayes. "Some philosophical problems from the standpoint of artificial intelligence". En: *Readings in artificial intelligence* (1969), págs. 431-450 (vid. pág. 142).

[72] W Muller-Freienfels. "Agency". En: *Encyclopedia Britannica*. Internet version. Encyclopedia Britannica, Inc., 1999 (vid. pág. 10).

[73] J Negrete-Martínez, PP González-Pérez y A Guerra-Hernández. *Pericia Artificial: Una aproximación incremental a los Sistemas Expertos*. Xalapa, Ver., México: Universidad Veracruzana, 1996 (vid. págs. 9, 115, 138).

[74] A Newell. "The Knowledge Level". En: *AI Magazine* 2 (1981), págs. 1-20 (vid. págs. 1, 8, 9, 24, 25).

[75] A Newell. *Unified Theories of Cognition*. Cambridge, MA, USA: Harvard University Press, 1990 (vid. pág. 14).

[76] A Newell y HA Simon. "Computer Science As Empirical Inquiry: Symbols and Search". En: *Commun. ACM* 19.3 (1976), págs. 113-126. ISSN: 0001-0782. DOI: 10.1145/360018.360022. URL: http://doi.acm.org/10.1145/360018.360022 (vid. págs. 9, 23, 30).

[77] U Nilsson y J Maluszynski. *Logic, Programming and Prolog*. 2nd. John Wiley & Sons Ltd, 2000 (vid. págs. 74, 91, 112).

[78] P Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Burlington, MA, USA: Morgan Kauffman Publishers, 1992 (vid. pág. 9).

[79] A Omicini, A Ricci y M Viroli. "Artifacts in the A&A meta-model for multi-agent systems". En: *Autonomous Agents and Multi-Agent Systems* 17.3 (2008), págs. 432-456 (vid. pág. 15).

[80] G Ortiz-Hernández et al. "A Namespace Approach for Modularity in BDI Programming Languages". En: *Engineering Multi-Agent Systems, 4th International Workshop, EMAS 2016. Singapore, Singapore, May 9–10. Revised, Selected, and Invited Papers*. Ed. por M Baldoni et al. Vol. 10093. Lecture Notes in Artificial Intelligence. Berlin, Germany: Springer Verlag, 2016, págs. 117-135 (vid. pág. 180).

[81] J Pearl, M Glymour y NP Jewell. *Causal inference in statistics*. Wiley, 2016 (vid. pág. 139).

[82] FJ Pelletier. "A Brief History of Natural Deduction". En: *History and Philosophy of Logic* 20 (1999), págs. 1-31 (vid. pág. 45).

[83] CR Perrault y JF Allen. "A Plan-Based Analysis of Indirect Speech Acts". En: *American Journal of Computational Linguistics* 6.3-4 (1980), págs. 167-182 (vid. pág. 171).

[84] GD Plotkin. "A note on inductive generalization". En: *Machine Intelligence* 5 (1970), págs. 153-163 (vid. pág. 88).

[85] GD Plotkin. "A further note on inductive generalization". En: *Machine Intelligence* 6 (1971), págs. 104-224 (vid. pág. 88).

[86] GD Plotkin. *A Structural Approach to Operational Semantics*. Inf. téc. DAIMI FN-19. University of Aarhus, 1981. URL: citeseer.ist.psu.edu/article/plotkin81structural.html (vid. págs. 163, 178).

[87] D Poole. "Logic, knowledge representation, and Bayesian decision theory". En: *Computational Logic—CL 2000*. Springer, 2000, págs. 70-86 (vid. pág. 139).

[88] AS Rao y MP Georgeff. "Decision Procedures for BDI Logics". En: *Journal of Logic and Computation* 8.3 (1998), págs. 293-342 (vid. pág. 158).

[89] A Rao. "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language". En: *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Ed. por R van Hoe. Eindhoven, The Netherlands, 1996 (vid. págs. 158, 178).

[90] JA Rawls. *A Theory of Justice*. New York, NY, USA: Oxford University Press, 1973 (vid. pág. 14).

[91] R Reiter. "Logic and Databases". En: ed. por H Gallaire y J Minker. Springer, 1978. Cap. On closed world data bases, págs. 55-76 (vid. pág. 208).

[92] A Ricci, M Piunti y M Viroli. "Environment programming in multi-agent systems: an artifact-based perspective". En: *Autonomous Agents and Multi-Agent Systems* 23.2 (2011), págs. 158-192. DOI: 10.1007/s10458-010-9140-7 (vid. pág. 15).

[93] JA Robinson. "A Machine-Oriented Logic based on the Resolution Principle". En: *Journal of the ACM* 12.1 (1965), págs. 23-41 (vid. págs. 81, 92).

[94] S Russell y P Norvig. *Artificial Intelligence: A Modern Approach*. Fourth, global. New York, NY, USA: Pearson, 2022 (vid. pág. 1).

[95] SJ Russell y P Norvig. *Artificial Intelligence: A Modern Approach*. Third. Prentice Hall Series in Artificial Intelligence. USA: Prentice Hall, 2009 (vid. págs. 8, 10, 16, 18, 22, 26, 27, 45, 65, 66, 91).

[96] SJ Russell y D Subramanian. "Provably Bounded-Optimal Agents". En: *Journal of Artificial Intelligence Research* 2 (1995), págs. 575-609 (vid. págs. 11, 26, 27).

[97] RB Scherl y HJ Levesque. "Knowledge, action, and the frame problem". En: *Artificial Intelligence* 144.1 (2003), págs. 1-39. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/S0004-3702(02)00365-X. URL: http://www.sciencedirect.com/science/article/pii/S000437020200365X (vid. pág. 142).

[98] Y Shoham. "Agent-Oriented Programming". En: *Artificial Intelligence* 60 (1993), págs. 51-92 (vid. pág. 159).

[99] Y Shoham. *Artificial Intelligence Techniques in Prolog*. San Francisco, CA, USA: Morgan Kaufmann, 1994 (vid. pág. 8).

[100] Y Shoham. "Logics of Intention and the Database Perspective". En: *Journal of Philosophical Logic* 38.6 (2009) (vid. pág. 27).

[101] HA Simon. *Models of Bounded Rationality*. Vol. 2. Cambridge, MA, USA: The MIT Press, 1982 (vid. págs. 26, 27).

[102] MP Singh. *Multiagent Systems: A theoretical framework for intentions, know-how, and communication*. Vol. 799. Lecture Notes in Computer Sciences. Berlin, Germany: Springer Verlag, 1994 (vid. pág. 175).

[103] L Sterling y E Shapiro. *The Art of Prolog*. Cambridge, MA, USA: The MIT Press, 1999 (vid. págs. 8, 190, 208).

[104] LE Sucar. *Probabilistic Graphical Models: Principles and Applicatons*. Advances in Computer Vision and Pattern Recognition. London, UK: Springer-Verlag London, 2015 (vid. pág. 139).

[105] H Torkzyner. *Magritte: Ideas and Images*. New York, NY, USA: Harry N. Abrams, Inc., 1979 (vid. pág. 1).

[106] AM Turing. "On the Computable Numbers, with Applications to the Entscheidungsproblem". En: *Proceedints of the London Mathematical Society*. Vol. 42. series 2. 1936, págs. 230-265 (vid. pág. 82).

[107] F Varela. *Invitation aux Science Cognitives*. Paris, France.: Editions du Seuil, 1989 (vid. pág. 1).

[108] R Vieira et al. "On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language". En: *Journal of Artificial Intelligence Research* 29 (2007), págs. 221-267 (vid. págs. 171, 180).

[109] J Von Neumann y O Morgesten. *Theory of Games and Economic Behavior*. Princeton, NJ, USA: Princeton University Press, 1947 (vid. págs. 26, 27).

[110] G Weiß. *Multiagent Systems*. Second. Intelligent Robotics and Autonomous Agents. Cambridge, MA, USA: The MIT Press, 2013 (vid. págs. 27, 208).

[111] M Wooldridge. *Reasoning about Rational Agents*. Cambridge, MA, USA: The MIT Press, 2000 (vid. pág. 26).

[112] M Wooldridge. *An Introduction to MultiAgent Systems*. 2nd. West Sussex, England: John Wiley & Sons, LTD, 2009 (vid. págs. 8, 10, 11, 19, 26, 27).

[113] M Wooldridge y N Jennings. "Intelligent Agents: Theory and practice". En: *The Knowledge Engineering Review* 10.2 (1995), págs. 115-152 (vid. págs. 10, 13, 14).