UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Exposición de los fundamentos del sistema de inferencia inductiva CIGOL

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO(A) EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A N:

VÍCTOR DIDIER GUTIÉRREZ BASULTO
YAZMÍN ANGÉLICA IBÁÑEZ GARCÍA

FACULTAD DE CIENCIAS
UNAM

DIRECTOR DE TESIS: Dr. David Arturo Rosenblueth Laguette.

2005

UNIVERSIDAD NACIONAL AVANZADA DE MEXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito: Exposición de los fundamentos del sistema de inferencia inductiva CIGOL

realizado por Víctor Didier Gutiérrez Basulto
Yazmín Angélica Ibáñez García

con número de cuenta 09711081-5
09711059-2
, quien cubrió los créditos de la carrera de: Lic. en Ciencias Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director Propietario	Dr. David Arturo Rosenblueth Laguette
Propietario	Dr. Francisco Hernández Quiroz
Propietario	Dr. Julio César Peralta Estrada
Suplente	Lic. Elías Samra Hassan
Suplente	Mtro. Miguel Carrillo Barajas

Consejo Departamental de Matemáticas

Dr. Francisco Hernández Quiroz

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL DE MATEMÁTICAS

A nuestros padres y hermanos

Agradecimientos

Agradecemos al Dr. David Rosenblueth por el tiempo y esfuerzo dedicados a la dirección de esta tesis. Asimismo, por sus invaluables consejos que enriquecieron este trabajo y que nos permitieron conocer las dificultades y satisfacciones de la investigación.

Durante el tiempo que trabajamos bajo la dirección del Dr. Rosenblueth logramos un considerable crecimiento académico, que no hubiese sido posible sin su paciencia y dedicación.

Agradecemos al Dr. Stephen Muggleton por proporcionarnos el código de su sistema de inferencia inductiva CIGOL, que permitió llevar a cabo esta tesis.

Agradecemos a los doctores Francisco Hernández Quiroz, Julio Peralta Estrada, al maestro Miguel Carrillo Barajas y al licenciado Elías Samra Hassan, sus útiles comentarios y valiosas sugerencias que permitieron el mejoramiento de esta tesis. Especialmente queremos agradecer el sincero interés mostrado

hacia nuestro trabajo (i.e., dedicaron tiempo y esfuerzo en leer nuestro trabajo).

Agradecemos al Instituto de Investigación en Matemáticas Aplicadas y en Sistemas (IIMAS), y a la UNAM, quienes nos otorgaron el espacio, y los recursos materiales y humanos necesarios para llevar a buen término este proyecto de tesis.

Por último, pero no menos importante, agradecemos profundamente a nuestras familias por el apoyo y la paciencia que nos han brindado durante el trayecto de nuestra vida.

A nuestros padres agradecemos los ejemplos de perseverancia, dedicación, esfuerzo y disciplina dados a lo largo de nuestra vida. Así como por los valores inculcados; y el cariño y amor con el que crecimos.

A nuestros hermanos, por la comprensión, el cariño y respeto con los que hemos crecido juntos.

Yazmín A. Ibáñez García
Víctor D. Gutiérrez Basulto
junio, 2005

Índice general

Índice de figuras

Capítulo 1

Introducción

Uno de los objetivos de esta tesis es dar un panorama general de la programación lógica inductiva. En particular presentar la resolución inversa, considerada una de las primeras técnicas de la programación lógica inductiva. Para entender esta técnica se describe la implementación del sistema de inferencia inductiva CIGOL, desarrollado por Stephen Muggleton [MB88a].

En este capítulo se presenta una introducción a la programación lógica inductiva. En dicha introducción se establece la relación de la programación lógica inductiva con la programación lógica y el aprendizaje de máquina. Después, se presenta una descripción del trabajo desarrollado. Posteriormente, se elabora un resumen del contenido de esta tesis. Finalmente, se aclaran la notación y el uso de algunas palabras dentro de este trabajo.

1.1. Programación lógica inductiva

La programación lógica inductiva toma elementos tanto del aprendizaje de máquina como de la programación lógica. Del aprendizaje de máquina inductivo toma su objetivo: desarrollar herramientas y técnicas que permitan inducir hipótesis a partir de observaciones y sintetizar nuevo conocimiento obtenido por la experiencia. De la programación lógica toma el lenguaje que permite representar las hipótesis y las observaciones. El uso de este lenguaje le ayuda a superar dos de las principales limitaciones de las técnicas clásicas de aprendizaje de máquina:

1. el uso de un formalismo limitado para representar el conocimiento (esencialmente lógica proposicional),

2. dificultades al usar conocimiento previo sustancial durante el proceso de aprendizaje.

Superar la primera limitación es importante porque el conocimiento en muchos campos de estudio puede representarse solamente dentro de la lógica de primer orden o un formalismo equivalente, y no dentro de la lógica proposicional. Un problema en que dicha limitación se hace evidente es en el campo de síntesis de programas lógicos a partir de ejemplos. Muchos programas lógicos no pueden definirse al usar solamente lógica proposicional. Además, la lógica ofrece un formalismo elegante para representar el conocimiento y, por lo tanto, incorporar dicho conocimiento dentro de la tarea de inducción.

La programación lógica inductiva extiende la teoría de la programación lógica al investigar la inducción en lugar de deducción como regla básica

de inferencia. Antes del advenimiento de la programación lógica inductiva, la teoría de la programación lógica enfatizaba la inferencia *deductiva* a partir de fórmulas lógicas provistas por el usuario. La teoría de la programación lógica inductiva, en cambio, describe la inferencia *inductiva* de programas lógicos a partir de instancias y conocimiento previo. De este modo, la programación lógica inductiva puede contribuir a la práctica de la programación lógica, al proveer herramientas que ayuden a los programadores lógicos a desarrollar programas.

La programación lógica inductiva se distingue de otras áreas de investigación de inferencia inductiva como inducción de gramáticas e inducción de autómatas finitos por su énfasis en el uso de una representación universal. Las representaciones universales prometen un rango mayor de aplicaciones. Podría decirse que los programas lógicos son más fáciles de manipular por un algoritmo de aprendizaje de máquina que otras representaciones universales, como máquinas de Turing o programas en LISP. Esto se debe al hecho de que en la lógica clausal se pueden hacer cambios a un programa al agregar o quitar ya sea cláusulas completas o literales dentro de una cláusula.

1.2. Trabajo desarrollado

El trabajo desarrollado en esta tesis se puede dividir de la siguiente manera:

Se modificó el código de CIGOL proporcionado por Stephen Muggleton que corría bajo Quintus Prolog 2.4.2 para que corriera bajo Sicstus Prolog

3.11.1. Esto se hizo ya que la versión referida de Quintus Prolog no está disponible actualmente y tal como se encontraba el código no corría bajo versiones actuales ni de Quintus ni de Sicstus Prolog.

Una vez que se realizó esta modificación, se llevó a cabo el análisis del código de CIGOL que dio como resultado la formulación de algunos algoritmos que aparecen en el capítulo 5, los cuales se basan en resultados teóricos que aparecen en dicho capítulo.

En el capítulo 4 se presenta un lema relacionado con la formulación de los operadores de resolución inversa.

Además, se realizó una recopilación de resultados principales de la programación lógica y de la programación lógica inductiva.

1.3. Breve descripción del contenido

El capítulo 2 presenta los fundamentos de la programación lógica que serán necesarios para los propósitos de esta tesis.

El capítulo 3 comienza con una descripción de la programación lógica inductiva y continúa con el planteamiento del problema que intenta resolver esta disciplina. Después se describen la técnica de resolución inversa y los operadores que la conforman. Finalmente se enumeran algunas aplicaciones de la programación lógica inductiva.

El capítulo 4 contiene los algoritmos de los operadores que invierten resolución, implementados en CIGOL. Además, se ejemplifica el funcionamiento del sistema.

En el capítulo 5 se describe de manera detallada la implementación de CIGOL y se establece su relación con algunos resultados teóricos.

El capítulo 6 presenta algunas notas finales.

1.4. Notación

Los autores de esta tesis utilizan subrayado sencillo al escribir las palabras en inglés.

Dentro de este trabajo, se usan las palabras *software*, *implementar* e *implementación*, todas ellas aceptadas por la Real Academia de la lengua Española (RAE).

Capítulo 2

Programación lógica

En este capítulo se establece un panorama general de la programación lógica. Se da una breve introducción al concepto de programación lógica y sus antecedentes, se define la sintaxis y se establece la semántica del lenguaje de la programación lógica.

El contenido de este capítulo se basa en [Apt97] [Hog90] [Llo87]. Se supondrá que el lector está familiarizado con algunos conceptos básicos de la lógica de primer orden.

El objetivo de este capítulo no es profundizar en el estudio de la programación lógica, sino establecer un conjunto básico de conceptos que se utilizarán en lo sucesivo.

2.1. Antecedentes

Como primera aproximación, la *programación lógica* es un formalismo computacional que combina dos principios:

- La *lógica* para representar el conocimento.

- La *inferencia* para manipular la representación del conocimiento.

En términos de la solución de problemas, el primer principio trata con la representación de suposiciones y conclusiones, mientras que el segundo trata con el establecimiento de conexiones lógicas entre las suposiciones y conclusiones. De manera general, el objetivo es *inferir* una conclusión a partir de las suposiciones dadas y hacerlo de una manera computacionalmente viable.

En términos concretos, el formalismo estándar de la programación lógica usa un fragmento particular —*lógica clausal*— de la lógica de predicados de primer orden como lenguaje para la representación del conocimiento, y una regla de inferencia particular —*resolución*— como mecanismo de manipulación de la representación del conocimiento.

La programación lógica representa un punto de convergencia en las disciplinas de lógica, demostración automática de teoremas y ciencia de la computación. En particular, la lógica contribuyó con sistemas simbólicos que gozaban de economía de sintaxis y para los cuales las nociones de *derivar* e *interpretar* enunciados podían expresarse independientemente una de la otra, y aún así relacionarse. Esto se debe en particular a Gottlob Frege, cuyo trabajo introdujo lo que ahora se considera como la formulación estándar de la lógica de primer orden, y a Alfred Tarski, quien aclaró confusiones semánticas entre verdad y demostración.

La teoría de la *lógica clausal*, y un teorema en particular —teorema de Herbrand— se debe a Jacques Herbrand; el descubrimiento de la resolución —un paso fundamental para la automatización en la demostración de teoremas de la lógica clausal— se debe a J. Alan Robinson. La aplicación de estos conocimientos en la programación, ciencia de la computación e inteligencia artificial fue iniciada por Carl Hewitt, Alain Colmerauer y Robert Kowalski.

El formalismo adecuado para la programación y representación del conocimiento se introdujo en 1974 por Kowalski [Kow74]. La diferencia principal de la programación lógica con los demostradores automáticos de teoremas basados en resolución, es que la primera puede usarse no sólo para demostrar, sino también para calcular. De hecho, la programación lógica ofrece un nuevo paradigma de programación el cual originalmente se implementó en Prolog, un lenguaje de programación introducido a principios de los años setenta por un grupo dirigido por Colmerauer. Desde entonces, el paradigma de la programación lógica ha inspirado el diseño de nuevos lenguajes de programación que se han usado exitosamente para abordar diversos problemas computacionalmente complejos.

2.2. Sintaxis del lenguaje de la programación lógica

A continuación se define un tipo de fórmulas especiales dentro del lenguaje de primer orden, dado un "alfabeto". Estas fórmulas constituyen la llamada forma clausal. La forma clausal es una de las muchas *formas normales* de la

lógica de primer orden y es el sublenguaje en el cual los programas lógicos se escriben convencionalmente. La característica que tienen en común todas las formas normales es hacer la estructura de los enunciados *regular* [Hog90]. Dada cualquier fórmula de la lógica de primer orden A, se puede construir una fórmula B en forma clausal. Dicha forma clausal no siempre es equivalente a la fórmula A. Sin embargo, se puede decir que B es inconsistente si y solo si A es inconsistente [Gal86].

Definición Un *alfabeto* consta de las siguientes clases disjuntas de símbolos:

- un conjunto infinito numerable de variables
- un conjunto de símbolos de función
- un conjunto de símbolos de predicado
- un conjunto de conectivos $(\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow\})$
- un conjunto de cuantificadores

Definición Un *término* se define inductivamente como sigue:

- Una variable es un término.
- Si f es un símbolo de función de aridad $n \geq 0$ y $t_1, \ldots, t_n$ son términos, entonces $f(t_1, \ldots, t_n)$ es un término.

Un símbolo de función de aridad 0 se llama *constante* y en ese caso se omiten los paréntesis. Un término que no contiene variables se llama *aterrizado.*

En este trabajo, las variables se denotan por las letras minúsculas u, v, w, x, y, z; las constantes, por las letras minúsculas a, b, c, d, e; las letras minúsculas para denotar los símbolos de función de aridad positiva son f, g, h, k. Todas las letras que se usan para denotar términos pueden tener subíndices.

Definición Si p es un símbolo de predicado de aridad $n \geq 0$ y $t_1, \ldots, t_n$ son términos, entonces, $p(t_1, \ldots, t_n)$ es un *átomo*. Si $n = 0$ se omiten los paréntesis.

Definición 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.

Definición Una *cláusula* es un conjunto de literales

$$\{A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m\}$$

donde cada A_i, B_j es un átomo, $0 \leq i \leq n$ y $0 \leq j \leq m$. Si $n = 0$ y $m = 0$ la cláusula se llama *vacía* y se denota como $\square$.

Por conveniencia se adoptará la notación de Kowalski para cláusulas

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

Definición Una *cláusula definida* es una cláusula que contiene exactamente una literal positiva A.

Este tipo de cláusula se denota como

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

A se llama la *cabeza* de la cláusula y $\{\neg B_1, \ldots, \neg B_n\}$ el *cuerpo*. La cabeza de una cláusula C también puede denotarse como C_{cabeza} y el cuerpo como C_{cuerpo}.

Definición Una *cláusula unitaria* es una cláusula de la forma

$$A \leftarrow$$

esto es, una cláusula definida con el cuerpo vacío.

Definición Una *meta* es una cláusula de la forma

$$\leftarrow B_1, \ldots, B_n \qquad n \geq 0$$

es decir, una cláusula que no tiene cabeza (no tiene literales positivas). Cada B_i, $1 \leq i \leq n$, se denomina *submeta*. Si $n = 0$ se denota como $\square$.

Definición Un *programa lógico* es un conjunto finito de cláusulas definidas.

En lo sucesivo se hará referencia a un programa lógico de esta manera o simplemente como *programa*.

Definición El *lugar* dentro de un término o literal se denota por una tupla y se define recursivamente como sigue. El término en el lugar $\langle a_1 \rangle$ dentro de $f(t_1, \ldots, t_n)$ es t_{a_1}. El término en el lugar $\langle a_1, \ldots, a_m \rangle$, $m > 1$, dentro de $f(t_1, \ldots, t_n)$ es el término en el lugar $\langle a_2, \ldots, a_m \rangle$ dentro de t_{a_1}. La definición de lugar puede extenderse a *lugar dentro de una cláusula*, si se supone que se tiene un orden fijo de las literales de una cláusula.

Se dice que E es una *expresión*, si es un término, una literal o una cláusula. Con $vars(E)$, se denota el conjunto de las variables que ocurren en E.

2.3. Unificación

Dentro de la programación lógica, las variables representan valores desconocidos. Los valores asignados a las variables son términos. Estos valores son asignados por medio de ciertas sustituciones, llamadas "unificadores". Al proceso de calcular estos unificadores se llama *unificación*. Por tanto, la unificación forma un mecanismo básico mediante el cual un procedimiento de prueba utiliza los programas lógicos para calcular. En esta sección se abordarán algunos conceptos básicos para entender este mecanismo.

Definición Una *sustitución* θ es un mapeo finito de variables a términos, que asigna a cada variable x en su dominio, que se denota como $dom(\theta)$, un término t diferente de x. Se denota como

$$\{x_1/t_1, \ldots, x_n/t_n\} \qquad n \geq 0$$

donde

- $x_1, \ldots, x_n$ son variables distintas,
- $t_1, \ldots, t_n$ son términos,
- $1 \leq i \leq n,\ x_i \neq t_i$.

Si $n = 0$ la sustitución se llama *sustitución identidad* y se denota como ε.

Considérese una sustitución $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$. Si $t_1, \ldots, t_n$ son todos aterrizados, θ se llama *sustitución aterrizante*.

La *aplicación* de una sustitución $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ a una expresión E, denotada como $E\theta$, es la expresión obtenida al remplazar simultáneamente cada ocurrencia de x_i en E por t_i.

De manera informal, puede decirse que las variables $x_1, \ldots, x_n$ se ligan a $t_1, \ldots, t_n$ respectivamente.

Definición Sea E una expresión. Un *renombramiento* de E es una sustitución de variables a variables $\{x_1/y_1, \ldots, x_n/y_n\}$ tal que $\{x_1, \ldots, x_n\} \subseteq vars(E)$, con $y_i \neq y_j$ si $i \neq j$ e $i, j \in [1, n]$ y $(vars(E) \setminus \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$.

Definición Sean E una expresión y θ una sustitución. $E\theta$ se llama una *instancia* de E. Se dice que una instancia es *aterrizada* si no contiene variables. Si θ es un renombramiento, entonces se dice que es una *variante* de E.

Sea E una expresión. Se denotará como $aterrizados(E)$ al conjunto de todas las instancias aterrizadas de E. Asimismo, si S es un conjunto de expresiones, $aterrizados(S)$ es el conjunto de todas las instancias aterrizadas de todos los elementos de S.

Definición La *composición* de dos sustituciones $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ y $\eta = \{v_1/t_1, \ldots, v_n/t_n\}$, denotada como $\theta\eta$, es la sustitución obtenida del conjunto

$$\{u_1/s_1\eta, \ldots, u_m/s_m\eta, v_1/t_1, \ldots, v_n/t_n\}$$

eliminando cualquier $u_i/s_i\eta$ para el cual $u_i = s_i\eta$ y eliminando cualquier v_j/t_j para el cual $v_j \in \{u_1, \ldots, u_m\}$.

Ejemplo Sean $\theta = \{x/f(y), y/z\}$ y $\eta = \{x/a, y/b, z/y\}$, entonces,

$$\theta\eta = \{x/f(b), z/y\}.$$

◁

Informalmente, la unificación es el proceso de hacer términos idénticos mediante ciertas sustituciones. Este trabajo se limitará a "unificadores más generales". La presentación formal de la unificación comienza con la siguiente definición necesaria para establecer la noción de unificador más general.

Definición Sean θ y τ sustituciones. Se dice que θ es *más general* que τ si para alguna sustitución η se tiene que $\tau = \theta\eta$.

Definición Sean θ una sustitución, y E_1 y E_2 expresiones que no tengan variables en común. Si $E_1\theta = E_2$ y $dom(\theta) \subseteq vars(E_1)$, se dice que θ es la *diferencia* o *θ-diferencia* de E_1 y E_2 y se denota como $E_1 -_\theta E_2$.

El siguiente es el concepto clave de esta sección.

Definición Sean θ una sustitución y E_1 y E_2 expresiones.

- θ es un *unificador* de E_1 y E_2 si $E_1\theta = E_2\theta$. Si existe un unificador de E_1 y E_2, se dice que E_1 y E_2 son *unificables*.
- Se dice que θ es un *unificador más general* (umg) de E_1 y E_2, $umg(E_1, E_2)$, si es un unificador de E_1 y E_2 y es más general que todos los unificadores de E_1 y E_2.

Intuitivamente, un umg es una sustitución que hace dos expresiones iguales, pero que lo hace de "la forma más general", sin instanciar variables de manera innecesaria. Por lo tanto, θ es un umg si para cada unificador η existe una sustitución γ tal que $\eta = \theta\gamma$.

2.4. Semántica del lenguaje de la programación lógica

La semántica declarativa de un programa lógico está dada por la semántica usual de las fórmulas en lógica de primer orden. Esta sección se enfocará en la clase de las "interpretaciones de Herbrand". Se introducirá la noción de "modelo mínimo de Herbrand" de un programa. Este modelo en particular desempeña un papel central dentro de la teoría. Se establecerá que el modelo mínimo de Herbrand es precisamente el conjunto de átomos aterrizados que son "consecuencia lógica" de un programa.

Antes de dar las definiciones principales de esta parte, será conveniente entender la motivación de éstas. Para poder hablar acerca de la verdad o falsedad de una fórmula, en nuestro caso, de una cláusula o de un programa, es necesario dar significado a los símbolos utilizados para describirlos. El significado de algunos de ellos puede ser fijo, pero el significado dado a las constantes, los símbolos de función y de predicado pueden variar.

Una *preinterpretación* [Llo87] para un lenguaje de primer orden $\mathcal{L}$ consta de un conjunto no vacío D, llamado el *dominio* de la preinterpretacion, una asignación de un elemento en D a cada constante en el alfabeto asociado a $\mathcal{L}$ y una asignación para cada símbolo de función en dicho alfabeto de un mapeo que va de D^n a D. Una *interpretación* [Llo87] I de un lenguaje de primer orden $\mathcal{L}$ consta de una preinterpretación J con dominio D de $\mathcal{L}$ y de la asignación de una relación en D^n a cada símbolo de predicado de aridad

n.

Por tanto, una interpretación especifica un significado para cada símbolo en la fórmula. Se tendrá particular interés en interpretaciones para la cuales la fórmula exprese una afirmación "verdadera" en la interpretación. Tal interpretación se llama *modelo* de la fórmula.

Se dice que una cláusula se *satisface*, si tiene un modelo al menos. Asimismo, un conjunto de cláusulas P (programa) es *consistente* si existe una interpretación I que satisface cada cláusula en P.

Si se piensa en un programa P y en una meta M, dentro de un sistema de programación lógica, lo que se quiere demostrar es que $P \cup \{M\}$ es inconsistente. De hecho, si M es la meta $\leftarrow B_1, \ldots, B_n$, entonces, demostrar que $P \cup \{M\}$ es inconsistente es lo mismo que demostrar que $\{B_1, \ldots, B_n\}\theta$ es consecuencia lógica de P, para alguna sustitución aterrizante θ.

Por lo anterior, demostrar la inconsistencia de $P \cup \{M\}$, implica demostrar que *ninguna* interpretacion de $P \cup \{M\}$ es su modelo. Claramente, esto es complicado. Sin embargo, existe una clase de interpretaciones más pequeña y conveniente, estas interpretaciones son todas las que se necesitan estudiar para demostrar inconsistencia. Éstas son las llamadas "interpretaciones de Herbrand", que se definirán en esta sección.

Definición Sea $\mathcal{L}$ un lenguaje de primer orden. El *universo de Herbrand* $U_{\mathcal{L}}$ de $\mathcal{L}$ es el conjunto de todos los términos aterrizados que pueden formarse

con los símbolos función en el alfabeto asociado a $\mathcal{L}$. En el caso de que dicho alfabeto no tenga constantes, se añade alguna constante, por decir a, para formar términos aterrizados.

Ejemplo Considérese el programa

$$p(x) \leftarrow q(f(x), g(x))$$
$$r(y) \leftarrow$$

que tiene un lenguaje de primer orden subyacente $\mathcal{L}$ basado en los símbolos de predicado p, q, r y los símbolos de función f, g. El universo de Herbrand de $\mathcal{L}$ es

$$\{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), \ldots\}.$$

◁

Definición Sea $\mathcal{L}$ un lenguaje de primer orden. La *base de Herbrand* $B_{\mathcal{L}}$ de $\mathcal{L}$ es el conjunto de todos los átomos aterrizados que pueden formarse con los símbolos de predicado de $\mathcal{L}$.

Ejemplo Del ejemplo anterior, la base de Herbrand de $\mathcal{L}$ es

$$\{p(a), q(a, a), r(a), p(f(a)), p(g(a)), q(a, f(a)), \ldots\}.$$

◁

Definición La *preinterpretación de Herbrand* de $\mathcal{L}$ consta de:

- un dominio conformado por $U_{\mathcal{L}}$,

- un mapeo tal que si f es un símbolo de función de aridad n en el alfabeto de $\mathcal{L}$, entonces se le asigna el mapeo de $(U_{\mathcal{L}})^n$ a $U_{\mathcal{L}}$ que mapea la secuencia $t_1, \ldots, t_n$ de términos aterrizados al término aterrizado $f(t_1, \ldots, t_n)$.

Definición Sea $\mathcal{L}$ un lenguaje de primer orden. Una *interpretación de Herbrand* es una interpretación I basada en la preinterpretación de Herbrand de $\mathcal{L}$.

De esta manera, si p es un símbolo de predicado de aridad n en el alfabeto de $\mathcal{L}$ entonces se le asigna un conjunto p_I de tuplas con n términos aterrizados.

Definición Sean $\mathcal{L}$ un lenguaje de primer orden y $\mathcal{S}$ un conjunto de fórmulas cerradas de $\mathcal{L}$. Un *modelo de Herbrand* de $\mathcal{S}$ es una interpretación de Herbrand que es modelo de $\mathcal{S}$. Un modelo de Herbrand de $\mathcal{S}$ se denomina *modelo mínimo de Herbrand* $\mathcal{M}_{\mathcal{L}}$ de $\mathcal{S}$ si está incluido en cualquier modelo de Herbrand de $\mathcal{S}$.

De este modo cada interpretación de Herbrand de $\mathcal{L}$ se determina de manera única por la interpretación de los símbolos de predicado. Hay una correspondencia natural entre interpretaciones de Herbrand y subconjuntos de la base de Herbrand que se hace explícita mediante el mapeo que asigna a la interpretación de Herbrand I el conjunto de átomos aterrizados

$$\{p(t_1, \ldots, t_n) \mid p \text{ es un símbolo de predicado de aridad } n \text{ y } (t_1, \ldots, t_n) \in p_I\}.$$

Esto permite identificar interpretaciones de Herbrand de $\mathcal{L}$ con subconjuntos (posiblemente vacíos) de la base de Herbrand.

Teorema 2.4.1 ***(Teorema de Herbrand)*** [Doe94] *Sea P un programa. Las siguientes afirmaciones son equivalentes.*

1. *P tiene un modelo.*
2. *P tiene un modelo de Herbrand.*
3. *$aterrizados(P)$ es consistente.*

La siguiente definición establece la noción de verdad respecto a una interpretación de Herbrand. Se dice que E es verdad respecto a la interpretación I ($I \models E$).

Definición [Apt97] **(Noción de verdad)** Si I es una interpretación de Herbrand. Entonces

1. para un átomo A

$$I \models A \text{ sii } aterrizados(A) \subseteq I,$$

2. para una cláusula C

$$I \models C \text{ sii para toda } A \leftarrow B_1, \ldots, B_n \in aterrizados(C),$$

$$\{B_1, \ldots, B_n\} \subseteq I \text{ implica } A \in I.$$

En particular, para una interpretación de Herbrand I y un átomo aterrizado A, $I \models A$ si y sólo si $A \in I$. Entonces cada interpretación de Herbrand puede identificarse con el conjunto de átomos aterrizados que son verdaderos

en ella.

Un programa $P = \{C_1, \ldots, C_n\}$ es *verdadero* en I, denotado como $I \models P$, si y sólo si

$$\forall i \in [1, n] \quad I \models C_i.$$

A menudo será conveniente referirse, abusando del lenguaje, a la interpretación de un conjunto $\mathcal{S}$ de fórmulas en lugar de la del lenguaje de primer orden subyacente del cual provienen las fórmulas. Normalmente se asume que el lenguaje de primer orden subyacente se define por medio de los símbolos de función y de predicado que aparecen en $\mathcal{S}$. Bajo este entendido, se hará referencia al universo de Herbrand $U_{\mathcal{S}}$ y la base de Herbrand $B_{\mathcal{S}}$ de $\mathcal{S}$. En particular, el conjunto de fórmulas será con frecuencia un programa P. De este modo se hará referencia al universo de Herbrand U_P; a la base de Herbrand B_P, y al modelo mínimo de Herbrand $\mathcal{M}_P$ de P.

Sean P un programa y A un átomo. Se dice que A es *consecuencia lógica* de P, denotado como $P \models A$, si para toda interpretación I tal que $I \models P$ se tiene que $I \models A$. De igual manera si C es una cláusula, C es consecuencia lógica de P, denotado como $P \models C$, si para toda interpretación I tal que $I \models P$ se tiene que $I \models C$.

El siguiente teorema establece que los átomos en $\mathcal{M}_P$ son precisamente aquellos que son consecuencia lógica del programa. Este resultado se debe a van Emden y Kowalski [vEK76].

Teorema 2.4.2 *Sea P un programa. Entonces $\mathcal{M}_P = \{A \in B_P : P \models A\}$.*

Definición Sean P un programa y M una meta. Una *respuesta* para $P \cup \{M\}$ es una sustitución de las variables de M.

Definición Sean P un programa, M una meta $\leftarrow B_1, \ldots, B_k$ y θ una respuesta para $P \cup \{M\}$. Se dice que θ es una *respuesta correcta* para $P \cup \{M\}$ si $\{B_1, \ldots, B_k\}\theta$ es consecuencia lógica de P.

Teorema 2.4.3 *Sean P un programa y M una meta $\leftarrow B_1, \ldots, B_k$. Si θ es una respuesta para $P \cup \{M\}$ tal que $\{B_1, \ldots, B_k\}\theta$ es aterrizada, entonces las siguientes afirmaciones son equivalentes:*

1. *θ es correcta.*
2. *$\{B_1, \ldots, B_k\}\theta$ es verdadera respecto a cada modelo de Herbrand de P.*
3. *$\{B_1, \ldots, B_k\}\theta$ es verdadera respecto al modelo mínimo de Herbrand de P.*

Demostraciones de los teoremas presentados en esta sección se encuentran en [Llo87].

2.5. Inferencia

La lógica de primer orden provee métodos para deducir teoremas de una teoría. Esto puede caracterizarse como las fórmulas que son consecuencia lógica de los axiomas de la teoría, esto es, son verdaderas en toda interpretación que es modelo de los axiomas de la teoría. Los sistemas de programación lógica en los que estamos interesados usan "resolución" como única regla de

inferencia.

La "resolución" es una regla de inferencia aplicable a la lógica clausal. Brevemente se puede decir que a partir de dos cláusulas que tengan una forma apropiada la resolución deriva una nueva cláusula como consecuencia.

La siguiente es una definición de un tipo especial de resolución.

Definición [Llo87]. Sean $M = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ una meta y $C = A \leftarrow B_1, \ldots, B_q$ una cláusula definida. Entonces M' se deriva de M y C, al usar el umg θ, si y sólo si:

1. $vars(M) \cap vars(C) = \emptyset$.
2. A_m es un átomo, llamado el átomo *seleccionado*, en M.
3. θ es el umg de A_m y A.
4. M' es la meta $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

Se dice que M' es la *resolvente* de M y C.

El mecanismo descrito en la definición anterior se llama *paso de resolución-SLD*. Al iterar pasos de resolución-SLD se obtiene una *derivación-SLD*. La abreviatura SLD, viene de *Selection rule driven Linear resolution for Definite clauses* en inglés [Apt97].

Como se dijo, la resolución se usa para derivar las consecuencias de un programa lógico. Ahora bien, los demostradores de teoremas basados en resolución son sistemas de refutación. Esto es, la negación de la fórmula que se

va a demostrar se agrega a los axiomas y se intenta derivar una contradicción (cláusula vacía).

Dado un programa lógico P, una cláusula C es consecuencia de P si y sólo si se puede demostrar que $P \cup \{\neg C\}$ es inconsistente (*i.e.*, no tiene modelos).

Definición Sean P un programa lógico y M una meta. Una *refutación-SLD* de $P \cup \{M\}$ es una derivación-SLD finita que tiene como último resolvente la cláusula vacía $\square$.

Se dice que una derivación-SLD de $P \cup \{M\}$ es *exitosa* si dicha derivación es una refutación-SLD de $P \cup \{M\}$. Si $\theta_1, \ldots, \theta_n$ son los umgs tomados en cada paso de resolución-SLD entonces la composición $(\theta_1 \ldots \theta_n)$ restringida a las variables de M, se llama *respuesta calculada* de $P \cup \{M\}$ y $M\theta_1 \ldots \theta_n$ es la instancia calculada de M. Una derivación es *fallida* si el último resolvente no es la cláusula vacía y no se pueden aplicar más pasos de resolución-SLD.

Teorema 2.5.1 ***(Corrección de respuestas calculadas)*** [Llo87]. Sean P un programa y M una meta. Entonces cada respuesta calculada de $P \cup \{M\}$ es una respuesta correcta de $P \cup \{M\}$.

Corolario 2.5.2 [Llo87]. Sean P un programa lógico y M una meta. Si existe una refutación-SLD de $P \cup \{M\}$, entonces, $P \cup \{M\}$ es inconsistente.

Véanse demostraciones en [Llo87].

El resultado siguiente de completitud se demostró por primera vez en [Hil74].

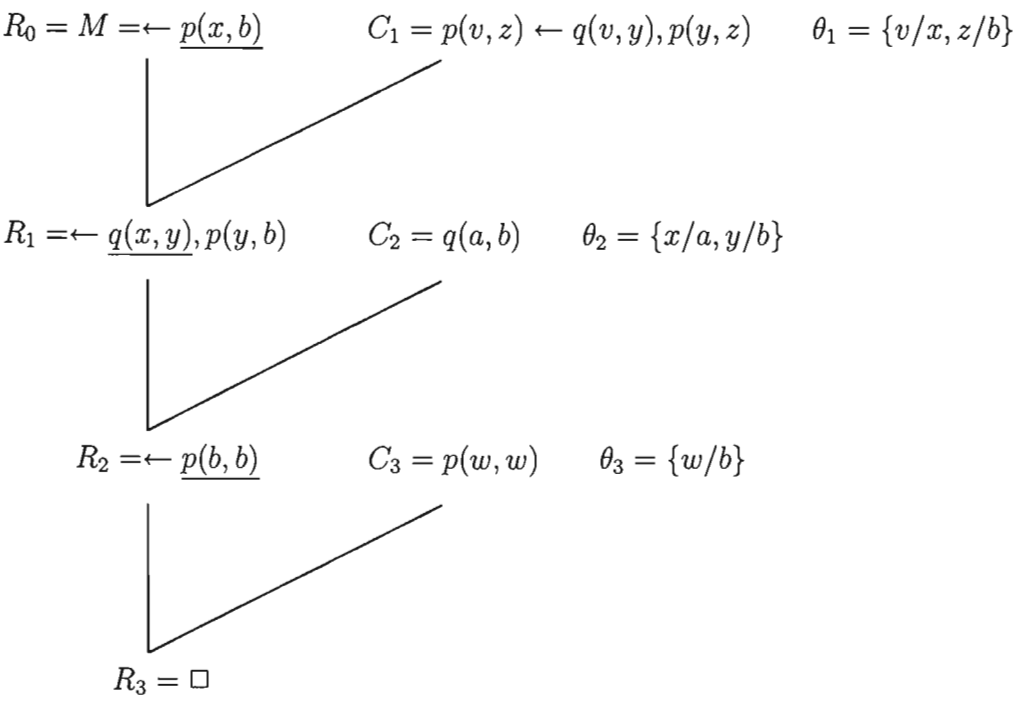

Figura 2.1: Una refutación-SLD de $P \cup \{M\}$

Teorema 2.5.3 *Sean P un programa y M una meta. Supóngase que $P \cup \{M\}$ es inconsistente, entonces, existe un refutación-SLD de $P \cup \{M\}$.*

Teorema 2.5.4 ***(Completitud de respuestas calculadas)*** *Sean P un programa y M una meta. Para cada respuesta correcta θ de $P \cup \{M\}$, existe una respuesta calculada σ para $P \cup \{M\}$ y una sustitución γ tales que θ es igual a la restricción de $\sigma\gamma$ a las variables de M.*

Ejemplo Considérese el siguiente programa P

1. $p(x,z) \leftarrow q(x,y), p(y,z)$

2. $p(x, x) \leftarrow$

3. $q(a, b) \leftarrow$

Sea $M = \leftarrow p(x, b)$. La figura 2.1 muestra una refutación-SLD de $P \cup \{M\}$, donde θ_i denota el umg que se usa en cada paso de resolución-SLD. Cada átomo seleccionado está subrayado. Esta refutación corresponde a la respuesta calculada $\{x/a\}$, ya que $\theta_1\theta_2\theta_3 = \{v/a, x/a, y/b, z/b, w/b\}$. ◁

Cuando se busca una derivación exitosa de una meta, las derivaciones-SLD se construyen con la intención de generar una refutación. Todas estas derivaciones forman un *espacio de búsqueda.* Una manera de organizar este espacio de búsqueda es dividir las derivaciones-SLD de acuerdo con la cláusula elegida para realizar cada paso de resolución-SLD. Esto lleva al concepto de *árbol-SLD.*

Definición Sean P un programa lógico y M una meta. Un *árbol-SLD* de $P \cup \{M\}$ es un árbol tal que

- Cada nodo del árbol es una meta.
- La raíz es M.
- Sea $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ $(k \geq 1)$ un nodo en el árbol y supóngase que A_m es el átomo seleccionado. Entonces para cada cláusula $A \leftarrow B_1, \ldots, B_q$ tal que A_m y A son unificables por medio de θ, el nodo tiene como hijo

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots B_q, A_{m+1}, \ldots, A_k)\theta$$

- Los nodos que son la cláusula vacía no tienen hijos.

Cada rama del árbol-SLD es una derivación de $P \cup \{M\}$ las ramas que corresponden a derivaciones exitosas se llaman *ramas exitosas*, las ramas de derivaciones infinitas se llaman *ramas infinitas*; y las ramas que corresponden a derivaciones fallidas, *ramas fallidas.*

Ejemplo Considérese el siguiente programa

1. $p(x, z) \leftarrow q(x, y), p(y, z)$
2. $p(x, x) \leftarrow$
3. $q(a, b) \leftarrow$

y la meta $\leftarrow p(x, b)$.

Las figuras 2.2 y 2.3 muestran respectivamente dos árboles-SLD para este programa y esta meta. En la figura 2.2 la regla de selección es tomar el átomo de más a la izquierda. En la figura 2.3 la regla de selección es tomar el átomo de más a la derecha. Los átomos seleccionados están subrayados y las ramas exitosas, fallidas e infinitas se muestran. Nótese que el primero es un árbol finito, mientras que el segundo es infinito. Cada árbol tiene dos ramas exitosas que corresponden a las respuestas $\{x/a\}$ y $\{x/b\}$, respectivamente. ◁

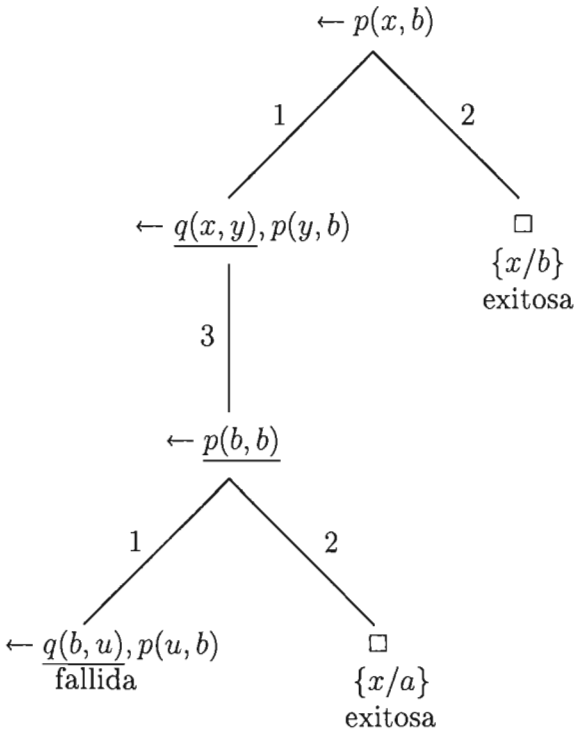

Figura 2.2: Un árbol-SLD finito

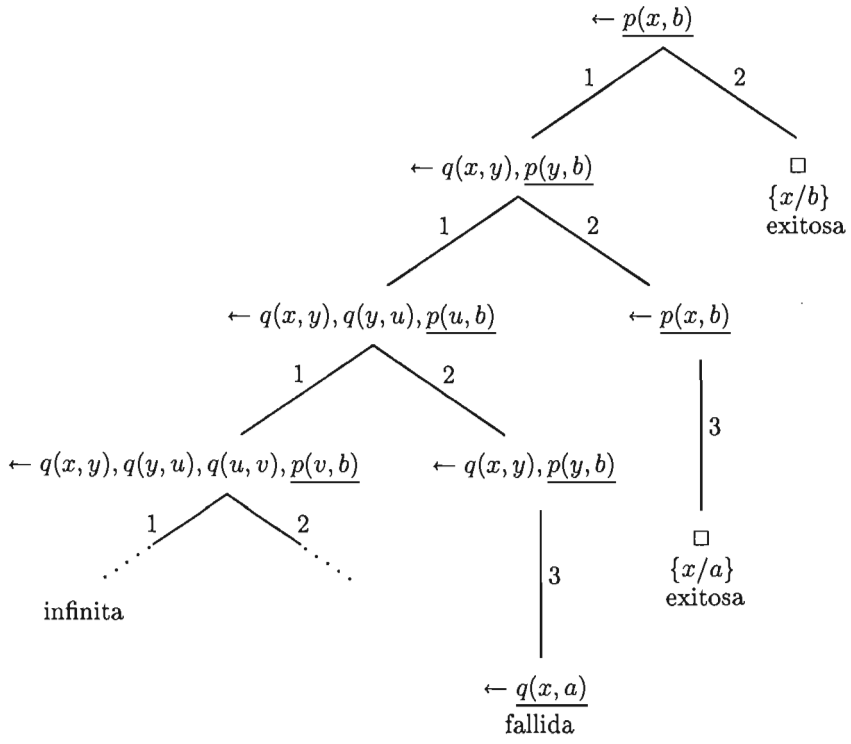

Figura 2.3: Un árbol-SLD infinito

Capítulo 3

Programación lógica inductiva

En este capítulo se ofrece un panorama general de la motivación detrás de la programación lógica inductiva (PLI) [BG96] [BM95] [LD94] [NCdW97], así como de los fundamentos teóricos de ésta [BG96] [LD94] [MR94] [NCdW97] [Sta93]. Se dedica una sección a dar una breve exposición de los métodos que se usan en la PLI para inferir teorías [BG96] [NCdW97] y se mencionan algunas aplicaciones de la PLI dentro de diversas áreas del conocimiento [LD94] [MR94].

3.1. Introducción

A pesar del éxito de la lógica deductiva, una pregunta ha surgido. Si todo el razonamiento humano o de máquinas proviene de axiomas lógicos, ¿Entonces de dónde provienen los axiomas lógicos? Una respuesta aceptada a esta pregunta es que los axiomas lógicos, que representan creencias generalizadas, pueden construirse de hechos particulares al usar razonamiento inductivo.

La historia del razonamiento inductivo interactúa con todo desarrollo de su contraparte deductiva. En la antigua Grecia el razonamiento inductivo desempeñó un papel clave en las discusiones dialécticas de Sócrates, las cuales se describen en el *Crito* de Platón. En estas discusiones los conceptos se desarrollaron y refinaron al utilizar una serie de ejemplos y contraejemplos tomados de la vida diaria.

En el siglo XVII Francis Bacon en su obra *Novum Organum* [Bac94] dio la primera descripción detallada del método científico inductivo. En siglos posteriores, muchos filósofos reanudaron el estudio de la inducción. David Hume [Hum00, Hum78] formuló lo que actualmente se conoce como el problema de inducción o el problema de Hume: ¿Cómo puede la inducción a partir de un número finito de casos resultar en conocimiento sobre la infinidad de casos para los que una regla general se aplica? ¿Qué justifica inferir una regla general a partir de un número finito de casos particulares? Sorpresivamente, Hume respondió que no existe tal justificación. Desde su punto de vista, es simplemente una cuestión psicológica de los seres humanos, que al observar que algún patrón particular se repite en casos distintos (sin observar contraejemplos del patrón), tendemos a esperar que este patrón aparezca en todos los casos similares. Hume señaló que esta expectativa es un hábito, análogo al hábito de un perro que corre a la puerta después de oír que su amo lo llama, esperando que su amo lo saque a pasear. Otros filósofos como John Stuart Mill [Mil58] intentaron encontrar una respuesta al problema que planteó Hume al establecer condiciones bajo las que una inferencia inductiva se justifica. Otros filósofos que abordaron el tema fueron Stanley Jevons [Jev07]

y Charles Sanders Peirce [Pei58].

En el siglo pasado, la inducción se retomó principalmente por filósofos y matemáticos que estaban involucrados en el desarrollo y aplicación de lógica formal. El tratamiento que le dieron a la inducción a menudo fue en términos de probabilidad o "grado de confirmación" de una teoría en particular o de hipótesis que se obtienen a partir de datos empíricos disponibles. Algunos de los que principalmente contribuyeron son Bertrand Russell [Rus48, Rus97], Rudolf Carnap [Car52, Car62], Carl Hempel [Hem75, Hem99], Hans Reichenbach [Rei49] y Nelson Goodman [Goo73].

En los años cincuenta y sesenta filósofos de la ciencia como Karl Popper [Pop02] [Lak87] descalificaron la inducción. Sin embargo, en esos mismos años se reconoció en el creciente campo de la inteligencia artificial, que en un sistema de inteligencia artificial no es necesario codificar todo el conocimiento de antemano. En cambio, es más eficiente proveer al sistema con un poco de conocimiento y con la habilidad de adaptarse a las distintas situaciones que se pudiera encontrar (para aprender de la experiencia). Así el estudio de la inducción pasó del campo de la filosofía al de la inteligencia artificial.

A pesar de que Gordon Plotkin en los años setenta y Ehud Shapiro en los años ochenta trabajaron en sistemas inductivos basados en computadoras dentro del marco de la lógica de primer orden, la mayoría de los resultados dentro del campo de *aprendizaje de máquina* se han obtenido a partir de sistemas que construyen hipótesis en el límite de la lógica proposicional.

La abducción es otro campo de estudio que debe mencionarse, debido a que éste tiene una estrecha relación con la inducción. La abducción se introdujo por el filósofo Charles Sanders Peirce [Pei58]. La forma lógica de la abducción es muy similar a la de la inducción [DK96, KKT93], las características que las distinguen son muy sutiles. Ambas proceden a patir de ejemplos y conocimiento previo, y el objetivo es encontrar una teoría que junto con el conocimiento previo "expliquen" dichos ejemplos. Sin embargo, la teoría que genera la abducción debe de ser un *hecho particular*, que junto con el conocimiento previo expliquen los ejemplos. Lo anterior difiere de la inducción, la cual genera una *teoría general.*

Un ejemplo informal es el siguiente. Suponga que es Robinson Crusoe en su isla, y que ve huellas de un humano extraño en la arena. Puesto que sabe que las huellas de humanos se producen por seres humanos, y que las huellas no son suyas, entonces puede concluir, basandose en el conocimiento previo, que alguien más ha visitado la isla. La hipótesis de que alguien más ha visitado la isla explica la presencia de las huellas (el ejemplo). El inferir esta explicación particular es un caso de abducción.

3.2. Fundamentos

La *programación lógica inductiva* es una disciplina que estudia la construcción de teorías clausales de primer orden a partir de "ejemplos" y "conocimiento previo". Esta disciplina se encuentra en la intersección de la programación

lógica y el aprendizaje de máquina [LD94]. La programación lógica inductiva tiene como propósito una formalización y el desarrollo de métodos para la inferencia inductiva de programas lógicos a partir de ejemplos.

El interés actual en el aprendizaje de máquina, como una subárea de la inteligencia artificial, es doble. Por un lado, a pesar de que no hay concenso sobre la naturaleza de la inteligencia, se está de acuerdo en que la capacidad de aprendizaje es crucial para cualquier comportamiento inteligente. El aprendizaje de máquina se justifica, de esta manera, desde un punto de vista científico como parte de la ciencia cognitiva. Por otro lado, las técnicas de aprendizaje de máquina pueden usarse exitosamente en herramientas de adquisición de conocimiento, la cual es una justificación desde el punto de vista de la ingeniería, a saber, se acepta generalmente que el problema principal en la construcción de sistemas expertos es la adquisición de conocimiento.

Diversos formalismos lógicos se han utilizado en sistemas de aprendizaje inductivo para representar ejemplos y descripciones de conceptos. Estos formalismos son similares a los que se usan para representar el conocimiento en general, y van de la lógica proposicional a la lógica de predicados de primer orden. Dentro de este ámbito se distingue entre sistemas que aprenden descripciones de atributos, y sistemas que aprenden descripciones de relaciones de primer orden.

Muchos algoritmos de aprendizaje inductivo ampliamente conocidos, como ID3 [Qui86] y AQ [Mic83] usan un lenguaje de atributo-valor para re-

presentar objetos y conceptos, y se llaman por tanto *sistemas de aprendizaje de atributo-valor.* También se llaman *sistemas de apredizaje proposicional* ya que un lenguaje de atributo-valor es equivalente en poder expresivo al cálculo proposicional. Tanto en ID3 como en AQ, los objetos se describen en términos de sus características globales, es decir, mediante valores de una colección fija de atributos. Para representar conceptos se usan árboles de decisión en ID3 y reglas *"if-then"* en AQ. Las dos principales limitaciones de los sistemas de aprendizaje proposicional son la expresividad limitada del formalismo de representación y la capacidad limitada para tomar en cuenta el conocimiento previo disponible.

Otra clase de sistemas de aprendizaje, llamados *sistemas de aprendizaje relacionales*, inducen descripciones de relaciones (definiciones de predicados). En estos sistemas los objetos pueden describirse estructuralmente, es decir, en términos de sus componentes y relaciones entre sus componentes. Las relaciones dadas constituyen el conocimiento previo. En los sistemas de aprendizaje relacionales, los lenguajes que se usan para representar ejemplos, conocimiento previo y descripciones de conceptos son típicamente fragmentos de la lógica de primer orden. En particular, el lenguaje de los programas lógicos [Llo87] provee suficiente expresividad para resolver un número significativo de problemas de aprendizaje de relaciones. Los sistemas de aprendizaje que inducen hipótesis en forma de programas lógicos se llaman *sistemas de programación lógica inductiva.*

Una de las ventajas principales de los sistemas de programación lógica in-

ductiva sobre los sistemas de aprendizaje atributo-valor es que los primeros permiten proveer conocimiento previo de un dominio específico que puede usarse durante el proceso de aprendizaje. El uso de conocimiento previo permite desarrollar una representación adecuada del problema y además introducir restricciones específicas del problema dentro del proceso de aprendizaje. En contraste los sistemas de aprendizaje de atributo-valor comúnmente no cuentan con conocimiento previo durante el proceso de aprendizaje. Así que si se tiene un sistema de programación lógica inductiva, y se quiere aprender, por ejemplo, a distinguir gráficas cíclicas de acíclicas, las gráficas se pueden introducir al representar sus aristas como conocimiento previo. Además, se puede proveer una definición recursiva de la noción de camino dentro de una gráfica. Si el problema es aprender acerca de las propiedades de compuestos químicos, las estructuras moleculares se pueden introducir como conocimiento previo en términos de los átomos y los enlaces entre ellos. Si la tarea es construir, de manera automática, un modelo de un sistema físico a partir de las observaciones de su comportamiento, las herramientas matemáticas que se consideren útiles para modelar el dominio del problema se incluyen en el conocimiento previo. El uso de la programación lógica inductiva implica el desarrollo de una buena representación de los ejemplos y del conocimiento previo pertinente. En la última sección de este capítulo se abordarán algunas aplicaciones de la programación lógica inductiva.

3.3. Planteamiento del problema y definiciones

Se comenzará con la introducción de la noción de un *ejemplo* del comportamiento entrada/salida de un programa lógico. Intuitivamente, un ejemplo debería describir el comportamiento entrada/salida de un programa en un caso específico. Para programas lógicos, esto se formaliza fácilmente al establecer los ejemplos como literales aterrizadas. La posibilidad de tener ejemplos no aterrizados es interesante, pero no se ha entendido suficientemente bien en la PLI. Cada uno de dichos ejemplos, de hecho, describe un conjunto posiblemente infinito de ejemplos aterrizados. El usuario entonces tiene la posibilidad de proveer la entrada al sistema de aprendizaje de una manera más concisa. Sin embargo, la mayoría de los enfoques sólo consideran ejemplos aterrizados.

Los ejemplos necesitan no sólo describir el comportamiento de un programa, sino también pueden proveer información negativa, es decir, cosas que el programa no debería calcular. En general, los sistemas de aprendizaje inductivo usan "ejemplos positivos" y "ejemplos negativos". Los *ejemplos positivos* son literales aterrizadas consideradas verdaderas en la interpretación deseada por el usuario, mientras que los *ejemplos negativos* son literales aterrizadas que el usuario trata como falsas en la interpretación deseada. El programa aprendido debe realizar cálculos que no contradigan los ejemplos. Si se considera un conjunto E^+ de ejemplos positivos y un conjunto E^- de ejemplos negativos, esto se hace preciso de la siguiente forma:

Se espera que el sistema de inferencia produzca un conjunto finito de cláusulas en un lenguaje $\mathcal{L}$ que explique los ejemplos positivos y excluya los

ejemplos negativos.

De manera más formal [Sta93], sea $\mathcal{M}$ la interpretación deseada y $\mathcal{L}_0$ el lenguaje de los ejemplos sobre los símbolos de predicado y de función en $\mathcal{M}$. Una *presentación completa de* $\mathcal{M}$ consiste en

$$E^+ = \{\varphi \in \mathcal{L}_0 \mid \mathcal{M} \models \varphi\} \text{ y } E^- = \{\varphi \in \mathcal{L}_0 \mid \varphi \notin E^+\}.$$

Los conjuntos de los ejemplos positivos $\mathcal{E}^+$ y negativos $\mathcal{E}^-$ que se proporcionan a los sistemas de inferencia inductiva son subconjuntos propios, en la práctica finitos, de E^+ y E^-. No obstante, para conjuntos finitos de ejemplos no hay necesidad de construir una hipótesis, ya que el conjunto de ejemplos positivos siempre puede usarse como un resultado correcto de la inferencia inductiva. Por lo tanto, se considera el caso límite en el que se da una presentación completa y posiblemente infinita de $\mathcal{M}$.

El objetivo de la inferencia inductiva dentro de este marco es encontrar un conjunto finito de fórmulas T en $\mathcal{L} \supseteq \mathcal{L}_0$ tal que

$$T \models E^+ \text{ y } T \not\models E^-$$

i.e., para toda fórmula $\varphi \in \mathcal{L}_0$, se tiene que si $\varphi \in E^+$ entonces $T \models \varphi$ y si $\varphi \notin E^+$ entonces $T \not\models \varphi$; que es equivalente a decir que para toda $\varphi \in \mathcal{L}$, $T \models \varphi$ si y sólo si $\varphi \in E^+$.

Esto último con T finito es exactamente la definición de que T sea una axiomatización finita de E^+. Por lo tanto, el problema de la inferencia inductiva en el límite es encontrar una axiomatización finita de la interpretación

deseada. Si la interpretación deseada no es finitamente axiomatizable dentro de un lenguaje $\mathcal{L}$, la inferencia inductiva puede no tener éxito. Además, ni siquiera es decidible si el conjunto de hechos aterrizados de E^+ válidos en $\mathcal{M}$ es finitamente axiomatizable dentro del lenguaje $\mathcal{L}$ [Sta93].

A un programa lógico B dado como entrada al sistema de inferencia se le llama comúnmente *conocimiento previo* y representa una parte del programa (axiomatización finita) deseado, que el usuario ya tiene disponible o se puede programar directamente.

Se ilustrará la tarea de la PLI con un ejemplo. Imagínese que se está aprendiendo sobre las relaciones dentro del círculo familiar, y ya se tiene conocimiento de las relaciones *progenitor* y *mujer*. Dicho conocimiento puede representarse por el siguiente conjunto de cláusulas.

$$B = \left\{ \begin{array}{l} progenitor(ana, maría) \leftarrow \\ progenitor(ana, tomás) \leftarrow \\ progenitor(tomás, evelinda) \leftarrow \\ progenitor(tomás, ignacio) \leftarrow \\ mujer(ana) \leftarrow \\ mujer(maría) \leftarrow \\ mujer(evelinda) \leftarrow \end{array} \right.$$

Se supone que se tiene también conocimiento de los siguientes hechos (ejemplos positivos) concernientes a la relación entre hijas y progenitores

específicos, representado de la siguiente manera.

$$\mathcal{E}^{+} = \begin{cases} hija(maría, ana) \leftarrow \\ hija(evelinda, tomás) \leftarrow \end{cases}$$

Además se sabe cuándo la relación de *hija* no se cumple para individuos particulares (ejemplos negativos).

$$\mathcal{E}^{-} = \begin{cases} hija(tomás, ana) \leftarrow \\ hija(evelinda, ana) \leftarrow \end{cases}$$

Al tomar B como cierto y compararlo con los hechos en $\mathcal{E}^{+}$ y $\mathcal{E}^{-}$ se puede pensar que la siguiente relación se cumple.

$$P = \{hija(x, y) \leftarrow mujer(x), progenitor(y, x)$$

3.4. Herramientas de la PLI

Una palabra clave en el aprendizaje de máquina es *generalización* [BG96]. Aun cuando no hay un total acuerdo en la comunidad de aprendizaje de máquina sobre el significado de generalización, a menudo se le considera como la construcción de una descripción general de un conjunto de ejemplos (positivos), tal que puede utilizarse para predecir la clasificación de un nuevo tipo de datos. El objetivo de esto es obtener una descripción más concisa de los datos que, después de un esfuerzo inicial para construirla, pueda utilizarse sin cambios posteriores.

Intuitivamente, se dice que una cláusula C_1 es una generalización de otra cláusula C_2 si todo modelo de C_1 es también modelo de C_2, pero lo contrario

no es necesariamente cierto, es decir, no todo modelo de C_2 es modelo de C_1. En el marco de la lógica esto puede hacerse preciso de diferentes maneras, como $C_1 \models C_2$ (semánticamente) o $C_1 \rightarrow C_2$ (sintácticamente). Normalmente se desea que la generalización sea una tarea que vaya de lo "específico a lo general". Si los ejemplos ("lo específico") son el punto de partida, y la descripción aprendida ("lo general") es el objetivo, dicho objetivo es un conjunto de reglas que implican los ejemplos dados y quizá otros ejemplos. Como se puede observar en el ejemplo de la sección anterior, $\mathcal{E}^+$ es "lo específico" y $P \cup B$ es "lo general".

Existen algunos métodos que se basan en la generalización de un conjunto de ejemplos (positivos), y que encajan perfectamente con la noción de generalización, sin importar qué operador de generalización utilicen. Estos métodos, comúnmente se conocen en la literatura en inglés como bottom-up. Por otro lado, también existen los métodos conocidos como top-down, en la literatura en inglés, que se basan en la especialización de una hipótesis general. En este trabajo sólo se explicarán algunos métodos bottom-up. Estos métodos pueden clasificarse respecto a la forma en que realizan la generalización. Los operadores de generalización clásicos están representados por reglas como "tirar condiciones", remplazar constantes por variables y conjunciones por disyunciones, y se han utilizado extensamente en el aprendizaje de máquina [Mic83]. Sin embargo, todas las anteriores son reglas heurísticas y pueden fallar en la tarea de dar una generalización correcta. En la PLI los operadores inductivos se han desarrollado al invertir reglas deductivas debidamente formalizadas, a saber, unificación y resolución. Invertir estas reglas

ha traído como consecuencia la construcción de sistemas con una base teórica sólida.

A continuación se presentan tres modelos de generalización: "generalización más específica"; "generalización más específica relativa"; y "subsunción generalizada".

3.4.1. Modelos de generalización

Gordon Plotkin [Plo70] y John Reynolds [Rey70] fueron, probablemente, los primeros en analizar rigurosamente la noción de generalización como automatización del proceso de inferencia inductiva.

Definición Sean D y C cláusulas. Se dice que C *subsume o* θ*-subsume* a D si existe una sustitución θ tal que $C\theta \subseteq D$ (*i.e.*, cada literal en $C\theta$ aparece también en D).

Ejemplo Algunos ejemplos de θ-subsunción:

- $C = p(x) \leftarrow q(a)$ θ-subsume a $D = p(a) \leftarrow p(f(x)), q(a)$
- La cláusula vacía $\square$ subsume a cualquier cláusula, pues el conjunto vacío es subconjunto del conjunto de literales de cualquier cláusula.
- La única cláusula que subsume a la cláusula vacía $\square$ es, la cláusula vacía $\square$ misma.

◁

Un modelo de generalización usado por Plotkin y Reynolds establece que una cláusula C es *más general* que una cláusula D si C θ-subsume a D. Usualmente esto se escribe como $C \succeq D$.

Definición C es la *generalización más específica* (gme) del conjunto de cláusulas $\{D_1, \ldots, D_n\}$ bajo *θ-subsunción* si $C \succeq D_i$, para toda $1 \leq i \leq n$, y si para algúna otra F sucede que $F \succeq D_i$, entonces $F \succeq C$.

Considérense las siguientes cláusulas:

$$C_1 = tiene\_alas(águila) \leftarrow ave(águila)$$
$$C_2 = tiene\_alas(halcón) \leftarrow ave(halcón)$$

Al usar la definición anterior se puede ver que la gme para C_1 y C_2 es:

$$C_3 = tiene\_alas(x) \leftarrow ave(x).$$

Un algoritmo para calcular la generalización más específica basado en [Plo70] es el siguiente.

Algoritmo 3.4.1 *Sean E_1 y E_2 expresiones tales que ambas son términos, o literales con el mismo símbolo de predicado y signo (i.e., ambas son positivas, o negativas).*

1. *Asignar E_i a V_i, asignar ε, la sustitución identidad, a σ_i, $i \in \{1, 2\}$.*

2. *Determinar si existen subtérminos* t_1, t_2 *que tengan el mismo lugar en* V_1, V_2, *respectivamente, y tales que* $t_1 \neq t_2$ *y* t_1 *y* t_2 *tengan diferentes símbolos de función o bien, al menos uno de ellos sea una variable.*

3. *Si no existen tales* t_1, t_2 *entonces parar.* V_1 *es la generalización más específica de* $\{E_1, E_2\}$ *y* $V_1 = V_2$, $V_i\sigma_i = E_i$, $i \in \{1, 2\}$.

4. *Si sí existen tales* t_1, t_2, *entonces escoger una variable* x *que no ocurra ni en* V_1 *ni en* V_2 *y remplazar cada ocurrencia de* t_1 *y* t_2 *en el mismo lugar en* V_1 *y* V_2, *respectivamente, por la variable elegida* x.

5. *Cambiar* σ_i *por* $\{x/t_i\}\sigma_i$, $i \in \{1, 2\}$

6. *Ir al paso 2.*

◇

Ejemplo Se usará el algoritmo 3.4.1 para encontrar la generalización más específica del conjunto

$$\{p(f(a, g(y)), x, g(y)), p(h(a, g(x)), x, g(x))\}.$$

Inicialmente se tiene que

$$V_1 = p(f(a, g(y)), x, g(y))$$
$$V_2 = p(h(a, g(x)), x, g(x)).$$

Se toma $t_1 = y$, $t_2 = x$, y z como la nueva variable. Entonces después de *4* se tiene que

$$V_1 = p(f(a, g(z)), x, g(z))$$
$$V_2 = p(h(a, g(z)), x, g(z))$$

y después de *5*; que

$$\sigma_1 = \{z/y\}, \sigma_2 = \{z/x\}.$$

A continuación, se toma $t_1 = f(a, g(z))$, $t_2 = h(a, g(z))$ y w como la nueva variable. Después de *4* y *5*, se tiene que

$$\begin{aligned} V_1 = &\ p(w, x, g(z)) = V_2 \\ \sigma_1 = &\ \{w/f(a, g(z))\}\{z/y\} \\ = &\ \{w/f(a, g(y)), z/y\} \\ \sigma_2 = &\ \{w/h(a, g(z))\}\{z/x\} \\ = &\ \{w/h(a, g(x)), z/x\} \end{aligned}$$

El algoritmo termina con $p(w, x, g(z))$ como la gme. ◁

En el marco de PLI, la generalización más específica puede proveer una base teórica para la generalización. Sin embargo, la generalización más específica no es muy útil para el problema de la PLI. Por ejemplo, dadas dos cláusulas:

$$C_1 = tío(x, y) \leftarrow hermano(x, padre(y))$$
$$C_2 = tío(x, y) \leftarrow hermano(x, madre(y))$$

la gme correspondiente es:

$$gme(C_1, C_2) = tío(x, y) \leftarrow hermano(x, z)$$

lo cual no resulta muy informativo para definir la relación *tío*. De hecho, se está más interesado en encontrar una generalización del conjunto de ejemplos en relación con el conocimiento previo o a la teoría. Por tanto, si en el ejemplo anterior se sabe que $progenitor(padre(x), x)$ es verdadero y $progenitor(madre(x), x)$

es verdadero, entonces la generalización más específica de C_1 y C_2 *relativa* a este conocimiento resulta ser:

$$C = tío(x, y) \leftarrow hermano(x, u), progenitor(u, y)$$

Se dice que una cláusula C es más general que una cláusula D respecto a una teoría T si $T \cup \{C\} \models D$. Como un caso especial, T puede ser un conjunto de cláusulas definidas (o programa). Este caso es importante para la PLI, porque se podría tener un programa lógico B de manera parcial, y necesitar redefinir este programa mediante la adición de otras cláusulas que permita derivar nuevos ejemplos. El programa lógico parcial al que se hace referencia es el conocimiento previo.

La noción de generalización relativa a una teoría T dada fue estudiada y definida por Plotkin [Plo71] como la extensión de la generalización más específica bajo θ-subsunción.

Definición Una cláusula C es *más general que una cláusula D respecto a una teoría T*, si $T \models C\theta \rightarrow D$ para alguna sustitución θ [Plo71]. Se dice que C es la generalización de D relativa a T, $C \succeq_T D$. Se utilizará **gr** para referirse a la generalización relativa.

La relación entre la generalización relativa y una refutación se da en el siguiente teorema:

Teorema 3.4.1 [Plo71] *Sean C y D cláusulas y P un programa lógico. $C \succeq_P D$ si y solo si C ocurre a lo más una vez en alguna refutación que demuestra $P \models C \rightarrow D$.*

Definición Una cláusula C es la *generalización más específica* de las cláusulas D_1 y D_2 *relativa* a una teoría T si y solo si

1. $C \succeq_T D_1$ y $C \succeq_T D_2$
2. y si $C' \succeq_T D_1$ y $C' \succeq_T D_2$ entonces $C' \succeq_T C$

La generalización más específica relativa puede verse como una forma de evitar algunos resultados sin sentido, generados por la gme, mediante el uso del conocimiento previo. Sin embargo, aun la noción de gr puede llevar a algunas conclusiones contraintuitivas. Por ejemplo [Bun88], de acuerdo con la definición anterior, se puede verificar que la cláusula:

$$C = pequeño(x) \leftarrow gato(x)$$

es más general que la cláusula:

$$D = mascota\_mimosa(x) \leftarrow peludo(x), gato(x)$$

respecto al programa P:

$$mascota(x) \leftarrow gato(x)$$
$$mascota\_mimosa(x) \leftarrow pequeño(x), peludo(x), mascota(x)$$

informalmente, se puede ver como sigue: suponga que para toda x se tiene que si x es un *gato*, entonces x es *pequeño* (*i.e.*, C es verdadera). Si se usa la primera cláusula de P también se tiene que si x es un *gato*, entonces x es una *mascota*. De esta manera, si x es un gato *peludo*, entonces x es *pequeño*, *peludo* y una *mascota*, y de la segunda cláusula de P se tiene que x es una *mascota_mimosa* (*i.e.*, D es verdadera).

De manera más formal, se tiene que $C \succeq_P D$ puesto que $P \models C\varepsilon \rightarrow D$.

Suponga ahora que una cláusula C es más general que una cláusula D si cualquier interpretación de C puede utilizarse para generar al menos las mismas conclusiones que D. Entonces, las cláusulas C y D mencionadas en el ejemplo no se encuentran en una relación de generalización, ya que definen conceptos diferentes.

Para evitar las anomalías que puede generar la generalización relativa, Wray Buntine [Bun88] introdujo la noción de *subsunción generalizada* como un caso especial de la generalización más específica relativa, restringida a cláusulas definidas. En el marco de la subsunción generalizada, una cláusula definida C es más general que una cláusula definida D respecto a un programa P si, siempre que D pueda utilizarse para explicar un ejemplo, C también puede explicar ese ejemplo. Esto es, C se puede aplicar en cualquier situación en la que D es aplicable esto se formaliza como: el conjunto de átomos "cubiertos" por C es un super conjunto de los átomos cubiertos por D. La noción de *cubrir* se define a continuación.

Definición Se dice que una cláusula definida $A \leftarrow B_1, \ldots, B_n$ *cubre* a un átomo A_1 en una interpretación I si existe una sustitución aterrizante θ tal que $A\theta$ es idéntica a A_1 e $I \models \{B_1, \ldots, B_n\}\theta$. Un conjunto de cláusulas definidas *cubre* a un átomo A en la interpretación I si alguna cláusula en dicho conjunto cubre a A en I.

La definición anterior formaliza la noción de que una cláusula sea la "responsable" directa de demostrar que un átomo es verdadero.

Ejemplo Sean $C = p(f(x)) \leftarrow p(x)$ y $A = p(f(a))$. Entonces, C cubre a A en la interpretación $I = \{p(a)\}$, ya que si $\theta = \{x/a\}$, entonces, $C_{cuerpo}\theta = p(a)$ es verdadera en I y $C_{cabeza}\theta = A$. Nótese que A no necesita ser verdadera en I para aplicar la definición de cubre. Asimismo, C no cubre a A en $I = \{p(f(a))\}$, aun cuando A es verdadero en I.

$C = p(x)$ cubre a $A = p(a)$ en cualquier interpretación de Herbrand. ◁

Ahora se puede definir la subsunción generalizada.

Definición Una cláusula definida C es *más general* que una cláusula definida D *respecto a un programa* P, si para cualquier modelo de Herbrand I de P, y para cualquier átomo A, C cubre a A en I siempre que D cubra a A en I [Bun88]. Esto se denota como $C \succeq_P^B D$. Se dice que C es una *generalización* de D, y que D es una *especialización* de C.

Esta definición se extiende a un conjunto de cláusulas. El conjunto de cláusulas definidas $\{C_1, \ldots, C_n\} \succeq_P^B \{D_1, \ldots, D_m\}$ respecto al programa lógico P si para todo modelo de Herbrand I de P, y para cualquier átomo A, $\{C_1, \ldots, C_n\}$ cubre a A en I siempre que el conjunto $\{D_1, \ldots, D_m\}$ lo cubre.

Ejemplo [NCdW97] Sea P el siguiente conjunto de cláusulas:

$$mascota(x) \leftarrow gato(x)$$
$$mascota(x) \leftarrow perro(x)$$
$$pequeño(x) \leftarrow gato(x)$$

y suponga que:

$$C = mascota\_mimosa(x) \leftarrow pequeño(x), mascota(x)$$
$$D = mascota\_mimosa(x) \leftarrow gato(x)$$

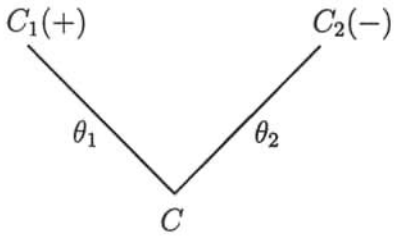

Figura 3.1: Paso simple de resolución

entonces, se puede demostrar que $C \succeq_P^B D$. Suponga que $\mathcal{M}$ es un modelo de Herbrand de P, y D cubre a algún átomo $A = mascota\_mimosa(t)$ en $\mathcal{M}$. Entonces, para $\theta = \{x/t\}$, $D_{cuerpo}\theta$ es verdadero en $\mathcal{M}$. Puesto que $\mathcal{M}$ es un modelo de P, en particular de la primera y tercera cláusula de P, $mascota(t)$ y $pequeño(t)$ también deben ser verdaderas. Entonces, $C_{cuerpo}\theta = \{pequeño(t), mascota(t)\}$ es verdadero en $\mathcal{M}$ y $C_{cabeza}\theta = A$, así que C cubre a A en $\mathcal{M}$. Por lo tanto, C es más general que D respecto a P. Dicho en lenguaje natural: si las mascotas pequeñas son mascotas mimosas entonces, los gatos son mascotas mimosas, puesto que se sabe que los gatos son mascotas pequeñas. ◁

3.4.2. Un método de programación lógica inductiva

En esta sección se tratará un método cuyos operadores inductivos invierten la regla de resolución. Como se dijo antes, existen otros métodos que invierten otras reglas deductivas.

Resolución inversa

La noción de resolución inversa se introdujo en el trabajo seminal de Stephen Muggleton y Wray Buntine [MB88a]. Sin embargo, algunos trabajos previos ya utilizaban una versión simplificada de la resolución inversa [Mug87, SB86]. Desde el primer trabajo de resolución inversa se han presentado varios enfoques, que han involucrado diferentes lenguajes de representación y métodos de aprendizaje, pero se ha mostrado que estos trabajos son esencialmente una forma especial de la teoría de resolución inversa [LN92]. Más aún, Muggleton [Mug91] ha descrito una perspectiva que unifica la resolución inversa y la noción de generalización más específica relativa.

Se comenzará por recordar la regla de resolución y se establecerá la notación.

Dadas dos cláusulas C_1 y C_2, la regla de resolución permite deducir una cláusula nueva de la siguiente manera. Al suponer que las dos cláusulas son disjuntas respecto a sus variables, y tomar las literales l_1 y l_2 en C_1 y C_2 respectivamente, sea θ el unificador más general de l_1 y l_2 tal que $\neg l_1\theta = l_2\theta$. Ya que l_1 y l_2 son disjuntas respecto a las variables, esto puede reescribirse como $\neg l_1\theta_1 = l_2\theta_2$, con $\theta = \theta_1\theta_2$. Entonces, al aplicar la regla de resolución a las cláusulas C_1 y C_2, la *resolvente* C puede expresarse como:

$$C = (C_1 - \{l_1\})\theta_1 \cup (C_2 - \{l_2\})\theta_2 \tag{3.1}$$

Esto se denotará como $C = C_1 \cdot C_2$ y se representa como en la figura 3.1

Así como la resolución es la base para los sistemas deductivos, la inversión de la resolución es la base para construir sistemas inductivos. Esto es, si una hipótesis correcta, junto con el conocimiento previo, puede utilizarse en una demostración por resolución de algunos ejemplos, entonces, la hipótesis puede inducirse del conocimiento previo y de los ejemplos al invertir el proceso de resolución.

Sin embargo, algunos problemas surgen al invertir la resolución. Suponga que se tienen las tres cláusulas siguientes, donde C es la resolvente de C_1 y C_2:

$$\begin{aligned} C_1 &= B \leftarrow E, F \\ C_2 &= A \leftarrow B, G, D \\ C &= A \leftarrow E, F, G, D \end{aligned}$$

Dadas C y C_1, entonces, C_2 es claramente una solución al problema de encontrar una cláusula que produzca C al resolver con C_1. Sin embargo, las siguientes cláusulas también serían una solución correcta al mismo problema:

$$\begin{aligned} C_2' &= A \leftarrow B, G, D, E \\ C_2'' &= A \leftarrow B, G, D, F \\ C_2''' &= A \leftarrow B, G, D, E, F \end{aligned}$$

Esto es, en general no se tiene una única solución a menos que se asuma C_1 y C_2 disjuntas respecto a sus variables proposicionales. El problema se agrava si se trata con cláusulas de primer orden. Considérese el siguiente ejemplo. Dadas:

$$\begin{aligned} C_1 &= \leftarrow más\_pesado(martillo, pluma) \\ C &= \leftarrow más\_denso(martillo, pluma), más\_largo(martillo, pluma) \end{aligned}$$

muchas soluciones para C_2 son posibles, de la más específica:

$$C_2 = más\_pesado(martillo, pluma) \leftarrow más\_denso(martillo, pluma), más\_largo(martillo, pluma)$$

a la más general:

$$C_2 = más\_pesado(x, y) \leftarrow más\_denso(x, y), más\_largo(x, y)$$

De hecho, para generar C_2 se tiene que decidir cómo y si los términos involucrados en la resolución se transforman a variables.

Estos problemas se afrontaron por primera vez en el sistema de aprendizaje CIGOL [MB88a], que utiliza tres operadores para construir cláusulas a partir de ejemplos dados y del conocimiento previo. Las cláusulas construidas se presentan al supervisor (humano) para determinar si se aceptan, en cuyo caso se añaden al conocimiento previo.

3.5. Operadores para invertir la resolución

Como se explicó anteriormente, la idea básica de la *resolución inversa*, introducida como una técnica de generalización para PLI por Muggleton y Buntine [MB88a], es invertir la regla de resolución de la inferencia deductiva, por ejemplo, invertir el procedimiento de prueba con resolución-SLD para programas lógicos. Para lograr esto, se han desarrollado algunos operadores que invierten la resolución. Como se mencionó antes, el sistema de aprendizaje CIGOL [MB88a] utiliza tres operadores para construir cláusulas a partir

de ejemplos y el conocimiento previo (posiblemente vacío) dados.

Esta sección tiene como propósito exponer la formulación de los operadores que se tienen dentro de la PLI para invertir la resolución.

3.5.1. Los operadores 'V'

Los árboles de derivación pueden verse como un número de 'V's conectadas, donde cada 'V' representa un paso de resolución, como en la figura 3.1. La resolución puede derivar la cláusula en la base de la 'V' (la resolvente), dadas las cláusulas de los brazos. A diferencia de la resolución, la resolución inversa puede construir una de las dos cláusulas de los brazos, dadas la otra y la resolvente. Se introducirán primero, de manera informal, los dos operadores que llevan a cabo la inversión de la resolución, y que se conocen con el nombre de operadores 'V', a saber, *absorción* e *identificación.*

Para aplicar ya sea absorción o identificación es necesario definir una operación inversa a la resolución. Esto es, si se tienen dos cláusulas C y C_1, que cumplen ciertas condiciones, se debe construir una cláusula C_2 tal que $C = C_1 \cdot C_2$. En el caso de cláusulas proposicionales C_2 es única bajo la suposición de que C_1 y C_2 no comparten variables proposicionales. Para cláusulas de primer orden, C_2 no es única en general. Sin embargo, se pueden encontrar restricciones exactas para su construcción.

Identificación

Sea C_1 la cláusula que contiene la literal positiva en (3.1). El operador *identificación* construye C_1, dadas la resolvente C y la cláusula C_2 que contiene el átomo seleccionado. La idea es "identificar" parte del cuerpo de C_2 dentro del cuerpo de C por medio de un unificador adecuado. Entonces, al suponer que las cabezas de C y C_2 son unificables mediante θ y exactamente una literal l en el cuerpo de C_2 no ocurre en el cuerpo de C. El operador identificación puede construir C_1 al usar como cabeza l y, como cuerpo la parte de C que no se encuentra en C_2.

Por ejemplo, dadas:

$$C = p_1(a) \leftarrow q_1(a), q_2(a), q_3(a)$$
$$C_2 = p_1(a) \leftarrow p_2(a), q_3(a)$$

se tiene que $l = p_2(a)$ y que $C - C_2 = \{q_1(a), q_2(a)\}$. Por tanto,

$$C_1 = p_2(a) \leftarrow q_1(a), q_2(a)$$

es una posible solución para $C = C_1 \cdot C_2$.

Absorción

Sea C_2 la cláusula que contiene la literal negativa en (3.1). El operador *absorción* construye C_2, dadas la resolvente C y la cláusula C_1. El uso del término absorción es claro si se habla de cláusulas definidas: el cuerpo de C_1 se "absorbe" dentro del cuerpo de C (después de aplicar un unificador adecuado) y se remplaza con su cabeza. Por ejemplo, dadas:

$$\begin{aligned} C &= p_1(a) \leftarrow q_1(a), q_2(a), q_3(a) \\ C_1 &= p_2(x) \leftarrow q_1(x), q_2(x) \end{aligned}$$

el cuerpo de C_1, $\{q_1(x), q_2(x)\}$, se "absorbe" dentro del cuerpo de C después de la sustitución $\theta = \{x/a\}$ y se remplaza por la instancia de su cabeza $p_2(x)\theta$, esto da, como posible solución:

$$C_2 = p_1(a) \leftarrow p_2(a), q_3(a)$$

Claramente, C es la resolvente de C_1 y C_2. Otros sistemas que se basan en la inversión de la resolución usan nombres como *generalización* [Ban92] o *saturación* [RP90, Rou92] para este operador.

De hecho, de estos dos operadores, sólo el operador absorción está implementado en el sistema CIGOL. Lo mismo pasa, en general, para otros sistemas de PLI. En realidad, se tiene más interés en construir una generalización (cláusula C_2) del concepto en la base del operador 'V', mientras que la cláusula C_1 normalmente forma parte del conocimiento previo.

Ahora bien, si C_2 contiene la literal negativa, entonces, mediante una simple manipulación algebraica de (3.1) se obtiene

$$C_2 = (C - (C_1 - \{l_1\})\theta_1)\theta_2^{-1} \cup \{l_2\} \tag{3.2}$$

Se hace la suposición de que $(C_1 - \{l_1\})\theta_1$ y $(C_2 - \{l_2\})\theta_2$ no contienen literales en común. Ahora, puesto que $\theta_1\theta_2 = umg(\neg l_1, l_2)$, se sabe que $\neg l_1\theta_1 = l_2\theta_2$ y de esta manera se obtiene

$$l_2 = \neg l_1\theta_1\theta_2^{-1} \tag{3.3}$$

Al sustituir (3.3) en (3.2) se obtiene

$$C_2 = (C - (C_1 - \{l_1\})\theta_1)\theta_2^{-1} \cup \{\neg l_1\}\theta_1\theta_2^{-1}$$

$$= ((C - (C_1 - \{l_1\})\theta_1) \cup \{\neg l_1\}\theta_1)\theta_2^{-1} \tag{3.4}$$

Una *sustitución inversa* θ^{-1} consiste en remplazar términos por variables, de tal manera que para cada literal l, sucede que $l\theta\theta^{-1} = l$. Para calcular una sustitución inversa se debe decidir qué términos y subtérminos deben remplazarse por la misma variable, y cuáles por distintas variables. Como consecuencia, esto puede resultar en una explosión combinatoria.

Nótese que las restricciones para invertir la resolución descritas en (3.4) no se limitan a cláusulas definidas, sino que se cumplen para cláusulas arbitrarias de primer orden. Dadas sólo C y C_1 hay tres indeterminaciones en (3.4), a saber, l_1, θ_1 y θ_2^{-1}. Para reducir el número de indeterminaciones, se asume que C_1 es una cláusula unitaria, *i.e.*, $C_1 = \{l_1\}$. Esto simplifica (3.4) a

$$C_2 = (C \cup \{\neg l_1\}\theta_1)\theta_2^{-1} \tag{3.5}$$

y se reducen las indeterminaciones a θ_1 y θ_2^{-1}. Si se intenta construir θ_2^{-1} de manera no-determinista, de la definición de sustitución inversa, se debe decidir qué términos dentro de $(C \cup \{\neg l_1\}\theta_1)$ deben mapearse a variables distintas. Sean t un término en C y $t_1\theta_1$ en $\{\neg l_1\}\theta_1$ que se mapean a la misma variable v dentro de θ_2^{-1}, donde t_1 es un término que ocurre en $\{\neg l_1\}$. Esto implica que $t_1\theta_1 = t$. En este caso, claramente, la θ-diferencia de t_1 y t está definida y es un subconjunto de θ_1.

La suposición de que C_1 sea unitaria limita fuertemente el tipo de teorías que pueden aprenderse. De hecho, sólo pasos de resolución donde al menos una de las cláusulas es unitaria pueden invertirse. Otros sistemas (como IRES [RP90] y LFP2 [Wir89]) pueden eludir tal restricción.

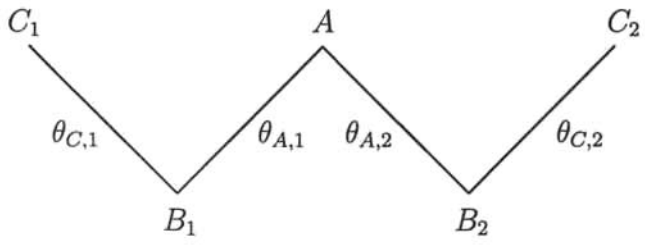

Figura 3.2: Dos pasos de resolución con una cláusula en común A

3.5.2. Invención de predicados: operadores 'W'

Al combinar dos resoluciones 'V' se obtiene una 'W' como se muestra en la figura 3.2. Al suponer que C_1 y C_2 pueden resolverse en una literal común l dentro de A para producir B_1 y B_2 respectivamente, los operadores 'W' construyen las cláusulas A, C_1 y C_2 dadas B_1 y B_2. Cuando l es negativa, el operador toma el nombre de *intra-construcción*, mientras que cuando l es positiva se llama *inter-construcción*. Nótese que la literal respecto a la que se resuelve no aparece en la resolvente, por lo tanto, las cláusulas A, C_1 y C_2 pueden contener una literal cuyo símbolo de predicado no aparece en B_1 ni en B_2. Es en ese sentido que el operador 'W' "inventa" un nuevo predicado.

De los operadores 'W', CIGOL sólo usa el operador de intra-construcción, el cual se discutirá en esta sección.

La 'W' en la figura 3.2 puede extenderse a múltiples cláusulas como sigue. Sean $BB = \{B_1, \ldots, B_n\}$ y $CC = \{C_1, \ldots, C_n\}$ dos conjuntos de cláusulas, y l una literal negativa en A tal que si l es la literal respecto a la que se

resuelve entonces $B_i = A \cdot C_i$, $1 \leq i \leq n$, al aplicar (3.1) se obtiene

$$B_i = (A - \{l\})\theta_{A,i} \cup (C_i - \{l_i\})\theta_{C,i} \tag{3.6}$$

donde $\theta_{A,i}\theta_{C,i}$ es el **umg** de $\neg l$ y l_i, *i.e.*,

$$\neg l\theta_{A,i} = l_i\theta_{C,i} \tag{3.7}$$

para simplificar suponga que

$$C_i = \{l_i\} \tag{3.8}$$

al sustituir en (3.6) se obtiene

$$B_i = (A - \{l\})\theta_{A,i} \tag{3.9}$$

De esta manera, se escoge $B = (A - \{l\})$ tal que $B = gme(\{B_1, \ldots, B_n\})$. Dada esa elección se tiene que

$$\theta_{A,i} = B -_\theta B_i \tag{3.10}$$

De (3.7) puede verse que $vars(\neg l) \subseteq \bigcup dom(\theta_{A,i})$ para $1 \leq i \leq n$. La elección más simple de $\neg l$ es $\neg l = p(v_1, \ldots, v_m)$ donde $\{v_1, \ldots, v_m\} = \bigcup dom(\theta_{A,i})$ y p es un nuevo símbolo de predicado. Claramente, ésta es una literal más general y por tanto $\neg l$ debe subsumir a l_i. De ahí que al unificar $\neg l$ y l_i se obtenga l_i pero de acuerdo con (3.7) al unificar $\neg l$ y l_i se obtiene $l_i\theta_{C,i}$, por lo tanto, $\theta_{C,i}$ debe ser la sustitución identidad. Al aplicar este hecho a (3.8) y (3.7) se tiene que

$$C_i = \{l_i\} = \{\neg l\}\theta_{A,i}. \tag{3.11}$$

Puesto que se está construyendo la definición de un nuevo predicado, vale la pena verificar que no se introducirán términos que ocurran en "posiciones"

"irrelevantes" dentro de la definición.

Se dice que, el término t_i ocurre en la *posición* i, $1 \leq i \leq n$, dentro de una expresión de la forma $f(t_1, \ldots, t_n)$ o $p(t_1, \ldots, t_n)$.

Una posición j es *irrelevante* en una literal l si el término que ocurre en dicha posición no comparte variables con el resto de los términos que ocurren en l.

Se dice que, una posición j es *irrelevante* en un conjunto de literales $L = \{l_1, \ldots, l_q\}$ que tienen el mismo símbolo de predicado y aridad, si j es irrelevante en l_k para toda k, $1 \leq k \leq q$.

Ejemplo Sean $C_1 = p(x_1, x_1, y_1)$ y $C_2 = p(x_2, s(x_2), y_2)$. La posición 3 es irrelevante en C_1 puesto que el término y_1 (que ocurre en la posición 3) no comparte variables con el resto de los términos en C_1. Análogamente, la posición 3 es irrelevante en C_2. De lo anterior se tiene que la posición 3 es irrelevante en $\{C_1, C_2\}$. ◁

Al remover las variables de $\neg l$ y los términos de C_i que ocurran en las posiciones que son irrelevantes en el conjunto $\{l_1, \ldots, l_n\}$, no se afecta la semántica de ningún predicado subsecuente que contenga a p [MB88a]. Por último, si se reescribe (3.9) se obtiene

$$A = B_i\theta_{A,i}^{-1} \cup \{l\}$$

$$= B \cup \{l\} \tag{3.12}$$

En el capítulo siguiente se mostrarán los algoritmos para absorción e intra-construcción en CIGOL.

3.6. Aplicaciones de la PLI

Se dice que algunas técnicas computacionales, como las redes neuronales, imitan el aprendizaje humano. En algún sentido las redes neuronales, junto con otras técnicas como la regresión estadística, se pueden ver como el uso de una forma de inferencia inductiva. No obstante, a diferencia de las redes neuronales, los algoritmos de la PLI generan reglas más fáciles de entender por la humanos. Esto hace a la PLI particularmente apropiada para la tarea de formación de teorías científicas.

El uso de un formalismo lógico relacional ha permitido la aplicación exitosa de sistemas de la PLI en campos en los cuales los conceptos que han de aprenderse no son fácilmente descritos en un lenguaje de atributo-valor. Desde la perspectiva de la ingeniería de software, también vale la pena notar la reducción en el tiempo de desarrollo y mantenimiento de sistemas construidos inductivamente. Esto hace pensar que las tecnologías basadas en inferencia inductiva desempeñarán un papel de importancia cada vez mayor en el desarrollo futuro de software. Estas aplicaciones incluyen el descubrimiento científico, la adquisición de conocimiento y asistentes de programación; un asistente de programación es una herramienta que ayuda al programador en el diseño e implementación de software. A continuación se dan algunos ejemplos.

3.6.1. Adquisición de conocimiento

El problema principal al construir sistemas expertos es la adquisición de conocimiento. La adquisición de conocimiento en los sistemas expertos toma mucho tiempo ya que es necesario observar y entrevistar expertos en cierto campo, los cuales a menudo son incapaces de formular su conocimiento de manera adecuada para su uso en una máquina. Este problema se conoce como *cuello de botella en la adquisición de conocimiento*. El aprendizaje de máquina contribuye a la apertura de este cuello de botella por medio del desarrollo de herramientas que ayuden y automaticen parcialmente el proceso de adquisición de conocimiento.

Por un lado, la PLI se interesa en el uso de lenguajes cada vez más poderosos en la descripción de conceptos, para facilitar el uso de un dominio específico de conocimiento. La PLI puede utilizarse para construir conocimiento previo para los sistemas expertos que se basan en modelos cualitativos, para los cuales las técnicas de aprendizaje proposicionales no son suficientes.

Por otro lado, un esfuerzo significativo se ha dedicado al desarrollo de un conjunto de técnicas de la PLI. En la comunidad de sistemas expertos se ha reconocido que es difícil resolver el problema de adquisición de conocimiento mediante el uso de una sola técnica, por lo cual es más apropiado tener un conjunto de técnicas y seleccionar una de ellas de acuerdo con la naturaleza del dominio y el tipo de conocimiento que se está investigando.

3.6.2. Descubrimiento de conocimiento en bases de datos

El descubrimiento de conocimiento en bases de datos se relaciona con la extracción de información (implícita, previamente conocida y potencialmente útil) que se almacena en bases de datos. Uno de los principales problemas en el descubrimiento de conocimiento en las bases de datos son los datos "ruidosos". El problema de inducir de manera intencional definiciones de relaciones al tomar en cuenta las dependencias sobre otras relaciones ha recibido recientemente una atención especial, y dicho problema se ha convertido en uno de los temas principales en la investigación actual en la PLI.

3.6.3. Descubrimiento de conocimiento científico

Existe un paralelismo entre el descubrimiento de teorías científicas y la construcción de conocimiento base para los sistemas expertos. El proceso de descubrimiento que realiza un científico y el proceso de adquisición de conocimiento que realiza un ingeniero del conocimiento usualmente dependen del número de observaciones o ejemplos que generalizan, al utilizar el conocimiento sobre el dominio. El resultado es una teoría inicialmente desconocida y que se puede considerar una nueva parte o una nueva formulación del conocimiento. En la generación de una nueva teoría, es útil realizar experimentos que podrían confirmar o rechazar la teoría actualmente creada. También, la teoría que se genera es, de hecho, una hipótesis que tiene que probarse. La PLI puede ayudar a los científicos e ingenieros del conocimiento en todas las fases de la construcción de una teoría.

La PLI puede utilizarse en los siguientes escenarios del descubrimiento científico:

- en la generación empírica o interactiva de teorías lógicas a partir de observaciones específicas,
- en la generación interactiva de experimentos cruciales durante el descubrimiento de una teoría lógica, y
- en demostrar, de manera sistemática, una teoría lógica dada, al sugerir experimentos que podrían rechazar la teoría.

3.6.4. Síntesis de programas

En varios lugares se dan argumentos en favor de una interpretación más rigurosa de la programación lógica inductiva como la intersección entre la inducción y la programación lógica [LD94], *i.e.*, PLI es la intersección de técnicas e intereses en inducción y programación lógica. Esta interpretación permite importar la PLI a la programación lógica, y exportar la programación lógica a técnicas inductivas en general, lo que permite una retroalimentación.

La síntesis y la transformación de programas lógicos intentan desarrollar técnicas para derivar programas eficientes a partir de una especificación (síntesis) o de una implementación ineficiente (transformación). Usualmente la síntesis y la transformación de programas lógicos emplean técnicas deductivas para llevar a cabo su propósito. Se mostrará una alternativa en la cual se puede usar inducción. Ésta tiene la ventaja de que la síntesis de programas lógicos a partir de especificaciones incompletas es plausible. Por ejemplo, en

el marco de la síntesis y transformación de programas, el predicado *ordenar* puede especificarse por:

$$S = ordenar(x, y) \leftrightarrow permutación(x, y), ordenado(y)$$

con las definiciones de *permutación* y *ordenado* en el conocimiento previo de la teoría B. El propósito sería mejorar la definición del ordenamiento por permutaciones hacia una definición de un predicado de ordenamiento más eficiente, a saber, quicksort, ordenamiento por inserción u ordenamiento de la burbuja. Las técnicas para lograr este propósito en la síntesis y la transformación de programas lógicos son básicamente deductivas, por ejemplo, doblado-desdoblado o técnicas de inducción matemática.

En el marco de la PLI, se puede atacar este problema al tomar:

$$\mathcal{E}^+ = \begin{cases} permutación(x, y) \leftarrow ordenar(x, y) \\ ordenado(y) \leftarrow ordenar(x, y) \\ ordenar(x, y) \leftarrow permutación(x, y), ordenado(y) \end{cases}$$

B incluye, entonces, *ordenado* y *permutación* y posiblemente otros predicados como *partición*, *concatenar*, *miembro*, etc.; y con restricciones sintácticas de modo que *permutación* y *ordenado* no se usarán en las hipótesis. Cualquier definición de *ordenado* que satisfaga las condiciones será equivalente a la especificación (*i.e.*, al ordenamiento por permutaciones) y por tanto correcto. También, dependiendo de los predicados del conocimiento previo de la teoría y de las restricciones sintácticas sobre las cláusulas que pueden inferirse, podrían derivarse distintas definiciones para *ordenar*. Por ejemplo, si *partición* se da en el conocimiento previo de la teoría, entonces se podría

inducir *quicksort* [MR94]. Esto muestra que la PLI se puede usar para derivar programas lógicos de especificaciones completas. Al relajar la evidencia, la PLI también puede inducir programas a partir de especificaciones incompletas, lo cual no es posible en la mayoría de los enfoques de síntesis de programas. Por ejemplo, la tercera cláusula en $\mathcal{E}^+$ podría remplazarse por pocos ejemplos. Una desventaja de utilizar las técnicas de la PLI para la síntesis de programas, es que no existe garantía de que la hipótesis inducida sea más eficiente en la práctica que la especificación original. Esto debe verificarse empíricamente.

Un caso extremo de la aplicación de la PLI a esta área es calcular la síntesis únicamente a partir de ejemplos. A pesar de que este cálculo automático se ha utilizado por muchos investigadores de la PLI, para probar e ilustrar sus técnicas, se cree que la síntesis de programas basada únicamente en ejemplos no es un camino prometedor para la PLI, debido a que se necesitarían muchos ejemplos antes de poder inducir la definición correcta.

Capítulo 4

El sistema de aprendizaje CIGOL

Este capítulo comienza con una descripción general del sistema inductivo CIGOL (LOGIC al revés) basado en resolución inversa descrito en [MB88a]. Después se presentan unos algoritmos no-deterministas que implementan los operadores 'V' y 'W', se enuncia y se demuestra el lema 4.3.1, relacionado con el operador 'W' que es contribución de los autores de este trabajo, hasta donde sabemos. Por último, se muestran algunos ejemplos del funcionamiento del sistema.

4.1. Introducción

Un concepto puede aprenderse solamente si se puede representar. Más que esto, un lenguaje apropiado para la representación facilita una descripción simple y elegante del concepto que se quiere aprender. En [MB88a] se describe un sistema llamado CIGOL que genera automáticamente su propio lenguaje de representación para representar de manera eficiente los concep-

tos que serán aprendidos. Primero, se provee a CIGOL con conocimiento previo adecuado en forma de cláusulas. Después se le presentan una secuencia de cláusulas unitarias aterrizadas que representan instancias positivas del concepto que será aprendido. Una vez que cada ejemplo se ha presentado, CIGOL presenta al usuario una secuencia de hipótesis que tienen una de las dos formas siguientes: ya sea "¿X es verdadera?" o bien "¿Cómo debería llamar al siguiente concepto?". El primer tipo de pregunta involucra una generalización que podría usarse para derivar ejemplos previos. Nótese que cada respuesta negativa del usuario agrega una instancia negativa al "conjunto de entrenamiento" de CIGOL que de otro modo consistiría únicamente de instancias positivas. El segundo tipo de pregunta permite la introducción de nuevos predicados que hacen posible una representación más clara del concepto que será aprendido. Puesto que estas formas de generalización están ligadas, la introducción de un nuevo predicado es seguida por una generalización adicional y/o una descomposición en subconceptos relacionados.

La generalidad del enfoque que se usa permite a CIGOL exhibir un número de facetas distintas del aprendizaje de máquina. De esta manera CIGOL se clasifica dentro de los sistemas que llevan a cabo

1. formación inductiva de conceptos,

2. inducción constructiva,

3. descubrimiento,

4. generalización de ejemplos al usar el conocimiento previo.

CIGOL es capaz de aprender no sólo conceptos estructuralmente simples, sino también fragmentos más complejos de programas. Los distintos mecanismos para formar hipótesis empleados por CIGOL se basan en invertir un paso de resolución.

4.2. Un algoritmo para absorción

Al recordar lo dicho en el capítulo anterior, los operadores 'V' construyen alguna de las cláusulas en los brazos de la 'V' dadas la cláusula en el otro brazo y la cláusula de la base. Esto se puede ver en la figura 3.1, donde la literal respecto a la que se resuelve es positiva $(+)$ en C_1 y negativa $(-)$ en C_2.

Como se dijo en el capítulo anterior el operador 'V' que implementa CIGOL es absorción, por lo cual únicamente se mostrará un algoritmo para dicho operador.

La construcción de las sustituciones en (3.5) sugiere un algoritmo no-determinista para construir C_2 dadas C y $C_1 = \{l_1\}$.

Una *partición* Π de un conjunto S es un conjunto de conjuntos tales que dos bloques (elementos) diferentes de Π no tienen elementos en común, y la unión de los bloques de Π es igual a S.

Algoritmo 4.2.1 *Un algoritmo no-determinista para calcular absorción.*

Entrada: C *y* $C_1 = \{l_1\}$ *cláusulas*

Sea $TP = \{(t,p) \mid t$ *es un término que ocurre en el lugar* p *en* $(C \cup \{\neg l_1\})\}$
Sea TP' *un subconjunto de* TP
Construir una partición Π *de* TP' *como sigue*
Sea cada bloque $B = \{(r,p_1),\ldots,(r,p_n)\} \cup \{(s,q_1),\ldots,(s,q_m)\}$ *de* Π
tal que s *subsume a* r *y todos los pares*
(r,p_i) *corresponden a términos de* C *y*
(s,q_j) *corresponden a términos de* $\{\neg l_1\}$
Sea $\theta_1 = \bigcup(s -_\theta r)$ *para todos los bloques* B
construidos como se indica arriba
Sea $\theta_2^{-1} = \{(r,\{p_1,\ldots,p_n,q_1,\ldots,q_m\})/v \mid$ *para todos los bloques* $B\}$
donde todas las v *son variables distintas que no ocurren en* $(C \cup \{\neg l_1\})$

Salida: $C_2 = (C \cup \{\neg l_1\}\theta_1)\theta_2^{-1}$

◇

Puesto que el algoritmo 4.2.1 es no-determinista, es necesario reformularlo como un algoritmo basado en búsqueda para poder ejecutarlo. En el siguiente capítulo se describirá la implementación del algoritmo 4.2.1 en CIGOL basada en búsqueda "eligiendo al mejor".

Para entender cómo opera el algoritmo 4.2.1, considérese el siguiente ejemplo.

Ejemplo Sean $C = men\_ig(x, s(s(x)))$ y $C_1 = men\_ig(y, s(y))$, donde $s(z)$ es el sucesor de z. Entonces,

$(C \cup \{\neg l_1\}) = men\_ig(x, s(s(x))) \leftarrow men\_ig(y, s(y))$ y

$TP = \{(x, \langle 1,1\rangle), (x, \langle 1,2,1,1\rangle), (s(x), \langle 1,2,1\rangle),$
$(s(s(x)), \langle 1,2\rangle), (y, \langle 2,1\rangle), (y, \langle 2,2,1\rangle), (s(y), \langle 2,2\rangle)\}$

Al elegir de manera no-determinista $TP' = \{(s(s(x)), \langle 1,2\rangle), (s(y), \langle 2,2\rangle)\}$ y tomar la partición trivial de TP', $\{\{(s(s(x)), \langle 1,2\rangle), (s(y), \langle 2,2\rangle)\}\}$ nos queda

$\theta_1 = \{y/s(x)\}$,

$\theta_2^{-1} = \{(s(s(x)), \{\langle 1,2\rangle, \langle 2,2\rangle\})/w\}$ y

$C_2 = men\_ig(x, w) \leftarrow men\_ig(s(x), w)$ ◁

Nótese que esto satisface $C = C_1 \cdot C_2$.

Sin embargo, esto no es cierto para cualquier C y C_1 [NCF91]. Por ejemplo, sean $C = q(f(a), f(b))$, $C_1 = p(f(u), f(v))$, entonces, puede construirse una cláusula $C_2 = q(x, y) \leftarrow p(x, y)$ con el algoritmo anterior, con $\theta_1 = \{u/a, v/b\}$. La resolvente de C_1 y C_2 es $q(f(u), f(v))$, que es más general que C respecto a θ-subsunción. Del ejemplo anterior, se puede concluir que el algoritmo 4.2.1 puede construir una cláusula C_2, tal que C no es la resolvente de C_1 y C_2.

Esto no es lo único que el algorimto 4.2.1 no puede hacer. Si se considera $C_1 = p(x)$, $C = q(v, g(v))$ y $C_2 = q(v, g(v)) \leftarrow p(h(v))$, C_2 no se puede construir con el algoritmo 4.2.1, a pesar de que $C = C_1 \cdot C_2$.

De igual manera, si $C_1 = p(x, y)$, $C = q(u, f(w))$ y $C_2 = q(u, f(w)) \leftarrow p(u, u)$. Es claro que $C = C_1 \cdot C_2$. Sin embargo, no es posible construir C_2 con el algoritmo, pues las ocurrencias (s, q_j) de C_1 que están en un bloque deberían tener el mismo término s. En el ejemplo, para poder sustituir x e y por la misma variable u en C_2, x e y tendrían que estar en el mismo bloque, lo que no es posible ya que x e y son diferentes.

Mas allá de lo mostrado con estos ejemplos, quedan las siguientes interrogantes:

- ¿Qué tipo de subconjuntos TP' de términos deben usarse para construir las particiones, y cómo deben ser éstas dado dicho subconjunto?
- Puesto que distintas particiones pueden inducir distintas cláusulas. ¿Será posible ver que una cláusula inducida es más general que otra, solamente comparando la partición asociada?

En [NCF91] se formaliza la noción de sustitución inversa por medio de *mapeos consistentes* de términos, introducidos (bajo el nombre de *funciones consiguientes*) por Shan-Hwei Nienhuys-Cheng [NC90]. Además, se propone un nuevo algoritmo de absorción que puede construir todas las C_2 tales que $C = C_1 \cdot C_2$. El algoritmo se basa en restricciones sobre el subconjunto TP', y la construcción de las particiones sobre este conjunto.

4.3. Un algoritmo para intra-construcción

Como se mencionó en el capítulo anterior, al combinar dos resoluciones 'V' se obtiene una 'W' como en la figura 3.2. Al suponer que C_1 y C_2 pueden

resolverse en una literal común l contenida en A para generar los resolventes B_1 y B_2 respectivamente. Los métodos descritos en este capítulo construyen las cláusulas A, C_1 y C_2 dadas B_1 y B_2. Del capítulo anterior se sabe que existen dos tipos de operadores 'W' dependiendo del signo de l, a saber, intra-construcción cuando l es negativa e inter-construcción si l es positiva. Puesto que el único operador 'W' que utiliza CIGOL es intra-construcción, en esta sección se dará un algoritmo que implementa dicho operador.

Las ecuaciones (3.10), (3.11), (3.12) son la base del siguiente algoritmo no-determinista.

Algoritmo 4.3.1 *Un algoritmo no-determinista para calcular intra-construcción*

Entrada: $BB = \{B_1, \ldots, B_n\}$ *un conjunto de cláusulas*

Sea $B = gme(BB)$ *y* $\theta_{A,i} = B -_\theta B_i$ *para* $1 \leq i \leq n$

Se construye $\neg l = p(v_1, \ldots, v_m)$ *donde* $\{v_1, \ldots, v_m\} = \bigcup dom(\theta_{A,i})$ *y* p *es un nuevo símbolo de predicado*

Se eliminan de $\neg l$ *las variables que ocurren en las posiciones irrelevantes en el conjunto* $\{\neg l\theta_{A,1}, \ldots, \neg l\theta_{A,n}\}$

Salida: $A = B \cup \{l\}$ *y* $C_i = \{\neg l\}\theta_{A,i}$

◇

Ejemplo Dada la entrada

$$BB = \{B_1, B_2\}$$

$$B_1 = min(x_1, [s(x_1) \mid y_1]) \leftarrow min(x_1, y_1)$$

$$B_2 = min(x_2, [s(s(x_2)) \mid y_2]) \leftarrow min(x_2, y_2)$$

Entonces,

$$B = gme(B_1, B_2) = min(x, [s(y) \mid z]) \leftarrow min(x, z)$$

Al tomar las *θ-diferencias* se tiene que

$$B -_\theta B_1 = \theta_{A,1} = \{x/x_1, y/x_1, z/y_1\}$$

$$B -_\theta B_2 = \theta_{A,2} = \{x/x_2, y/s(x_2), z/y_2\}.$$

Puesto que ninguno de los términos por los que se sustituye z comparte variables con ninguna de las otras sustituciones, dichos términos ocurren en una posición irrelevante en las definiciones de C_1 y C_2. Por lo tanto pueden eliminarse de l.

Entonces,

$A = min(x, [s(y) \mid z]) \leftarrow min(x, z), p(x, y) = B \cup \{l\}$

$C_1 = p(x_1, x_1) = \neg l\theta_{A,1}$

$C_2 = p(x_2, s(x_2)) = \neg l\theta_{A,2}$ ◁

El siguiente lema, presentado de manera original por los autores de esta tesis (hasta donde se sabe), establece que el algoritmo 4.3.1 es correcto.

Lema 4.3.1 *Sean* $BB = \{B_1, \ldots, B_n\}$ *un conjunto de cláusulas y* $B = gme(BB)$. *Sean*

$$\theta_{A,i} = B -_\theta B_i, 1 \leq i \leq n$$

y

$$A = B \cup \{\neg l\}$$

donde $\neg l = p(v_1, \ldots, v_n)$ *y* $\{v_1, \ldots, v_n\} = \bigcup dom(\theta_{A,i})$. *Entonces,* $A \cdot C_i = B_i$, *donde* $C_i = \{l_i\}$, *y* $l_i = \neg l\theta_{A,i}$.

Demostración De la definición de resolución se tiene que

$$A \cdot C_i = (A - \{\neg l\})\theta_1 \cup (C_i - \{l_i\})\theta_2 \tag{4.1}$$

tal que $\theta_1\theta_2$ es el umg de $\neg l$ y l_i, *i.e.*, $\neg l\theta_1 = l_i\theta_2$.

Puesto que $l_i = \neg l\theta_{A,i}$, entonces $umg(\neg l, l_i) = \theta_{A,i} = \theta_1\theta_2$. Por lo tanto $\theta_1 = \theta_{A,i}$ y $\theta_2 = \varepsilon$. Al aplicar esto a (4.1) se obtiene:

$$\begin{aligned} A \cdot C_i &= (A - \{\neg l\})\theta_{A,i} \cup (C_i - \{l_i\})\varepsilon \\ &= B\theta_{A,i} \cup \emptyset \\ &= B\theta_{A,i} \\ &= B_i \end{aligned}$$

∎

4.4. Operador de truncamiento

Los operadores que se han descrito hasta ahora no tratan el caso en que el paso de resolución considerado, tiene como resolvente la cláusula vacía. El operador de truncamiento que se describe en esta sección trata este caso.

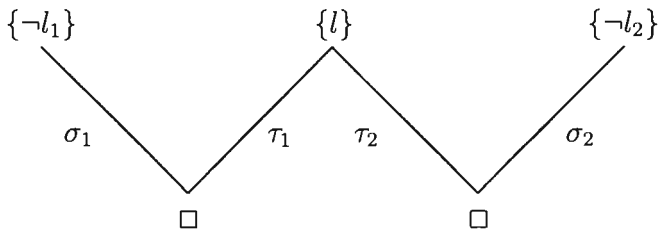

Figura 4.1: Operador de truncamiento

La figura 4.1 muestra esquemáticamente este operador. Dadas las cláusulas unitarias $\{\neg l_1\}$ y $\{\neg l_2\}$ el operador de truncamiento construye l. En [MB88b] se muestra que es suficiente para la completitud del operador cubrir sólo el caso en que τ_1 y τ_2 son la sustitución identidad, en cuyo caso l subsume a ambas $\neg l_1$ y $\neg l_2$. Más aún, sin pérdida de completitud la operación puede hacerse de manera determinista si $l = gme(\neg l_1, \neg l_2)$. En CIGOL esta operación se extendió para tratar conjuntos arbitrariamente grandes de cláusulas unitarias.

4.5. Sesiones de CIGOL

En esta sección se ilustrará el modo en que opera CIGOL mediante sesiones en las que aprende e inventa diversos predicados.

4.5.1. Pertenencia a una lista

En las figuras 4.2 y 4.3 se muestra una sesión en la que CIGOL aprende la relación de "ser miembro" de una lista, entre una lista y un elemento.

La entrada proporcionada por el supervisor se encuentra después de `!-`. En la primera línea CIGOL se llama desde Prolog. El primer ejemplo establece que `a` es miembro de la lista `[a,b,c]`. Nótese que a pesar de que ésta es una cláusula unitaria aterrizada, los ejemplos pueden tomar la forma de cláusulas arbitrarias no aterrizadas. Puesto que no hay otros ejemplos, CIGOL espera más información. Dado el segundo ejemplo, CIGOL aplica el operador de truncamiento y pregunta si la gme, `m(A,[A | B])`, es verdadera, *i.e.*, ¿Algo es miembro de una lista si es el primer elemento de esa lista? Este tipo de preguntas formuladas por CIGOL deben considerarse universalmente cuantificadas. El número `(-3)` después de la palabra `TRUNCATION` representa la compactación producida en términos del "tamaño" (ver siguiente capítulo). El supervisor indica que la generalización es verdadera. En los siguientes dos ejemplos el supervisor muestra a CIGOL que algo es miembro de una lista si es el segundo o el tercer elemento de dicha lista. CIGOL utiliza el operador de absorción para obtener la cláusula general recursiva `m(A,[B | C]):- m(A,C)`. CIGOL usa un demostrador de teoremas basado en resolución-SLD de profundidad acotada para demostrar al usuario que `m(A,[B |C]):- m(A,C)` no puede derivarse de las anteriores.

4.5.2. Problema del arco

El problema del arco consiste en describir la estructura de un arco. En las figuras 4.4 y 4.5 se muestra una sesión de CIGOL en la que el sistema inventa un predicado auxiliar a partir de un conocimiento previo para describir el arco.

La estructura específica del arco que se quiere aprender en este ejemplo tiene dos columnas en cada lado. El arco está coronado con una viga. Los arcos se representan simbólicamente por términos que consisten en tuplas de la forma `(Column1, beam, Column2)`. En la sesión de las figuras 4.4 y 4.5, el usuario consulta un archivo de cláusulas llamado "`arch`". El supervisor entonces da un nuevo ejemplo de un arco. Esto conlleva tres aplicaciones de truncamiento no exitosas seguidas de la aplicación de intra-construcción. El operador de intra-construcción sugiere un nuevo predicado, `p15`, y muestra la llamada de `p15` con cuatro instancias del nuevo predicado. El supervisor reconoce y nombra `p15` como `column`. Ahora CIGOL aplica el operador de absorción para producir una generalización recursiva de dos de las cláusulas para `column`, las cuales confirma el supervisor. CIGOL puede seguir infiriendo el resto de la descripción de arco, pero aquí sólo se muestra esta parte por razones ilustrativas.

```
| ?- cigol.
!- m(a,[a,b,c]).
[Truncating m(a,[a,b,c])]
[Time spent is 0]
[Developing absorption roots]
[Developing intra-construction roots]
[Time spent is 0]
I know:
m(a,[a,b,c]).
!- m(1,[1]).
[Truncating m(1,[1])]
[Time spent is 0]
[Time spent is 0]
[Trying to disprove [m(A,[A|B])]]
[Trying to prove [m(A,[A|B])]]
TRUNCATION: (-3)
Is m(A,[A|B]) always true? y.
[Developing absorption roots]
[Developing intra-construction roots]
[Time spent is 0]
I know:
m(A,[A|B]).

  ....
```

Figura 4.2: CIGOL aprende pertenencia de una lista.

```
....

!- m(A,[B,A|C]).
[Truncating m(A,[B,A|C])]
[Developing absorption roots]
[Developing intra-construction roots]
I know:
m(A,[A|B]).
m(A,[B,A|C]).
!- m(A,[B,C,A|D]).
[Truncating m(A,[B,C,A|D])]
[Developing absorption roots]
[Developing intra-construction roots]
[Trying to disprove [(m(A,[B|C]):-m(A,C))]]
[Trying to prove [(m(A,[B|C]):-m(A,C))]]
ABSORPTION: (-12)
        New clauses:[(m(A,[B|C]):-m(A,C))]
cover new facts: [m(A,[B,C,D,A|E])] ...
Are new clauses always true? y.
[Developing absorption roots]
[Developing intra-construction roots]
[Time spent is 0]
I know:
m(A,[A|B]).
m(A,[B|C]):-m(A,C).
```

Figura 4.3: CIGOL aprende pertenencia de una lista (continuación).

```
?- cigol.
!- [arch].
[consulting arch]
I know:
        arch(([ ],beam,[ ])).
        arch(([block],beam,[block])).
        arch(([brick],beam,[brick])).

!- arch(([block,brick],beam,[block,brick])).
[Truncating arch(([block,brick],beam,[block,brick]))]
TRUNCATION: (-18)
Is arch(([block|A],beam,[block|A])) always true? n.
TRUNCATION: (-27)
Is arch(([A|B],beam,[A|B])):-not(nonarch1(A,B)) always true? n.
TRUNCATION: (-27)
Is arch(([A|B],beam,[A|B])):-not(nonarch2(A,B)) always true? n.
TRUNCATION: (-34)
Is arch((A,beam,A)):-not(nonarch3(A)) always true? n.

  ....
```

Figura 4.4: Problema del arco.

```
   ....
INTRA-CONSTRUCTION
        arch((A,beam,A)):-p15(A).
        p15([ ]).
        p15([block]).
        p15([block,brick]).
        p15([brick]).

What shall I call p15? column.

ABSORPTION: (-8)

   New clauses:[(column([block|A]):-column(A))]
   cover new facts:
    [column([block,block]),column([block,block,block,brick]),
     column([block,block,brick])] ...

Are new clauses always true? y.
I know:
        arch((A,beam,A)):-column(A).
        column([ ]).
        column([block|A]):-column(A).
        column([brick]).
        not((arch((A,beam,A)):-not(nonarch3(A)))).
        not(arch(([block|A],beam,[block|A]))).
        not((arch(([A|B],beam,[A|B])):-not(nonarch1(A,B)))).
        not((arch(([A|B],beam,[A|B])):-not(nonarch2(A,B)))).
```

Figura 4.5: Problema del arco (continuación).

Capítulo 5

Implementación de CIGOL

Este capítulo comienza con una descripción a alto nivel del sistema CIGOL [MB88a]. Después se dan algunos fundamentos teóricos [Bun88] [MB88a] que permiten introducir el concepto de "redundancia lógica". Dicho concepto es la base para la reducción de los programas generados por CIGOL. Además, los autores de este trabajo proponen un teorema que permite formular un algoritmo para realizar dicha reducción. Finalmente, se dan los fundamentos teóricos [Pea84] de la búsqueda que permite atacar el no-determinismo de los algoritmos 4.2.1, que calcula absorción, y 4.3.1, que calcula intra-construcción.

Parte del material de este capítulo se basa en el análisis que los autores de este trabajo hicieron del código de CIGOL. Tal es el caso de la formulación del teorema 5.2.4 y del algoritmo 5.3.2.

5.1. Descripción a alto nivel de CIGOL

CIGOL es un sistema interactivo codificado en Prolog que construye de manera incremental programas lógicos a partir de ejemplos presentados por un supervisor humano. La forma de interactuar del sistema es haciendo preguntas al supervisor para verificar las generalizaciones hechas por los operadores de truncamiento y absorción. CIGOL además pide al supervisor el nombre para cada nuevo predicado introducido por el operador de intra-construcción. El siguiente algoritmo [MB88a] da una descripción general de CIGOL.

Algoritmo 5.1.1

Sean $P = \emptyset$ *y* $N = \emptyset$

Forever Do

Sea $I = term\text{-}indexing(P)$

Lee un ejemplo E *del supervisor*

$E' = best\text{-}agreed\text{-}truncation(E, P, N, I)$

$inverse\text{-}prove(\{E'\}, P, N, I)$

$reduce\text{-}clauses(P, N, I)$

Repeat

◇

En el algoritmo de arriba, P es el programa que se está construyendo. El conjunto de cláusulas N actúa como un conjunto de contraejemplos de los predicados que se están infiriendo, y se compone de cláusulas propuestas por los procedimientos *best-agreed-truncation* e *inverse-prove*, que el supervisor rechazó. El procedimiento *reduce-clauses* remueve cláusulas redundantes al utilizar un algoritmo basado en el concepto de redundancia de

Buntine [Bun88] como se explicará más adelante en este capítulo.

El procedimiento *term-indexing*(P) construye una indexación I a partir del programa P que consta de tres partes: TI para truncamiento, AI para absorción e II para intra-construcción.

En TI se tiene una lista de las cláusulas unitarias de P clasificadas por el símbolo de predicado y la aridad de éste. Cada clase tiene un representante, que es la cláusula más general de esa clase de acuerdo con θ-subsunción.

En AI se tienen, por una parte, los términos de todas las cláusulas en P clasificados por su símbolo de función y número de argumentos. Por otra parte, se tiene una lista de pares de modo que para cada cláusula unitaria y para cada término que ocurra en ella existe un par cuyo primer elemento es dicho término, y el segundo es una lista de los términos que ocurren en el resto de las cláusulas de P y que θ-subsumen a ese término.

Por último, en II se tienen todas las cláusulas de P clásificadas por el símbolo de predicado y la aridad de las literales de la cláusula, respetando el signo, y el orden en que aparecen en el caso de las literales que pertenecen al cuerpo. En cada clase los términos que ocurren en las cláusulas pertenecientes a dicha clase, se clasifican de acuerdo con la posición en que aparecen dentro de las literales.

El operador de truncamiento se aplica únicamente a ejemplos nuevos dados como cláusulas unitarias. El procedimiento *best-agreed-truncation* genera un subconjunto P' de las cláusulas unitarias en P, las cuales se indexan al usar TI, y una cláusula unitaria E' tal que E' es la generalización más específica de $P' \cup \{E\}$. P' se encuentra al usar búsqueda "eligiendo-al-mejor", como se explicará más adelante en este capítulo.

Los operadores de absorción e intra-construcción se aplican en el procedimiento *inverse-prove*. Este procedimiento trabaja de manera similar al mecanismo de reducción de metas de Prolog. La diferencia es que mientras Prolog construye refutaciones de las metas a partir de un programa dado, CIGOL utiliza resolución inversa para construir una refutación de "abajo hacia arriba" a partir de los ejemplos truncados. Las cláusulas en las hojas de un árbol de derivación de CIGOL se añaden a P. Las cláusulas en los nodos internos del árbol de derivación se borran de P, ya que estas cláusulas son implicadas por P. El algoritmo que describe el procedimiento *inverse-prove* es el siguiente.

Algoritmo 5.1.2 *El procedimiento inverse-prove*

Entrada: Metas Gs, programa P, contraejemplos N e I
ForEach *meta G en Gs*
    $NewGs = best\text{-}agreed\text{-}operator(G, P, N, I)$
    $inverse\text{-}prove(NewGs, P, N, I)$
Repeat

◇

El procedimiento *best-agreed-operator* implementa los algoritmos 4.2.1 y 4.3.1 usa búsqueda eligiendo al mejor como se explicará más adelante. Este procedimiento trabaja de la misma manera que *best-agreed-truncation* excepto que usa AI e II para indexar los términos y literales de P y la búsqueda tiene como objetivo encontrar el operador que se va a aplicar, a saber, absorción o intra-construcción.

5.2. Un modelo de redundancia

En esta sección se profundizará en los fundamentos teóricos [Bun88] del procedimiento *reduce-clauses* y el demostrador de teoremas usado por CIGOL.

Para ayudar a la formulación de una noción de redundancia, se utiliza el modelo de generalización propuesto por Buntine ($\succeq_P^B$) [Bun88], que se presentó en el capítulo 3.

En esta sección se utlizará la palabra *cláusula* para denotar *cláusula definida.*

Definición Una cláusula C es *equivalente* a una cláusula D *respecto a* un programa P, si $C \succeq_P^B D$ y $D \succeq_P^B C$. Esto se denota como $C \equiv_P^B D$.

"$\equiv_P^B$" es una relación de equivalencia puesto que "$\succeq_P^B$" es transitiva y reflexiva. La *clase de equivalencia* bajo "$\equiv_P^B$" de C, denotada como $[C]_P^B$, es el conjunto formado por todas las cláusulas D tales que $C \equiv_P^B D$. De manera natural, estos conceptos pueden extenderse a programas.

El siguiente teorema da pauta a un algoritmo para probar la subsunción generalizada.

Teorema 5.2.1 [Bun88]*(**Prueba para "$\succeq_P^B$")***

1. *Sean $C_1, \ldots, C_n$ y $D_1, \ldots, D_m$ cláusulas y P un programa lógico. $\{C_1, \ldots, C_n\} \succeq_P^B \{D_1, \ldots, D_m\}$ si y sólo si para toda j, $1 \leq j \leq m$ existe i, $1 \leq i \leq n$ y $C_i \succeq_P^B D_j$.*

2. *Sean C y D dos cláusulas que no comparten variables y P un programa lógico. Sea θ una sustitución que aterriza las variables de D al usar nuevas constantes que no ocurren en C, D, ni P. $C \succeq_P^B D$ si y sólo si, para alguna sustitución σ, $C_{cabeza}\sigma$ es idéntica a D_{cabeza} y*

$$P \cup \{D_{cuerpo}\theta\} \models C_{cuerpo}\sigma\theta \tag{5.1}$$

De acuerdo con el teorema 5.2.1, para comparar dos conjuntos de cláusulas es suficiente comparar cláusulas individuales en los conjuntos. Cada cláusula en un conjunto más específico deberá ser subsumida por alguna cláusula en el conjunto más general. Al comparar dos cláusulas, (5.1) puede probarse por un sistema de programación lógica.

Cualquier prueba de subsunción generalizada es sólo semidecidible. Esto es, la terminación de la prueba se garantiza sólo cuando realmente $C \succeq_P^B D$. Sin embargo, si el programa P no contiene recursión, se garantiza la terminación [Bun88].

El siguiente teorema establece la conexión específica entre la subsunción generalizada e implicación lógica.

Teorema 5.2.2 [Bun88] *Sean D cualquier cláusula que no sea tautología y P un programa lógico. $P \models D$ si y solo si $C \succeq_P^B D$ para alguna cláusula C en P, esto es, $P \succeq_P^B D$.*

Corolario 5.2.3 *Si $C \succeq_P^B D$, entonces, $P \cup \{C\} \models P \cup \{D\}$.*

Como consecuencia del corolario anterior, cuando una cláusula en un programa lógico se remplaza por una cláusula más general, al menos las mismas metas tendrán éxito en el nuevo programa.

Se considerará un modelo de redundancia llamado *redundancia lógica*, introducido por Buntine [Bun88]. Con este modelo, la redundancia se determina según las reglas ideales de la lógica, independientemente de la implementación particular del sistema de conocimiento o del sistema de programación lógica del que se trate.

Antes de considerar la redundancia lógica con mayor detalle, se considerarán las cuestiones que ésta ignora. En términos prácticos, una regla o alguna condición en una regla es redundante en un sistema de conocimiento particular si al removerla, el sistema puede continuar funcionando como debe y continuará haciéndolo después de que se haga cualquier extensión factible al conocimiento del sistema. Además de la forma del conocimiento disponible, es necesario considerar varios aspectos cuando se determina la redundancia:

1. El tipo de preguntas que se hacen al sistema por el supervisor.

2. La completitud del componente de razonamiento del sistema, esto es dados recursos suficientes, el sistema es capaz de generar todas las conclusiones posibles.

3. Las limitaciones de recursos y tiempo bajo las que opera el sistema.

4. El subconjunto de la base de conocimiento del sistema que es correcto o completo.

Con redundancia lógica se hacen las siguientes suposiciones:

1. Cualquiera de las preguntas posibles pueden hacerse al sistema.

2. El componente de razonamiento del sistema es completo.

3. El sistema tiene recursos ilimitados.

4. Ninguna parte de la base de conocimento es completa, sin embargo un subconjunto determinado es correcto.

5.2.1. Redundancia lógica

En un programa lógico existen al menos dos tipos de redundancia lógica. Una cláusula puede ser reduntante dentro de un programa, así como también un átomo puede serlo dentro de una cláusula.

Una cláusula D es *redundante* en un programa lógico P si $P - \{D\} \models D$. Tal cláusula puede removerse del programa lógico y una implementación particular tendrá exactamente las mismas metas exitosas, si se supone que se tienen recursos ilimitados (para construir las refutaciones de todas las metas

que se tenían antes de remover D) y el procedimiento de prueba es completo. Por ejemplo, la tercera y la cuarta cláusulas en el programa

$$member(x, [x \mid y])$$
$$member(x, [z \mid y]) \leftarrow member(x, y)$$
$$member(x, [z, x])$$
$$member(1, [3, 2, 1])$$

son redundantes porque ambas son consecuencia lógica de las primeras dos cláusulas del programa. Del teorema 5.2.2, si una cláusula es redundante, siempre existe otra cláusula en el programa lógico que puede considerarse principalmente responsable de hacerla redundante. Varias cláusulas como ésas pueden coexistir.

Para verificar si una cláusula D es redundante en un programa lógico P, en CIGOL se usa un demostrador de teoremas de profundidad acotada que construye una refutación-SLD de $P \cup \{D_{cabeza}\}\theta$, donde θ es una sustitución que aterriza las variables que ocurren en D_{cabeza} con constantes que no ocurren ni en D ni en P. La cota del demostrador está dada por el máximo de las "profundidades" de las metas. Además de dicha cota, se tiene un límite de tiempo para construir la refutación-SLD de modo que si se rebasa el tiempo establecido la demostración falla.

Definición La *profundidad* de una literal $l = p(t_1, \ldots, t_n)$, $profundidad(l)$ se define como $1 + max(\{profundidad(t_1), \ldots, profundidad(t_n)\})$, donde la profundidad de un término t se define como sigue:

- Si t es una variable, entonces, $profundidad(t) = 0$.
- Si t es una constante numérica, entonces, $profundidad(t) = 1 + t$.
- Si t es un término de la forma $f(t_1, \ldots, t_n)$, entonces, $profundidad(t) = 1 + max(\{profundidad(t_1), \ldots, profundidad(t_n)\})$.

En CIGOL el procedimiento *reduce-clauses* utiliza un algoritmo, similar al algoritmo de reducción que aparece en [Bun88], para eliminar cláusulas redundantes en el programa. El siguiente teorema lo proponen los autores de esta tesis basándose en el análisis del código de CIGOL y en el algoritmo de reducción antes mencionado. El teorema provee el algoritmo de reducción usado en CIGOL.

Teorema 5.2.4 ***(Algoritmo de reducción)*** *El siguiente algoritmo acepta como entrada un programa lógico P. Al suponer que todas las verificaciones de subsunción generalizada terminan, el programa P' tiene las mismas metas exitosas que P cuando termine el algoritmo.*

Paso 1. Hacer P' igual a P.
Paso 2. Para cada cláusula D en P',
si $P' - \{D\} \succeq_P^B D$ hacer P' igual a $P' - \{D\}$.

Demostración Recuérdese que si M es una meta exitosa en P, entonces, $P \models \exists M$. Por tanto basta demostrar que los modelos de P y P' son los mismos, *i.e.*, $P \models P'$ y $P' \models P$.

Puesto que se tiene que $P' \subseteq P$, entonces, $P \models P'$.

Por otro lado se tiene que $P' \succeq_P^B (P - P')$, por tanto para toda $D \in (P - P')$ existe $C \in P'$ tal que $C \succeq_P^B D$.

Sean $D \in (P - P')$ y $C \in P'$ tales que $C \succeq_P^B D$ entonces por el corolario 5.2.3 $P' \cup \{C\} \models P' \cup \{D\}$. Como D es arbitraria se tiene que lo anterior es cierto para toda $D \in (P - P')$, por lo tanto $P' \models P' \cup (P - P')$, es decir, $P' \models P$. ∎

Dentro de las cláusulas, un segundo tipo de redundancia es posible. Si las cláusulas se construyen mediante un algoritmo sin la consideración apropiada para la semántica subyacente, por ejemplo, si las cláusulas se enumeran durante la inducción, éstas pueden contener átomos que no contribuyen de manera efectiva al éxito de la cláusula. En CIGOL este tipo de redundancia no se elimina.

Una demostración de los resultados que aparecen en esta sección se encuentran en [Bun88].

5.3. Búsqueda eligiendo al mejor

En esta sección se detallará una versión del algoritmo general de búsqueda [Pea84] en el que se basa CIGOL.

Existen ejemplos en los cuales la familiaridad con el dominio del proble-

ma permite considerar ciertas direcciones de búsqueda más prometedoras que otras. Esto se logra al utilizar conocimiento más allá del establecido dentro de las definiciones del estado y de las operaciones, que producen conjuntos más refinados de soluciones potenciales. Ahora se verá cómo este conocimiento heurístico puede utilizarse en la formalización de una búsqueda sistemática.

El escenario más natural para utilizar la información heurística es en la decisión de qué nodo debe ser el siguiente en expandirse. Lo que diferencia "eligiendo-al-mejor" del resto de las estrategias es que esta estrategia debe escoger el mejor de entre todos los nodos encontrados hasta el momento sin importar en qué lugar dentro del árbol parcial, construido hasta el momento, se encuentre dicho nodo. Para explicar esto se puede utilizar la metáfora de Patrick Winston [Win77] "eligiendo-al-mejor trabaja como un equipo de alpinistas que buscan el punto más alto en una sección de la montaña y que mantienen contacto por radio. El subequipo que se encuentra más alto es siempre el que se mueve y se divide en sub-subequipos cada vez que hay una bifurcación en el camino". Para mejorar esta metáfora se debe añadir que los alpinistas emplean varios instrumentos para calcular la altura actual de todos los subequipos y la altitud de los caminos que se encuentran más adelante, y que los instrumentos ocasionalmente se corrompen por la interferencia. La interferencia representa el hecho de que la información heurística, por su naturaleza, ocasionalmente puede ser engañosa.

La promesa de un nodo puede estimarse de varias maneras. Una de ellas consiste en evaluar la dificultad de resolver un subproblema representado

por un nodo. Otra forma es estimar la calidad del conjunto de soluciones candidatas codificadas por el nodo, esto es, aquellos contenidos en el camino encontrado que lleva a ese nodo. Una tercera alternativa es considerar la cantidad de información que se espera obtener por la expansión de un nodo dado y la importancia de esta información para guiar en general la estrategia de búsqueda. En todos estos casos, la promesa de un nodo se estima numéricamente mediante una *función heurística de evaluación* $f(n)$ la cual, en general, dependerá de la descripción de n, la descripción del objetivo, de la información reunida por la búsqueda hasta ese punto, y más importante, de cualquier conocimiento extra sobre el dominio del problema.

Varias estrategias de eligiendo-al-mejor difieren en el tipo de función de evaluación que emplean. El algoritmo que se describe a continuación es igual para todas las estrategias ya que no existen restricciones sobre la naturaleza de $f(\cdot)$. Se asume únicamente que el espacio de búsqueda es una gráfica del espacio de estados, que el nodo que se selecciona para la expansión es aquel que tiene la menor $f(\cdot)$, y que cuando dos caminos se dirigen al mismo nodo el mayor $f(\cdot)$ se descarta. Este algoritmo se llama eligiendo-al-mejor.

Algoritmo 5.3.1 *Algoritmo eligiendo-al-mejor*

1. *Poner los nodos iniciales S en un conjunto que se llama $OPEN$ de nodos sin expandir.*

2. *Si $OPEN$ es vacío, terminar con falla; no existe solución.*

3. *Quitar de $OPEN$ un nodo n en el cual f es mínima (romper empates arbitrariamente), y colocarlo en un conjunto que se llama $CLOSED$*

donde se encuentran los nodos que ya se expandieron.

4. *Si n es un nodo objetivo, salir exitosamente con la solución obtenida mediante la localización del camino a lo largo de los apuntadores del objetivo al nodo inicial.*

5. *Expandir el nodo n, para generar todos sus sucesores con apuntadores hacia atrás a n.*

6. *Para cada sucesor n' de n:*

   a) *Calcular $f(n')$.*

   b) *Si n' no estaba en OPEN ni en CLOSED, añadirlo a OPEN. Asignar el cálculo de $f(n')$ al nodo n'.*

7. *Saltar al paso 2*

◇

5.3.1. *best-agreed-truncation* y *best-agreed-operator*

Como se mencionó antes, CIGOL usa búsqueda eligiendo-al-mejor en los procedimientos *best-agreed-truncation* y *best-agreed-operator.*

En *best-agreed-truncation* el objetivo de la búsqueda es encontrar un conjunto de cláusulas unitarias $P' \subseteq P$ y una cláusula unitaria E', dado un ejemplo E, de modo que E' es la gme de $\{P' \cup E\}$. La búsqueda se detiene en un mínimo local. En este punto CIGOL realiza una verificación para ver si se puede demostrar que E' es verdadero o falso, al utilizar P y N. Esta

verificación se hace mediante demostración de teoremas de profundidad acotada, descrito en la sección anterior. Cuando no se puede demostrar que sea verdadero ni falso, se cuestiona al supervisor sobre la veracidad de E'. Si el supervisor rechaza la generalización entonces E' se añade a N y la búsqueda se reanuda. De otra manera P es igual a $(P - P') \cup \{E'\}$, I se actualiza y *best-agreed-truncation* regresa E'.

En *best-agreed-operator* la búsqueda se usa para encontrar qué operador es más conveniente aplicar, si absorción o intra-construcción.

Si el operador elegido de acuerdo con la heurística es absorción, se busca un conjunto de cláusulas $P'' \subseteq P$ y un conjunto de cláusulas Q tal que cada elemento de Q puede obtenerse al resolver un elemento de P'' con una cláusula unitaria en P. Una vez que se detiene la búsqueda (en un mínimo local), CIGOL trata de verificar si los hechos que son consecuencia de $(P - P'') \cup Q$ y que no son consecuencia de P, son falsos al utilizar P y N, si esta verificación tiene éxito Q se agrega a N. Si no, CIGOL trata de verificar que todas las cláusulas de Q son falsas o verdaderas al utilizar P y N, en caso de que sean falsas entonces Q se agrega a N.

Por último, si ninguna de las verificaciones anteriores tuvo éxito, entonces se cuestiona al supervisor sobre la veracidad de Q. Si Q resulta verdadero, entonces Q remplazará a P''. De otra manera, Q se agregará a N y la búsqueda se reanudará.

Si el operador elegido es intra-construcción, se busca un conjunto $BB \subseteq P$ y una generalización B de ese conjunto, además, se construye una literal negativa l con un nuevo símbolo de predicado, para construir un conjunto S similar al conjunto de cláusulas de salida del algoritmo 4.3.1. Después se pregunta al supervisor el nombre del nuevo predicado. En este punto el supervisor también puede rechazar el resultado de la aplicación de intra-construcción, en ese caso P continúa igual y se reanuda la búsqueda. De otro modo, P es igual $(P - BB) \cup S$.

En *best-agreed-truncation* se usa como heurística la minimización del "tamaño" de $(P - P') \cup \{E'\}$. En *best-agreed-operator* se usa como heurística la minimización de $(P - P'') \cup Q$ si el operador es absorción y la minimización de $(P - BB) \cup S$ en el caso de intra-construcción.

El tamaño de los objetos sintácticos se define como sigue.

Definición Sea t un término. El tamaño de t, $tamaño(t)$ es:

- Si t es una variable, entonces,

$$tamaño(t) = 1$$

- Si t es un término de la forma $f(t_1, \ldots, t_n)$, entonces,

$$tamaño(t) = 1 + tamaño(\{t_1, \ldots, t_n\})$$

Sea l una literal cuyos argumentos son $\{t_1, \ldots, t_n\}$, entonces,

$$tamaño(l) = 1 + tamaño(\{t_1, \ldots, t_n\})$$

Sea $\mathscr{E}$ un conjunto de expresiones, entonces,

- si $\mathscr{E} = \emptyset$

$$tamaño(\mathscr{E}) = 1$$

- si $\mathscr{E} = \{E_1, \ldots, E_n\}$

$$tamaño(\mathscr{E}) = 1 + \sum_{i=1}^{n} tamaño(E_1, \ldots, E_n)$$

El espacio de búsqueda para estos dos procedimientos consiste en nodos de la siguiente forma $n = (V, PC, Cs, Op)$ donde V es el valor de la función heurística para ese nodo, PC contiene la información necesaria para aplicar el operador indicado por Op, y Cs contiene un conjunto de cláusulas de entrada para intra-construcción.

En *best-agreed-truncation* se usa como función heurística la siguiente:

$$f_1(n) = \left\{ \; tamaño(E') + tamaño(G) - tamaño(D) \right.$$

donde E' es el conjunto de cláusulas agregadas, D el conjunto de cláusulas subsumidas por E', y G es el conjunto de los términos que ocurren en D y que se generalizan, después de aplicar el operador.

En *best-agreed-operator* se usa como función heurística la siguiente:

$$f_2(n) = \begin{cases} tamaño(A) - tamaño(D) & \text{si } |A| \geq 1 \text{ y } |D| \geq 1 \\ \infty & \text{en otro caso} \end{cases}$$

donde A es el conjunto de cláusulas agregadas y D el conjunto de cláusulas borradas, después de aplicar el operador.

La búsqueda en *best-agreed-truncation* sigue de manera general el algoritmo 5.3.1. En *best-agreed-operator* antes de comenzar a buscar una solución se hacen dos búsquedas independientes, una que busca solamente sobre nodos de absorción y otra sobre nodos de intra-construcción. Estas búsquedas siguen el algoritmo 5.3.1, pero no realizan el paso *4*, y se detienen cuando ningún elemento de OPEN es menor que el menor elemento de CLOSED. Después se unen los conjuntos OPEN y CLOSED de cada búsqueda, y se aplica el algoritmo 5.3.2 sobre los conjuntos resultantes.

Algoritmo 5.3.2 *Algoritmo eligiendo-al-mejor'*

Entrada: conjuntos OPEN, CLOSED

1. *Si* $CLOSED \neq \emptyset$, *tomar un nodo* $n \in CLOSED$ *tal que* $f(n)$ *es mínima en CLOSED, si no ir 5*

2. *Si* $OPEN \neq \emptyset$ *tomar un nodo* $n' \in OPEN$ *tal que* $f(n')$ *es mínima en OPEN*

   a) *Si* $f(n) < f(n')$ *ir a 3*

   b) *Si* $f(n) \geq f(n')$ *ir a 6*

3. *Si n es un nodo objetivo terminar con éxito y n es la solución.*

4. *Eliminar n de CLOSED y de OPEN*

5. *Si* $OPEN = CLOSED = \emptyset$ *salir con falla*

6. *Tomar un nodo n' de $OPEN$ tal que $f(n')$ es mínima en $OPEN$ y ponerlo en $CLOSED$*

7. *Expandir n y poner los nodos resultantes en $OPEN$*

8. *Regresar al paso 1*

◇

Los autores de esta tesis obtuvieron esta variante del algoritmo eligiendo-al-mejor mediante el análisis del código de CIGOL.

Capítulo 6

Notas finales

Este capítulo presenta algunas notas finales sobre la motivación para realizar este trabajo, el trabajo futuro y retos de la PLI, y además se presentan algunas conclusiones.

6.1. Motivación

Como se ha mencionado a lo largo de este trabajo, el artículo de Muggleton y Buntine [MB88a] es uno de los artículos seminales en el área de la programación lógica inductiva. Es por esto que nos interesamos en las aportaciones de dicho artículo, a saber, resolución inversa e invención de predicados, para desarrollar el trabajo aquí presentado. Consideramos que estudiar las bases de un área de conocimiento, en este caso la programación lógica inductiva, es la mejor manera de incursionar en dicha área, por tanto esta tesis puede ser de interés para quienes deseen introducirse en el campo de la programación lógica inductiva.

Esperamos que en el futuro se incluyan temas como éste dentro de los programas del nivel licenciatura de la UNAM, para incentivar, de esta manera, el estudio de la programación lógica inductiva, de modo que se generen grupos y proyectos de investigación en esta área.

6.2. Trabajo futuro

En lo que respecta al trabajo futuro, podemos considerar los siguientes puntos.

- La modificación del código de CIGOL, de modo que utilice el algoritmo de absorción mencionado en el capitulo 4.
- Mejorar la implementación de los algoritmos de búsqueda en CIGOL al cambiar la representación del espacio de estados y la heurística.
- Estudiar la relación de los operadores 'V' y 'W' con algunos de los operadores de transformación de programas.
- Estudiar otras técnicas más modernas de la programación lógica inductiva [AC04, BMO+01, Mug95, TNM02].

6.3. Retos de la PLI

A pesar de las ventajas que tiene la PLI sobre otras técnicas de aprendizaje de máquina, algunas aplicaciones actuales y potenciales destacan los siguientes defectos en la tecnología actual de la PLI [Pag00] .

- Otras técnicas como modelos ocultos de Markov, redes de Bayes y redes dínamicas de Bayes, y bigramas y trigramas pueden representar, de manera expresa, las probabilidades inherentes en tareas como etiquetado de componentes sintácticos del habla, alineamiento de proteinas, maniobras en robots, etc. Pocos sistemas de PLI son capaces de representar o tratar con probabilidades.

- Los sistemas de PLI tienen requerimientos mayores de tiempo y espacio que otros sistemas de aprendizaje, que hacen difícil su aplicación a conjuntos grandes de datos. Enfoques alternativos como búsqueda estocástica y procesamiento paralelo necesitan explorarse.

- La PLI funciona bien cuando los datos y conocimiento previo pueden expresarse en la lógica de primer orden; pero, ¿Qué puede hacerse cuando los datos son imágenes, audio, video, etc.? Dentro de la PLI es necesario aprender de la programación lógica con restricciones en lo que concierne a la incorporación de técnicas de propósito específico para el manejo de formatos especiales de datos.

- Dentro del descubrimiento de conocimiento científico, por ejemplo, en el dominio de la bioinformática, sería benéfico que los sistemas de PLI pudieran colaborar con científicos, en lugar de procesar todos los datos juntos. Si la PLI no da este paso, otras clases de asistentes de colaboración científica se desarrollarán, y ocuparán la posición de la PLI dentro de estos dominios.

6.4. Conclusiones

La PLI ha despertado gran interés en la comunidad de aprendizaje de máquina y en la comunidad de inteligencia artificial debido a sus fundamentos lógicos, su habildad para utilizar el conocimiento previo y por lo comprensible de sus resultados. Pero la razón primordial por la que se ha despertado este interés es el éxito de las aplicaciones de la PLI.

A pesar de esto, la PLI necesita de avances significativos para mantener este éxito, para lograr dichos avances se requieren contribuciones de otras áreas de la lógica computacional. La PLI necesita de expertos en lógica, probabilidad, satisfacción de restricciones, programación lógica con restricciones y paralelismo. La incorporación de estos expertos a la PLI permitirá superar los retos anteriormente planteados.

Bibliografía

[AC04] N. Angelopoulos y J. Cussens. Extended stochastic logic programs for informative priors over C&RTs. En R. Camacho, R. King, y A. Srinivasan, editores, *Proceedings of the work-in-progress track of the Fourteenth International Conference on Inductive Logic Programming (ILP04)*, pp. 7–11, Porto, Portugal, Septiembre 2004.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall international series in computer science, 1997.

[Bac94] F. Bacon. *Novum Organum.* Open Court, Chicago, IL, 1994. Editado y traducido por P. Urbach, J. Gibson. Publicado por primera vez en 1620.

[Ban92] R. B. Banerji. Learning theoretical terms. En S. Muggleton, editor, *Inductive Logic Programming.* Academic Press, Londres, 1992.

[BG96] F. Bergadano y D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, 1996.

[BM95] I. Bratko y S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM*, 38(11):65–70, 1995.

[BMO+01] C.H. Bryant, S.H. Muggleton, S.G. Oliver, D.B. Kell, P. Reiser, y R.D. King. Combining inductive logic programming, active learning and robotics to discover the function of genes. *Electronic Transactions in Artificial Intelligence*, 5-B1(012):1–36, Noviembre 2001.

[Bun88] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[Car52] R. Carnap. *The Continuum of Inductive Methods.* The University of Chicago Press, Chicago, IL, 1952.

[Car62] R. Carnap. *Logical Foundations of Probability.* The University of Chicago Press, Chicago, IL, 1962.

[DK96] Y. Dimopoulos y A. Kakas. Abduction and inductive learning. En L. De Raedt, editor, *Advances in Inductive Logic Programming*, pp. 144–171. IOS, 1996.

[Doe94] K. Doets. *From Logic to Logic Programming.* MIT Press, Cambridge, MA, 1994.

[Gal86] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, New York, NY, 1986.

[Goo73] N. Goodman. *Fact, Fiction and Forecast.* Bobbs Merril, Indianapolis, 1973.

[Hem75] C. G. Hempel. *Confirmación, Inducción y Creencia Racional.* Paidos, Buenos Aires, 1975. Traducción al español de Néstor Miguez.

[Hem99] C. G. Hempel. *Filosofía de la Ciencia.* Alianza, Madrid, 1999. Traducción al español de Alfredo Deaño.

[Hil74] R. Hill. LUSH-resolution and its completeness. Dcl Memo 78, School of Artificial Intelligence, University of Edinburgh, Agosto 1974.

[Hog90] J. C. Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[Hum78] D. Hume. *A Treatise of Human Nature.* Claredon, Oxford, 1978. Publicado por primera vez 1739-1740.

[Hum00] D. Hume. *An Equiry Concerning Human Understanding.* Claredon, 2000. Editado por Tom L. Beachamp. Publicado por primera vez en 1748.

[Jev07] W. S. Jevons. *The Principles of Science: A Treatise.* Macmillan, Londres, 1907.

[KKT93] A. C. Kakas, R. A. Kowalski, y F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kow74] R. Kowalski. Predicate logic as programming language. En *IFIP Congress*, pp. 569–574, 1974. Reimpreso en *Computers for Artifi-*

cial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73.

[Lak87] I. Lakatos. *Matemáticas, ciencia y epistemología.* Alianza, Madrid, 2a edición, 1987. Traducción al español de Diego Ribes Nicolás.

[LD94] N. Lavrač y S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2a edición, 1987.

[LN92] C. X. Ling y M. A. Narayan. A critical comparison of various methods. En S. Muggleton, editor, *Inductive Logic Programming*, pp. 131–143. Academic Press, Londres, 1992.

[MB88a] S. Muggleton y W. Buntine. Machine invention of first order predicates by inverting resolution. *Proc. of the Fifth Int. Conf. on Machine Learning*, pp. 339–352, 1988.

[MB88b] S. Muggleton y W. Buntine. Towards constructive induction in first-order predicate calculus. *Turing Institute Working Paper*, 1988.

[Mic83] R. Michalski. A theory and methodology of inductive logic programming. *Machine Learning: An Artificial Intelligence Approach*, 1:83–134, 1983.

[Mil58] J. S. Mill. *A system of logic, Ratiocinative and Inductive.* Harper, New York, NY, 1858.

[MR94] S. Muggleton y L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20, 1994.

[Mug87] S.H. Muggleton. Duce, an oracle based approach to constructive induction. En *IJCAI-87*, pp. 287–292. Kaufmann, 1987.

[Mug91] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[Mug95] S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[NC90] S-H. Nienhuys-Cheng. Consequent functions and inverse resolutions. Reporte técnico Eur-CS-90-03, Erasmus University Rotterdam, Mayo 1990.

[NCdW97] S.-H. Nienhuys-Cheng y R. de Wolf. *Foundations of Inductive Logic Programming*, volumen 1228 de *LNAI.* Springer-Verlag, Febrero 1997.

[NCF91] S-H. Nienhuys-Cheng y P.A. Flach. Consistent term mappings, term partitions and inverse resolution. En Y. Kodratoff, editor, *EWSL91*, volumen 482 de *LNAI*, pp. 361–374. Springer-Verlag, 1991.

[Pag00] D. Page. ILP: Just do it. En J. Cussens y A. Frisch, editores, *Inductive Logic Programming*, volumen 1866 de *LNAI*, pp. 3–18. Springer-Verlag, 2000.

[Pea84] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison Wesley, 1984.

[Pei58] C. S. Peirce. *Collected Papers*, volumen I-VII. Harvard University Press, Cambridge, MA, 1958. Editado por C. Harstshorne y P. Weiss.

[Plo70] G.D. Plotkin. A note on inductive generalization. En *Machine Intelligence*, volumen 5, pp. 153–163. Edinburgh University Press, 1970.

[Plo71] G.D. Plotkin. *Automatic Methods of Inductive Inference.* PhD tesis, Edinburgh University, 1971.

[Pop02] K. R. Popper. *The logic of Scientific Discovery.* Rutledge, Londres, 2002.

[Qui86] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Rei49] H. Reichenbach. *The theory of probability.* University of California Press, Berkeley, CA, 1949.

[Rey70] J.C. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. *Machine Intelligence*, 5:135–171, 1970.

[Rou92] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. En S. Muggleton, editor, *Inductive Logic Programming*, pp. 63–92. Academic Press, 1992.

[RP90] C. Rouveirol y J. F. Puget. Beyond inversion of resolution. En B. Porter y R.J. Mooney, editores, *ML90 - Proc. of the 7th International Conference on Machine Learning*, pp. 122–131, Palo Alto, 1990. Morgan Kaufmann.

[Rus48] B. Russell. *Human Knowlwdge: Its Scope and Limits.* Simon y Schuster, New York, NY, 1948.

[Rus97] B. Russell. *The Problems of Philosophy.* Oxford University Press, New York, NY, 1997. Publicado por primera vez en 1912.

[SB86] C. Sammut y R. Banerji. Learning concepts by asking questions. En R.S. Michalski, J.G. Carbonell, y T.M. Mitchell, editores, *Machine Learning: An Artificial Intelligence Approach, volumen 2*, pp. 167–191. Morgan Kaufmann, 1986.

[Sta93] I. Stahl. Predicate invention in ILP: An overview. En P. B. Brazdil, editor, *Machine Learning: ECML-93 - Proc. of the European Conference on Machine Learning*, pp. 313–322. Springer, Berlín, Heidelberg, 1993.

[TNM02] A. Tamaddoni-Nezhad y S.H. Muggleton. A genetic algorithms approach to ILP. En *Proceedings of the 12th International Conference on Inductive Logic Programming*, pp. 285–300. Springer-Verlag, 2002.

[vEK76] M. H. van Emden y R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[Win77] P. H. Winston. *Artificial Intelligence.* Addison-Wesley, 1977.

[Wir89] R. Wirth. Completing logic programs by inverse resolution. En K. Morik, editor, *EWSL89.* Pitman, 1989.