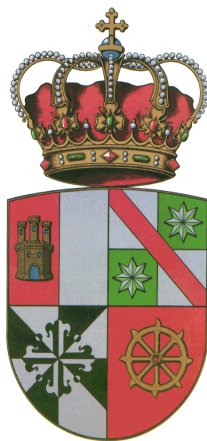


Universidad de Castilla-La Mancha

Departamento de Sistemas Informáticos



Técnicas de calibrado y despliegado de
programas lógicos difusos integrados

TESIS DOCTORAL

Presentada por:

José Antonio Riaza Valverde

Dirigida por:

Ginés Moreno Valverde (UCLM)

Pascual Julián Iranzo (UCLM)

Técnicas de calibrado y despliegado de programas lógicos difusos integrados

José Antonio Riaza Valverde

Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha



Memoria presentada para optar al título de:

Doctor en Informática

Dirigida por:

Ginés Moreno Valverde
Pascual Julián Iranzo

Tribunal de lectura:

Presidente:	Germán Vidal Oriola	(UPV)
Vocal:	Jesús Manuel Almendros Jiménez	(UAL)
Secretaria:	Hermenegilda Macià Soler	(UCLM)

Albacete, 7 de noviembre de 2022.

Resumen

FASILL (acrónimo de «*Fuzzy Aggregators and Similarity Into a Logic Language*») es un lenguaje de programación lógico difuso que integra y extiende en un mismo lenguaje las capacidades de otros lenguajes difusos. FASILL cuenta con anotaciones de grados de verdad implícitas y explícitas y combina un algoritmo de unificación débil basado en relaciones de similitud, con un amplio repertorio de conectivas difusas, cuyas funciones de verdad pueden ser definidas sobre un retículo completo. El objetivo de esta tesis es el diseño e introducción de técnicas de calibrado y desplegado de programas lógicos difusos en el sistema FASILL, con el propósito de avanzar en el desarrollo de un entorno completo de programación lógica difusa que incluya utilidades para la compilación, ejecución, optimización y depuración de este tipo de programas.

El calibrado de programas lógicos difusos es una técnica automática que permite a los programadores ajustar los pesos y las conectivas en las reglas y hechos de un programa, al disponer de un conjunto de casos de prueba con los valores esperados para una serie de objetivos. Con este fin hemos diseñado una extensión simbólica de FASILL que permite introducir valores y conectivas simbólicas, cuya interpretación puede ser pospuesta hasta conocer sus valores concretos, sobre la que introducimos diversos algoritmos de calibrado. Además, mostramos algunas aplicaciones del calibrado en distintos dominios como la verificación de circuitos digitales, el aprendizaje automático y la web semántica.

Por otro lado, el desplegado de programas es una técnica de transformación automática que permite optimizar los programas aplicando pasos de computación sobre los cuerpos de las reglas. Esta transformación, que es clásica en otros paradigmas como el lógico y el funcional, presenta un reto especial en un marco tan amplio y flexible como FASILL ya que, en general, no siempre es posible desplegar un programa FASILL garantizando la preservación de las respuestas computadas difusas. Por lo tanto, además de introducir la transformación de desplegado en el lenguaje FASILL caracterizamos los programas que pueden ser desplegados de forma segura.

Todos los desarrollos de esta tesis vienen acompañados de demostraciones formales de sus propiedades fundamentales, al tiempo que se proporcionan los detalles de implementación de estas técnicas sobre el sistema FASILL: una implementación de alto nivel escrita en Prolog, desarrollada en nuestro grupo de investigación DEC-TAU, que pretende servir como una herramienta sobre la que implementar, ejecutar y medir todos nuestros avances.

Agradecimientos

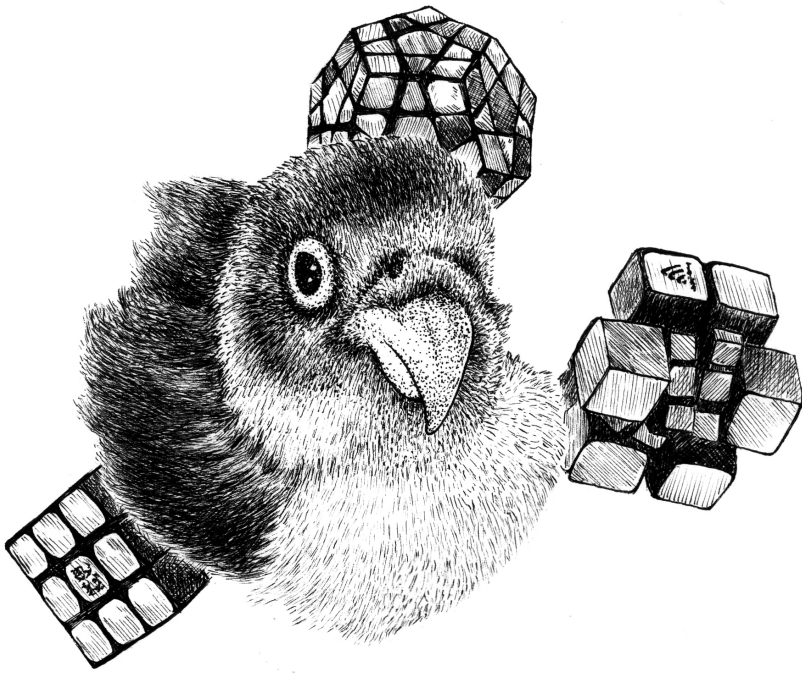
En primer lugar, gracias a la Universidad de Castilla-La Mancha por financiar este trabajo mediante un contrato predoctoral para la formación de personal investigador en el marco del Plan Propio de I+D+i, cofinanciado por el Fondo Social Europeo, publicado en el DOCM con número de anuncio 2018/12504 y número de resolución 2019/451.

Gracias también al Ministerio de Ciencia e Innovación por financiar los proyectos «Métodos rigurosos para el internet del futuro» (MERINET), con referencia TIN2016-76843-C4-2-R, y «Análisis y validación de software y recursos web» (SAFER), con referencia PID2019-104735RB-C42, que nos han permitido publicar y difundir los resultados de nuestra investigación.

Quiero agradecer especialmente a mis directores de tesis Ginés Moreno y Pascual Julián, y a los profesores Jaime Penabad y Juan Antonio Guerrero de la Universidad de Castilla-La Mancha, su inestimable ayuda y la amistad que me han demostrado en este período de tiempo. Su experiencia y su inquietud profesional han sido determinantes en el desarrollo de este trabajo.

Además, gracias a los compañeros Jesús Almendros y Antonio Becerra de la Universidad de Almería y Germán Vidal de la Universidad Politécnica de Valencia. Las contribuciones nacidas de estas colaboraciones nos han permitido ampliar y mejorar los resultados de nuestra investigación.

Por último, gracias a los compañeros Fernando Sáenz de la Universidad Complutense de Madrid y Josep Silva de la Universidad Politécnica de Valencia por sus exhaustivas revisiones de este trabajo y por sus comentarios sobre la memoria.



A Lambda.

Índice general

1. Introducción	1
1.1. Motivación y objetivos	1
1.2. Contribuciones	2
1.3. Estructura de la memoria	3
2. Fundamentos	7
2.1. Lógica	7
2.1.1. Sistemas formales	7
2.1.2. Lógica de proposiciones	9
2.1.3. Lógica de predicados	13
2.2. Programación lógica	19
2.2.1. Sustituciones	19
2.2.2. Unificación	20
2.2.3. Resolución	21
2.2.4. El lenguaje Prolog	23
2.3. Lógica difusa	25
2.3.1. Conjuntos difusos	26
2.3.2. Interpretaciones y conectivas difusas	26
2.3.3. Relaciones de similitud	29
2.4. Programación lógica difusa	30
2.4.1. Unificación débil	31
2.4.2. Extendiendo la unificación	35
2.4.3. Extendiendo la resolución	36
3. El lenguaje FASILL	39
3.1. Sintaxis	39
3.2. Semántica operacional	40
3.3. Semántica declarativa	45
3.4. El sistema FASILL	47
3.4.1. Interfaz	47
3.4.2. Predicados incorporados	49
3.4.3. Detalles de implementación	53
3.4.4. Pruebas y evaluación de rendimiento	64
3.5. Conclusiones	70

4. La extensión simbólica sFASILL	73
4.1. Sintaxis	73
4.2. Sustituciones simbólicas	74
4.3. Relaciones de similitud simbólicas	75
4.4. Semántica operacional	77
4.5. Detalles de implementación	82
4.6. Conclusiones	88
5. Calibrado de programas lógicos difusos	91
5.1. Motivación y antecedentes	92
5.2. Introducción	93
5.3. Calibrado de programas en el sistema FASILL	94
5.4. Calibrado basado en satisfacibilidad	101
5.5. Detalles de implementación	104
5.6. Pruebas y evaluación de rendimiento	112
5.6.1. Equivalencia de circuitos combinacionales	114
5.6.2. Regresión lineal	119
5.6.3. Redes neuronales	123
5.6.4. La web semántica	126
5.7. Conclusiones	133
6. Despliegado de programas lógicos difusos	137
6.1. Motivación y antecedentes	137
6.2. Despliegado de programas FASILL	138
6.2.1. Problemas de corrección debidos a relaciones de similitud	142
6.2.2. Problemas de completitud debidos a pasos de fallo	145
6.3. Corrección y completitud del desplegado	148
6.4. Detalles de implementación	158
6.5. Pruebas y evaluación de rendimiento	163
6.5.1. Despliegado difuso de programas simbólicos	165
6.6. Conclusiones	167
7. Conclusiones y trabajo futuro	171
7.1. Conclusiones	171
7.2. Trabajo futuro	175
A. Retículos completos en FASILL	179
A.1. Retículo en el intervalo unitario	179
A.2. Retículo booleano	181
A.3. Retículo en la recta real extendida	182
A.4. Retículo para FSA-SPARQL	183
Bibliografía	187

Índice de figuras

2.1. Reglas de transición de la relación de unificación (\Rightarrow_{mgu}).	21
2.2. Árbol de derivación.	23
2.3. Reglas de transición de la relación de unificación débil (\Rightarrow_{wmgu}).	32
3.1. Árbol de derivación generado por el sistema FASILL en línea.	43
3.2. Consola interactiva del sistema FASILL.	48
3.3. Área de entrada del sistema FASILL en línea.	49
3.4. Área de salida del sistema FASILL en línea.	50
3.5. Grafo de dependencias funcionales del sistema FASILL.	53
3.6. Problema de las n reinas en FASILL y su versión compilada a Prolog.	67
3.7. Comparación de los tiempos de ejecución del sistema FASILL resolviendo objetivos y unificando términos.	68
4.1. Árbol de derivación simbólico generado por el sistema FASILL en línea.	79
4.2. Consola interactiva del sistema FASILL ejecutando un programa simbólico.	82
4.3. Área de entrada del sistema FASILL en línea cargando un programa simbólico.	83
4.4. Área de salida del sistema FASILL en línea ejecutando un programa simbólico.	84
5.1. Calibrado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el comando <code>:tuning</code>	105
5.2. Área de calibrado del sistema FASILL en línea.	106
5.3. Comparación de los tiempos de ejecución de los algoritmos de calibrado básico y simbólico en función del número de sustituciones simbólicas consideradas (con 100 átomos).	113
5.4. Comparación de los tiempos de ejecución de los algoritmos de calibrado básico y simbólico en función del número de átomos en los objetivos de los casos de prueba (con 1000 sustituciones simbólicas).	113
5.5. Dos circuitos combinacionales equivalentes.	114
5.6. Circuito de comprobación de equivalencia (<i>miter</i>).	116

5.7.	Diagrama de dispersión del conjunto de datos del estudio [Pie46].	121
5.8.	Modelo de regresión lineal del ejemplo 5.13.	122
5.9.	Comparación de los tiempos de ejecución del algoritmo de calibrado basado en satisfacibilidad ajustando modelos de regresión lineal.	123
5.10.	Diagrama de un modelo de neurona artificial.	124
5.11.	Funciones de verdad asociadas a las funciones de activación utilizadas en redes neuronales.	125
5.12.	Conjuntos difusos basados en una función de pertenencia trapezoidal.	130
5.13.	Conjuntos difusos para el atributo “jonrones”.	131
6.1.	Comparación de los árboles de derivación para la ejecución de un mismo objetivo en el programa original y en el programa desplegado.	140
6.2.	Árbol de derivación para el objetivo \mathcal{G}_0 en $\mathcal{P}_0^{(8)}$	142
6.3.	Despliegado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el comando <code>:unfold</code>	159
6.4.	Despliegado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el predicado incorporado <code>unfold/1</code>	159
6.5.	Área de despliegado del sistema FASILL en línea.	160
6.6.	Despliegado de programas en el sistema FASILL en línea.	160
6.7.	Programas FASILL de prueba.	164
6.8.	Comparación de los tiempos de ejecución de los programas originales frente a los programas desplegados.	165
6.9.	Comparación de los tiempos de ejecución de varios programas FASILL en función del número de pasos de despliegado.	166
6.10.	Comparación de los tiempos de ejecución del algoritmo de calibrado básico en función del número de pasos de despliegado aplicados.	168
A.1.	Intervalo unitario como subconjunto de la recta real.	179
A.2.	Funciones de verdad de las conectivas difusas definidas en el retículo unitario (I, \leq)	180
A.3.	Diagrama de Hasse del conjunto ordenado $\mathcal{B} = \{0, 1\}$	181
A.4.	Recta real extendida $\overline{\mathbb{R}}$	182
A.5.	Funciones de verdad de las conectivas difusas definidas en el retículo de la recta real extendida $(\overline{\mathbb{R}}, \leq)$	183
A.6.	Funciones de verdad de las conectivas difusas definidas en el retículo de FSA-SPARQL.	184

Índice de tablas

2.1. Definición de las conectivas lógicas.	10
2.2. Operadores predefinidos de Prolog.	25
3.1. Operadores predefinidos de FASILL.	55
3.2. Tiempo medio de ejecución (en milisegundos) de un programa FASILL arbitrario con n hechos tras 100 ejecuciones, en función de la t-norma utilizada en la derivación.	65
3.3. Tiempo medio de ejecución (en milisegundos) y número de inferencias del problema de las n reinas ejecutado en el intérprete FASILL y en SWI-Prolog (tanto la versión compilada del programa FASILL como el programa original en Prolog) tras 50 ejecuciones.	66
3.4. Tiempo medio de ejecución (en milisegundos) y número de inferencias de una consulta FSA-SPARQL con n resultados, ejecutado por el intérprete FASILL y por SWI-Prolog (al compilarlo) tras 50 ejecuciones.	68
3.5. Tiempo de ejecución (en milisegundos) y número de inferencias del algoritmo para computar el unificador más general, al unificar términos FASILL arbitrarios de n símbolos aleatoriamente anidados tras 1000 ejecuciones.	69
3.6. Tiempo de ejecución (en milisegundos) y número de inferencias del algoritmo para computar el unificador débil más general, al unificar débilmente términos FASILL arbitrarios de n símbolos aleatoriamente anidados tras 1000 ejecuciones.	69
3.7. Tiempo medio de ejecución (en milisegundos) y número de inferencias del problema de las n reinas tras 50 ejecuciones. Tanto la versión compilada del programa FASILL, que llama explícitamente al algoritmo de unificación débil, como el programa Prolog original son ejecutados sobre SWI-Prolog.	70
4.1. Operadores predefinidos de sFASILL.	84

5.1.	Tiempo medio de ejecución (en milisegundos) de los métodos de calibrado básico y simbólico tras 50 ejecuciones, en función del número de átomos en la derivación (k) y del número de sustituciones simbólicas consideradas, para una constante simbólica y un caso de prueba.	112
5.2.	Tabla de verdad de las funciones lógicas implementadas por los circuitos combinatoriales C_A , C_B y C_C	119
5.3.	Tiempo medio de ejecución (en milisegundos) de los algoritmos de calibrado simbólico y basado en satisfacibilidad, tras 50 ejecuciones, al comprobar la equivalencia de circuitos combinatoriales en función del número de entradas.	119
5.4.	Conjunto de datos del estudio [Pie46].	120
5.5.	Tiempo medio de ejecución (en segundos) del algoritmo de calibrado basado en satisfacibilidad en Z3 para regresión lineal en función del número de variables explicativas y del número de casos de prueba (k).	123
5.6.	Respuesta a la consulta SPARQL del ejemplo 5.16.	128
5.7.	Respuesta a la consulta FSA-SPARQL del ejemplo 5.17.	129
6.1.	Tiempo medio de ejecución (en milisegundos) y número de inferencias al ejecutar los programas FASILL de prueba mostrados en la figura 6.7 tras 50 ejecuciones.	165
6.2.	Tiempo medio de ejecución (en milisegundos) y número de inferencias al buscar todas las f.c.a. para el predicado <code>mul/3</code> utilizando el programa de prueba mostrado en la figura 6.7e tras 50 ejecuciones, dependiendo del número de pasos de desplegado realizados sobre el programa original.	166
6.3.	Tiempo medio de ejecución (en milisegundos) al calibrar un programa tras 50 ejecuciones, dependiendo del número de pasos de desplegado realizados sobre el programa original y del número de sustituciones simbólicas consideradas.	167

Capítulo 1

Introducción

En los últimos años, la lógica difusa ha jugado un papel fundamental en el desarrollo de aplicaciones informáticas en campos tan diversos como los sistemas expertos, la medicina o el control industrial. Con el objetivo de facilitar el desarrollo de tales aplicaciones, más recientemente ha surgido el interés por diseñar lenguajes declarativos (en particular, lenguajes lógicos) difusos que incorporen entre sus recursos expresivos el tratamiento de información imprecisa de forma natural. En la última década, nuestro grupo de investigación DEC-TAU de la Universidad de Castilla-La Mancha ha producido numerosos trabajos en esta línea, dando forma al lenguaje FASILL, que incorpora y extiende las capacidades de otros lenguajes lógicos difusos como MALP –que modifica la noción de cláusula al añadir pesos y conectivas difusas definidas en un retículo–, o Bousi~Prolog –que extiende el mecanismo de unificación al introducir relaciones de similitud–.

1.1. Motivación y objetivos

El objetivo fundamental de esta tesis es el diseño e introducción de técnicas de calibrado y desplegado de programas lógicos difusos en el sistema FASILL, con el propósito de avanzar en el desarrollo de un entorno completo de programación lógica difusa que incluya utilidades para la compilación, ejecución, optimización y depuración de este tipo de programas. El calibrado de programas lógicos difusos es una técnica automática que permite a los programadores ajustar los pesos y las conectivas en las reglas y hechos de un programa, al disponer de un conjunto de casos de prueba con los valores esperados para una serie de objetivos. Por otro lado, el desplegado de programas lógicos difusos es una técnica de transformación automática que permite optimizar los programas aplicando pasos de computación sobre los cuerpos de las reglas.

Podemos resumir los objetivos de la tesis en los siguientes puntos.

- (1) Implementar y mantener un entorno de programación lógico difuso que nos permita incorporar todas las características y transformaciones que estudiamos en nuestras investigaciones.

- (2) Extender la sintaxis y la semántica del lenguaje FASILL para incorporar constantes (valores y conectivas) simbólicas que no pertenecen al retículo asociado al programa, con el fin de posponer la evaluación de las mismas hasta que sus valores son conocidos.
- (3) Definir distintos métodos de calibrado de programas lógicos difusos y estudiar su rendimiento y su utilidad en diversos dominios de aplicación de interés.
- (4) Adaptar al lenguaje FASILL la transformación de desplegado (clásica en otros paradigmas declarativos) y caracterizar los programas que pueden desplegarse de manera segura, proporcionando las demostraciones de corrección y completitud de la transformación.

1.2. Contribuciones

Las principales contribuciones de esta tesis se resumen a continuación.

- Hemos demostrado algunas propiedades sobre la unificación débil y las sustituciones [JIMR22b], como la idempotencia de las sustituciones generadas por el algoritmo de unificación débil (véase la proposición 2.4), o la relación entre el operador de composición paralela débil (véase la definición 2.41) y la composición estándar de sustituciones (véase la proposición 2.5). Estos resultados son de vital importancia para la demostración de corrección del desplegado.
- Hemos dado una especificación de nuestro lenguaje de programación lógico difuso FASILL, no solo desde el punto de vista teórico, sino también práctico [JIMR20], proporcionando una descripción detallada de nuestra implementación de alto nivel escrita en Prolog. Además, hemos llevado a cabo una serie de pruebas para medir el rendimiento y analizar el sobre-coste que conlleva introducir cada una de las componentes difusas en un lenguaje lógico. En esta línea, hemos dotado al sistema FASILL de diversas características y funcionalidades extrapoladas del estándar ISO Prolog en base a la experiencia adquirida durante el diseño y desarrollo del sistema Tau Prolog [Ria22a, Ria22b].
- Hemos diseñado una extensión simbólica del lenguaje FASILL [MPRV17], que permite la introducción de valores y conectivas desconocidas en las reglas de los programas y en las relaciones de similitud, con el fin de poder delegar la evaluación de las mismas hasta que sus valores son conocidos. Además, proporcionamos algunas demostraciones que garantizan la existencia de las mismas respuestas computadas difusas en un programa SFASILL con independencia de cuándo se reemplacen dichas constantes simbólicas por valores y conectivas concretas del retículo (véanse los teoremas 4.1, 4.2 y 4.3).

- Hemos introducido diversas técnicas de calibrado de programas lógicos difusos [MPRV17, MR20, MR21] que permiten ajustar automáticamente los pesos y las conectivas de un programa a partir de un conjunto de casos de prueba especificados por el usuario. En particular, hemos diseñado tres algoritmos de calibrado distintos: (1) un método basado en la semántica operacional de FASILL que sienta las bases del calibrado (véase el algoritmo 5.1), (2) una mejora del primer método basada en la semántica operacional de la extensión simbólica sFASILL que calcula derivaciones más cortas y puede reducir considerablemente el espacio de búsqueda (véase el algoritmo 5.3) y (3) un método basado en resolutores de satisfacibilidad que permite superar algunas limitaciones del calibrado en FASILL (véase la sección 5.4).
- Hemos utilizado las técnicas de calibrado en diversos dominios de aplicación de interés, como la verificación de la equivalencia de circuitos digitales y el aprendizaje automático (regresión lineal y redes neuronales) [RM20, MPR19b]. Además, hemos integrado el lenguaje FASILL con una extensión difusa del lenguaje de consulta SPARQL para generar conjuntos difusos y calibrar consultas FSA-SPARQL [AJBTMR19, AJBTMR21]. En esta línea, hemos utilizado el calibrado de consultas FSA-SPARQL en el ámbito de las redes sociales [AJBTMR22].
- Hemos adaptado la transformación de desplegado (clásica de otros paradigmas) al lenguaje lógico difuso FASILL [MPR17, JIMR22a] y hemos caracterizado los programas que pueden ser desplegados de forma segura mediante una serie de condiciones (véase la definición 6.7). Hemos dado las demostraciones de corrección y completitud de esta transformación (véanse los teoremas 6.1, 6.2 y 6.3) garantizando la preservación de las respuestas computadas difusas y la eficiencia del desplegado.
- Hemos desarrollado una herramienta web que permite ejecutar, calibrar [MR17] y desplegar [MR19a] programas FASILL, accesible desde la URL <https://dectau.uclm.es/fasill/sandbox>. Además, hemos relacionado las técnicas de calibrado y ejecución simbólica con las técnicas de desplegado difuso [MPR19a, MR19b].
- Hemos desarrollado herramientas para el diseño y análisis de retículos y relaciones de similitud [GMM⁺17, GMRS18] que permiten generar código compatible con el sistema FASILL para los retículos completos y las relaciones de similitud, además de verificar determinadas propiedades de los mismos.

1.3. Estructura de la memoria

En el capítulo 1 hemos establecido la motivación y los objetivos fundamentales de la tesis, y hemos destacado las principales aportaciones del trabajo

realizado. Tras este primer capítulo introductorio, la estructura de la memoria se organiza como sigue.

- En el capítulo 2 presentamos los fundamentos necesarios para abordar los conceptos estudiados en la tesis. En particular, se revisan las nociones básicas sobre los sistemas formales, la programación lógica y la lógica difusa. El capítulo termina con una sección sobre programación lógica difusa, en la que se abordan diferentes aproximaciones para *difuminar* un lenguaje de programación lógico, y sienta las bases para la definición de FASILL. Además, en esta sección proporcionamos algunos resultados nuevos sobre las propiedades del algoritmo de unificación débil, el operador de composición paralela débil y la composición estándar de sustituciones.
- En el capítulo 3 describimos en detalle nuestro lenguaje lógico difuso FASILL, tanto de forma teórica como práctica. Aquí, describimos la sintaxis, la semántica operacional y la semántica declarativa del lenguaje, y proporcionamos una descripción general de los aspectos prácticos del sistema FASILL, desde su uso a través de la consola interactiva o la interfaz web, hasta sus detalles de implementación de alto nivel en Prolog. Terminamos este capítulo con una serie de pruebas para evaluar el sobrecoste que conlleva introducir cada una de las componentes difusas en un lenguaje lógico.
- En el capítulo 4 introducimos una extensión simbólica del lenguaje FASILL, denominada sFASILL, que permite utilizar (en las reglas y relaciones de similitud de un programa) valores y conectivas simbólicas que no pertenecen al retículo asociado al programa. Esta extensión simbólica será de gran utilidad en capítulos posteriores para el diseño de algoritmos eficientes de calibrado de programas lógicos difusos. Para ello, en este capítulo presentamos algunos resultados fundamentales sobre la preservación de las respuestas computadas difusas de los programas simbólicos.
- En el capítulo 5 presentamos una de las aportaciones fundamentales de la tesis: el calibrado de programas lógicos difusos. Formalizamos el concepto de calibrado y especificamos diversos algoritmos para tal fin, que implementamos en el sistema FASILL para evaluar su rendimiento. El capítulo termina con una sección donde se muestran varios usos del calibrado en distintos dominios de aplicación relevantes como son la verificación de la equivalencia de circuitos combinatoriales, la web semántica y el entrenamiento de redes neuronales y modelos de regresión lineal.
- En el capítulo 6 presentamos la segunda aportación fundamental de la tesis: el despliegado de programas lógicos difusos. Damos una definición de despliegado para programas FASILL y mostramos los problemas de corrección y completitud que puede ocasionar esta transformación. En esta línea, caracterizamos los programas FASILL que pueden desplegarse sin problemas, y proporcionamos la demostración de corrección y completitud del despliegado para estos programas. El capítulo concluye con una

sección sobre la implementación de esta transformación en el sistema FASILL, y una serie de pruebas para determinar la ganancia de eficiencia de los programas desplegados.

- Por último, en el capítulo 7 recogemos las conclusiones del trabajo realizado y mostramos algunas líneas de trabajo que ya se han iniciado en nuestro grupo o que pueden seguirse en un futuro para ampliar y mejorar las técnicas de calibrado y despliegado de programas lógicos difusos estudiadas en esta tesis.

Capítulo 2

Fundamentos

Los lenguajes de programación declarativos se apoyan en algún tipo de lógica que se usa como modelo de cómputo, donde los programas se especifican como conjuntos de fórmulas y la computación se entiende como una forma de inferencia en dicha lógica. En particular, la programación lógica se basa en fragmentos de la lógica de predicados: los programas son conjuntos de cláusulas y el mecanismo computacional es el principio de resolución. El objetivo de este capítulo es revisar las nociones principales sobre la lógica de predicados y la programación lógica al tiempo que definimos conceptos fundamentales para el desarrollo de este trabajo. El capítulo termina con una introducción a la lógica difusa, que enlaza con la definición formal del lenguaje FASILL que se proporciona en el siguiente capítulo.

2.1. Lógica

Como preludeo al estudio de la programación lógica, en esta sección se presentan algunas ideas sobre los sistemas deductivos en general, que se abordan aquí mediante sistemas formales a dos niveles: (1) la lógica proposicional, que se ocupa de proposiciones simples que se contemplan como un todo indivisible y que pueden ser combinadas mediante conectivas; y (2) la lógica de predicados, donde se efectúa un análisis interno de las proposiciones para observar qué afirman y los objetos de quienes se realizan las afirmaciones.

2.1.1. Sistemas formales

Los sistemas formales son un marco matemático utilizado para formalizar los conceptos de *teorema* y *demostración* [Hod13]. Según [CFC⁺58], un *sistema formal* se define como un cuerpo de teoremas generados por reglas precisas y referentes a objetos indeterminados (que carecen de significado). En tales sistemas, la deducción procede por medio de reglas que son arbitrarias, pero que están explícitamente formuladas. Por lo tanto, la validez de una proposición en

un sistema formal no requiere ningún principio intuitivo o previo, ni siquiera los de la lógica. No obstante, siguiendo a Carnap [Car42], algunos autores añaden además un componente semántico a los sistemas formales [JIA07] que resulta de especial interés para el estudio de los lenguajes de programación (que pueden diseñarse y estructurarse como sistemas formales dotados de una sintaxis, un comportamiento operacional y una semántica).

Definición 2.1 (Sistema formal, [JIA07, Hod13]). Un *sistema formal* S es un modelo de razonamiento matemático formado por los siguientes componentes:

- a) **Lenguaje formal.** Un lenguaje formal es un conjunto de símbolos y fórmulas sintácticamente correctas. Para definir un lenguaje formal se necesita un conjunto (infinito numerable) de símbolos a utilizar en las construcciones de S , denominado *alfabeto*, y reglas que establezcan qué cadenas de símbolos son fórmulas bien formadas en S .
- b) **Cálculo deductivo.** Un cálculo deductivo es un conjunto de fórmulas y reglas que permiten obtener nuevas fórmulas atendiendo sólo a aspectos sintácticos de los símbolos:
 - b.1) **Axiomas.** Los axiomas son un conjunto (posiblemente vacío) de fórmulas bien formadas de S que se admiten dentro del sistema sin necesidad de demostración.
 - b.2) **Reglas de inferencia.** Las reglas de inferencia son un conjunto finito de reglas que permiten obtener nuevas fórmulas a partir de fórmulas anteriormente deducidas.
- c) **Semántica formal.** La semántica estudia la adscripción de significado a los símbolos y fórmulas del lenguaje. Para ello, se introducen reglas que permiten asignar significado a los símbolos del alfabeto y las expresiones del lenguaje formal.

Ejemplo 2.1 (El rompecabezas MU, [Hof79]). El *rompecabezas MU* es un problema que involucra un sistema formal denominado MIU , donde se pide que partiendo de la cadena MI , se consiga derivar la cadena MU en base a unas reglas. Este sistema se formaliza como sigue.

Lenguaje formal. El sistema MIU utiliza solo tres símbolos: M , I , U . Cualquier cadena compuesta por estos símbolos es una fórmula bien formada: MU , UIM , $MUUMUU$, ...

Axiomas. El único axioma es MI .

Reglas de inferencia. Hay cuatro reglas de inferencia:

$$\frac{xI}{xIU} \text{ (R1)} \quad \frac{Mx}{Mxx} \text{ (R2)} \quad \frac{xIIIy}{xUy} \text{ (R3)} \quad \frac{xUy}{xy} \text{ (R4)}$$

Una vez introducidos los sistemas formales, es posible definir qué es una demostración y qué es un teorema. Nótese que ambos son conceptos sintácticos.

Definición 2.2 (Teorema, [Hod13, JIA07]). Sea S un sistema formal y Γ un conjunto de fórmulas bien formadas. Una sucesión finita A_1, \dots, A_n de fórmulas bien formadas de S es una *deducción* en S a partir de Γ si para todo $k \in \{1, \dots, n\}$, se cumple una de las siguientes condiciones:

- a) A_k es un axioma de S ;
- b) $A_k \in \Gamma$;
- c) $k > 1$ y A_k es la conclusión de una regla de inferencia cuyas hipótesis están entre las fórmulas previas A_1, \dots, A_{k-1} .

El último miembro A_n se dice que es *deducible* en S a partir de Γ , denotado por $\Gamma \vdash_S A_n$, donde los elementos de Γ son las *premisas* y A_n es la *conclusión* de la deducción. Una *demostración* es una deducción sin premisas. Si A_1, \dots, A_n es una demostración en S , se dice que A_n es un *teorema* de S , denotado por $\vdash_S A_n$.

Ejemplo 2.2. La fórmula $MIIUU$ es un teorema del sistema formal MIU . Para demostrarlo, es suficiente con dar la deducción:

(1)	MI	Axioma
(2)	MII	R2
(3)	$MIIII$	R2
(4)	$MIIIIU$	R1
(5)	$MIUU$	R3
(6)	$MIUUIUU$	R2
(7)	$MIIUU$	R4

Es decir, $\vdash_{MIU} MIIUU$.

Se dice que un sistema formal es: *consistente*, si de él no se desprende ninguna contradicción (es imposible demostrar una fórmula y su negación); *correcto*, si toda fórmula demostrable sin premisas es válida; *completo*, si toda fórmula que es válida es demostrable sin premisas; y *decidible*, si existe un procedimiento finito para comprobar si una fórmula es válida o no [JIA07].

2.1.2. Lógica de proposiciones

Las *proposiciones* (o *enunciados*) son los componentes del discurso que tienen un significado completo. Un enunciado declarativo es aquel que posee un valor de verdad conocido y pueden formar enunciados compuestos por medio de conectivas lógicas. La *lógica proposicional* es el estudio de las relaciones de diverso tipo que se establecen entre los enunciados (en el plano semántico) y entre las formas enunciativas (en el plano sintáctico).

Lenguaje formal

A continuación, se describe el lenguaje formal de la lógica de proposiciones dando su alfabeto y las reglas de formación de las fórmulas bien formadas.

Alfabeto. El alfabeto de la lógica de proposiciones está formado por un conjunto finito de *variables proposicionales* $\mathcal{V} = \{p_1, p_2, p_3, \dots\}$ y las conectivas lógicas \neg (negación), \wedge (conjunción), \vee (disyunción), \rightarrow (condicional material) y \leftrightarrow (bicondicional). Además, se suelen incluir algunos signos de puntuación como “(” y “)”.

Fórmulas bien formadas. Las fórmulas bien formadas de la lógica de proposiciones se definen inductivamente conforme a las siguientes reglas:

- (1) Toda variable proposicional es una fórmula bien formada.
- (2) Si A y B son fórmulas bien formadas, también lo son $\neg A$, $\neg B$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ y $(A \leftrightarrow B)$.¹

Semántica

La lógica proposicional se rige por el *principio de bivalencia*, según el cual toda proposición –simple o compuesta– tiene un único valor de verdad. Cada variable proposicional toma un valor de verdad, que representamos aquí mediante los símbolos 1 (verdadero) o 0 (falso). Además, a cada conectiva se le asigna una *función de verdad*, que sirve para interpretar las fórmulas bien formadas y darles un valor de verdad.

Definición 2.3 (Función de verdad, [Hod13]). Una *función de verdad* n -aria es una función $F: \{1, 0\}^n \rightarrow \{1, 0\}$. Una *tabla de verdad* es una representación por extensión de una función de verdad.

Ejemplo 2.3. Las funciones de verdad de las conectivas lógicas $\neg, \wedge, \vee, \rightarrow, \text{ y } \leftrightarrow$ están descritas en la tabla 2.1 mediante sus tablas de verdad. Por ejemplo, la función de verdad de la negación queda definida como:

$$F_{\neg}(1) = 0; \quad F_{\neg}(0) = 1.$$

Ahora, dada un fórmula A en la que aparecen las variables proposicionales p_1, p_2, \dots, p_n , tras asignar un valor de verdad a cada una de estas variables, es posible calcular el valor de verdad de A mediante las funciones de verdad $F_{\neg}, F_{\wedge}, F_{\vee}, F_{\rightarrow}$ y F_{\leftrightarrow} .

¹Nótese que los símbolos A y B no pertenecen al lenguaje formal de la lógica proposicional. Son variables metalingüísticas utilizadas para definir esquemas de fórmulas.

Tabla 2.1: Definición de las conectivas lógicas.

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Definición 2.4 (Asignación de verdad, [Hod13]). Una *asignación de verdad* (o simplemente *asignación*) es una función $\phi : \mathcal{V} \rightarrow \{1, 0\}$ que asigna a cada variable proposicional un valor de verdad.

Definición 2.5 (Valoración, [JI04]). Dada una asignación de verdad ϕ , una *valoración* ϑ_ϕ es una extensión de ϕ a la que se le añade la interpretación de las conectivas lógicas para poder asignar un valor de verdad a una fórmula A . La función ϑ_ϕ se define inductivamente como:

- (1) $\vartheta_\phi(A) = \phi(A)$ si $A \in \mathcal{V}$;
- (2) $\vartheta_\phi(\neg A) = F_\neg(\vartheta_\phi(A))$;
- (3) $\vartheta_\phi(A \wedge B) = F_\wedge(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (4) $\vartheta_\phi(A \vee B) = F_\vee(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (5) $\vartheta_\phi(A \rightarrow B) = F_\rightarrow(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (6) $\vartheta_\phi(A \leftrightarrow B) = F_\leftrightarrow(\vartheta_\phi(A), \vartheta_\phi(B))$.

Ejemplo 2.4. Sea A la fórmula $\neg(p_1 \vee p_2) \rightarrow (p_1 \rightarrow p_3)$ y ϕ la asignación de verdad definida como $\phi(p_1) = \phi(p_2) = 1$ y $\phi(p_3) = 0$. El valor de verdad de A bajo la valoración ϑ_ϕ es el siguiente:

$$\begin{aligned}
 \vartheta_\phi(A) &= \vartheta_\phi(\neg(p_1 \vee p_2) \rightarrow (p_1 \rightarrow p_3)) \\
 &= F_\rightarrow(\vartheta_\phi(\neg(p_1 \vee p_2)), \vartheta_\phi(p_1 \rightarrow p_3)) \\
 &= F_\rightarrow(F_\neg(\vartheta_\phi(p_1 \vee p_2)), F_\rightarrow(\vartheta_\phi(p_1), \vartheta_\phi(p_3))) \\
 &= F_\rightarrow(F_\neg(F_\vee(\vartheta_\phi(p_1), \vartheta_\phi(p_2))), F_\rightarrow(\vartheta_\phi(p_1), \vartheta_\phi(p_3))) \\
 &= F_\rightarrow(F_\neg(F_\vee(\phi(p_1), \phi(p_2))), F_\rightarrow(\phi(p_1), \phi(p_3))) \\
 &= F_\rightarrow(F_\neg(F_\vee(1, 1)), F_\rightarrow(1, 0)) \\
 &= 1
 \end{aligned}$$

En lo que sigue, denotaremos una valoración simplemente por ϑ cuando no queramos hacer énfasis en la asignación ϕ empleada. El concepto de valoración permite definir la equivalencia de dos fórmulas desde el punto de vista lógico.

Definición 2.6 (Equivalencia lógica, [JI04]). Dadas dos fórmulas A y B , se dice que son *lógicamente equivalentes*, denotado por $A \Leftrightarrow B$, si y sólo si para toda asignación ϕ , $\vartheta_\phi(A) = \vartheta_\phi(B)$.

Además, se dice que una fórmula A es: una *tautología*, si toma el valor de verdad 1 para toda valoración; una *contradicción*, si toma el valor de verdad 0 para toda valoración; y una *contingencia*, si toma el valor de verdad 1 para algunas valoraciones y 0 para otras. También se dice que A es *satisfacible* si toma el valor de verdad 1 para alguna valoración, o *insatisfacible* en caso contrario.

Ejemplo 2.5. La fórmula mostrada en el ejemplo 2.4 es una tautología, ya que su valor de verdad es 1 para cualquier valoración, como se observa en la siguiente tabla de verdad.

p_1	p_2	p_3	$\neg(p_1 \vee p_2)$	$p_1 \rightarrow p_3$	$\neg(p_1 \vee p_2) \rightarrow (p_1 \rightarrow p_3)$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	0	1	1

Saber a qué categoría pertenece una fórmula es decidible: basta con calcular la tabla de verdad de la fórmula correspondiente, como en el ejemplo 2.5. No obstante, confirmar la corrección de un argumento mediante este método es inabordable cuando el número de variables proposicionales crece. Por esta razón, a continuación se introduce un cálculo deductivo para el sistema formal de la lógica proposicional con el fin de poder demostrar la validez de un argumento con métodos puramente sintácticos.

Cálculo deductivo

Como se indicó al principio del capítulo, muchas decisiones que se toman al caracterizar un sistema formal son arbitrarias. Por ello, existen diferentes axiomatizaciones –equivalentes– del cálculo proposicional [II65]. Aquí, se presenta uno cuyos axiomas se deben a Łukasiewicz.

Axiomas. Sean A , B y C fórmulas bien formadas, los siguientes son axiomas del sistema:²

$$(L1) (A \rightarrow (B \rightarrow A))$$

$$(L2) ((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$$

$$(L3) (((\neg A) \rightarrow (\neg B)) \rightarrow (B \rightarrow A))$$

Reglas de inferencia. Sólo hay una regla de inferencia, denominada *modus ponens*:

$$\frac{A, A \rightarrow B}{B} \text{ (MP)}$$

Nótese que en este sistema sólo se usan dos de las conectivas lógicas: la negación (\neg) y el condicional material (\rightarrow).³ El resto pueden obtenerse por equivalencia utilizando estas dos conectivas:

²Téngase en cuenta que **L1**, **L2** y **L3** no son propiamente fórmulas sino *metafórmulas*, por lo que, en realidad, representan *esquemas de axiomas*, es decir, conjuntos infinitos de axiomas, uno por cada instancia que se derive de la metafórmula al concretar las metavariables A , B y C .

³Se dice que $\{\neg, \rightarrow\}$ es un conjunto *adecuado* de conectivas, ya que cualquier función de verdad de la lógica proposicional puede expresarse como una fórmula bien formada utilizando sólo estas conectivas.

- $(A \wedge B)$ es equivalente a $(\neg(A \rightarrow (\neg B)))$;
- $(A \vee B)$ es equivalente a $((\neg A) \rightarrow B)$;
- y $(A \leftrightarrow B)$ es equivalente a $(\neg((A \rightarrow B) \rightarrow (\neg(B \rightarrow A))))$.

Ejemplo 2.6. Para toda fórmula bien formada A , la (meta)fórmula $(A \rightarrow A)$ es un teorema del sistema formal de la lógica proposicional, ya que es posible construir una deducción sin premisas de $(A \rightarrow A)$:

(1)	$((A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)))$	L2
(2)	$(A \rightarrow ((A \rightarrow A) \rightarrow A))$	L1
(3)	$((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$	MP _{1,2}
(4)	$(A \rightarrow (A \rightarrow A))$	L1
(5)	$(A \rightarrow A)$	MP _{3,4}

Limitaciones

El siguiente ejemplo revela las limitaciones de la lógica proposicional, que usa exclusivamente propiedades de las conectivas sin ocuparse del análisis interno de las proposiciones.

Ejemplo 2.7. El argumento correcto que sigue:

$$\frac{\begin{array}{l} \text{Todos los hombres son mortales,} \\ \text{Sócrates es hombre} \end{array}}{\therefore \text{Sócrates es mortal}}$$

reproduce un ejemplo clásico de silogismo que se formaliza en la lógica de proposiciones mediante el argumento $(p_1, p_2 \therefore p_3)$, donde p_1 y p_2 son las premisas y p_3 es la conclusión. Ahora bien, en la lógica referida no se puede deducir la conclusión p_3 a partir de esas premisas. Para concluir que “Sócrates es mortal” es necesario un análisis más preciso de las oraciones y un lenguaje formal más rico.

La lógica de predicados, que se aborda a continuación, extiende la lógica de proposiciones incorporando el concepto de predicado y de término, que formalizan el predicado verbal y el sujeto del enunciado y aportan, como es de esperar, mayor expresividad y capacidad deductiva al lenguaje lógico.

2.1.3. Lógica de predicados

La *lógica de predicados* (que aquí será sinónimo de *lenguaje de primer orden*) responde a la necesidad de disponer un lenguaje formal específico que capte la riqueza del lenguaje natural y la exprese con la mayor precisión posible, superando las limitaciones de la lógica proposicional que no interviene en analizar los enunciados elementales [JIA07].

Lenguaje formal

El alfabeto de cada formalismo está constituido por dos clases de símbolos: los comunes a todo formalismo y los propios a él.

Alfabeto. El alfabeto de la lógica de predicados está formado por un conjunto infinito numerable de variables $\mathcal{V} = \{x_1, x_2, x_3, \dots\}$, las conectivas lógicas $\neg, \wedge, \vee, \rightarrow$ y \leftrightarrow , y los cuantificadores \forall (para todo) y \exists (existe). También se suelen incluir algunos signos de puntuación como “(” y “)”. Además, cada formalismo puede disponer desde uno hasta una cantidad infinita numerable de: símbolos de *constante* $\mathcal{C} = \{a_1, a_2, a_3, \dots\}$, símbolos de *función* n -arios $\mathcal{F} = \{f_1^n, f_2^n, f_3^n, \dots\}$, y símbolos de *predicado* n -arios $\mathcal{P} = \{p_1^n, p_2^n, p_3^n, \dots\}$.

Antes de especificar las reglas de formación de las fórmulas bien formadas en la lógica de predicados, es necesario introducir los conceptos de *término* y *fórmula atómica*.

Definición 2.7 (Término). Un *término* es una expresión en la que intervienen símbolos de variable, símbolos de constante y símbolos de función, que se define conforme a las siguientes reglas:

- (1) Todo símbolo de variable o constante es un término.
- (2) Si t_1, t_2, \dots, t_n son términos y f^n es un símbolo de función n -ario, entonces $f^n(t_1, t_2, \dots, t_n)$ es un término.

Denotamos por \mathcal{T} el conjunto de todos los términos.

Definición 2.8 (Fórmula atómica). Una *fórmula atómica* (o *átomo*) es una expresión en la que intervienen términos y símbolos de predicado, que se define conforme a la siguiente regla:

- (1) Si t_1, t_2, \dots, t_n son términos y p^n es un símbolo de predicado n -ario, entonces $p^n(t_1, t_2, \dots, t_n)$ es una fórmula atómica.

Los términos son los elementos de la lógica de predicados que se interpretan como objetos de un universo de discurso, mientras que las fórmulas atómicas son las expresiones más simples del lenguaje que se interpretan como enunciados. Cuando un término o átomo no contiene variables, se dice que es *básico*.

Fórmula bien formada. Las fórmulas bien formadas de la lógica de predicados se definen inductivamente conforme a las siguientes reglas:

- (1) Toda fórmula atómica es una fórmula bien formada.
- (2) Si A y B son fórmulas bien formadas, también lo son $\neg A, \neg B, (A \wedge B), (A \vee B), (A \rightarrow B)$ y $(A \leftrightarrow B)$.
- (3) Si x es un símbolo de variable y A es una fórmula bien formada, también lo son $(\forall x)A$ y $(\exists x)A$.

El *radio de acción* de un cuantificador en la fórmula $(\forall x)A$ o $(\exists x)A$ es A . Se dice que la ocurrencia de una variable x en una fórmula A es *ligada* si aparece dentro del radio de acción de un cuantificador. En caso contrario, se dice que la ocurrencia es *libre*. Además, una fórmula bien formada es *cerrada* si en ella no aparecen variables libres, o *abierta* en caso contrario.

A continuación se introduce un importante tipo de fórmulas denominadas *cláusulas*, que tienen un papel fundamental en la demostración automática de teoremas, como se verá más adelante.

Definición 2.9 (Literal). Un *literal* es una fórmula atómica o su negación. Un *literal positivo* es una fórmula atómica. Un *literal negativo* es su negación.

Definición 2.10 (Cláusula). Una *cláusula* es una fórmula de la forma:

$$(\forall x_1) \dots (\forall x_n)(L_1 \vee \dots \vee L_m)$$

donde L_1, \dots, L_m son literales y x_1, \dots, x_n son todas las variables que ocurren en L_1, \dots, L_m .

Semántica

Al igual que en la lógica proposicional, para poder discutir sobre la verdad o falsedad de una fórmula en un lenguaje de primer orden, es necesario asignar un significado a sus símbolos. Por lo tanto, interpretar un formalismo consiste en seleccionar un dominio al que se referirán las variables, y asignar significados a los símbolos de constante, función y predicado del formalismo [Llo12].

Definición 2.11 (Interpretación, [JIA07]). Una interpretación \mathcal{I} es un par $(\mathcal{D}_{\mathcal{I}}, \mathcal{J})$ que consiste en un conjunto no vacío $\mathcal{D}_{\mathcal{I}}$ (el dominio de \mathcal{I}), y una aplicación \mathcal{J} que asigna:

- A cada símbolo de constante a_i , un elemento del dominio $\mathcal{J}(a_i) \in \mathcal{D}_{\mathcal{I}}$.
- A cada símbolo de función n -ario f_i^n , una función $\mathcal{J}(f_i^n) : \mathcal{D}_{\mathcal{I}}^n \rightarrow \mathcal{D}_{\mathcal{I}}$.
- A cada símbolo de predicado n -ario p_i^n , una relación $\mathcal{J}(p_i^n) \subseteq \mathcal{D}_{\mathcal{I}}^n$.

Una vez se ha interpretado el formalismo, se puede discutir la verdad o falsedad de una fórmula. No obstante, falta un paso previo si la fórmula no es cerrada: asignar valores a las variables que aparecen libres.

Definición 2.12 (Asignación en \mathcal{I}). Una *asignación en \mathcal{I}* es una función $\phi : \mathcal{V} \rightarrow \mathcal{D}_{\mathcal{I}}$ que asigna a cada variable un elemento del dominio de interpretación. Una asignación puede extenderse a todos los términos del lenguaje inductivamente como sigue:

- $\phi(a) = \mathcal{J}(a)$ si $a \in \mathcal{C}$;
- $\phi(f_i^n(t_1, \dots, t_n)) = \mathcal{J}(f_i^n)(\phi(t_1), \dots, \phi(t_n))$ si $f_i^n \in \mathcal{F}$ y t_1, \dots, t_n son términos.

Definición 2.13 (Valoración en \mathcal{I}). Dada una asignación ϕ en \mathcal{I} , una *valoración* ϑ_ϕ en \mathcal{I} es una extensión de ϕ a la que se le añade la interpretación de las conectivas lógicas, los cuantificadores y las fórmulas atómicas para poder asignar un valor de verdad a una fórmula. La función ϑ_ϕ se define inductivamente como:

- (1) $\vartheta_\phi(\neg A) = F_\neg(\vartheta_\phi(A))$;
- (2) $\vartheta_\phi(A \wedge B) = F_\wedge(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (3) $\vartheta_\phi(A \vee B) = F_\vee(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (4) $\vartheta_\phi(A \rightarrow B) = F_\rightarrow(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (5) $\vartheta_\phi(A \leftrightarrow B) = F_\leftrightarrow(\vartheta_\phi(A), \vartheta_\phi(B))$;
- (6) $\vartheta_\phi((\forall x)A) = 1$ si y solo si $\vartheta_{\phi'}(A) = 1$ para toda asignación ϕ' que difiere de ϕ solo en el valor de x , en caso contrario $\vartheta_\phi((\forall x)A) = 0$;
- (7) $\vartheta_\phi((\exists x)A) = 1$ si y solo si $\vartheta_{\phi'}(A) = 1$ para alguna asignación ϕ' que difiere de ϕ solo en el valor de x , en caso contrario $\vartheta_\phi((\exists x)A) = 0$;
- (8) $\vartheta_\phi(p^n(t_1, \dots, t_n)) = 1$ si y solo si $(\phi(t_1), \dots, \phi(t_n)) \in \mathcal{J}(p^n)$, en caso contrario $\vartheta_\phi(p^n(t_1, \dots, t_n)) = 0$.

Se dice que una fórmula A es: *verdadera* en \mathcal{I} , si toda valoración en \mathcal{I} satisface A ; *falsa* en \mathcal{I} , si no existe ninguna valoración en \mathcal{I} que satisfaga A ; *lógicamente válida*, si para toda interpretación \mathcal{I} , A es verdadera en \mathcal{I} ; *insatisfacible*, si para toda interpretación \mathcal{I} , A es falsa en \mathcal{I} ; y *satisfacible*, si existe una interpretación \mathcal{I} y una valoración en ella que satisface A en \mathcal{I} .

Nótese que, cuando la fórmula es cerrada, su valor de verdad no depende de una asignación en particular. En tal caso, se puede hablar de la satisfacibilidad de una fórmula en una interpretación sin ambigüedad [Llo12].

Definición 2.14 (Modelo). Sea \mathcal{I} una interpretación de un lenguaje de primer orden \mathcal{L} y A una fórmula cerrada de \mathcal{L} . Entonces, \mathcal{I} es *modelo* de A si A es verdadera en \mathcal{I} . Sea Γ un conjunto de fórmulas cerradas de \mathcal{L} , \mathcal{I} es modelo de Γ si es modelo para cada una de las fórmulas de Γ .

Ejemplo 2.8. Dada la fórmula $(\forall x_1)(\forall x_2)p_1^2(f_1^2(x_1, x_2), f_1^2(x_2, x_1))$, una posible interpretación \mathcal{I} es aquella que asigna como dominio el conjunto de los números naturales \mathbb{N} , y que al símbolo de función f_1^2 le asigna la función *suma* ($+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$), y al símbolo de predicado p_1^2 la relación de *igualdad* ($= \subset \mathbb{N} \times \mathbb{N}$). En \mathcal{I} , la fórmula expresa el enunciado verdadero “para todo par de números naturales, su suma es conmutativa”.

En programación lógica, un programa es un conjunto de cláusulas Δ y el interés reside en, dado un objetivo \mathcal{G} , conocer si \mathcal{G} se puede deducir a partir de Δ . Por lo tanto, el concepto de *consecuencia lógica* es de especial interés.

Definición 2.15 (Consecuencia lógica). Sea Γ un conjunto de fórmulas cerradas y A una fórmula cerrada de un lenguaje de primer orden \mathcal{L} . Entonces, A es *consecuencia lógica* de Γ , denotado $\Gamma \vDash A$, si para toda interpretación \mathcal{I} de \mathcal{L} , si \mathcal{I} es modelo de Γ entonces \mathcal{I} es modelo de A .

Proposición 2.1. Sea Γ un conjunto de fórmulas cerradas y A una fórmula cerrada de un lenguaje de primer orden \mathcal{L} . Entonces, A es consecuencia lógica de Γ si y solo si $\Gamma \cup \{\neg A\}$ es insatisfacible.

Así, el principal problema a resolver ahora es el de demostrar la insatisfacibilidad de $\Delta \cup \{\neg \mathcal{G}\}$, donde Δ es un programa y \mathcal{G} un objetivo. Según la definición, esto implicaría mostrar que toda interpretación \mathcal{I} no es modelo de $\Delta \cup \{\neg \mathcal{G}\}$, no obstante, esto es inviable ya que se pueden formar infinitos dominios de interpretación y es imposible considerarlos todos [Llo12]. Por suerte, existe un dominio de interpretación concreto en el que basta explorar la insatisfacibilidad de una fórmula para probar su falsedad sobre cualquier interpretación. Este dominio se denomina *universo de Herbrand*.

Teorema de Herbrand

Un conjunto de fórmulas Γ genera un lenguaje de primer orden \mathcal{L} cuyo alfabeto está constituido por los símbolos de variable, constante, función y predicado que aparecen en dichas fórmulas.

Definición 2.16 (Universo de Herbrand). Dado un lenguaje de primer orden \mathcal{L} , el *universo de Herbrand* $\mathcal{U}_{\mathcal{L}}$ de \mathcal{L} es el conjunto de todos los términos básicos de \mathcal{L} . (En caso de que \mathcal{L} no contenga símbolos de constante, se añade una constante arbitraria a para poder formar términos básicos).

Definición 2.17 (Base de Herbrand). Dado un lenguaje de primer orden \mathcal{L} , la *base de Herbrand* $\mathcal{B}_{\mathcal{L}}$ de \mathcal{L} es el conjunto de todos los átomos básicos de \mathcal{L} .

Ejemplo 2.9. Sea Δ el conjunto de cláusulas $\{p_1^1(a_1), (\forall x_1)(p_1^1(f_1^1(x_1)) \vee \neg p_1^1(x_1))\}$ cuyo lenguaje de primer orden \mathcal{L} subyacente contiene los símbolos de constante $\mathcal{C} = \{a_1\}$, los símbolos de función $\mathcal{F} = \{f_1^1\}$ y los símbolos de predicado $\mathcal{P} = \{p_1^1\}$. Entonces, el universo de Herbrand de \mathcal{L} es

$$\mathcal{U}_{\mathcal{L}} = \{a_1, f_1^1(a_1), f_1^1(f_1^1(a_1)), f_1^1(f_1^1(f_1^1(a_1))), \dots\}.$$

Además, la base de Herbrand de \mathcal{L} es

$$\mathcal{B}_{\mathcal{L}} = \{p_1^1(a_1), p_1^1(f_1^1(a_1)), p_1^1(f_1^1(f_1^1(a_1))), p_1^1(f_1^1(f_1^1(f_1^1(a_1))))\}.$$

Definición 2.18 (Interpretación de Herbrand, [JIA07]). Dado un lenguaje de primer orden \mathcal{L} , una *interpretación de Herbrand* de \mathcal{L} es una interpretación $\mathcal{I} = (\mathcal{U}_{\mathcal{L}}, \mathcal{J})$ de \mathcal{L} , donde el dominio es el propio universo de Herbrand para \mathcal{L} , y \mathcal{J} es una aplicación que asigna:

- a) A cada símbolo de constante a_i de \mathcal{L} , el mismo símbolo a_i : $\mathcal{J}(a_i) = a_i$.

- b) A cada símbolo de función n -ario f_i^n de \mathcal{L} , una función $\mathcal{J}(f_i^n) : \mathcal{U}_{\mathcal{L}}^n \rightarrow \mathcal{U}_{\mathcal{L}}$ que asocia a cada secuencia de términos básicos t_1, \dots, t_n de $\mathcal{U}_{\mathcal{L}}$ el término básico $f_i^n(t_1, \dots, t_n)$ de $\mathcal{U}_{\mathcal{L}}$.
- c) A cada símbolo de predicado n -ario p_i^n de \mathcal{L} , cualquier subconjunto $\mathcal{J}(p_i^n)$ de n -tuplas de términos básicos de $\mathcal{U}_{\mathcal{L}}^n$.

Dado que para las interpretaciones de Herbrand el dominio y el significado de los símbolos de constante y de función son fijos, una interpretación de Herbrand queda caracterizada por un subconjunto de la base de Herbrand, que representa los átomos básicos que son ciertos en dicha interpretación.

Definición 2.19 (Modelo de Herbrand). Sea \mathcal{L} un lenguaje de primer orden y Γ un conjunto de fórmulas cerradas de \mathcal{L} . Un *modelo de Herbrand* de Γ es una interpretación de Herbrand de \mathcal{L} que es modelo de Γ .

Una de las propiedades más importantes de las interpretaciones de Herbrand se establece en las siguientes proposiciones.

Proposición 2.2. *Sea Δ un conjunto de cláusulas satisfacible. Entonces, Δ tiene un modelo de Herbrand.*

Proposición 2.3. *Sea Δ un conjunto de cláusulas. Entonces, Δ es insatisfacible si y solo si Δ no tiene ningún modelo de Herbrand.*

A partir de estos resultados se infiere que para probar la insatisfacibilidad de un conjunto de cláusulas Δ , basta con analizar si Δ es insatisfacible para toda interpretación de Herbrand. Más aún, el *teorema de Herbrand* [Her30] establece que un conjunto de cláusulas Δ es insatisfacible si y solo si existe un conjunto de *instancias básicas* de Δ que es insatisfacible.

Definición 2.20 (Instancia básica). Sea Δ un conjunto de cláusulas de un lenguaje de primer orden \mathcal{L} . Una *instancia básica* de una cláusula C de Δ es la cláusula obtenida tras sustituir cada una de las variables de C por elementos de $\mathcal{U}_{\mathcal{L}}$.

Esto sugiere un procedimiento de prueba basado en criterios semánticos en el que, para comprobar la insatisfacibilidad de un conjunto de cláusulas Δ , se genera un conjunto Δ'_i de instancias básicas hasta un determinado nivel i , y se prueba la insatisfacibilidad de Δ'_i con algún método de satisfacibilidad de la lógica de proposiciones. Si Δ es insatisfacible, Δ'_i será insatisfacible para algún conjunto de instancias básicas de un determinado nivel i y el procedimiento terminará. No obstante, si Δ es satisfacible, el procedimiento puede no terminar.

Ejemplo 2.10. Dado el conjunto de cláusulas Δ del ejemplo 2.9, supongamos que queremos comprobar que el objetivo $\mathcal{G} \equiv p_1^1(f_1^1(f_1^1(a_1)))$ es consecuencia lógica de Δ . Por la proposición 2.1, esto es equivalente a demostrar que $\Delta' = \Delta \cup \{\neg \mathcal{G}\}$ es insatisfacible.

- (1) $\mathcal{U}_0 = \{a_1\}$, $\Delta'_0 = \{p_1^1(a_1), p_1^1(f_1^1(a_1)) \vee \neg p_1^1(a_1), \neg p_1^1(f_1^1(f_1^1(a_1)))\}$. Δ'_0 no es insatisfacible.

- (2) $\mathcal{U}_1 = \{a_1, f_1^1(a_1)\}$, $\Delta'_1 = \{p_1^1(a_1), p_1^1(f_1^1(a_1)) \vee \neg p_1^1(a_1), p_1^1(f_1^1(f_1^1(a_1))) \vee \neg p_1^1(f_1^1(a_1)), \neg p_1^1(f_1^1(f_1^1(a_1)))\}$. Δ'_1 es insatisfacible.

Por lo tanto, Δ' es insatisfacible y \mathcal{G} es consecuencia lógica de Δ .

2.2. Programación lógica

Para evitar la ineficiencia de la implementación directa del teorema de Herbrand, que genera de manera sistemática conjuntos de instancias básicas, en esta sección se introduce el *principio de resolución* de Robinson [Rob65]: una regla de inferencia para fórmulas en forma clausal que –junto con la *unificación*– constituye un cálculo deductivo completo, y proporciona un método sintáctico de prueba por refutación.

El objetivo fundamental de la programación lógica es la ejecución eficiente de programas lógicos, computando para qué valores de las variables de una cláusula objetivo se cumple que el objetivo es consecuencia lógica del programa. Para ello, se hace uso de un tipo particular de cláusulas: las *cláusulas de Horn*.

Definición 2.21 (Cláusula de Horn). Una *cláusula de Horn* (o *cláusula definida*) es una cláusula con, como mucho, un literal positivo. El literal positivo se denomina *cabeza*, y el resto de literales se denominan *cuerpo*.

Por lo tanto, una cláusula de Horn es una cláusula con la siguiente forma: $A_1 \vee \dots \vee A_k \vee \neg B_1 \vee \dots \vee \neg B_n$ con ($k = 1$ o $k = 0$) y $n \geq 0$. En el ámbito de la programación lógica, las cláusulas de Horn se suelen expresar en forma de implicación inversa: $A_1 \vee \dots \vee A_k \leftarrow B_1 \wedge \dots \wedge B_n$, que es una fórmula lógicamente equivalente a la anterior y permite una lectura más apropiada desde el punto de vista procedural.

Definición 2.22 (Programa lógico definido). Un *programa definido* Π es un conjunto de cláusulas definidas con cabeza.

Ejemplo 2.11. El conjunto de cláusulas

$$\Pi = \begin{cases} C_1 : & \text{app}(\text{nil}, x_1, x_1) & \leftarrow \\ C_2 : & \text{app}(\text{cons}(x_1, x_2), x_3, \text{cons}(x_1, x_4)) & \leftarrow \text{app}(x_2, x_3, x_4) \end{cases}$$

es un ejemplo de programa lógico definido. Las cláusulas C_1 y C_2 definen el predicado *app*.

2.2.1. Sustituciones

El principal objetivo de computar *enlaces* se lleva a cabo por medio de la unificación sintáctica de términos. Estos enlaces se formalizan y manipulan mediante el concepto de *sustitución*.

Definición 2.23 (Sustitución). Sea \mathcal{L} un lenguaje de primer orden. Una *sustitución* es una aplicación $\theta : \mathcal{V} \rightarrow \mathcal{T}$ que asigna un término de \mathcal{L} a cada variable del conjunto \mathcal{V} .

La notación $\{x_1/t_1, \dots, x_k/t_k\}$ hace referencia a una sustitución que asigna a cada variable x_i el término correspondiente t_i , para todo $1 \leq i \leq k$, y que al resto de variables asigna la propia variable. Se denomina *dominio* de la sustitución al conjunto de variables reemplazadas, $dom(\theta) = \{x \in \mathcal{V} : \theta(x) \neq x\}$. Denotamos por $\theta[X]$ a la sustitución obtenida tras restringir el dominio de θ al conjunto de variables $X \subseteq \mathcal{V}$.

Definición 2.24 (Instancia, [CL14]). La aplicación de una sustitución $\theta = \{x_1/t_1, \dots, x_n/t_n\}$ a una expresión E , denotado $E\theta$, se obtiene reemplazando —simultáneamente— cada ocurrencia de x_i en la expresión E por el término t_i . Se dice que $E\theta$ es una *instancia* de E .

Como toda aplicación, las sustituciones pueden componerse. Dadas dos sustituciones θ y σ , su composición es la sustitución $\theta\sigma$ tal que $E(\theta\sigma) = (E\theta)\sigma$. Las sustituciones conforman un monoide bajo la composición, ya que es una operación interna, asociativa y tiene elemento neutro: la *sustitución identidad* (denotada id), que asigna cada variable a ella misma.

Además, la composición induce una relación de preorden (esto es, una relación reflexiva y transitiva) sobre las sustituciones. Dadas dos sustituciones θ y σ , se dice que θ es más general que σ , denotado $\theta \leq \sigma$, si y solo si existe una sustitución δ tal que $\theta = \sigma\delta$.

Definición 2.25 (Sustitución idempotente). Una sustitución θ es *idempotente* si y solo si $\theta\theta = \theta$. Equivalentemente, θ es idempotente si y solo si $dom(\theta) \cap vars(ran(\theta)) = \emptyset$ [Pal90].

Estas sustituciones son de especial interés, ya que todas las sustituciones computadas por el principio de resolución son idempotentes.

Definición 2.26 (Renombramiento). Una sustitución ρ es un *renombramiento* si es una aplicación biyectiva $\rho : \mathcal{V} \rightarrow \mathcal{V}$. Es decir, ρ es un renombramiento si existe una sustitución inversa ρ^{-1} tal que $\rho\rho^{-1} = \rho^{-1}\rho = id$.

Dadas dos expresiones E_1 y E_2 , se dice que son *variantes* si existen dos renombramientos θ y σ tal que $E_1 = E_2\theta$ y $E_2 = E_1\sigma$.

2.2.2. Unificación

La unificación es un proceso por el que dos o más expresiones se vuelven sintácticamente idénticas mediante la aplicación de una sustitución, denominada *unificador*. Este tipo de sustituciones sintetiza la noción de cómputo en el contexto de la programación lógica [JIA07].

Definición 2.27 (Unificador). Una sustitución θ es un *unificador* de un conjunto de expresiones $S = \{E_1, \dots, E_n\}$ si y solo si $E_1\theta = \dots = E_n\theta$. Se dice que el conjunto S es *unificable* si existe un unificador para él.

Definición 2.28 (Unificador más general). Un unificador θ de un conjunto de expresiones S es un *unificador más general* (o *m.g.u.*) de S si y solo si, para cualquier otro unificador θ' de S , se verifica que $\theta \leq \theta'$.

Este proceso fue estudiado por primera vez por J. Herbrand en teoría de la demostración [Her30], y posteriormente fue denominado *unificación* por J. A. Robinson en su trabajo preliminar sobre demostración automática de teoremas en la lógica de primer orden [Rob65]. Desde entonces, se han propuesto diversos algoritmos de unificación. Aquí, se presenta el algoritmo dado por A. Martelli y U. Montanari en [MM82], reformulado como un sistema de transición de estados equipado con una relación de unificación (denotada \Rightarrow_{mgu}). La relación de unificación \Rightarrow_{mgu} es la mínima relación definida por el conjunto de reglas de transición dadas en la figura 2.1.

$$\frac{\langle \{f(t_1, \dots, t_n) = f(s_1, \dots, s_n)\} \cup S, \theta \rangle}{\langle \{t_1 = s_1, \dots, t_n = s_n\} \cup S, \theta \rangle} \quad (1)$$

$$\frac{\langle \{x = x\} \cup S, \theta \rangle}{\langle S, \theta \rangle} \quad (2) \quad \frac{\langle \{x = t\} \cup S, \theta \rangle, \quad x \notin \text{vars}(t)}{\langle S\{x/t\}, \theta\{x/t\} \rangle} \quad (3)$$

$$\frac{\langle \{t = x\} \cup S, \theta \rangle}{\langle \{x = t\} \cup S, \theta \rangle} \quad (4) \quad \frac{\langle \{x = t\} \cup S, \theta \rangle, \quad x \in \text{vars}(t)}{\langle \text{fallo}, \theta \rangle} \quad (5)$$

$$\frac{\langle \{f_1(t_1, \dots, t_n) = f_2(s_1, \dots, s_n)\} \cup S, \theta \rangle, \quad f_1 \neq f_2}{\langle \text{fallo}, \theta \rangle} \quad (6)$$

Figura 2.1: Reglas de transición de la relación de unificación (\Rightarrow_{mgu}).

La unificación de las expresiones E_1 y E_2 se obtiene por medio de una secuencia de transformaciones, empezando por el estado inicial $\langle G \equiv \{E_1 = E_2\}, id \rangle$:

$$\langle G, id \rangle \Rightarrow_{mgu} \langle G_1, \theta_1 \rangle \Rightarrow_{mgu} \dots \Rightarrow_{mgu} \langle G_n, \theta_n \rangle.$$

Cuando se alcanza el estado final $\langle G_n, \theta_n \rangle$, con $G_n = \emptyset$, las expresiones E_1 y E_2 son unificables con m.g.u. θ_n , denotado $\text{mgu}(E_1, E_2) = \theta_n$. Si las expresiones E_1 y E_2 no son unificables, el algoritmo termina con un estado de fallo $\langle G_n, \theta_n \rangle \equiv \langle \text{fallo}, \theta_n \rangle$.

2.2.3. Resolución

El principal inconveniente de los métodos de prueba basados en el teorema de Herbrand es que requerían generar conjuntos de cláusulas básicas hasta obtener una insatisficible, dando lugar al problema denominado “explosión combinatoria”. Este problema de la explosión combinatoria fue eliminado por J. A. Robinson mediante la introducción de la unificación [Rob65]. Dadas dos cláusulas C_1 y C_2 sin variables en común, la regla de resolución puede resolverlas si contienen literales complementarios:

$$\frac{L_1 \in C_1, L_2 \in C_2, \text{vars}(C_1) \cap \text{vars}(C_2) = \emptyset, \text{mgu}(L_1, \neg L_2) = \theta}{((C_1 - \{L_1\}) \cup (C_2 - \{L_2\}))\theta} \quad (\text{RES})$$

Dado un conjunto de cláusulas Δ , si existe una deducción de la cláusula vacía \square a partir de Δ usando la regla de resolución, entonces Δ es insatisfacible. El problema es que no hay una estrategia fija para guiar la búsqueda de dicha deducción, y la aplicación sin restricciones de la regla de resolución puede generar cláusulas redundantes o irrelevantes para los objetivos de la prueba [CL14].

Este problema es mitigado en programación lógica mediante la estrategia de resolución *SLD* (acrónimo de «resolución lineal selectiva para cláusulas definidas»): un refinamiento del procedimiento de refutación por resolución descrito originalmente por R. Kowalski [Kow74] donde, tras cada paso de resolución sobre un objetivo (cuyos literales son todos negativos) con una cláusula del programa (que solo contiene un literal positivo), se obtiene otro objetivo como cláusula central.

Definición 2.29 (Regla de computación). Una *regla de computación* (o *función de selección*) es una función φ que, cuando se aplica a un objetivo \mathcal{G} , selecciona un átomo de \mathcal{G} .

A continuación, se define la semántica operacional de la programación lógica como un sistema de transición de estados. Denotamos por $C[A]$ a una fórmula donde A es una subexpresión (normalmente un átomo) que ocurre en C , mientras que $C[A/A']$ significa el reemplazo de A por A' en C .

Definición 2.30 (Resolución SLD). Sea $\langle \mathcal{G}, \theta \rangle$ un estado, donde \mathcal{G} es un objetivo y θ una sustitución, y sea Π un programa lógico definido. Dada una función de selección φ , definimos la resolución SLD como un sistema de transición cuya relación de transición \Rightarrow_{SLD} es la menor relación binaria que satisface:

$$\frac{\langle \mathcal{G}, \theta \rangle, \varphi(\mathcal{G}) = A, (H \leftarrow B) \in \Pi, \text{mgu}(A, H) = \sigma}{\langle \mathcal{G}[A/B]\sigma, \theta\sigma \rangle} \quad (\text{SLD})$$

Una *derivación SLD* es una secuencia $\langle \mathcal{G}_0, id \rangle \Rightarrow_{SLD} \langle \mathcal{G}_1, \theta_1 \rangle \Rightarrow_{SLD} \dots \Rightarrow_{SLD} \langle \mathcal{G}_n, \theta_n \rangle$ de pasos de resolución. Una *refutación SLD* es una derivación de éxito, es decir, que conduce a la cláusula vacía: $\langle \mathcal{G}_0, id \rangle \Rightarrow_{SLD} \dots \Rightarrow_{SLD} \langle \square, \theta \rangle$, donde θ es la *respuesta computada* en la derivación.

Ejemplo 2.12. Sea Π el programa lógico definido mostrado en el ejemplo 2.11, y sea \mathcal{G} el objetivo $\leftarrow \text{app}(x_1, x_2, \text{cons}(a, \text{cons}(b, \text{nil})))$. La siguiente es una refutación SLD para el objetivo \mathcal{G} :

$$\begin{aligned} \langle \leftarrow \text{app}(x_1, x_2, \text{cons}(a, \text{cons}(b, \text{nil}))), id \rangle & \Rightarrow_{SLD}^{C_2} \\ \langle \leftarrow \text{app}(x_3, x_4, \text{cons}(b, \text{nil})), \{x_1/\text{cons}(a, x_3), x_2/x_4\} \rangle & \Rightarrow_{SLD}^{C_2} \\ \langle \leftarrow \text{app}(x_5, x_6, \text{nil}), \{x_1/\text{cons}(a, \text{cons}(b, x_5)), x_2/x_6\} \rangle & \Rightarrow_{SLD}^{C_1} \\ \langle \square, \{x_1/\text{cons}(a, \text{cons}(b, \text{nil})), x_2/\text{nil}\} \rangle & \end{aligned}$$

Entonces, la sustitución $\{x_1/\text{cons}(a, \text{cons}(b, \text{nil})), x_2/\text{nil}\}$ es una respuesta computada para el objetivo \mathcal{G} . Tal y como se muestra en la figura 2.2, hay otras dos respuestas computadas asociadas a este objetivo.

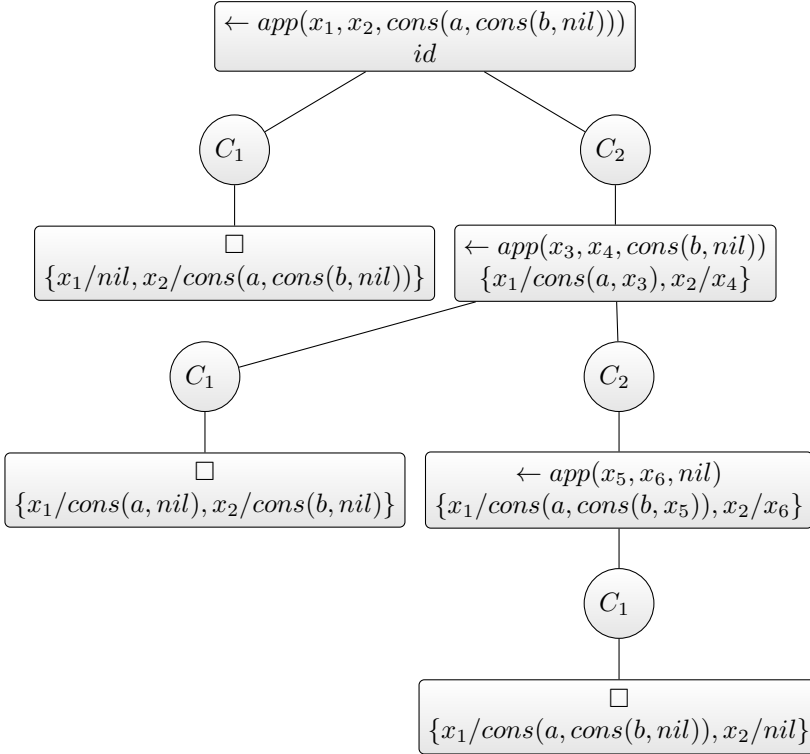


Figura 2.2: Árbol de derivación.

Un aspecto importante del método de resolución SLD es que es independiente de la función de selección utilizada. El siguiente resultado justifica que la selección de un átomo en un objetivo se puede realizar arbitrariamente, sin afectar al resultado de la computación.

Teorema 2.1 (Independencia de la regla de computación, [JIA07]). *Sea Π un programa lógico definido y sea \mathcal{G} un objetivo definido. Si existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ con respuesta computada θ , usando una regla de computación φ , entonces existe también una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ con respuesta computada θ' , usando cualquier otra regla de computación φ' , tal que $\mathcal{G}\theta'$ es una variante de $\mathcal{G}\theta$.*

2.2.4. El lenguaje Prolog

Prolog (abreviatura de «*PRO*grammation en *LOG*ique») es un lenguaje de programación lógica de propósito general, diseñado en 1972 por A. Colmerauer

y P. Roussel en la Universidad de Aix-Marseille [CR96]. En 1995 se publicó el estándar ISO Prolog (ISO/IEC 13211-1) [ISO95] con el fin de promover la portabilidad de los programas Prolog entre los distintos sistemas, especificando la sintaxis, la semántica operacional, los predicados incorporados y la representación de la entrada/salida producida por Prolog.

Sintaxis

El único tipo de dato en Prolog es el *término*. Los términos pueden ser constantes (átomos⁴ o números), variables o términos compuestos:

- (1) Un átomo es cualquier secuencia de caracteres encerrada entre comillas simples; cualquier secuencia de caracteres alfanuméricos y guiones bajos precedida de un carácter en minúsculas; cualquier secuencia formada por los caracteres +, -, *, /, \, ^, >, <, =, :, ., ?, @, #, \$ y &; o uno de los siguientes átomos especiales: !, ;, [], {}.
- (2) Una variable es cualquier secuencia de caracteres alfanuméricos y guiones bajos precedida de un carácter en mayúsculas o de un guion bajo. Una variable llamada “_” se dice que es una *variable anónima*, que unifica con cualquier término sin ligar su valor a la variable.
- (3) Un término compuesto es una estructura formada por un átomo seguido de una serie de términos encerrados entre paréntesis y separados por comas.

Los operadores en Prolog son simplemente una conveniencia de notación y pueden ser definidos por el usuario. En la tabla 2.2 se muestran los operadores predefinidos. Las instrucciones en Prolog se codifican como cláusulas definidas sucedidas de un punto, donde las conectivas lógicas se representan por los operadores (,)/2 (conjunción), (;)/2 (disyunción), (\+)/1 (negación) y (: -)/2 (implicación inversa).

Ejemplo 2.13. El programa lógico definido mostrado en el ejemplo 2.11 puede ser codificado en Prolog como sigue:

```

1  append([], X, X).
2  append([H|T], X, [H|S]) :- append(T, X, S).
```

Semántica operacional

En esencia, un programa Prolog se ejecuta utilizando el mecanismo operacional de la resolución SLD, alejándose de algunos de los supuestos formales de esta. En particular, la regla de computación de Prolog siempre selecciona el átomo situado más a la izquierda dentro del objetivo a resolver, e intenta

⁴En Prolog, un átomo es un identificador sin significado inherente, que se utiliza para representar una constante. No se debe confundir con un átomo (en el sentido de fórmula atómica) en lógica de primer orden.

Tabla 2.2: Operadores predefinidos de Prolog.

Prioridad	Asociatividad	Operadores
1200	xfx	:- -->
1200	fx	:- ?-
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+
700	xfx	= \=
700	xfx	== \== @< @=< @> @>=
700	xfx	=..
700	xfx	is == =\= < =< > >=
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfx	**
200	xfy	^
200	fy	- + \

unificarlo con las cabezas de las cláusulas del programa, considerándolas en el orden textual en el que aparecen dentro del programa. A la hora de unificar dos expresiones, Prolog omite la comprobación de ocurrencia de variables (en las reglas de transición **(3)** y **(5)** de la figura 2.1) por motivos de eficiencia. Además, utiliza una regla de búsqueda en profundidad con vuelta atrás automática.

Predicados incorporados

El estándar ISO Prolog define un gran repertorio de predicados incorporados y estructuras de control que proporcionan mecanismos que no pueden obtenerse en Prolog puro, o que simplemente proveen de facilidades al programador. Entre otros, se incluyen predicados para la unificación y comparación de términos, verificación de tipos, evaluación y comparación aritmética, creación y destrucción de cláusulas, búsqueda de todas las soluciones, procesamiento de átomos, control de ficheros y entrada/salida, manejo de excepciones, etcétera. Puede consultarse un listado completo en [ISO95].

2.3. Lógica difusa

En esta sección se revisan las nociones básicas de la *lógica difusa*, formuladas por primera vez por L. A. Zadeh en su artículo sobre conjuntos difusos [Zad75] con la intención de incorporar a la lógica formal los predicados de carácter impreciso del lenguaje ordinario, lo cual ha permitido iniciar la construcción del razonamiento aproximado.

2.3.1. Conjuntos difusos

La generalización de la *función característica* [Zad65] de un conjunto clásico a un conjunto difuso es la base sobre la que se formalizan otros conceptos, como las interpretaciones difusas o las conectivas lógicas difusas. Un *conjunto clásico* A –cuya relación de pertenencia tiene un carácter discreto– puede definirse a partir de su función característica $\chi_A : \mathcal{U} \rightarrow \{1, 0\}$, de modo que $A = \{x : \chi_A(x) = 1\}$. Así cada elemento x del universo de discurso \mathcal{U} o bien pertenece al conjunto A o no:

$$\forall x \in \mathcal{U} : x \in A \vee x \notin A; \quad A \subseteq \mathcal{U}.$$

En cambio, en los conjuntos difusos la relación de pertenencia tiene un grado. Un *conjunto difuso* $A = \{\mu_A(x)/x : x \in \mathcal{U} \wedge \mu_A(x) > 0\}$ se define a partir de su función característica $\mu_A : \mathcal{U} \rightarrow [0, 1]$, donde cada elemento del universo de discurso $x \in \mathcal{U}$ tiene asociado un grado de pertenencia $\mu_A(x)$ que representa la compatibilidad de x con la característica (predicado) que define el conjunto A . Dado un conjunto clásico A , se observa que la pertenencia difusa es una generalización de la clásica determinada por la función característica χ_A . De esta observación se deduce que todo conjunto clásico es un conjunto difuso; es decir, la noción de conjunto difuso extiende la de conjunto clásico. Además, de forma más general, los grados de pertenencia de los elementos de un conjunto difuso se pueden tomar de un *retículo completo*.

Definición 2.31 (Retículo completo). Un *retículo completo* es un conjunto ordenado (L, \leq) tal que todo subconjunto $S \subseteq L$ tiene ínfimo y supremo. En particular, existen el ínfimo y el supremo de L , denotados respectivamente por \perp y \top .

Así, dado un retículo completo (L, \leq) , un conjunto difuso $A = \{\mu_A(x)/x : x \in \mathcal{U} \wedge \mu_A(x) > \perp\}$ se define a partir de una función característica $\mu_A : \mathcal{U} \rightarrow L$. En lo que sigue, no haremos distinciones entre la noción de conjunto difuso y su función característica, y por conjunto difuso nos referiremos a una función de \mathcal{U} en L .

2.3.2. Interpretaciones y conectivas difusas

Desde el punto de vista semántico, el concepto esencial de la lógica difusa es el de *interpretación difusa*, cuya definición es análoga a la definición 2.11 de interpretación pero asociando a cada símbolo de predicado un conjunto difuso en lugar de una relación.

Definición 2.32 (Interpretación difusa, [JIMP18]). Sea (L, \leq) un retículo completo. Una *interpretación difusa* \mathcal{I} es un par $(\mathcal{D}_{\mathcal{I}}, \mathcal{J})$ que consiste en un conjunto no vacío $\mathcal{D}_{\mathcal{I}}$ (el dominio de \mathcal{I}), y una aplicación \mathcal{J} que asigna:

- a) A cada símbolo de constante a_i , un elemento del dominio $\mathcal{J}(a_i) \in \mathcal{D}_{\mathcal{I}}$.
- b) A cada símbolo de función n -ario f_i^n , una función $\mathcal{J}(f_i^n) : \mathcal{D}_{\mathcal{I}}^n \rightarrow \mathcal{D}_{\mathcal{I}}$.

- c) A cada símbolo de predicado n -ario p_i^n , un conjunto difuso $\mathcal{J}(p_i^n) : \mathcal{D}_{\mathcal{I}}^n \rightarrow L$.

Dada una asignación ϕ en la interpretación difusa $\mathcal{I} = (\mathcal{D}_{\mathcal{I}}, \mathcal{J})$, la *valoración difusa* ϑ_ϕ es una extensión de ϕ a la que se le añade la interpretación de las conectivas difusas, los cuantificadores y las fórmulas atómicas para poder asignar un grado de verdad a una fórmula. Sea $p^n(t_1, \dots, t_n)$ un átomo tal que $\mathcal{J}(p^n) = A$. Entonces, el valor de verdad de $p^n(t_1, \dots, t_n)$ en \mathcal{I} es el grado de verdad determinado por el conjunto difuso A :

$$\vartheta_\phi(p^n(t_1, \dots, t_n)) = A(\phi(t_1), \dots, \phi(t_n)).$$

Una vez se han interpretado las expresiones elementales, es necesario dar significado a las conectivas lógicas difusas con el fin de otorgar un grado de verdad a las expresiones compuestas. Por ejemplo, en el retículo $([0, 1], \leq)$, la conjunción y la disyunción se definen habitualmente por el mínimo y el máximo, respectivamente:

$$\begin{aligned}\vartheta_\phi(A \wedge B) &= \text{mín}\{\vartheta_\phi(A), \vartheta_\phi(B)\}; \\ \vartheta_\phi(A \vee B) &= \text{máx}\{\vartheta_\phi(A), \vartheta_\phi(B)\};\end{aligned}$$

pero, de manera más general, la función de verdad de la conjunción y la disyunción difusas se puede definir también por todo un conjunto de funciones: las *normas triangulares* y las *conormas triangulares*, introducidas por B. Schweizer y A. Sklar [SS83], y cuyas definiciones –generalizadas a retículos completos– proporcionamos a continuación.

Definición 2.33 (Norma triangular). Sea (L, \leq) un retículo completo. Una *norma triangular* (o *t-norma*) es una función $T : L^2 \rightarrow L$ asociativa, conmutativa y monótona creciente en cada componente (esto es, si $x_1 \leq y_1$ y $x_2 \leq y_2$, entonces $T(x_1, x_2) \leq T(y_1, y_2)$) que satisface la condición de frontera $T(x, \top) = T(\top, x) = x$ para todo $x \in L$.

Definición 2.34 (Conorma triangular). Sea (L, \leq) un retículo completo. Una *conorma triangular* (o *t-conorma*) es una función $S : L^2 \rightarrow L$ asociativa, conmutativa y monótona creciente en cada componente que satisface la condición de frontera $S(x, \perp) = S(\perp, x) = x$ para todo $x \in L$.

La función de verdad de la negación difusa se define habitualmente como sigue en el retículo $([0, 1], \leq)$:

$$\vartheta_\phi(\neg A) = 1 - \vartheta_\phi(A);$$

pero, al igual que ocurre con la conjunción y la disyunción difusas, también es posible plantear la negación de forma más general.

Definición 2.35 (Negación fuerte). Sea (L, \leq) un retículo completo. Una *negación fuerte* es una función $N : L \rightarrow L$ continua y estrictamente decreciente, que satisface la condición de frontera $N(\perp) = \top$ y la propiedad $N(N(x)) = x$ para todo $x \in L$.

Dadas una t-norma T y una negación fuerte N en un retículo (L, \leq) , la función $S_N : L^2 \rightarrow L$ definida como $S_N(x, y) = N(T(N(x), N(y)))$ es una t-conorma denominada N -dual de T . De forma análoga, se puede definir una t-norma N -dual a partir de una t-conorma y una negación fuerte.

Ejemplo 2.14. Los pares de t-normas y t-conormas (duales) básicos en el retículo $([0, 1], \leq)$ son los definidos en las lógicas de Gödel, de Łukasiewicz, y del producto (véase [CFF97]):

$$\begin{aligned} T_{godel}(x, y) &= \min\{x, y\}; & S_{godel}(x, y) &= \max\{x, y\}; \\ T_{luka}(x, y) &= \max\{0, x + y - 1\}; & S_{luka}(x, y) &= \min\{x + y, 1\}; \\ T_{prod}(x, y) &= xy; & S_{prod}(x, y) &= x + y - xy. \end{aligned}$$

Es común combinar t-normas y t-conormas para producir nuevas conectivas [Miz89a, Miz89b, Tur92, FC98, DKMS07, KMP04]. Las t-normas y t-conormas son casos particulares de los operadores de agregación [DP85, Yag94]. En [KK99] se considera su definición más general, caracterizada del siguiente modo en un retículo completo.

Definición 2.36 (Operador de agregación, [KK99]). Sea (L, \leq) un retículo completo. Un *operador de agregación* (o *agregador*) es una función $A : L^n \rightarrow L$ monótona creciente⁵ que satisface las condiciones de frontera $A(\top, \dots, \top) = \top$ y $A(\perp, \dots, \perp) = \perp$.

A las condiciones de la definición anterior se añaden en ocasiones otras como la continuidad, la simetría o la idempotencia.

Ejemplo 2.15. Además de las t-normas y las t-conormas, otros ejemplos de operadores de agregación son la media aritmética, la media geométrica o el modificador lingüístico “muy”:

$$A_{aver}(x, y) = (x + y)/2; \quad A_{geom}(x, y) = \sqrt{xy}; \quad A_{very}(x) = x^2.$$

La implicación difusa también admite diversas formulaciones (no equivalentes), que no siempre extienden la implicación de la lógica clásica [TdCC00]. La manera más frecuente de interpretar la implicación difusa en el retículo $([0, 1], \leq)$ es la siguiente:

$$\vartheta_\phi(A \rightarrow B) = \max\{\min\{\vartheta_\phi(A), \vartheta_\phi(B)\}, 1 - \vartheta_\phi(A)\}.$$

Respecto a la interpretación de los cuantificadores \forall y \exists , habitualmente se definen respectivamente por el ínfimo y el supremo:

$$\begin{aligned} \vartheta_\phi((\forall x)A) &= \inf\{\vartheta_{\phi'}(A) : \text{para todo } \phi' \text{ que difiere de } \phi \text{ solo en el valor de } x\}; \\ \vartheta_\phi((\exists x)A) &= \sup\{\vartheta_{\phi'}(A) : \text{para todo } \phi' \text{ que difiere de } \phi \text{ solo en el valor de } x\}. \end{aligned}$$

⁵Si $(x_1, \dots, x_n), (y_1, \dots, y_n) \in L^n$ tal que $x_i \leq y_i$ para todo $1 \leq i \leq n$, entonces $A(x_1, \dots, x_n) \leq A(y_1, \dots, y_n)$.

2.3.3. Relaciones de similitud

Una relación de equivalencia es una relación binaria reflexiva, simétrica y transitiva que permite relacionar y agrupar los elementos de un conjunto en clases de equivalencia. La noción de *relación de similitud* [Zad71, Yin94] es una generalización de las relaciones de equivalencia, donde los elementos del conjunto se relacionan con un determinado grado. Formalizamos a continuación el concepto de relación de similitud, en el contexto de un retículo completo.

Definición 2.37 (Relación de similitud). Dado un dominio \mathcal{U} y un retículo completo (L, \leq) con una t-norma \wedge fija, una *relación de similitud* $\mathcal{R} : \mathcal{U} \times \mathcal{U} \rightarrow L$ es una relación binaria difusa sobre \mathcal{U} , que es reflexiva ($\mathcal{R}(x, x) = \top$ para todo $x \in \mathcal{U}$), simétrica ($\mathcal{R}(x, y) = \mathcal{R}(y, x)$ para todo $x, y \in \mathcal{U}$) y transitiva ($\mathcal{R}(x, y) \wedge \mathcal{R}(y, z) \leq \mathcal{R}(x, z)$ para todo $x, y, z \in \mathcal{U}$).

En lo que sigue, utilizaremos el símbolo \wedge exclusivamente para denotar la t-norma asociada a una relación de similitud.

Sea \mathcal{L} un lenguaje de primer orden sobre un alfabeto Σ . Dada una relación de similitud \mathcal{R} sobre Σ , la relación \mathcal{R} se puede extender a los términos de \mathcal{L} por inducción estructural de la forma usual [Ses02]. Denotamos por $\hat{\mathcal{R}}$ a la extensión de \mathcal{R} definida inductivamente conforme a las siguientes reglas:

- (1) Si $x \in \mathcal{V}$ es una variable, entonces $\hat{\mathcal{R}}(x, x) = \top$.
- (2) Si $f_1^n, f_2^n \in \mathcal{F}^n$ son símbolos de función n -arios, y $t_1, \dots, t_n, s_1, \dots, s_n$ son términos, entonces $\hat{\mathcal{R}}(f_1^n(t_1, \dots, t_n), f_2^n(s_1, \dots, s_n)) = \mathcal{R}(f_1^n, f_2^n) \wedge (\bigwedge_{i=1}^n \hat{\mathcal{R}}(t_i, s_i))$.
- (3) En cualquier otro caso, el grado de similitud es \perp .

La relación de similitud se extiende análogamente a las fórmulas atómicas de \mathcal{L} .

Ejemplo 2.16. Sea L el retículo completo $([0, 1], \leq)$ y \wedge la t-norma de Gödel. La siguiente matriz caracteriza una relación de similitud \mathcal{R} definida sobre el universo $\mathcal{U} = \{\text{ritz}, \text{atlantis}, \text{metro}, \text{taxi}, \text{bus}\}$:

\mathcal{R}	ritz	atlantis	metro	taxi	bus
ritz	1	0.6	0	0	0
atlantis	0.6	1	0	0	0
metro	0	0	1	0.4	0.5
taxi	0	0	0.4	1	0.4
bus	0	0	0.5	0.4	1

donde es fácil comprobar que \mathcal{R} es reflexiva, simétrica y transitiva. En particular, se observa que $\mathcal{R}(\text{metro}, \text{bus}) \wedge \mathcal{R}(\text{bus}, \text{taxi}) = \min\{0.5, 0.4\} = 0.4 \leq \mathcal{R}(\text{metro}, \text{taxi}) = 0.4$. Además, la extensión $\hat{\mathcal{R}}$ determina que $\text{close}(\text{ritz}, \text{taxi})$

y $\text{close}(\text{atlantis}, \text{metro})$ son similares con un grado de 0.4, ya que:

$$\begin{aligned} \hat{\mathcal{R}}(\text{close}(\text{ritz}, \text{taxi}), \text{close}(\text{atlantis}, \text{metro})) &= \\ \mathcal{R}(\text{close}, \text{close}) \wedge \mathcal{R}(\text{ritz}, \text{atlantis}) \wedge \mathcal{R}(\text{taxi}, \text{metro}) &= \\ 1 \wedge 0.6 \wedge 0.4 &= \\ 0.4. & \end{aligned}$$

Una relación binaria difusa arbitraria puede ser transformada en una relación de similitud siguiendo el algoritmo originalmente descrito en [JI08], que adaptamos aquí a un retículo completo con una t-norma fija \wedge . Para ello, introducimos primero el concepto de *esquema de similitud*.

Definición 2.38 (Esquema de similitud). Dado un dominio \mathcal{U} y un retículo completo L , un *esquema de similitud* \mathcal{S} es un conjunto de ecuaciones de la forma $x \sim y = v$, donde $x, y \in \mathcal{U}$ y $v \in L$.

El algoritmo 2.1 construye una relación de similitud a partir de un esquema de similitud aplicando los cierres reflexivo, simétrico y transitivo sobre el esquema inicial.

Ejemplo 2.17. Considérese la relación de similitud \mathcal{R} mostrada en el ejemplo 2.16, definida sobre el universo $\mathcal{U} = \{\text{ritz}, \text{atlantis}, \text{metro}, \text{taxi}, \text{bus}\}$. El cierre del siguiente esquema de similitud \mathcal{S} con la t-norma de Gödel genera la relación de similitud \mathcal{R} :

$$\mathcal{S} = \begin{cases} \text{metro} \sim \text{bus} = 0.5 \\ \text{bus} \sim \text{taxi} = 0.4 \\ \text{atlantis} \sim \text{ritz} = 0.6 \end{cases}$$

2.4. Programación lógica difusa

La *programación lógica difusa* integra la lógica difusa y la programación lógica pura, con el objetivo de crear sistemas formales que permitan razonar sobre problemas que tienen un carácter inherentemente vago o impreciso. Aunque el campo de la programación lógica difusa tiene un largo recorrido, que comienza con el trabajo preliminar de R. Lee en 1972 [Lee72], todavía no es un área bien asentada, y por lo tanto no hay un estándar claro, sino distintas aproximaciones. Como es de esperar, no hay una única forma de difuminar el proceso de resolución basado en unificación sintáctica en el que se apoya la programación lógica clásica. En particular, podemos destacar dos tendencias:

- (1) Por un lado, los lenguajes que reemplazan el algoritmo de unificación sintáctica por un algoritmo de unificación difuso, en la línea de SiLog [FGS00], Likelog [LSS01] y Bousi~Prolog [JIRM17].
- (2) Por otro lado, los lenguajes cuya semántica operacional es una extensión de la resolución SLD que permite propagar los grados de verdad anotados en las reglas de los programas, en la línea de Fril [BMP95], f-Prolog [LL90], Prolog-ELF [IK85], Fuzzy Prolog [MSD89] y MALP [MOAV04].

Algoritmo 2.1: Cierre de un esquema de similitud

Datos: Un esquema de similitud \mathcal{S} en \mathcal{U} y una t-norma \wedge
Resultado: Una relación de similitud \mathcal{R} en \mathcal{U}
Sea \mathcal{R} una relación binaria difusa en \mathcal{U} con todas sus entradas a \perp ;
para cada $x \in \mathcal{U}$ **hacer**
 | $\mathcal{R}(x, x) \leftarrow \top$;
fin
para cada $x \in \mathcal{U}$ **hacer**
 | **para cada** $y \in \mathcal{U}$ **hacer**
 | **si** $(x \sim y = v) \in \mathcal{S}$ **entonces**
 | $\mathcal{R}(x, y) \leftarrow v$;
 | $\mathcal{R}(y, x) \leftarrow v$;
 | **fin**
 | **fin**
fin
para cada $x \in \mathcal{U}$ **hacer**
 | **para cada** $y \in \mathcal{U}$ **hacer**
 | **para cada** $z \in \mathcal{U}$ **hacer**
 | $\mathcal{R}(x, y) \leftarrow \sup\{\mathcal{R}(x, y), \mathcal{R}(x, z) \wedge \mathcal{R}(z, y)\}$;
 | **fin**
 | **fin**
fin
devolver \mathcal{R} ;

En esta tesis nos centramos en el desarrollo de FASILL, un lenguaje de programación lógico difuso con anotaciones de grados de verdad implícitas y explícitas, una gran variedad de conectivas y unificación por similitud, que integra y extiende en un mismo lenguaje las capacidades de otros lenguajes difusos como MALP y Bousi~Prolog. En esta sección, se describen las principales características de algunos de estos lenguajes lógicos difusos como preámbulo a la introducción del lenguaje FASILL.

2.4.1. Unificación débil

Existen diversas propuestas [AG98, GS00] para obtener mecanismos de unificación más flexibles que la unificación sintáctica, pero la más general de ellas es la *unificación débil* [Ses02], mediante la cual se pueden implementar el resto de aproximaciones [JIRM10]. La unificación débil consiste en la sustitución de la igualdad sintáctica por una relación de similitud (o de proximidad) \mathcal{R} , que relaciona los símbolos del alfabeto de un lenguaje de primer orden \mathcal{L} , y que se extiende a los términos y átomos de \mathcal{L} como vimos en la sección 2.3.3. Así, el unificador más general se reemplaza por el concepto de unificador *débil* más general (w.m.g.u.) y se introduce un algoritmo de unificación débil para computarlo [Ses02]. Este algoritmo especifica que dos expresiones (términos o

fórmulas atómicas) $f_1^n(t_1, t_2, \dots, t_n)$ y $f_2^n(s_1, s_2, \dots, s_n)$ unifican débilmente si los símbolos de la raíz, f_1^n y f_2^n , están relacionados con un determinado grado de similitud $\mathcal{R}(f_1^n, f_2^n) > \perp$, y cada uno de los argumentos t_i y s_i unifican débilmente. Entonces, existe un unificador débil de dichas expresiones incluso si los símbolos de sus raíces no son sintácticamente iguales ($f_1^n \neq f_2^n$).

Definición 2.39 (Unificador débil). Dada una relación de similitud \mathcal{R} , una sustitución θ es un *unificador débil* de dos expresiones E_1 y E_2 si y solo si $\hat{\mathcal{R}}(E_1\theta, E_2\theta) > \perp$.

La relación de preorden inducida por la composición de sustituciones puede generalizarse para definir una versión débil de la misma. Dadas dos sustituciones θ y σ , y sea \mathcal{R} una relación de similitud, se dice que θ es débilmente más general que σ , denotado $\theta \preceq_{\mathcal{R}} \sigma$, si y solo si existe una sustitución δ tal que $\mathcal{R}(x\theta, x\sigma\delta) > \perp$ para todo $x \in \mathcal{V}$.

Definición 2.40 (Unificador débil más general). Un unificador débil θ de dos expresiones E_1 y E_2 es un *unificador débil más general* (o *w.m.g.u.*) de E_1 y E_2 si y solo si, para cualquier otro unificador débil θ' , se verifica que $\theta \preceq_{\mathcal{R}} \theta'$.

En esta sección se detalla el algoritmo de unificación débil dado por M. Sessa en [Ses02], reformulado como un sistema de transición de estados equipado con una relación de unificación débil (denotada \Rightarrow_{wmgu}), junto con algunas propiedades que demostramos en nuestras aportaciones [JIMR22a, JIMR22b]. La relación de unificación débil \Rightarrow_{wmgu} es la mínima relación definida por el conjunto de reglas de transición dadas en la figura 2.3.

$$\frac{\langle \{f(t_1, \dots, t_n) = g(s_1, \dots, s_n)\} \cup S, \theta, v \rangle, \mathcal{R}(f, g) > \perp}{\langle \{t_1 = s_1, \dots, t_n = s_n\} \cup S, \theta, \mathcal{R}(f, g) \wedge v \rangle} \quad (1)$$

$$\frac{\langle \{x = x\} \cup S, \theta, v \rangle}{\langle S, \theta, v \rangle} \quad (2) \quad \frac{\langle \{x = t\} \cup S, \theta, v \rangle, x \notin \text{vars}(t)}{\langle S\{x/t\}, \theta\{x/t\}, v \rangle} \quad (3)$$

$$\frac{\langle \{t = x\} \cup S, \theta, v \rangle}{\langle \{x = t\} \cup S, \theta, v \rangle} \quad (4) \quad \frac{\langle \{x = t\} \cup S, \theta, v \rangle, x \in \text{vars}(t)}{\langle \text{fallo}, \theta, v \rangle} \quad (5)$$

$$\frac{\langle \{f(t_1, \dots, t_n) = g(s_1, \dots, s_n)\} \cup S, \theta, v \rangle, \mathcal{R}(f, g) = \perp}{\langle \text{fallo}, \theta, v \rangle} \quad (6)$$

Figura 2.3: Reglas de transición de la relación de unificación débil (\Rightarrow_{wmgu}).

La unificación débil de las expresiones E_1 y E_2 se obtiene por medio de una secuencia de transformaciones empezando por el estado inicial $\langle G \equiv \{E_1 = E_2\}, id, \top \rangle$:

$$\langle G, id, \top \rangle \Rightarrow_{wmgu} \langle G_1, \theta_1, v_1 \rangle \Rightarrow_{wmgu} \dots \Rightarrow_{wmgu} \langle G_n, \theta_n, v_n \rangle.$$

Cuando se alcanza el estado final $\langle G_n, \theta_n, v_n \rangle$, con $G_n =$, las expresiones E_1 y E_2 son débilmente unificables con w.m.g.u. θ_n y grado de unificación v_n , denotado $\text{wmgur}_{\mathcal{R}}(E_1, E_2) = \langle \theta_n, v_n \rangle$. En ocasiones, escribiremos $\text{wmgur}_{\mathcal{R}}(E_1, E_2) = \theta_n$ con grado de unificación $\text{deg}(E_1, E_2) = v_n$. Si las expresiones E_1 y E_2 no son unificables, el algoritmo termina con un estado de fallo $\langle G_n, \theta_n, v_n \rangle \equiv \langle \text{fallo}, \theta_n, v_n \rangle$.

Ejemplo 2.18. Dada la relación de similitud \mathcal{R} del ejemplo 2.16, tenemos que:

$$\text{wmgur}_{\mathcal{R}}(\text{close}(\text{ritz}, \text{taxi}), \text{close}(\text{atlantis}, \text{metro})) = \langle \text{id}, 0.4 \rangle;$$

siendo id la sustitución identidad, mientras que:

$$\text{wmgur}_{\mathcal{R}}(\text{close}(\text{ritz}, x), \text{close}(\text{atlantis}, \text{metro})) = \langle \{x/\text{metro}\}, 0.6 \rangle.$$

El *teorema de unificación débil*, demostrado en [Ses02, pág. 412], establece que si dos expresiones son débilmente unificables, el algoritmo de unificación débil es capaz de computar un w.m.g.u. de dichas expresiones. Más aún, el algoritmo de unificación débil es capaz de comprobar si un conjunto de ecuaciones de expresiones $S = \{E_1 = E'_1, \dots, E_n = E'_n\}$ es débilmente unificable.⁶ Aunque el w.m.g.u. de un conjunto de expresiones S no es único (igual que en el caso clásico), podemos pensar que lo es con respecto a la relación de equivalencia débil $\sim_{\mathcal{R}}$.⁷ La clase de equivalencia correspondiente al w.m.g.u. de S es $\text{wmgur}_{\mathcal{R}}^{\sim}(S) = \{\theta' : \theta = \text{wmgur}_{\mathcal{R}}(S) \text{ y } \theta' \sim_{\mathcal{R}} \theta\}$. Así, el algoritmo de unificación débil computa un representante de la clase del w.m.g.u. En ocasiones, denotamos al w.m.g.u. representante del conjunto S por $\text{wmgur}_{\mathcal{R}}(S)$. Por supuesto, cada $\theta \in \text{wmgur}_{\mathcal{R}}^{\sim}(S)$ tiene un grado de unificación $v = \bigwedge_{i=1}^n \text{deg}(E_i\theta, E'_i\theta)$.

Como en el caso clásico, el unificador débil más general es una sustitución idempotente [JIMR22a, JIMR22b].

Proposición 2.4. *El algoritmo de unificación débil computa sustituciones idempotentes.*

Demostración. Dado un conjunto de ecuaciones de expresiones débilmente unificable y sea G_0 el conjunto inicial de problemas de unificación, por el teorema de unificación débil existe una secuencia de transiciones desde la configuración inicial tal que:

$$\langle G_0, \text{id}, \top \rangle \Rightarrow_{\text{wmgur}} \langle G_1, \theta_1, v_1 \rangle \Rightarrow_{\text{wmgur}} \dots \Rightarrow_{\text{wmgur}} \langle \emptyset, \theta_n, v_n \rangle.$$

Nótese que todos los enlaces se generan por la regla de transición **(3)** de la figura 2.3. Dado que siempre que se genera un enlace x/t (donde $x \notin \text{vars}(t)$) en un paso i -ésimo, este es aplicado a G_{i-1} para obtener el siguiente conjunto de problemas de unificación $G_i = G_{i-1}\{x/t\}$, la variable x se elimina de G_i y del resto de sus estados sucesores. Por lo tanto, θ_n es idempotente. \square

⁶Bajo esta notación, el w.m.g.u. de dos expresiones E_1 y E_2 , $\text{wmgur}_{\mathcal{R}}(E_1, E_2)$, se puede denotar también por $\text{wmgur}_{\mathcal{R}}(\{E_1 = E_2\})$.

⁷ $\sigma \sim_{\mathcal{R}} \theta$, si y solo si existe un renombramiento ρ tal que, para toda variable $x \in \mathcal{V}$, $\mathcal{R}(x\theta, x\sigma\rho) > \perp$.

E. Eder [Ede85] se planteó la posibilidad de computar los m.g.u. de un conjunto de ecuaciones de expresiones S , calculando los m.g.u. de determinados subconjuntos de S por separado –posiblemente en paralelo– y entonces componiéndolos para obtener el m.g.u. de S . Más tarde, C. Palamidessi [Pal90] reformuló el problema. Para terminar esta sección, extendemos el operador de *composición paralela* definido por Palamidessi y estudiamos su relación con la composición estándar de sustituciones. Este operador trabaja sobre el conjunto cociente de sustituciones idempotentes inducido por la relación de equivalencia débil $\sim_{\mathcal{R}}$, que denotamos por $\text{ISUBST}/\sim_{\mathcal{R}}$. Los elementos de $\text{ISUBST}/\sim_{\mathcal{R}}$ son las clases de equivalencia débil de las sustituciones idempotentes θ , denotadas por θ^{\sim} .

La *representación ecuacional* de una sustitución $\theta = \{x_1/t_1, \dots, x_n/t_n\}$ es el conjunto de ecuaciones $\widehat{\theta} = \{x_1 = t_1, \dots, x_n = t_n\}$, que expresan un problema de unificación débil.

Definición 2.41 (Composición paralela débil). El operador de *composición paralela débil* es una función $\uparrow: \text{ISUBST}/\sim_{\mathcal{R}} \times \text{ISUBST}/\sim_{\mathcal{R}} \rightarrow \text{ISUBST}/\sim_{\mathcal{R}}$ definida como:

$$\theta_1^{\sim} \uparrow \theta_2^{\sim} = \text{wmgu}_{\mathcal{R}}^{\sim}(\widehat{\theta}_1 \cup \widehat{\theta}_2);$$

donde $\text{wmgu}_{\mathcal{R}}^{\sim}$ se restringe a sustituciones idempotentes.

El operador \uparrow está bien definido, en el sentido de que no depende de la elección de los elementos en las clases de equivalencia. Por lo tanto, por simplicidad, en lo que sigue indicaremos la operación $\theta_1^{\sim} \uparrow \theta_2^{\sim}$ como $\theta_1 \uparrow \theta_2$. La siguiente proposición establece la relación existente entre la composición paralela débil y la composición estándar de sustituciones. Este resultado tendrá un papel fundamental en la prueba de corrección de la transformación de desplegado (véase la sección 6.3).

Proposición 2.5 (Composición paralela débil, [JIMR22a, JIMR22b]). *Sean θ_1 y θ_2 dos sustituciones idempotentes tal que $(\widehat{\theta}_1 \cup \widehat{\theta}_2)$ es un conjunto de ecuaciones débilmente unificable. Entonces:*

$$\theta_1 \uparrow \theta_2 = \theta_1 \text{wmgu}_{\mathcal{R}}^{\sim}(\widehat{\theta}_2 \theta_1) = \theta_2 \text{wmgu}_{\mathcal{R}}^{\sim}(\widehat{\theta}_1 \theta_2).$$

Demostración. Nos centramos en la primera igualdad, ya que la segunda es completamente simétrica. Empezamos por la definición del operador de composición paralela débil y aplicamos el algoritmo de unificación débil. Sea $\widehat{\theta}_1 = \{x_1 = t_1, \dots, x_n = t_n\}$, entonces, dado que $\theta_1 \uparrow \theta_2 = \text{wmgu}_{\mathcal{R}}^{\sim}(\widehat{\theta}_1 \cup \widehat{\theta}_2)$ y $(\widehat{\theta}_1 \cup \widehat{\theta}_2)$ es un conjunto de ecuaciones débilmente unificable, es posible realizar la siguiente secuencia (de éxito) de pasos de transición con la relación de unificación débil $\Rightarrow_{\text{wmgu}}$. Nótese que si θ_1 es idempotente, $\text{dom}(\theta_1) \cap \text{vars}(\text{ran}(\theta_1)) = \emptyset$. Entonces, para cada ecuación $(x_i = t_i) \in \widehat{\theta}_1$, $x_i \notin \text{vars}(t_i)$, y es posible realizar los primeros n pasos aplicando la regla de transición **(5)** de la figura 2.3:

$$\langle \{x_1 = t_1, \dots, x_n = t_n\} \cup \widehat{\theta}_2, id, \top \rangle \Rightarrow_{\text{wmgu}}^n \langle \widehat{\theta}_2 \theta_1, \theta_1, \top \rangle \Rightarrow_{\text{wmgu}}^* \langle \emptyset, \theta_1 \sigma, v \rangle.$$

Entonces, la secuencia de pasos de transición $\langle \widehat{\theta}_2\theta_1, id, \top \rangle \Rightarrow_{wmg_u}^* \langle \emptyset, \vartheta, v \rangle$ también es posible y $\widehat{\theta}_2\theta_1$ es débilmente unificable con w.m.g.u. $wmg_u_{\mathcal{R}}(\widehat{\theta}_2\theta_1) = \sigma \in wmg_u_{\mathcal{R}}(\widehat{\theta}_2\theta_1)$. Por lo tanto, $\theta_1 \uparrow \theta_2 = \theta_1 wmg_u_{\mathcal{R}}(\widehat{\theta}_2\theta_1)$. \square

2.4.2. Extendiendo la unificación

Los lenguajes lógicos difusos que reemplazan el algoritmo de unificación sintáctica por un algoritmo de unificación difuso permiten unificar términos y predicados que no son sintácticamente iguales, siempre y cuando los símbolos involucrados sean similares en cierta medida. En estos lenguajes, el mecanismo de resolución SLD permanece intacto, con ligeras modificaciones para incorporar los grados de similitud introducidos por la unificación difusa. En esta línea, Bousi~Prolog es uno de los lenguajes más interesantes.

El lenguaje Bousi~Prolog [JIRMG09, JIRM09a, JIRM09b, JIRM17] es una extensión de Prolog que reemplaza el algoritmo de unificación clásico de la semántica operacional de la programación lógica por un algoritmo de unificación débil, y cuya sintaxis permite declarar relaciones de similitud (en $[0, 1]$) entre las reglas del programa. La regla de resolución de Bousi~Prolog, denominada *resolución WSLD* (acrónimo de «resolución lineal selectiva débil para cláusulas definidas»), es similar a la resolución SLD de la programación lógica clásica, excepto por el uso del algoritmo de unificación débil y por la incorporación de los grados de proximidad. Durante el proceso de resolución WSLD, se asigna un grado de proximidad a cada cláusula objetivo, que se computa a partir de los grados producidos en los pasos de unificación débil. Inicialmente, la cláusula objetivo a resolver tiene asociado el mayor grado de proximidad: 1. Entonces, tras cada paso de resolución WSLD este grado va decreciendo progresivamente si, en la unificación débil de los literales complementarios de las cláusulas resueltas, los símbolos a unificar débilmente tienen valores menores que los anteriores grados obtenidos hasta el momento. Basándonos en la definición 2.30 de la resolución SLD, un paso de resolución WSLD puede definirse de la siguiente forma.

Definición 2.42 (Resolución WSLD). Sea $\langle \mathcal{G}, \theta, \alpha \rangle$ un estado, donde \mathcal{G} es un objetivo, θ una sustitución y $\alpha \in (0, 1]$ un grado de proximidad, y sea Π un programa lógico definido y \mathcal{R} una relación de similitud. Dada una función de selección φ , definimos la resolución WSLD como un sistema de transición cuya relación de transición \Rightarrow_{WSLD} es la menor relación binaria que satisface:

$$\frac{\langle \mathcal{G}, \theta, \alpha \rangle, \varphi(\mathcal{G}) = A, (H \leftarrow B) \in \Pi, wmg_u_{\mathcal{R}}(A, H) = \langle \sigma, \beta \rangle}{\langle \mathcal{G}[A/B]\sigma, \theta\sigma, \min\{\alpha, \beta\} \rangle} \text{ (WSLD)}$$

De forma análoga al caso clásico, una *derivación WSLD* es una secuencia $\langle \mathcal{G}_0, id, 1 \rangle \Rightarrow_{WSLD} \langle \mathcal{G}_1, \theta_1, \alpha_1 \rangle \Rightarrow_{WSLD} \dots \Rightarrow_{WSLD} \langle \mathcal{G}_n, \theta_n, \alpha_n \rangle$. Una *refutación WSLD* es una derivación de éxito, es decir, que conduce a la cláusula vacía: $\langle \mathcal{G}_0, id, 1 \rangle \Rightarrow_{WSLD} \dots \Rightarrow_{WSLD} \langle \square, \theta, \alpha \rangle$, donde θ es la *respuesta computada* en la derivación y α es el grado de proximidad asociado a dicha respuesta.

2.4.3. Extendiendo la resolución

Generalmente, en los lenguajes que extienden la regla de resolución clásica, los programas son conjuntos de cláusulas con grados de verdad explícitamente anotados. Las computaciones y los grados de verdad se propagan mediante una semántica que es una extensión del principio de resolución, mientras que el mecanismo de unificación sintáctica permanece intacto. Más concretamente, un programa lógico difuso podría estar compuesto por reglas de la forma:

$$H \leftarrow [\alpha] \langle B_1, \beta_1 \rangle \wedge \langle B_2, \beta_2 \rangle \wedge \dots \wedge \langle B_n, \beta_n \rangle$$

donde α es el grado de verdad anotado de la regla, y β_1, \dots, β_n son los grados de verdad anotados de los átomos del cuerpo. Las anotaciones del cuerpo restringen la aplicabilidad de toda la regla: si para cada $1 \leq i \leq n$, B_i puede probarse con un grado de verdad $\alpha_i \geq \beta_i$, entonces la regla es aplicable y el átomo H se puede deducir con un grado de verdad $\alpha \wedge (\bigwedge_{i=1}^n \alpha_i)$, donde \wedge es una conectiva difusa (usualmente una t-norma). Desde nuestro punto de vista, MALP es uno de los lenguajes más interesantes en esta línea.

El lenguaje MALP [MOAV01a, MOAV01b, MOAV04, JIMP09, JIMOA17] (acrónimo de «*Multi-Adjoint Logic Programming*») es un marco de programación lógico difuso en el que los grados de verdad se toman de un retículo multi-adjunto, y cuya sintaxis permite incorporar distintas lógicas en las reglas del programa. Un retículo multi-adjunto es, en esencia, un conjunto ordenado (L, \leq) equipado con una serie de conectivas difusas (implicaciones, conjunciones, disyunciones y agregadores) con la particularidad de que cada símbolo de implicación \leftarrow_i debe tener asociado una conjunción adjunta $\&_i$, utilizada para modelar la regla de inferencia *modus ponens* en un entorno difuso.⁸ Una regla MALP es de la forma $\langle H \leftarrow B, r \rangle$, donde la cabeza H es una fórmula atómica, el cuerpo B es una fórmula construida a partir de átomos y grados de verdad de L combinados mediante las conectivas difusas definidas en el retículo, y $r \in L$ es el grado de verdad asociado a la regla.

Al contrario que en la programación lógica clásica, donde el mecanismo operacional se basa en refutación, en [MOAV04] se define la semántica operacional del lenguaje lógico multi-adjunto como un sistema de transición de estados dividido en dos fases: una primera fase donde se explotan los átomos y una segunda fase de corte interpretativo donde se evalúan las conectivas.

Definición 2.43 (Paso de computación admisible, [MOAV04]). Sea $\langle \mathcal{G}, \theta \rangle$ un estado, donde \mathcal{G} es un objetivo y θ una sustitución. Dado un programa multi-adjunto $\mathcal{P} = \langle \Pi, L \rangle$, donde Π es un conjunto de reglas y L es un retículo multi-adjunto, definimos un *paso de computación admisible* como un sistema de transición de estados cuya relación de transición \rightsquigarrow es la menor relación binaria que satisface las siguientes reglas:

⁸El par de conectivas $\langle \leftarrow_i, \&_i \rangle$ es un par adjunto si \leftarrow_i es una implicación difusa, $\&_i$ es una conjunción difusa, y se satisface la condición de adjunción: $x \leq (y \leftarrow_i z) \leftrightarrow (x \&_i z) \leq y$ para todo $x, y, z \in L$.

(1) *Paso admisible con una regla* (denotado \rightsquigarrow_{AS_1}):

$$\frac{\langle \mathcal{G}[A], \theta \rangle, \langle H \leftarrow_i B, r \rangle \in \Pi, \text{mgu}(A, H) = \sigma}{\langle \mathcal{G}[A/(r \&_i B)]\sigma, \theta\sigma \rangle} \quad (\mathbf{AS1})$$

(2) *Paso admisible con un hecho* (denotado \rightsquigarrow_{AS_2}):

$$\frac{\langle \mathcal{G}[A], \theta \rangle, \langle H \leftarrow, r \rangle \in \Pi, \text{mgu}(A, H) = \sigma}{\langle \mathcal{G}[A/r]\sigma, \theta\sigma \rangle} \quad (\mathbf{AS2})$$

(3) *Paso admisible de fallo* (denotado \rightsquigarrow_{AS_3}):

$$\frac{\langle \mathcal{G}[A], \theta \rangle, \nexists \langle H \leftarrow B, r \rangle \in \Pi : \text{mgu}(A, H) \neq \text{fallo}}{\langle \mathcal{G}[A/\perp], \theta \rangle} \quad (\mathbf{AS3})$$

Una *derivación admisible* es una secuencia de longitud arbitraria $\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle \mathcal{G}', \theta \rangle$. Cuando \mathcal{G}' no contiene más átomos, el estado $\langle \mathcal{G}', \theta \rangle$ es una *respuesta computada admisible* para esa derivación.

Después de una secuencia de pasos admisibles, un objetivo se transforma en una fórmula que no contiene átomos, y puede ser interpretada directamente en el retículo multi-adjunto L para asignar un grado de verdad final a la respuesta computada admisible.

Capítulo 3

El lenguaje FASILL

El lenguaje FASILL (acrónimo de «*Fuzzy Aggregators and Similarity Into a Logic Language*») combina un algoritmo de unificación débil, basado en relaciones de similitud, junto con un amplio repertorio de conectivas difusas cuyas funciones de verdad pueden ser definidas sobre un retículo completo. En este capítulo se presenta la sintaxis y las semánticas operacional y declarativa de FASILL, junto con algunos detalles de implementación del lenguaje, tal y como se recogen en nuestras aportaciones [JIMP18, JIMR20, JIMR22a, JIMR22b].

3.1. Sintaxis

En esta sección se introduce la sintaxis del lenguaje FASILL y la noción de programa. A continuación, se da el alfabeto y las reglas de formación de las fórmulas bien formadas.

Alfabeto. FASILL es un lenguaje de primer orden construido sobre un alfabeto Σ que contiene los elementos de un conjunto infinito numerable de variables, símbolos de función y de predicado con su aridad asociada y un amplio conjunto de conectivas Σ_L^C : t-normas ($\&$), t-conormas (\mid), agregadores ($\@$) y el símbolo de implicación (\leftarrow). Consideraremos los símbolos de constante como símbolos de función de aridad 0. Asumimos también que el lenguaje admite un conjunto de literales Σ_L^T que son interpretados asociándoles los elementos de un retículo completo (L, \leq) . Denotamos por Σ_L al conjunto de literales y conectivas $\Sigma_L^T \cup \Sigma_L^C$.

Fórmulas bien formadas. El lenguaje combina variables y símbolos de función para construir términos de la forma usual. Del mismo modo, combina términos y símbolos de predicado para construir fórmulas atómicas. Las fórmulas bien formadas se construyen inductivamente conforme a las siguientes reglas:

- (1) Todo elemento del conjunto de literales, $r \in \Sigma_L^T$, es una fórmula bien formada.¹
- (2) Toda fórmula atómica es una fórmula bien formada.
- (3) Si $\zeta^n \in \Sigma_L^C$ es un símbolo de conectiva y A_1, \dots, A_n son fórmulas bien formadas, también lo es $\zeta^n(A_1, \dots, A_n)$.

Definición 3.1 (Regla FASILL). Dado un retículo completo L , una *regla* FASILL es una fórmula bien formada de la forma $A \leftarrow B$, donde A (la cabeza) es un átomo y B (el cuerpo) es una fórmula bien formada construida a partir de átomos, literales de Σ_L^T y conectivas de Σ_L^C (excluyendo \leftarrow).

Las reglas en un programa FASILL tienen el mismo papel que las cláusulas en los programas Prolog o MALP [MOAV04], indicando que un cierto predicado relaciona algunos términos (la cabeza) si algunas condiciones (el cuerpo) se mantienen.

Definición 3.2 (Programa FASILL). Un *programa* FASILL es una tupla $\langle \Pi, \mathcal{R}, L \rangle$, donde Π es un conjunto de reglas construidas sobre un alfabeto Σ , \mathcal{R} es una relación de similitud (cuyo dominio es el conjunto de símbolos de función y de predicado de Σ) y L es un retículo completo.

Ejemplo 3.1. El retículo $L = ([0, 1], \leq)$, la relación de similitud \mathcal{R} mostrada en el ejemplo 2.16 y el siguiente conjunto de reglas Π forman un programa FASILL $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$:

$$\Pi = \left\{ \begin{array}{ll} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_4 : \text{good_hotel}(x) & \leftarrow @_{\text{aver}}(@_{\text{very}}(\text{close}(x, y)), \text{cheap}(y)) \end{array} \right.$$

Una L -expresión es una fórmula bien formada de \mathcal{L} que está compuesta únicamente por literales de Σ_L^T y conectivas de Σ_L^C . Dada una L -expresión E , $\vartheta_L(E)$ denota el grado de verdad $v \in L$ obtenido tras interpretar las conectivas en E . La interpretación de una L -expresión se define inductivamente como:

- (1) $\vartheta_L(r) = v$ si y solo si $v \in L$ es la interpretación de $r \in \Sigma_L^T$;
- (2) $\vartheta_L(\zeta^n(A_1, \dots, A_n)) = F_{\zeta^n}(\vartheta_L(A_1), \dots, \vartheta_L(A_n))$ si y solo si ζ^n es una conectiva de Σ_L^C interpretada mediante la función de verdad $F_{\zeta^n} : L^n \rightarrow L$ y, para todo $1 \leq i \leq n$, A_i es una L -expresión.

3.2. Semántica operacional

A continuación, se introduce la semántica operacional de FASILL como un sistema de transición de estados, en el cual se distinguen dos fases: una primera

¹Nótese que un elemento $r \in \Sigma_L^T$ se interpretará como el grado de verdad $r \in L$.

fase operacional similar a la resolución SLD (véase la definición 2.30) en la que se resuelven todos los átomos; y una segunda fase de carácter interpretativo, en la que se evalúan las conectivas para calcular el grado de verdad final.

Definición 3.3 (Paso de computación, [JIMP17]). Sea $\langle \mathcal{G}, \theta \rangle$ un estado, donde \mathcal{G} es un objetivo y θ una sustitución. Dado un programa $\langle \Pi, \mathcal{R}, L \rangle$ y una t-norma \wedge fija en L (también asociada a \mathcal{R}), definimos un *paso de computación* como un sistema de transición de estados cuya relación de transición \rightsquigarrow es la menor relación binaria que satisface las siguientes reglas:

(1) *Paso de éxito*² (denotado \rightsquigarrow_{SS}):

$$\frac{\langle \mathcal{G}[A], \theta \rangle, (H \leftarrow B) \in \Pi, \text{wmg}\mathbf{u}_{\mathcal{R}}(A, H) = \langle \sigma, r \rangle}{\langle \mathcal{G}[A/(r \wedge B)]\sigma, \theta\sigma \rangle} \text{ (SS)}$$

(2) *Paso de fallo* (denotado \rightsquigarrow_{FS}):

$$\frac{\langle \mathcal{G}[A], \theta \rangle, \nexists (H \leftarrow B) \in \Pi : \text{wmg}\mathbf{u}_{\mathcal{R}}(A, H) = \langle \sigma, r \rangle, r > \perp}{\langle \mathcal{G}[A/\perp], \theta \rangle} \text{ (FS)}$$

(3) *Paso interpretativo* (denotado \rightsquigarrow_{IS}):

$$\frac{\langle \mathcal{G}[\zeta(r_1, \dots, r_n)], \theta \rangle, \vartheta_L(\zeta(r_1, \dots, r_n)) = r_{n+1}}{\langle \mathcal{G}[\zeta(r_1, \dots, r_n)/r_{n+1}], \theta \rangle} \text{ (IS)}$$

Como en el caso clásico, una *derivación* es una secuencia de longitud arbitraria $\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle \mathcal{G}', \theta \rangle$. Cuando $\mathcal{G}' = r \in L$, el estado $\langle r, \sigma \rangle$ es una *respuesta computada difusa* (f.c.a., del inglés «fuzzy computed answer») para esa derivación, donde $\sigma = \theta[\text{vars}(\mathcal{G})]$ y r es el grado de verdad asociado a dicha f.c.a.

Al igual que en la resolución SLD, asumimos la existencia de una función de selección para decidir qué átomo del objetivo resolver en el siguiente paso de computación. Tal y como establece el teorema 2.1, en programación lógica pura la función de selección es independiente. No obstante, en FASILL esto no es generalmente cierto, como se muestra en los siguientes ejemplos. Por lo tanto, FASILL utiliza una regla de computación fija similar a la de Prolog que selecciona el átomo más a la izquierda, dando prioridad a los pasos de éxito y de fallo antes de aplicar cualquier paso interpretativo.

Ejemplo 3.2. Sea \mathcal{P} el programa mostrado en el ejemplo 3.1. La siguiente es una derivación que lleva a una respuesta computada difusa para el objetivo

²Por simplicidad, cuando la regla cuya cabeza unifica con el átomo seleccionado es un hecho, asumimos que su cuerpo es \top .

$\mathcal{G} = \text{good_hotel}(x)$ en \mathcal{P} :

$$\begin{aligned}
\mathcal{D}^{\mathcal{P}} : \langle \underline{\text{good_hotel}(x)}, id \rangle & \rightsquigarrow_{SS}^{R_4} \\
\langle @_{\text{aver}}(@_{\text{very}}(\underline{\text{close}(x_1, y_1)}), \underline{\text{cheap}(y_1)}), \{x/x_1, y/y_1\} \rangle & \rightsquigarrow_{SS}^{R_2} \\
\langle @_{\text{aver}}(@_{\text{very}}(0.7), \underline{\text{cheap}(taxi)}), \{x/hydropolis, y/taxi\} \rangle & \rightsquigarrow_{SS}^{R_1} \\
\langle @_{\text{aver}}(@_{\text{very}}(0.7), 0.8), \{x/hydropolis, y/taxi\} \rangle & \rightsquigarrow_{IS}^* \\
\langle 0.645, \{x/hydropolis, y/taxi\} \rangle &
\end{aligned}$$

Entonces, $\langle 0.645, \{x/hydropolis\} \rangle$ es una respuesta computada difusa para \mathcal{G} . Además, para este mismo objetivo existe otra derivación con respuesta computada difusa $\langle 0.605, \{x/ritz\} \rangle$ (véase la figura 3.1).

Ejemplo 3.3. Supongamos que la regla de computación selecciona el átomo más a la derecha en cada paso. Entonces, la siguiente es una derivación para el objetivo $\mathcal{G} = \text{good_hotel}(x)$ en \mathcal{P} :

$$\begin{aligned}
\mathcal{D}^{\mathcal{P}} : \langle \underline{\text{good_hotel}(x)}, id \rangle & \rightsquigarrow_{SS}^{R_4} \\
\langle @_{\text{aver}}(@_{\text{very}}(\underline{\text{close}(x_1, y_1)}), \underline{\text{cheap}(y_1)}), \{x/x_1, y/y_1\} \rangle & \rightsquigarrow_{SS}^{R_1} \\
\langle @_{\text{aver}}(@_{\text{very}}(\underline{\text{close}(x_1, taxi)}), 0.8), \{x/x_1, y/taxi\} \rangle & \rightsquigarrow_{SS}^{R_3} \\
\langle @_{\text{aver}}(@_{\text{very}}(0.4 \&_{\text{godel}} 0.9), 0.8), \{x/ritz, y/taxi\} \rangle & \rightsquigarrow_{IS}^* \\
\langle 0.48, \{x/ritz, y/taxi\} \rangle &
\end{aligned}$$

Se observa que ahora la respuesta computada difusa para *ritz* tiene asociado un grado de verdad de 0.48, mientras que en el ejemplo anterior es de 0.605. Por lo tanto, la función de selección no es independiente.

Aunque la regla de computación de FASILL no es independiente en general, en el sentido de que no es posible conmutar el orden en el que se resuelven los átomos, sí que hay una propiedad de independencia con respecto a los pasos interpretativos. De hecho, sólo es necesario fijar el orden de selección de los átomos (de izquierda a derecha) sin importar cuándo se evalúa una L -expresión mediante un paso interpretativo. El siguiente resultado [JIMR22a] será de utilidad para demostrar la corrección de la transformación de desplegado.

Lema 3.1 (Conmutación de pasos interpretativos). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y $\mathcal{G}_0[A, A']$ un objetivo, donde A es un átomo o una L -expresión y A' es una L -expresión. Entonces:*

$$\begin{aligned}
\mathcal{D}_0^{\mathcal{P}} : \langle \mathcal{G}_0[A], \theta_0 \rangle & \rightsquigarrow \langle \mathcal{G}_1[A'], \theta_1 \rangle \rightsquigarrow_{IS} \langle \mathcal{G}_2, \theta_1 \rangle \\
& \text{si y solo si} \\
\mathcal{D}_1^{\mathcal{P}} : \langle \mathcal{G}_0[A'], \theta_0 \rangle & \rightsquigarrow_{IS} \langle \mathcal{G}'_1[A], \theta_0 \rangle \rightsquigarrow \langle \mathcal{G}'_2, \theta'_1 \rangle
\end{aligned}$$

donde $\mathcal{G}_2 = \mathcal{G}'_2$ y $\theta_1 = \theta'_1$.

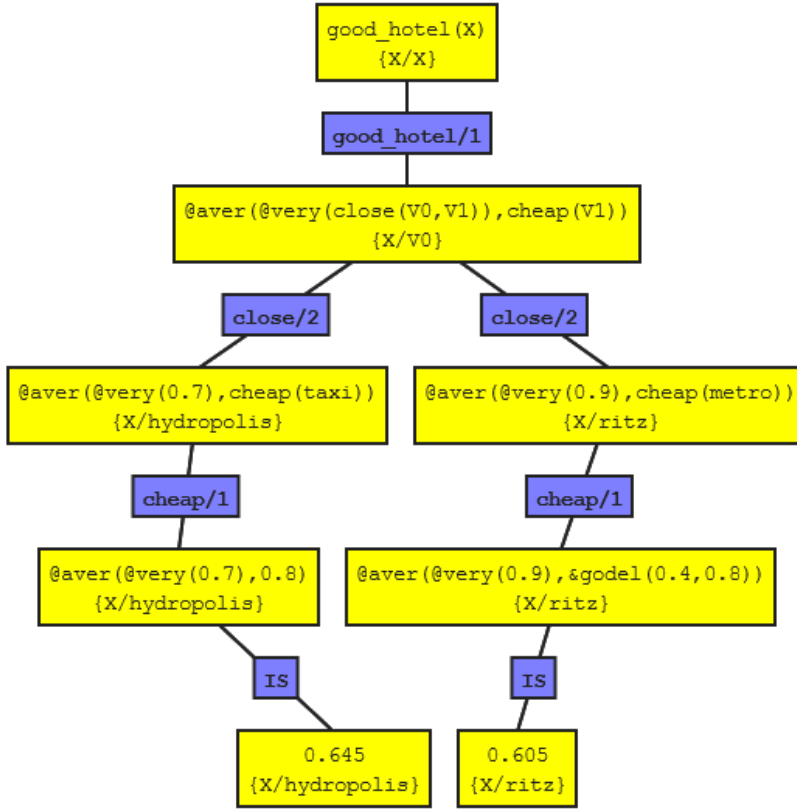


Figura 3.1: Árbol de derivación generado por el sistema FASILL en línea.

Demostración. Por claridad, subrayamos el átomo a resolver en cada paso de computación. Procedemos exhaustivamente con cada uno de los casos que pueden ocurrir en una derivación (A es un átomo que se resuelve con un paso \rightsquigarrow_{SS} o \rightsquigarrow_{FS} , o A es una L -expresión evaluada por medio de un paso \rightsquigarrow_{IS}).

- (1) A es un átomo y el primer paso de computación en \mathcal{D}_0^P es \rightsquigarrow_{SS} .
 En este caso suponemos que A es un átomo y existe una regla $R : (H \leftarrow B) \in \Pi$ tal que $\text{wmgu}_{\mathcal{R}}(A, H) = \langle \sigma, v \rangle$. Entonces:

$$\begin{aligned}
 \mathcal{D}_0^P : \langle \mathcal{G}_0[\underline{A}, A'], \theta_0 \rangle & \rightsquigarrow_{SS}^R \\
 \langle \mathcal{G}_0[A/(v \wedge B), \underline{A}']\sigma, \theta_0\sigma \rangle & \rightsquigarrow_{IS} \\
 \langle \mathcal{G}_0[A/(v \wedge B), A'/r']\sigma, \theta_0\sigma \rangle &
 \end{aligned}$$

si y solo si

$$\begin{aligned}
\mathcal{D}_1^{\mathcal{P}} : \quad & \langle \mathcal{G}_0[A, \underline{A}'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[\underline{A}, A'/r'], \theta_0 \rangle && \rightsquigarrow_{SS}^R \\
& \langle \mathcal{G}_0[A/(v \wedge B), A'/r']\sigma, \theta_0\sigma \rangle
\end{aligned}$$

donde $\vartheta_L(A') = r' \in L$.

- (2) A es un átomo y el primer paso de computación en $\mathcal{D}_0^{\mathcal{P}}$ es \rightsquigarrow_{FS} .
 En este caso suponemos que A es un átomo y no hay ninguna regla $(H \leftarrow B) \in \Pi$ tal que $\text{wmgur}_{\mathcal{R}}(A, H) = \langle \sigma, v \rangle, v > \perp$. Entonces:

$$\begin{aligned}
\mathcal{D}_0^{\mathcal{P}} : \quad & \langle \mathcal{G}_0[\underline{A}, A'], \theta_0 \rangle && \rightsquigarrow_{FS} \\
& \langle \mathcal{G}_0[A/\perp, \underline{A}'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[A/\perp, A'/r'], \theta_0 \rangle
\end{aligned}$$

si y solo si

$$\begin{aligned}
\mathcal{D}_1^{\mathcal{P}} : \quad & \langle \mathcal{G}_0[A, \underline{A}'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[\underline{A}, A'/r'], \theta_0 \rangle && \rightsquigarrow_{FS} \\
& \langle \mathcal{G}_0[A/\perp, A'/r'], \theta_0 \rangle
\end{aligned}$$

donde $\vartheta_L(A') = r' \in L$.

- (3) A es una L -expresión y el primer paso de computación en $\mathcal{D}_0^{\mathcal{P}}$ es \rightsquigarrow_{IS} .
 En este caso suponemos que A es una L -expresión. Entonces:

$$\begin{aligned}
\mathcal{D}_0^{\mathcal{P}} : \quad & \langle \mathcal{G}_0[\underline{A}, A'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[A/r, \underline{A}'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[A/r, A'/r'], \theta_0 \rangle
\end{aligned}$$

si y solo si

$$\begin{aligned}
\mathcal{D}_1^{\mathcal{P}} : \quad & \langle \mathcal{G}_0[A, \underline{A}'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[\underline{A}, A'/r'], \theta_0 \rangle && \rightsquigarrow_{IS} \\
& \langle \mathcal{G}_0[A/r, A'/r'], \theta_0 \rangle
\end{aligned}$$

donde $\vartheta_L(A) = r \in L$ y $\vartheta_L(A') = r' \in L$.

□

Teorema 3.1 (Independencia de la regla de computación difusa con respecto a los pasos interpretativos). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y $\mathcal{G}_0[A]$ un objetivo, donde A es una L -expresión. Entonces:*

$$\mathcal{D}_0^{\mathcal{P}} : \langle \mathcal{G}_0, \theta_0 \rangle \rightsquigarrow^n \langle \mathcal{G}_1[A], \theta_1 \rangle \rightsquigarrow_{IS} \langle \mathcal{G}_2, \theta_1 \rangle$$

si y solo si

$$\mathcal{D}_1^{\mathcal{P}} : \langle \mathcal{G}_0[A], \theta_0 \rangle \rightsquigarrow_{IS} \langle \mathcal{G}'_1, \theta_0 \rangle \rightsquigarrow^n \langle \mathcal{G}'_2, \theta'_1 \rangle$$

donde $\mathcal{G}_2 = \mathcal{G}'_2$ y $\theta_1 = \theta'_1$.

Demostración. Se demuestra fácilmente aplicando repetidamente el lema de conmutación de los pasos interpretativos (véase el lema 3.1). \square

3.3. Semántica declarativa

En esta sección describimos la semántica declarativa de FASILL como una extensión difusa del concepto clásico del modelo mínimo de Herbrand. Para simplificar el discurso de las propiedades de corrección y completitud, soslayamos la extensión umbralizada y más potente descrita en [JIMP17], al tiempo que razonamos únicamente a nivel de átomos básicos, aún cuando en [JIMP18] se hace un contraste más profundo a nivel de respuestas correctas y computadas.

Aquí, las nociones de *interpretación difusa* y *valoración difusa* son las dadas en la sección 2.3. Como es usual, para fórmulas cerradas es posible hablar de satisfacibilidad en una interpretación difusa sin referirnos a una asignación en particular. Además, con el fin de establecer cuándo una regla de un programa FASILL (que es cerrada y está universalmente cuantificada) es satisfacible por una interpretación difusa, damos la siguiente definición de *modelo*.

Definición 3.4 (Modelo de una regla, [JIMP18]). Una interpretación difusa \mathcal{I} es *modelo* de una regla $H \leftarrow B$ si y solo si, se verifica que $\vartheta_\phi(H) \geq \vartheta_\phi(B)$ para cualquier asignación ϕ .

Antes de definir la noción de modelo de un programa FASILL, es necesario introducir algunos conceptos preliminares. Primero introducimos algunas definiciones para tratar con algunos problemas que aparecen cuando las cabezas de las reglas son átomos no lineales.

Definición 3.5 (Átomo lineal). Un término o átomo es *lineal* cuando no contiene múltiples ocurrencias de la misma variable.

Dado un átomo no lineal A , la *linealización* de A (tal y como se describe en [CRARD08]) es un proceso en el cual se computa un par $\text{lin}(A) = \langle A_l, C_l \rangle$, donde A_l es un átomo lineal construido a partir de A reemplazando cada una de las n_i ocurrencias de la misma variable x_i por una nueva variable fresca y_k ($1 \leq k \leq n_i$), y C_l es un conjunto de restricciones de similitud $x_i \sim y_k$.³ Ahora,

³El operador $s \sim t$ establece la similitud entre los términos s y t , y cuando es interpretado, $\vartheta(s \sim t) = \hat{\mathcal{R}}(s, t)$.

dada una regla $R : H \leftarrow B$, tal que $\text{lin}(H) = \langle H_l, C_l \equiv \{x_1 \sim y_1, \dots, x_n \sim y_n\} \rangle$, entonces $\text{lin}(R) = H_l \leftarrow x_1 \sim y_1 \wedge \dots \wedge x_n \sim y_n \wedge B$.

Ejemplo 3.4. Sea R la regla FASILL $p(x, x) \leftarrow q(x)$. Entonces,

$$\text{lin}(R) = p(y_1, y_2) \leftarrow x \sim y_1 \wedge x \sim y_2 \wedge q(x),$$

y todas las reglas básicas que son similares a R pueden obtenerse como instancias básicas de $\text{lin}(R)$.

Para un conjunto de reglas Π , $\text{lin}(\Pi) = \{\text{lin}(R) : R \in \Pi\}$. Ahora es posible dar la noción de *programa extendido*.

Definición 3.6 (Programa extendido, [JIMP18]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL. Entonces, el conjunto de reglas similares a las reglas en $\text{lin}(\Pi)$:

$$\mathcal{K}(\Pi) = \{H \leftarrow \alpha \wedge B : (H' \leftarrow B) \in \text{lin}(\Pi), \hat{\mathcal{R}}(H, H') = \alpha \geq \perp\}$$

es el *programa extendido* de \mathcal{P} .

El programa extendido de \mathcal{P} refleja el significado inducido por la relación de similitud \mathcal{R} en el conjunto de reglas Π . Este significado se mide mediante los grados de similitud de las reglas que son similares a las reglas en $\text{lin}(\Pi)$: $\hat{\mathcal{R}}(H, H') = \alpha \in L$. Por otro lado, se observa que $\mathcal{K}(\Pi)$ es también un programa FASILL y, en general, $\text{lin}(\Pi) \subseteq \mathcal{K}(\Pi)$. El programa extendido $\mathcal{K}(\Pi)$ es solo un mecanismo de notación que nos permite decidir qué interpretaciones difusas son modelos de un programa, pero no juega ningún papel en la definición de la semántica operacional.

Definición 3.7 (Modelo de un programa, [JIMP18]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL. Una interpretación difusa \mathcal{I} es *modelo* de \mathcal{P} si y solo si, \mathcal{I} es modelo de toda regla $(H \leftarrow \alpha \wedge B) \in \mathcal{K}(\Pi)$. Esto es, $\vartheta_\phi(H) \geq \alpha \wedge \vartheta_\phi(B)$ para cualquier asignación ϕ .

En programación lógica, la semántica declarativa de un programa se suele formular en base al modelo mínimo de Herbrand, concebido como el ínfimo de un conjunto de interpretaciones de Herbrand.

Dado un programa FASILL $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$, este genera un lenguaje de primer orden \mathcal{L} , cuyo alfabeto se compone de los símbolos que aparecen en las reglas de Π y en las entradas de la relación de similitud \mathcal{R} , a partir de los cuales se genera el universo de Herbrand $\mathcal{U}_{\mathcal{L}}$. En una interpretación difusa de Herbrand, como en el caso clásico (véase la definición 2.18), los símbolos de constante y de función son interpretados como ellos mismos, mientras que los símbolos de relación n -arios son interpretados como relaciones difusas sobre $\mathcal{U}_{\mathcal{L}}$ (es decir, aplicaciones de $\mathcal{U}_{\mathcal{L}}^n$ a elementos del retículo L). Además, de forma análoga al caso clásico, es posible caracterizar una interpretación de Herbrand difusa como un subconjunto difuso de la base de Herbrand.

Definición 3.8 (Interpretación de Herbrand difusa, [JIMP18]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL. Una *interpretación de Herbrand difusa* es una aplicación $\mathcal{I} : \mathcal{B}_{\mathcal{P}} \rightarrow L$, donde $\mathcal{B}_{\mathcal{P}}$ es la base de Herbrand de \mathcal{P} .

Sea \mathcal{H} el conjunto de interpretaciones de Herbrand difusas cuya relación de orden se induce del orden de L :⁴

$$\mathcal{I}_1 \leq \mathcal{I}_2 \leftrightarrow \mathcal{I}_1(A) \leq \mathcal{I}_2(A), \forall A \in \mathcal{B}_{\mathcal{P}}.$$

Una interpretación de Herbrand difusa que es modelo de un programa \mathcal{P} es un *modelo de Herbrand difuso* de \mathcal{P} . En [JIMP17] se demuestra que, dado un programa FASILL \mathcal{P} , la interpretación difusa $\mathcal{I}_{\mathcal{P}} = \inf\{\mathcal{I}_j : \mathcal{I}_j \text{ es modelo de } \mathcal{P}\}$ es el modelo mínimo de Herbrand de \mathcal{P} .

Teorema 3.2 (Corrección, [JIMP17]). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y A un átomo básico. Si $\langle v, id \rangle$ es una f.c.a. para A en \mathcal{P} , entonces $v \leq \mathcal{I}_{\mathcal{P}}(A)$.*

Teorema 3.3 (Completitud, [JIMP17]). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y A un átomo básico. Si $\mathcal{I}_{\mathcal{P}}(A) = v$, entonces $v \leq \sup\{r \in L : \langle A, id \rangle \rightsquigarrow^* \langle r, id \rangle\}$.*

3.4. El sistema FASILL

Uno de los objetivos de esta tesis ha sido proporcionar una implementación del lenguaje FASILL, robusta y eficiente, que nos permita incorporar todas las características y transformaciones que estudiamos en nuestras investigaciones. Concluimos este capítulo con una sección sobre los detalles prácticos y de implementación del lenguaje FASILL que se recogen –con ligeras modificaciones– en nuestra aportación [JIMR20]. El sistema FASILL es una implementación de alto nivel escrita en Prolog que actualmente se ejecuta bajo SWI-Prolog [WSTL12], y que está públicamente disponible en la página web de nuestro grupo DECTAU.⁵ El código fuente se distribuye libremente en GitHub bajo la licencia BSD modificada de 3 cláusulas.⁶

3.4.1. Interfaz

El sistema FASILL es un intérprete interactivo del lenguaje de mismo nombre que permite cargar programas y consultar objetivos sobre estos. La figura 3.2 muestra la consola interactiva de FASILL ejecutando varios objetivos sobre el programa del ejemplo 3.1. El intérprete de comandos lee la entrada hasta el primer salto de línea, analiza el término FASILL y lo ejecuta como un objetivo. Tras la primera respuesta computada difusa, el usuario puede presionar la tecla “;” para buscar la siguiente respuesta, o cualquier otra tecla para abortar la búsqueda. Además, es posible ejecutar determinados comandos:

- `:exit` sale del sistema FASILL.
- `:help` muestra la lista de comandos disponibles.

⁴Es trivial comprobar que (\mathcal{H}, \leq) hereda la estructura de retículo completo de (L, \leq) .

⁵<https://dectau.uclm.es/fasill>

⁶<https://github.com/jariazavalverde/fasill>

```

fasill> consult('good_hotel.fpl').
<1.0, {}>

fasill> consult_sim('good_hotel.sim').
<1.0, {}>

fasill> metro ~ taxi.
<0.4, {}>

fasill> good_hotel(X).
<0.645, {X/hydropolis}> ;
<0.605, {X/ritz}>

fasill> findall(X-TD, truth_degree(good_hotel(X), TD), L).
<1.0, {L/[hydropolis-0.645, ritz-0.605]}>

```

Figura 3.2: Consola interactiva del sistema FASILL.

- `:lattice(Path)` carga la descripción de un retículo desde el fichero identificado por `Path`.
- `:license` muestra la licencia de FASILL.
- `:listing` lista las reglas cargadas en la sesión actual.

Aunque la carga de reglas y ecuaciones de similitud se lleva a cabo mediante predicados incorporados, es importante observar que no es posible cargar un nuevo retículo de esta forma, y por lo tanto se hace necesario utilizar el comando `:lattice`.

También es posible utilizar el sistema FASILL a través de la herramienta en línea [MR17, MR19a] mostrada en las figuras 3.3 y 3.4.⁷ Bajo el cuadro de texto que contiene las reglas del programa FASILL en la figura 3.3, se encuentra un segundo cuadro que modela el retículo de grados de verdad $([0, 1], \leq)$ cargado por defecto en el sistema. El tercer cuadro de texto contiene un conjunto de ecuaciones de similitud, cuyo cierre produce una relación de similitud asociada al programa FASILL. Finalmente, la figura 3.4 muestra el conjunto de respuestas computadas difusas y la representación textual del árbol de derivación asociado a la ejecución de un objetivo. Además, nuestra herramienta también es capaz de mostrar el árbol de derivación de forma gráfica, tal y como se ve en la figura 3.1.

⁷<https://dectau.uclm.es/fasill/sandbox>

</> Program

```

1 cheap(taxi) <- 0.8.
2 close(hydropolis, taxi) <- 0.7.
3 close(ritz, metro) <- 0.9.
4 good_hotel(X) <- @aver(@very(close(X, Y)), cheap(Y)).

```

Linearize program

Extend program

Unfold program

● Lattice

```

1 % Elements
2 member(X) :- number(X), 0 =< X, X =< 1.
3 members([0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]).
4
5 % Distance
6 distance(X,Y,Z) :- Z is abs(Y-X).

```

bool

unit

real

= Similarity Relation

```

1 atlantis ~ ritz = 0.6.
2 metro ~ bus = 0.5.
3 bus ~ taxi = 0.4.
4 ~tnorm = godel.

```

ⓘ Depth limit

0

✂ Cut value

Cut value (optional)

Running

Tuning

🚩 Goal

good_hotel(X).

Run goal

Figura 3.3: Área de entrada del sistema FASILL en línea.

3.4.2. Predicados incorporados

El sistema FASILL no es sólo un meta-intérprete que simula la semántica operacional del lenguaje FASILL, sino que cuenta además con un gran número de predicados incorporados, muchos de ellos inspirados en los predicados propuestos por el estándar ISO Prolog [ISO95]. En particular, FASILL cuenta con el mismo mecanismo para el manejo de excepciones, basado en los predicados

🔍 Fuzzy Computed Answers

```

1 <0.645, {X/hydropolis}>
2 <0.605, {X/ritz}>
3 execution time: 2 milliseconds

```

🌲 Derivation tree

```

1 GOAL <good_hotel(X), {X/X}>
2   good_hotel/1 <@aver(@very(close(V0,V1)),cheap(V1)), {X/V0}>
3     close/2 <@aver(@very(0.7),cheap(taxi)), {X/hydropolis}>
4       cheap/1 <@aver(@very(0.7),0.8), {X/hydropolis}>
5         IS <0.645, {X/hydropolis}>
6     close/2 <@aver(@very(0.9),cheap(metro)), {X/ritz}>
7       cheap/1 <@aver(@very(0.9),&godel(0.4,0.8)), {X/ritz}>
8         IS <0.605, {X/ritz}>

```

Draw derivation tree

Figura 3.4: Área de salida del sistema FASILL en línea.

incorporados `catch/3` y `throw/1`, y dispone de los mismos predicados para la comparación de términos, evaluación y comparación aritmética, procesamiento de átomos, etcétera.⁸ En esta sección describimos en detalle algunos de los predicados más relevantes, como las estructuras de control o los predicados de unificación.

Negación y confirmación

Al igual que Prolog, FASILL incluye un mecanismo de negación como fallo, mediante el uso del meta-predicado `(\+)/1`, donde el objetivo `\+(Goal)` falla con grado de verdad \perp si existe alguna respuesta computada difusa para `Goal` con grado de verdad distinto de \perp , o tiene éxito con grado de verdad \top sin ligar ninguna variable en caso contrario. Además, en analogía a este predicado FASILL cuenta con el meta-predicado incorporado `(+)/1`, que invoca un objetivo obligándolo a tener éxito, o a fallar sin la posibilidad de seguir con la derivación. Es decir, `(+)/1` invoca un objetivo inhibiendo la ejecución de un posible paso de fallo. El siguiente ejemplo ilustra el uso de este predicado.

Ejemplo 3.5. En ocasiones es necesario declarar reglas que no tienen un carácter inherentemente difuso o que no se benefician de la semántica operacional de los pasos de fallo, ya que llevan a derivaciones fallidas que no es necesario computar. Por ejemplo, el siguiente programa FASILL define una versión difusa

⁸Puede consultarse un listado completo de los predicados incorporados de FASILL en la URL <https://dectau.uclm.es/fasill/documentation#ref>.

de la ordenación por inserción, cuyo grado de verdad refleja lo ordenada que estaba la lista de entrada inicialmente.

```

1  sort([], []).
2  sort([X|Xs], Zs) <- sort(Xs, Ys) &prod insert(X, Ys, Zs).
3
4  insert(X, [], [X]).
5  insert(X, [Y|Ys], [X,Y|Ys]) <- +(X @=< Y).
6  insert(X, [Y|Ys], [Y|Zs]) <-
7      +(X @> Y) &prod insert(X, Ys, Zs) &prod 0.995.

```

Sin el uso de (+)/1, ante un objetivo como $\mathcal{G} = \text{sort}([3, 2, 1], xs)$, el sistema FASILL reportaría una respuesta con grado de verdad 0 para cada posible permutación de la lista de entrada, excepto en una permutación, la lista ordenada, que es la única respuesta computada difusa que se esperaría obtener para este objetivo: $\langle 0.985, \{xs/[1, 2, 3]\} \rangle$. El meta-predicado (+)/1 nos permite descartar prematuramente estas derivaciones fallidas.

Fallo y éxito

Mientras que Prolog cuenta con los predicados `true/0`, que siempre tiene éxito, y `false/0` (o `fail/0`), que siempre falla; FASILL define los predicados `top/0` y `bot/0`, que siempre tienen éxito con grado de verdad \top y \perp , respectivamente. Nótese que `bot/0` simplemente simula un paso de fallo y por lo tanto la derivación puede continuar. Sin embargo, es fácil imitar el comportamiento de `fail/0` en FASILL combinando el uso de (+)/1 y `bot/0`, lo que provocará el fin prematuro de la derivación.

```

1  fail <- +bot.

```

Llamadas a predicados

Al igual que Prolog, FASILL incluye meta-predicados para invocar objetivos, como `call/[1..]`, que invoca un objetivo con argumentos adicionales, y `once/1`, que busca sólo la primera respuesta computada difusa. Además, FASILL dispone de una versión difusa, `truth_degree/2` (y equivalentemente el operador infijo `on/2`), que permite al usuario obtener el grado de verdad asociado a un objetivo como un término más. El objetivo `truth_degree(Goal, TD)` (o `Goal on TD`) tiene éxito cuando TD es el grado de verdad asociado a una respuesta computada difusa para el objetivo `Goal`. El predicado `truth_degree/2` siempre tiene éxito con grado de verdad \top . La siguiente cláusula es una posible implementación de este predicado en Prolog, donde sólo hay dos posibles grados de verdad.

```

1  truth_degree(Goal, TD) :- Goal *-> TD = true ; TD = false.

```

Ejemplo 3.6. Sea \mathcal{P} el programa FASILL mostrado en el ejemplo 3.1. Entonces, el objetivo $\mathcal{G}_0 = (\text{good_hotel}(x) \text{ on } v)$ lleva a dos respuestas computadas difusas: $\langle 1.0, \{x/\text{hydropolis}, v/0.645\} \rangle$ y $\langle 1.0, \{x/\text{ritz}, v/0.605\} \rangle$. Una práctica común es invocar al grado de verdad tras el objetivo para restituir el grado de verdad de la respuesta computada difusa, por ejemplo, $\mathcal{G}_1 = (\text{good_hotel}(x) \text{ on } v \ \& \ \text{call}(v))$.

Conectivas clásicas

FASILL permite definir una gran variedad de conectivas en el retículo, entre ellas, conjunciones y disyunciones difusas. Además, también incluye la conjunción $(,)/2$ y la disyunción $(;)/2$ clásicas de Prolog como predicados incorporados. En el caso de la conjunción, el predicado $(,)/2$ es equivalente al uso de la t-norma fija \wedge del retículo, y se incluye simplemente por compatibilidad con Prolog.

```
1 ' , '(G1, G2) <- G1 & G2.
```

No obstante, el predicado incorporado $(;)/2$ no es equivalente al uso de una disyunción difusa, ya que este no agrega los grados de verdad de dos objetivos, sino que crea dos puntos de elección distintos, uno para cada objetivo.

```
1 ' ; '(G1, _) <- G1.
2 ' ; '(_, G2) <- G2.
```

Ejemplo 3.7. Sea L el retículo $([0, 1], \leq)$. Entonces, el objetivo $\mathcal{G}_0 = (0.3; 0.7)$ lleva a dos respuestas computadas difusas, $\langle 0.3, id \rangle$ y $\langle 0.7, id \rangle$; mientras que el objetivo $\mathcal{G}_1 = (0.3 \text{ |}_{\text{luka}} 0.7)$ lleva a una sola f.c.a. $\langle 1.0, id \rangle$.

FASILL también incluye el predicado incorporado $(->)/2$, que permite la ejecución condicional de objetivos. Su semántica es equivalente a la siguiente definición, donde se hace uso de $(+)/1$ para evitar la ejecución del cuerpo cuando no se satisface la condición.

```
1 ' -> '(If, Then) <- +once(If) & Then.
```

Al igual que en Prolog, la estructura de control `' ; ' ('->' (If, Then), Else)` (escrita normalmente como `If->Then;Else`) cambia la semántica del operador $(;)/2$ al combinarse con el condicional $(->)/2$.

Unificación sintáctica y unificación débil

Por defecto, el sistema FASILL utiliza el algoritmo de unificación débil y no realiza la comprobación de ocurrencia de variables. No obstante, tanto la unificación débil (`weak_unification`) como la ocurrencia de variables (`occurs_check`) pueden ser activadas (`true`) o desactivadas (`false`) mediante el uso de la directiva `set_fasill_flag/3`. Cuando se desactiva la unificación débil, el sistema

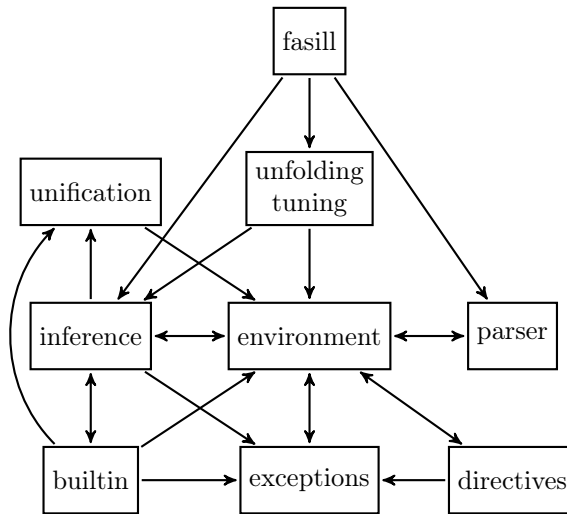


Figura 3.5: Grafo de dependencias funcionales del sistema FASILL.

utiliza el algoritmo de unificación sintáctica. Además, FASILL incluye los predicados incorporados `(=)/2` (unificación sintáctica), `(~)/2` (unificación débil), `unify_with_occurs_check/2` (unificación sintáctica con comprobación de ocurrencia de variables) y `weakly_unify_with_occurs_check/2` (unificación débil con comprobación de ocurrencia de variables).

3.4.3. Detalles de implementación

La implementación completa de FASILL consiste en más de 6000 líneas de código Prolog distribuidas en varios módulos. La figura 3.5 muestra la estructura del sistema FASILL mediante un grafo de dependencias funcionales, donde se han omitido algunos módulos por simplicidad. En este grafo, cada vértice (o nodo) representa un módulo, una flecha de un vértice x a otro vértice y indica que el módulo x importa el módulo y , y una flecha bidireccional refleja una referencia circular. La lista exhaustiva de los módulos que conforman el sistema FASILL es la siguiente:

- El módulo `fasill` exporta los predicados `fasill/0` y `fasill/1`, que implementan una consola interactiva, proporcionando la interfaz para el usuario.
- El módulo `fasill_arithmetic` contiene predicados para la evaluación de expresiones aritméticas. Este módulo se utiliza esencialmente por los predicados incorporados que evalúan expresiones aritméticas (como `is/2`).
- Los módulos `fasill_builtin` y `fasill_directives` contienen la definición de los predicados incorporados y de las directivas de FASILL, respectivamente.

- El módulo `fasill_environment` contiene predicados para manipular las reglas del programa, el retículo y la relación de similitud cargadas en la sesión actual.
- El módulo `fasill_exceptions` contiene predicados para la creación y gestión de errores.
- El módulo `fasill_inference` contiene la implementación de la semántica operacional de FASILL, y exporta predicados para simular pasos de inferencia y ejecutar objetivos.
- El módulo `fasill_linearization` contiene la implementación de la técnica de linealización mostrada en la sección 3.3 sobre la semántica declarativa de FASILL.
- El módulo `fasill_parser` contiene predicados para el análisis sintáctico de programas y objetivos FASILL.
- El módulo `fasill_sandbox` proporciona predicados para facilitar la ejecución inmediata de FASILL, cargando un programa y realizando una tarea específica (ejecutar un objetivo, desplegar o calibrar un programa, etcétera). Este módulo es utilizado esencialmente por la herramienta en línea.
- Los módulos `fasill_substitution` y `fasill_term` contienen predicados para la creación, manipulación y razonamiento de sustituciones y términos FASILL.
- Los módulos `fasill_tuning` y `fasill_tuning_smt` contienen la implementación de las técnicas de calibrado que se describirán en detalle en el capítulo 5.
- El módulo `fasill_unfolding` contiene la implementación de las técnicas de desplegado que se describirán en detalle en el capítulo 6.
- El módulo `fasill_unification` contiene predicados para la unificación sintáctica y unificación débil de términos FASILL.

Sintaxis

La fase de análisis de las reglas y de los esquemas de similitud ha sido implementada utilizando gramáticas de cláusulas definidas, mientras que los retículos son cargados como ficheros Prolog. El analizador sintáctico de FASILL es un analizador completamente nuevo implementado en Prolog; es decir, no es una mera parametrización apropiada del analizador de SWI-Prolog, ya que, aunque la sintaxis de FASILL y Prolog es muy cercana, existen algunas diferencias significativas. Por ejemplo, los operadores en FASILL pueden ser etiquetados (`&godel`, `@aver`, ...). Además, controlando el proceso de análisis, es posible un mejor tratamiento de los errores sintácticos y hacer frente a la sintaxis simbólica

Tabla 3.1: Operadores predefinidos de FASILL.

Prioridad	Asociatividad	Etiquetado	Operadores
1300	xfx	✗	<- :-
1300	fx	✗	<- :-
1200	xfx	✗	width
1100	xfy	✓	
1100	xfy	✗	;
1050	xfy	✗	->
1000	xfy	✓	&
1000	xfy	✗	, &
900	xfx	✗	on
700	xfx	✗	= \= ~ \~
700	xfx	✗	== \== @< @=< @> @>=
700	xfx	✗	=..
700	xfx	✗	is := =\= < =< > >=
500	yfx	✗	+ - /\ \/
400	yfx	✗	* / // rem mod << >>
200	xfx	✗	**
200	xfy	✗	^
200	fy	✗	- + \

utilizada en el proceso de calibrado de programas FASILL, como veremos más adelante.

De forma análoga a la tabla 2.2 que contiene los operadores de Prolog, la tabla 3.1 muestra los operadores definidos inicialmente en el sistema FASILL. Esta tabla dispone de una columna adicional para especificar si un operador debe ser etiquetado (✓) o no (✗). Las instrucciones en FASILL se codifican como reglas sucedidas de un punto, donde las conectivas lógicas difusas se representan por los operadores (&)/2 (conjunción), (|)/2 (disyunción), (@)/n (agregador) y (<-)/2 (implicación inversa).

Reglas

El predicado `program_consult/1` del módulo `fasill_environment` analiza un fichero de entrada en formato FASILL y lo compila a un conjunto de términos Prolog que se almacenan como hechos dinámicos del predicado `fasill_rule/3`, donde el primer argumento es la cabeza de la regla, el segundo es el cuerpo de la regla y el tercero es una lista que contiene información sobre la misma. Por ejemplo, la regla R_4 mostrada en el ejemplo 3.1 es compilada al siguiente término Prolog.

```

1  fasill_rule(
2      head(term(good_hotel, [var('X')])),
3      body(term('@'(aver), [
4          term('@'(very), [term(close, [var('X'), var('Y')])])],

```

```

5      term(elegant, [var('Y')]))),
6      [id(4), syntax(fasill)])

```

Aquí, utilizamos una representación básica (*ground*) para almacenar y manipular los términos FASILL con Prolog, donde:

- las variables se etiquetan como términos `var/1`, cuyo único argumento es un átomo que representa su identificador (por ejemplo, una variable con el identificador `X` se representa como `var('X')`);
- los números se etiquetan como términos `num/1`, cuyo único argumento es un número que representa su valor (por ejemplo, un número con valor `1` se representa como `num(1)`);
- y los átomos y términos compuestos se etiquetan como términos `term/2`, cuyo primer argumento es un término que representa su funtor, y el segundo es una lista que contiene sus argumentos etiquetados (por ejemplo, el átomo `p(a,X)` se representa como `term(p, [term(a, []), var('X')])`).

Retículos completos

Los retículos son descritos en el sistema FASILL mediante un conjunto de cláusulas Prolog, donde la definición de los siguientes predicados es obligatoria: `member/1`, que tiene éxito cuando el término que recibe representa un grado de verdad; `bot/1` y `top/1`, que devuelven el ínfimo y el supremo del retículo, respectivamente; y `leq/2`, que caracteriza la relación de orden.

Las conectivas difusas también se definen como predicados. El nombre de estos predicados va precedido por `and_`, `or_`, o `agr_`, en función del tipo de conectiva que representan (una conjunción, una disyunción o un agregador, respectivamente). La aridad del predicado es $n + 1$, donde n es la aridad de la conectiva y el último argumento representa el resultado de la evaluación de dicha conectiva tomando como entrada los n primeros parámetros.

Ejemplo 3.8. Las siguientes cláusulas Prolog modelan el retículo $([0, 1], \leq)$ en el sistema FASILL:

```

1  % Elements
2  member(X) :- number(X), 0 =< X, X =< 1.
3
4  % Ordering relation
5  leq(X,Y) :- X =< Y.
6
7  % Supremum and infimum
8  bot(0.0).
9  top(1.0).
10
11 % Binary operations
12 and_prod(X,Y,Z) :- Z is X*Y.

```

```

13 and_godel(X,Y,Z) :- Z is min(X,Y).
14 and_luka(X,Y,Z) :- Z is max(X+Y-1,0).
15 or_prod(X,Y,Z) :- Z is (X+Y)-(X*Y).
16 or_godel(X,Y,Z) :- Z is max(X,Y).
17 or_luka(X,Y,Z) :- Z is min(X+Y,1).
18
19 % Aggregators
20 agr_aver(X,Y,Z) :- Z is (X+Y)/2.
21 agr_geom(X,Y,Z) :- Z is sqrt(X*Y).
22 agr_very(X,Y) :- Y is X*X.

```

Relaciones de similitud

Una relación de similitud puede especificarse parcialmente estableciendo un conjunto de ecuaciones de similitud. Una ecuación de similitud es una declaración con forma textual $x/n \sim y/n = \alpha$, que se almacena en el sistema FASILL como un hecho dinámico del predicado `fasill_similarity/3`. Cada hecho representa una entrada $\mathcal{R}(x_n, y_n) = \alpha$ de la relación binaria difusa \mathcal{R} , y su lectura intuitiva es que dos símbolos n -arios, x e y , están relacionados con un cierto grado α . La especificación de la aridad para las constantes (`/0`) se puede omitir en la ecuación.

Nótese que si el usuario introduce dos o más ecuaciones conflictivas, el sistema FASILL mantiene el grado de similitud de la primera de ellas y muestra un mensaje de error. Por ejemplo, asumiendo que el usuario introduce las siguientes ecuaciones de similitud: `bus ~ taxi = 0.4`, `bus ~ taxi = 0.7` y `taxi ~ bus = 0.8`, entonces FASILL considera que el grado de similitud entre `bus` y `taxi` es 0.4, y reporta el siguiente mensaje de error al usuario:

```
warning(conflicting_equations(bus/0, taxi/0, [0.4, 0.7, 0.8])).
```

Cuando un conjunto de ecuaciones de similitud se carga en el entorno, el predicado `similarity_closure/0` es invocado para computar los cierres reflexivo, simétrico y transitivo con el fin de generar una relación de similitud completa.

```

1 similarity_closure :-
2     similarity_domain(Dom),
3     similarity_scheme(Scheme),
4     similarity_tnorm(Tnorm),
5     lattice_call_bot(Bot),
6     lattice_call_top(Top),
7     similarity_retract,
8     similarity_closure_check_equations(Dom, Eq),
9     similarity_closure_reflexive(Dom, Scheme, Tnorm, Bot, Top),
10    similarity_closure_symmetric(Dom, Scheme, Tnorm, Bot, Top),
11    similarity_closure_transitive(Dom, Scheme, Tnorm, Bot, Top),
12    assertz(fasill_similarity_tnorm(Tnorm)).

```

Este predicado primero busca el dominio \mathcal{U} de la relación de similitud \mathcal{R} que se va a generar; esto es, el conjunto de todos los símbolos que aparecen en las ecuaciones de similitud (evitando elementos duplicados), y recoge todas las ecuaciones de similitud en una lista de términos de la forma $\text{sim}(x,y,n,r)$. Después se elimina el esquema de similitud actual y se computan los cierres reflexivo, simétrico y transitivo.

El predicado `similarity_closure_reflexive/5` computa el cierre reflexivo de un esquema de similitud. Para ello, toma cada símbolo $x \in \mathcal{U}$ y establece que el símbolo x tiene un grado de similitud consigo mismo de \top . Esto garantiza la propiedad reflexiva de la relación de similitud, $\mathcal{R}(x,x) = \top$.

```

1 similarity_closure_reflexive(Dom, _, _, _, Top) :-
2   forall(
3     member(X/Arity, Dom),
4     assertz(fasill_similarity(X/Arity,X/Arity,Top))
5   ).

```

El predicado `similarity_closure_symmetric/5` computa el cierre simétrico de un esquema de similitud. Para ello, toma cada par de elementos $(x,y) \in \mathcal{U}^2$ y busca si hay una ecuación $x \sim y = \alpha$ o $y \sim x = \alpha$. Si existe, el predicado establece que el grado de similitud entre los símbolos x e y es α , y viceversa. Esto garantiza la propiedad simétrica de la relación de similitud, $\mathcal{R}(x,y) = \mathcal{R}(y,x)$.

```

1 similarity_closure_symmetric(Dom, Scheme, _, _, _) :-
2   forall(
3     ( member(X/Arity, Dom),
4       member(Y/Arity, Dom),
5       \+(fasill_similarity(X/Arity,Y/Arity,_,_)),
6       once( member(sim(X,Y,Arity,TD), Scheme)
7           ; member(sim(Y,X,Arity,TD), Scheme)
8         )
9     ), (
10      assertz(fasill_similarity(X/Arity,Y/Arity,TD)),
11      assertz(fasill_similarity(Y/Arity,X/Arity,TD))
12    )
13  ).

```

Finalmente, el predicado `similarity_closure_transitive/5` computa el cierre transitivo de un esquema de similitud. Para ello, toma cada tripla $(x,y,z) \in \mathcal{U}^3$ y establece que los símbolos x y z tienen un grado de similitud $(\mathcal{R}(x,y) \wedge \mathcal{R}(y,z)) \vee \mathcal{R}(x,z)$. Esto garantiza la propiedad transitiva de la relación de similitud, $\mathcal{R}(x,z) \geq \mathcal{R}(x,y) \wedge \mathcal{R}(y,z)$. El predicado `lattice_call_connective/3` se utiliza para invocar conectivas del retículo.

```

1 similarity_closure_transitive(Dom, _, Tnorm, Bot, _) :-
2   forall(

```

```

3      ( member(Y/Arity, Dom),
4        member(X/Arity, Dom), X \= Y,
5        member(Z/Arity, Dom), Z \= Y, X \= Z,
6        once( fasill_similarity(X/Arity,Z/Arity,TDxz)
7            ; TDxz = Bot),
8        once(fasill_similarity(X/Arity,Y/Arity,TDxy)),
9        once(fasill_similarity(Y/Arity,Z/Arity,TDyz))
10     ), (
11       lattice_call_connective('&'(Tnorm), [TDxy,TDyz], TDxyz),
12       lattice_call_connective('|'(Tnorm), [TDxz,TDxyz], TD),
13       retractall(fasill_similarity(X/Arity,Z/Arity,_)),
14       assertz(fasill_similarity(X/Arity,Z/Arity,TD))
15     )
16   ).

```

Nuestro algoritmo para el cálculo del cierre transitivo es una adaptación directa del algoritmo de Warshall para encontrar el camino mínimo en grafos dirigidos ponderados [War62], y tiene complejidad $\mathcal{O}(n^3)$, donde $n = |\mathcal{U}|$ es el número de elementos del dominio. Sin embargo, es importante aclarar que esto no produce un sobre coste en tiempo de ejecución, ya que los cierres se computan una única vez en tiempo de compilación. En tiempo de ejecución el sistema FASILL simplemente consulta las entradas de la relación de similitud resultante.

Unificación débil

La unificación débil se aplica tanto a términos como a átomos. El algoritmo de unificación débil ha sido implementado siguiendo el algoritmo de unificación sintáctica dado por Robinson [Rob65] que, en el contexto de una implementación de alto nivel, se comporta como el algoritmo de unificación de Martelli-Montanari [MM82].

El predicado `lambda_wmgu/5` computa el unificador débil más general umbralizado de dos expresiones. En general, el valor del umbral establecido por FASILL es \perp , pero el usuario puede establecer un umbral de unificación cambiando el valor de `lambda_cut` mediante la directiva o el predicado incorporado `set_fasill_flag/3` para que la unificación de dos expresiones sólo tenga éxito cuando el grado de unificación sea mayor o igual a un umbral dado.⁹

```

1  lambda_wmgu(A, B, Lambda, OccursCheck, WMGU) :-
2      lattice_call_top(Top),
3      empty_substitution(Sub),
4      lambda_wmgu(A, B, Lambda, OccursCheck, state(Top,Sub), WMGU).

```

⁹Por ejemplo, el usuario puede establecer el valor del umbral a 0.5 ejecutando el objetivo “`set_fasill_flag(lambda_cut, 0.5)`”.

El objetivo `lambda_wmgu(A, B, Lambda, OccursCheck, state(TD,WMGU))` tiene éxito cuando `WMGU` es el λ -`wmgu` _{\mathcal{R}} de `A` y `B`, donde `TD` \geq `Lambda` es el grado de unificación.

El predicado auxiliar `lambda_wmgu/6` es utilizado por `lambda_wmgu/5` para pasar de un estado a otro durante el proceso de unificación, empezando por el estado inicial $\langle \top, id \rangle$. Aquí, los dos primeros argumentos representan las expresiones a ser unificadas débilmente (en su representación básica), el tercer argumento es el valor del umbral (también en su representación básica), el cuarto argumento es un átomo que determina si se debe comprobar la ocurrencia de variables (`true`) o no (`false`), el quinto argumento es el estado actual de la unificación, y el sexto argumento es el siguiente estado de la unificación. Un estado es representado en FASILL mediante un término Prolog de la forma `state(TD,Sub)`, donde `TD` es el grado de unificación y `Sub` es el λ -`wmgu` _{\mathcal{R}} . FASILL utiliza la librería `assoc` de SWI-Prolog para manipular las sustituciones como listas de asociaciones (implementadas como árboles AVL). Como Prolog, FASILL permite el uso de variables anónimas “_” para unificar una expresión sin ligar ningún valor a la variable.

Por ejemplo, la siguiente cláusula corresponde a la regla de transición (3) de la Figura 2.3, donde se unifica una variable con un término que no contiene dicha variable.

```

1 lambda_wmgu(var(V), T, _, Occurs, state(TD,S0), state(TD,S3)) :-
2     !,
3     (Occurs == true -> occurs_check(V, T) ; true),
4     list_to_substitution([V-T], S1),
5     compose(S0, S1, S2),
6     put_substitution(S2, V, T, S3).
```

Pasos de inferencia

La semántica operacional de FASILL ha sido implementada como un sistema de transición de estados. El predicado `query/2` tiene éxito cuando el primer argumento es un objetivo (en su representación básica) y el segundo es una respuesta computada difusa para ese objetivo. Un estado es un término de la forma `state(Goal, Substitution)` y una f.c.a. es un estado donde el objetivo es un grado de verdad del retículo.

```

1 query(Goal, Answer) :-
2     init_substitution(Goal, Vars),
3     State = state(Goal, Vars),
4     derivation(top_level/0, State, Answer, _).
```

El estado inicial de la derivación está compuesto por el objetivo del argumento de `query/2` y por una sustitución inicial $\{x_0/x_0, \dots, x_n/x_n\}$ que liga cada variable que ocurre en el objetivo a ella misma. Por ejemplo, la sustitución inicial para

el objetivo “`elegant(X, Y)`” es codificada en Prolog como la lista de asociaciones `t('X', var('X'), >, t, t('Y', var('Y'), -, t, t))`, que indica que la variable `X` toma el valor `var('X')` y la variable `Y` toma el valor `var('Y')`. Estas listas se implementan como árboles binarios de búsqueda AVL.¹⁰

El predicado `derivation/4` realiza una derivación desde el estado inicial hasta una respuesta computada difusa o hasta una excepción sin capturar. Este predicado recibe el indicador¹¹ del predicado que realizó la llamada y un estado inicial, y devuelve el último estado y una lista que contiene información sobre los pasos de computación realizados (para fines de depuración).

```

1 derivation(From, State1, State2, Info) :-
2   current_fasill_flag(depth_limit, num(Depth)),
3   ( catch(
4     derivation(From, State1, State2, Depth, 0, Info),
5     exception(Error),
6     (State2 = exception(Error), Info = [])
7   )
8   *-> true
9   ; lattice_call_bot(Bot),
10     State1 = state(_, S0),
11     State2 = state(Bot, S0)
12   ).

```

Si no existe ninguna derivación, se genera la respuesta computada difusa $\langle \perp, \theta \rangle$, donde θ es la sustitución inicial.

El predicado auxiliar `derivation/6` va de un estado al siguiente hasta alcanzar una respuesta computada difusa o una excepción sin capturar. Este predicado toma dos parámetros adicionales para controlar la profundidad de la derivación que el usuario puede establecer cambiando el valor de la bandera `depth_limit` mediante `set_fasill_flag/3`.

```

1 derivation(_, State, State, Depth, Depth, []) :-
2   Depth > 0, !.
3 derivation(_, exception(Error), exception(Error), _, _, []) :-
4   throw(exception(Error)), !.
5 derivation(_, state(Goal,Sub), state(Goal,Sub), _, _, []) :-
6   is_fuzzy_computed_answer(Goal), !,
7   lattice_call_bot(Bot),
8   ( Bot == Goal ->
9     current_fasill_flag(failure_steps, term(true, []))
10   ; true ).
11 derivation(From, State1, State3, Depth, N, [X|Xs]) :-

```

¹⁰<https://www.swi-prolog.org/pldoc/man?section=assoc>

¹¹Un predicado con nombre `Pred` que recibe `N` argumentos se denota usualmente por su indicador `Pred/N`.

```

12     catch(
13         inference(From, State1, State2, X),
14         Error,
15         (State2 = exception(Error), !)),
16     succ(N, M),
17     derivation(From, State2, State3, Depth, M, Xs).

```

La primera cláusula devuelve el estado actual de la derivación si se ha llegado a la profundidad máxima. La segunda cláusula lanza una excepción si se ha alcanzado un error sin capturar. La tercera cláusula comprueba si se ha alcanzado una respuesta computada difusa, mediante `is_fuzzy_computed_answer/1`, que invoca al predicado `member/1` del retículo para comprobar si el objetivo es un grado de verdad. Si el grado de verdad asociado a esta f.c.a. es \perp , el sistema comprueba que los pasos de fallo están habilitados mediante la bandera `failure_steps`. Finalmente, la cuarta cláusula realiza un paso de inferencia mediante el predicado `inference/4`.

```

1  inference(From, State1, State2, Info) :-
2      operational_step(From, State1, State2, Info).
3  inference(From, state(Goal,Subs), State2, Info) :-
4      interpretable(Goal),
5      interpretive_step(From, state(Goal,Subs), State2, Info).

```

El predicado `inference/4` trata de realizar un paso operacional. Si no es posible, entonces realiza un paso interpretativo. Por lo tanto, los pasos interpretativos sólo son dados cuando todos los átomos del objetivo han sido resueltos. El predicado `interpretable/1` tiene éxito cuando un objetivo no contiene ningún átomo.

La fase operacional está dividida en dos tipos de pasos: pasos de éxito y pasos de fallo. Un paso de fallo se realiza sólo cuando no es posible realizar ningún paso de éxito. Esto es controlado mediante el uso del llamado *corte blando*¹² `(*->)/2`.

```

1  operational_step(From, State1, State2, Info) :-
2      ( success_step(From, State1, State2, Info) *->
3          true
4          ; failure_step(State1, State2, Info) ).

```

El predicado `success_step/4` funciona de manera similar a un paso SLD en un intérprete de Prolog. Primero, `success_step/4` selecciona el átomo más a la izquierda del objetivo mediante `select_atom/4` y comprueba si es un predicado incorporado o un predicado definido por el usuario. Si el átomo es un predicado

¹²El corte blando `(*->)/2` es una estructura de control que se comporta como el condicional `(->)/2` con la diferencia de que reevalúa la premisa.

incorporado, es resuelto por el predicado `eval_builtin_predicate/4` del módulo `fasill_builtin`. Si es un predicado definido por el usuario, siguiendo las reglas definidas en la sección 3.2, el sistema busca las reglas cuyas cabezas unifiquen débilmente con el átomo y reemplaza el átomo seleccionado por el cuerpo de la regla, componiendo el w.m.g.u. con la sustitución del estado actual. Si el predicado no existe, se lanza una excepción (`existence_error`).

```

1 success_step(From, state(G0,S0), state(G1,S1), Name2/Arity) :-
2   select_atom(G0, ExprVar, Var, Expr),
3   Expr = term(Name, Args),
4   length(Args, Arity),
5   (Name = Name2 ;
6     (current_fasill_flag(weak_unification, term(true, []))->
7       lattice_call_bot(Bot),
8       similarity_between(Name, Name2, Arity, Sim, _),
9       Name \= Name2, Sim \== Bot)
10  ),
11  % Builtin predicate
12  (is_builtin_predicate(Name2/Arity) -> (
13    eval_builtin_predicate(
14      Name2/Arity,
15      state(G0,S0),
16      selected(ExprVar, Var, Expr),
17      state(G1,S1)
18    )
19  ) ; (
20    % User-defined predicate
21    (program_has_predicate(Name2/Arity) -> (
22      lattice_tnorm(Tnorm),
23      lattice_call_top(Top),
24      program_clause(Name2/Arity, Rule),
25      Rule = fasill_rule(head(Head), Body, _),
26      rename([Head,Body], [HeadR,BodyR]),
27      unify(Expr, HeadR, _, state(TD,SubsExpr)),
28      (BodyR = empty ->
29        Var = TD ;
30        ( BodyR = body(Body_),
31          (TD == Top ->
32            Var = Body_ ;
33            Var = term('&'(Tnorm), [TD,Body_])))
34      ),
35      apply(SubsExpr, ExprVar, G1),
36      compose(S0, SubsExpr, S1)
37    ) ; (
38      % Undefined predicate
39      existence_error(procedure, Name/Arity, From, Error),
40      throw_exception(Error)

```

```

40         ))
41     )).

```

El predicado `failure_step/3` es invocado cuando un átomo falla y reemplaza el átomo seleccionado por el ínfimo del retículo. Nótese que el usuario puede desactivar los pasos de fallo estableciendo el valor de `failure_steps` a `false` mediante `set_fasill_flag/3`, o invocando el objetivo mediante el meta-predicado `(+)/1`.

```

1 failure_step(state(G0,S0), state(G1,S0), 'FS') :-
2     current_fasill_flag(failure_steps, term(true, [])),
3     lattice_call_bot(Bot),
4     select_atom(G0, G1, Bot, Atom),
5     Atom \= term('+', [_]).

```

Finalmente, el predicado `interpretive_step/3` es invocado cuando el objetivo no contiene ningún átomo. Este predicado selecciona la conectiva más a la izquierda cuyos argumentos son todos grados de verdad, y la evalúa llamando al predicado `lattice_call_conectiva/3`, que invoca a la conectiva adecuada del retículo y devuelve el resultado.

```

1 interpretive_step(From, state(G0,S0), state(G1,S0), 'IS') :-
2     ( select_expression(G0, G1, Var, Expr) ->
3       interpret(Expr, Var)
4     ; type_error(truth_degree, G0, From, Error),
5       throw_exception(Error) ).

```

Algunos predicados utilizan los términos `bot` y `top` como una representación básica del ínfimo y el supremo del retículo, respectivamente. Estos términos también son evaluados mediante la aplicación de pasos interpretativos.

```

1 interpret(bot, Bot) :- !, lattice_call_bot(Bot).
2 interpret(top, Top) :- !, lattice_call_top(Top).
3 interpret(sup(X, Y), Z) :- !, lattice_call_supremum(X, Y, Z).
4 interpret(term(Op, Args), Result) :-
5     lattice_call_connective(Op, Args, Result).

```

3.4.4. Pruebas y evaluación de rendimiento

Con el fin de determinar el sobrecoste introducido por nuestras técnicas de implementación que materializan el paradigma de programación lógico difuso sobre un sistema Prolog clásico, hemos medido el tiempo de ejecución y el número de inferencias de algunos programas FASILL,¹³ tanto interpretados como

¹³Los programas utilizados en nuestros experimentos pueden ejecutarse en la versión en línea de FASILL. El código fuente de las pruebas puede encontrarse en la URL <https://github.com/jariazaalverde/fasill/tree/master/test/experiments>.

Tabla 3.2: Tiempo medio de ejecución (en milisegundos) de un programa FASILL arbitrario con n hechos tras 100 ejecuciones, en función de la t-norma utilizada en la derivación.

n	Clásica	Gödel	Łukasiewicz	Producto
50	14	14	14	14
100	53	51	50	50
200	216	209	208	202
300	439	443	471	452
400	779	778	774	775
500	1209	1202	1200	1201
1000	4860	4763	4729	4887

compilados.¹⁴

Resolución

En nuestro primer experimento discutimos los resultados de utilizar varias t-normas diferentes a la conjunción clásica en un programa FASILL. Para ello, consideramos una regla general de la forma “ $p \leftarrow p_1 \wedge p_2 \wedge \dots \wedge p_n$ ” junto con un conjunto de hechos “ $p_i \leftarrow w_i$ ” que definen cada uno de los símbolos proposicionales p_i con un peso arbitrario $w_i \in [0, 1]$ generado aleatoriamente. La tabla 3.2 resume el tiempo medio de ejecución obtenido tras ejecutar diferentes instancias del programa FASILL propuesto, utilizando las diferentes t-normas de las lógicas de Gödel, de Łukasiewicz y del producto (todas ellas basadas en operaciones aritméticas). El contraste realizado con respecto al caso clásico (donde se utiliza un retículo con dos valores de verdad cuya t-norma es la conjunción de la lógica clásica), revela que la evaluación de diferentes t-normas no produce un sobrecoste significativo en tiempo de ejecución.

El sistema FASILL es capaz de compilar programas lógicos difusos a código Prolog estándar. La técnica de compilación fue concebida inicialmente para programas MALP [JIMP06], que pueden ser vistos como programas FASILL sin relaciones de similitud, y son suficientemente expresivos para los experimentos descritos a continuación. El punto clave de este código Prolog es extender cada átomo del programa con un argumento adicional, que será utilizado para contener el grado de verdad obtenido tras la evaluación de un átomo. En el caso de un hecho, el argumento adicional contiene el peso directamente. Por ejemplo, la regla “ $p(a) \leftarrow 0.5$ ” es traducida al hecho Prolog “ $p(a, 0.5)$ ”. Las conectivas difusas son representadas como predicados definidos en el retículo asociado al programa. Por ejemplo, la conectiva binaria $\&_{gödel}$ se define mediante el predicado `and_gödel/3` en el retículo, donde el tercer parámetro representa el resultado de la operación. Así, el resultado de com-

¹⁴Todas las pruebas de esta sección han sido ejecutadas en Swi-Prolog 8.0.6 (64 bits) utilizando un ordenador de sobremesa equipado con un procesador AMD Opteron™ @ 1593 MHz y 2.00 GB RAM.

Tabla 3.3: Tiempo medio de ejecución (en milisegundos) y número de inferencias del problema de las n reinas ejecutado en el intérprete FASILL y en SWI-Prolog (tanto la versión compilada del programa FASILL como el programa original en Prolog) tras 50 ejecuciones.

n	FASILL (interpretado)		FASILL (compilado)		Prolog	
	Tiempo	Inferencias	Tiempo	Inferencias	Tiempo	Inferencias
7	1869	18100580	2	8789	1	4182
8	8359	81092264	4	35763	2	17826
9	43459	402929204	31	158124	11	80825
10	216478	1990401248	142	724980	63	383746
11	1159475	11009507950	718	3656347	268	1973693
12	6847892	66065674305	3728	20021101	1466	10960122

pilar una regla como “ $p(x) \leftarrow q(x, y) \ \&_{godel} \ r(y)$ ” sería la cláusula Prolog “ $p(x, v_0) \leftarrow q(x, y, v_1), r(y, v_2), and_godel(v_1, v_2, v_0)$ ”. La figura 3.6 muestra un programa FASILL (adaptado de [SS94]), que es también un programa MALP y Prolog, junto a su versión compilada a Prolog.

Dado que FASILL subsume la sintaxis de Prolog, un programa escrito en Prolog puro puede ser ejecutado por el sistema FASILL sin ninguna modificación. La tabla 3.3 muestra el tiempo de ejecución y el número de inferencias realizadas por SWI-Prolog para resolver el problema de las n reinas mostrado en la figura 3.6, ejecutando: (1) el código original de Prolog en el intérprete FASILL; (2) la versión compilada del programa FASILL en SWI-Prolog; y (3) el código original de Prolog en SWI-Prolog. Nótese que para estas pruebas hemos deshabilitado los pasos de fallo. Como se observa en la figura 3.7a, se produce un sobrecoste significativo en la versión interpretada del código FASILL. Sin embargo, este sobrecoste se reduce drásticamente cuando compilamos el programa FASILL (sin similitudes) a código Prolog.

El programa anterior no tiene realmente una naturaleza difusa. Por este motivo, presentamos a continuación otra prueba para medir el rendimiento del sistema FASILL con una aplicación real. En particular, nos centramos en el lenguaje FSA-SPARQL, que proporciona mecanismos para expresar consultas difusas contra datos RDF y permite la transformación y difuminación de datos provenientes de APIs de redes sociales [AJBTM18]. La tabla 3.4 muestra el tiempo medio de ejecución y el número de inferencias realizadas por SWI-Prolog al ejecutar una consulta FSA-SPARQL con n resultados. En este caso, no comparamos el rendimiento con Prolog puro, ya que este no puede manejar grados de verdad distintos a **true** y **false**, y las consultas FSA-SPARQL utilizan el retículo real en el intervalo unitario $([0, 1], \leq)$ junto a una gran variedad de conectivas difusas. Nótese que la versión compilada funciona de manera muy eficiente, pero es importante reseñar que las técnicas de calibrado que introduciremos en el capítulo 5 (y que también usamos en el lenguaje FSA-SPARQL [AJBTMR19]) sólo están disponibles a la hora de interpretar un programa FASILL, y no en su versión compilada a Prolog.

Unificación

Para terminar esta sección, prestamos atención a la unificación. Como mencionamos anteriormente, hemos utilizado una adaptación del algoritmo de Mar-

```
:- set_fasill_flag(symbolic,false).
:- set_fasill_flag(failure_steps,false).
:- set_fasill_flag(weak_unification,false).

queens(N) :-
  queens(N, _),
  false ;
  true.
queens(N, Qs) :-
  gen_list(N, Qs),
  place_queens(N, Qs, _, _).
gen_list(0, []).
gen_list(N, [_|L]) :-
  N > 0,
  N1 is N-1,
  gen_list(N1, L).
place_queens(0, _, _, _).
place_queens(I, Qs, Ups, [_|Ds]) :-
  I > 0,
  I1 is I-1,
  place_queens(I1, Qs, [_|Ups], Ds),
  place_queen(I, Qs, Ups, Ds).
place_queen(Q, [Q|_], [Q|_], [Q|_]).
place_queen(Q, [_|Qs], [_|Ups], [_|Ds]) :-
  place_queen(Q, Qs, Ups, Ds).
```

(a) Programa original.

```
queens(N, TD) :-
  queens(N, _, TD1),
  TD1 \= false, TD2 = false,
  and_bool(TD1, TD2, TD),
  TD \= false ; TD = true.
queens(N, Qs, TD) :-
  gen_list(N, Qs, TD1), TD1 \= false,
  place_queens(N, Qs,_,_, TD2), TD2 \= false,
  and_bool(TD1, TD2, TD), TD \= false.
gen_list(0, [], true).
gen_list(N, [_|L], TD) :-
  N > 0, N1 is N-1,
  gen_list(N1, L, TD), TD \= false.
place_queens(0, _, _, true).
place_queens(I, Qs, Ups, [_|Ds], TD) :-
  I > 0, I1 is I-1,
  place_queens(I1, Qs, [_|Ups], Ds, TD1),
  TD1 \= false,
  place_queen(I, Qs, Ups, Ds, TD2),
  TD2 \= false,
  and_bool(TD1, TD2, TD), TD \= false.
place_queen(Q, [Q|_], [Q|_], [Q|_], true).
place_queen(Q, [_|Qs], [_|Ups], [_|Ds], TD) :-
  place_queen(Q, Qs, Ups, Ds, TD),
  TD \= false.
```

(b) Programa compilado.

```
queens(XN,TD) :-
  unify(XN,N,TD1), (queens(N,_,TD2),false ; TD2=1,true),
  degree_composition([TD2,TD1],TD).
queens(XN,XQs,TD) :-
  unify(XN,N,TD1), unify(XQs,Qs,TD2), gen_list(N,Qs,TD3), place_queens(N,Qs,_,_,TD4),
  degree_composition([TD4,TD3,TD1,TD2],TD).
gen_list(X,L,TD) :-
  unify(X,0,TD1), unify(L,[],TD2), degree_composition([TD1,TD2],TD).
gen_list(XN,XL,TD) :-
  unify(XN,N,TD1), unify(XL,[_|L],TD2), N>0, N1 is N-1, gen_list(N1,L,TD3),
  degree_composition([TD3,TD1,TD2],TD).
place_queens(X,Y,Z,L,TD) :-
  unify(X,0,TD1), unify(Y,_,TD2), unify(Z,_,TD3), unify(L,_,TD4),
  degree_composition([TD1,TD2,TD3,TD4],TD).
place_queens(XI,XQs,XUps,L,TD) :-
  unify(XI,I,TD1), unify(XQs,Qs,TD2), unify(XUps,Ups,TD3), unify(L,[_|Ds],TD4),
  I > 0, I1 is I-1, place_queens(I1,Qs,[_|Ups],Ds,TD5), place_queen(I,Qs,Ups,Ds,TD6),
  degree_composition([TD1,TD2,TD3,TD4,TD5,TD6],TD).
place_queen(X,Y,Z,L,TD) :-
  unify(X,Q,TD1), unify(Y,[Q|_],TD2), unify(Z,[Q|_],TD3), unify(L,[Q|_],TD4),
  degree_composition([TD1,TD2,TD3,TD4],TD).
place_queen(X,Y,Z,L,TD) :-
  unify(X,Q,TD1), unify(Y,[_|Qs],TD2), unify(Z,[_|Ups],TD3), unify(L,[_|Ds],TD4),
  place_queen(Q,Qs,Ups,Ds,TD5), degree_composition([TD1,TD2,TD3,TD4,TD5],TD).
```

(c) Programa compilado con unificación débil.

Figura 3.6: Problema de las n reinas en FASILL y su versión compilada a Prolog.

Tabla 3.4: Tiempo medio de ejecución (en milisegundos) y número de inferencias de una consulta FSA-SPARQL con n resultados, ejecutado por el intérprete FASILL y por SWI-Prolog (al compilarlo) tras 50 ejecuciones.

n	FASILL (interpretado)		FASILL (compilado)	
	Tiempo	Inferencias	Tiempo	Inferencias
1	19	178121	0	48
10	248	2450972	0	606
50	4480	42726732	1	7006
100	24049	220263932	7	24006
500	1913152	18194261532	89	520006
1000	13802164	137621008532	365	2040006

telli y Montanari [MM82] como descripción formal del algoritmo de unificación débil, que definimos como un sistema de transición de estados, y tiene una complejidad casi lineal en implementaciones de bajo nivel (es decir, mediante el uso de máquinas virtuales). Sin embargo, hemos implementado la unificación débil siguiendo la línea del algoritmo de unificación de Robinson porque es más simple de implementar. La complejidad de este algoritmo es exponencial respecto al número de símbolos en el peor caso. No obstante, se comporta bien en la práctica excepto para algunos casos atípicos (lo que explica que este algoritmo siga presente en muchas implementaciones actuales de demostradores de teoremas, intérpretes de Prolog, sistemas de tipos, etcétera). En [HV09] se realiza un amplio estudio comparando el rendimiento de diferentes métodos de unificación, donde se muestra que el algoritmo de Martelli y Montanari, aunque tiene una complejidad casi lineal en el peor caso, en la práctica se comporta peor que el

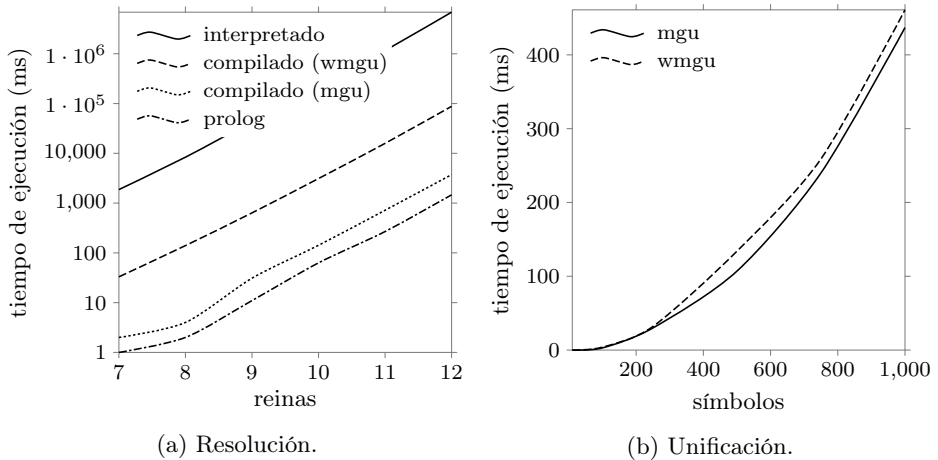


Figura 3.7: Comparación de los tiempos de ejecución del sistema FASILL resolviendo objetivos y unificando términos.

Tabla 3.5: Tiempo de ejecución (en milisegundos) y número de inferencias del algoritmo para computar el unificador más general, al unificar términos FASILL arbitrarios de n símbolos aleatoriamente anidados tras 1000 ejecuciones.

n	Tiempo (ms)			Inferencias		
	mínimo	media	máximo	mínimo	media	máximo
10	0	0	1	90	318	677
100	1	3	10	12997	41302	64948
250	10	30	90	102405	264125	404680
500	39	107	225	372728	1053805	1591275
750	76	240	493	756870	2376113	3497821
1000	101	437	778	1020895	4226163	6258932

Tabla 3.6: Tiempo de ejecución (en milisegundos) y número de inferencias del algoritmo para computar el unificador débil más general, al unificar débilmente términos FASILL arbitrarios de n símbolos aleatoriamente anidados tras 1000 ejecuciones.

n	Tiempo (ms)			Inferencias		
	mínimo	media	máximo	mínimo	media	máximo
10	0	0	3	443	609	852
100	1	4	32	17132	44403	73484
250	12	32	144	107739	290396	438447
500	39	134	294	408423	1119831	1720331
750	91	259	616	829200	2531031	3698443
1000	139	461	1121	1377969	4449438	6540874

algoritmo de Robinson. Por otro lado, hay que tener en cuenta que aquí hemos implementado la unificación débil (y el sistema FASILL completo) mediante SWI-Prolog. Por eso utilizamos técnicas de implementación de alto nivel y delegamos a SWI-Prolog las tareas complejas de la unificación débil, como ligar las variables a los términos y aplicar la composición de las sustituciones. De esta forma, la implementación de ambos algoritmos es muy cercana y se comportan de forma similar dentro de este marco. El sobrecoste de las adaptaciones difusas de estos algoritmos (cuando se basan en similitudes) con respecto a las versiones clásicas de los mismos se debe principalmente a los accesos ineludibles a las entradas de la relación de similitud, y a la aplicación de la t-norma utilizada para propagar los grados de verdad entre los símbolos.

Hemos realizado una prueba para medir el sobrecoste que introduce la unificación débil sobre la unificación sintáctica clásica, dado que los programas utilizados en las evaluaciones previas carecen de similitudes. El experimento muestra el tiempo de ejecución y el número de inferencias realizadas por SWI-Prolog al ejecutar el algoritmo de unificación implementado por FASILL tanto al unificar términos arbitrarios de n símbolos aleatoriamente anidados (véase la tabla 3.5) como al unificar débilmente este tipo de términos (véase la tabla 3.6).

Tabla 3.7: Tiempo medio de ejecución (en milisegundos) y número de inferencias del problema de las n reinas tras 50 ejecuciones. Tanto la versión compilada del programa FASILL, que llama explícitamente al algoritmo de unificación débil, como el programa Prolog original son ejecutados sobre SWI-Prolog.

n	FASILL (compilado + wmgü)		Prolog	
	Tiempo	Inferencias	Tiempo	Inferencias
7	33	426957	1	4182
8	140	1845388	2	17826
9	636	8430242	11	80825
10	3113	40186261	63	383746
11	15837	207367767	268	1973696
12	87825	1153940993	1466	10960122

Al comparar los tiempos de ejecución promedio de ambas tablas, concluimos que la unificación débil es ligeramente más lenta que la unificación sintáctica. Sin embargo, esta diferencia es prácticamente insignificante, como se muestra en la figura 3.7b.

Para terminar, hemos diseñado un último experimento para medir el sobre coste introducido por nuestro algoritmo de unificación débil con respecto al algoritmo de unificación sintáctica de SWI-Prolog. Este objetivo se logra mediante la producción de una variante del código compilado en la figura 3.6b donde se agregan llamadas explícitas al algoritmo de unificación débil (siguiendo técnicas similares a las utilizadas para traducir programas Bousi~Prolog [JISP20]), dando lugar al código mostrado en la figura 3.6c. Como se puede observar en la tabla 3.7, el algoritmo de unificación débil tiene un precio a pagar, pero es asequible. En relación a este punto, la figura 3.7a es muy reveladora, ya que muestra gráficamente el coste creciente que se produce al incluir varias características difusas en un lenguaje de programación lógico.

3.5. Conclusiones

En este capítulo hemos repasado los principales conceptos del lenguaje FASILL, desde su sintaxis y su semántica hasta los aspectos prácticos de su implementación en un lenguaje lógico de alto nivel. A continuación concretamos los diferentes propósitos de este capítulo.

- (1) Hemos introducido el lenguaje FASILL: un lenguaje de primer orden que combina un algoritmo de unificación débil, basado en relaciones de similitud, junto con un amplio repertorio de conectivas difusas cuyas funciones de verdad pueden ser definidas sobre un retículo completo.
- (2) Hemos descrito la semántica operacional de FASILL, que se concibe como un sistema de transición de estados con dos fases claramente diferenciadas: una primera fase de resolución difusa en la que se explotan los átomos

del objetivo, y una segunda fase interpretativa en la que se evalúan las conectivas.

- (3) Hemos descrito también su semántica declarativa, como una extensión difusa del concepto clásico de modelo mínimo de Herbrand [JIMP17, JIMP18], que nos permite dar una noción de las propiedades de corrección y completitud de FASILL.
- (4) Aunque la regla de computación de FASILL no es independiente en general, hemos demostrado que sí lo es respecto de los pasos interpretativos [JIMR22a]. Este resultado, que nos permite “reordenar” los pasos interpretativos en las derivaciones sin afectar a las respuestas computadas difusas de un programa, será de utilidad en la demostración de las propiedades fundamentales de las técnicas de calibrado y desplegado difuso que estudiamos en los siguientes capítulos.
- (5) Hemos presentado el sistema FASILL: una implementación de alto nivel del lenguaje FASILL que incluye características adicionales, como estructuras de control y predicados incorporados, y que permite activar o desactivar en tiempo de ejecución las diferentes componentes difusas del lenguaje. Siguiendo la línea de nuestra aportación [JIMR20], describimos su arquitectura y los detalles de implementación en Prolog.
- (6) Además, hemos mostrado el entorno de programación de FASILL, que se compone de una consola interactiva que se ejecuta sobre SWI-Prolog, y de una herramienta en línea, que permite ejecutar programas lógicos difusos en la web y visualizar de forma gráfica los árboles de derivación.
- (7) Por último, hemos diseñado una serie de experimentos sobre el sistema FASILL, para medir el sobrecoste que supone incorporar las distintas componentes difusas en un lenguaje lógico [JIMR20], tanto al extender el mecanismo de resolución, como al reemplazar el algoritmo de unificación sintáctica por un algoritmo de unificación débil basado en relaciones de similitud.

Por lo tanto, en este capítulo –junto al capítulo 2 donde proporcionamos algunos resultados novedosos sobre las propiedades del algoritmo de unificación débil, las sustituciones y el operador de composición paralela débil [JIMR22b]– hemos establecido las bases fundamentales sobre las que desarrollamos las técnicas de calibrado y desplegado difuso estudiadas en esta tesis, tanto de forma teórica como de forma práctica.

Capítulo 4

La extensión simbólica SFASILL

En este capítulo se introduce, en los términos que recogen nuestras propuestas [MPRV17, MR17, RM20, MR20, MR21], una extensión simbólica de la programación lógica difusa. En esencia, permitiremos que las reglas y las relaciones de similitud de los programas FASILL contengan algunos valores (grados de verdad) y conectivas indefinidas, que podrán ser calculadas automáticamente más tarde en tiempo de calibrado. En adelante, utilizaremos la abreviatura SFASILL (acrónimo de «*Symbolic Fuzzy Aggregators and Similarity Into a Logic Language*») para referirnos a estos programas.

4.1. Sintaxis

Denotaremos los objetos simbólicos mediante un superíndice “#” ($o^\#$) y, en el sistema FASILL, mediante un identificador que comienza por el carácter “#”. A continuación se presenta el lenguaje formal de la extensión simbólica del lenguaje FASILL.

Alfabeto. Dado un retículo completo L , consideramos un alfabeto aumentado $\Sigma_L^\#$ que genera un lenguaje de primer orden $\mathcal{L}^\# \supseteq \mathcal{L}$, que puede incluir un número infinito numerable de valores simbólicos $\Sigma_L^{T^\#} = \{v_1^\#, v_2^\#, \dots\}$ y conectivas simbólicas $\Sigma_L^{C^\#} = \{\zeta_1^\#, \zeta_2^\#, \dots\}$ cuya interpretación no se asocia a priori¹ a valores y conectivas de L . Llamamos *constantes simbólicas* de forma genérica al conjunto de valores simbólicos y conectivas simbólicas de $\mathcal{L}^\#$.

¹En el capítulo siguiente estudiaremos distintas técnicas de calibrado cuyo principal objetivo será establecer (a posteriori) conexiones entre elementos simbólicos y elementos concretos del retículo.

Fórmulas bien formadas. El lenguaje combina variables y símbolos de función para construir términos de la forma usual. Del mismo modo, combina términos y símbolos de predicado para construir fórmulas atómicas. Las fórmulas bien formadas se construyen inductivamente conforme a las siguientes reglas:

- (1) Todo elemento del conjunto de literales, $r \in \Sigma_L^T$, es una fórmula bien formada.
- (2) Todo valor simbólico, $v^\# \in \Sigma_L^{T^\#}$, es una fórmula bien formada.
- (3) Toda fórmula atómica es una fórmula bien formada.
- (4) Si $\zeta^n \in \Sigma_L^C$ es un símbolo de conectiva y A_1, \dots, A_n son fórmulas bien formadas, también lo es $\zeta^n(A_1, \dots, A_n)$.
- (5) Si $\zeta^\# \in \Sigma_L^{C^\#}$ es una conectiva simbólica y A_1, \dots, A_n son fórmulas bien formadas, también lo es $\zeta^\#(A_1, \dots, A_n)$.

Una $L^\#$ -expresión es una fórmula bien formada de $\mathcal{L}^\#$ que está compuesta únicamente por valores y conectivas de L y por constantes simbólicas. Denotamos por $\text{exp}_L^\#$ al conjunto de todas las $L^\#$ -expresiones en $\mathcal{L}^\#$. Dada una $L^\#$ -expresión $E^\#$, $\vartheta_L^\#(E^\#)$ es la nueva $L^\#$ -expresión obtenida tras evaluar todo lo que sea posible las conectivas en $E^\#$. En particular, si E no contiene ningún elemento o conectiva simbólica, entonces $\vartheta_L^\#(E) = \vartheta_L(E) \in L$. La interpretación de una $L^\#$ -expresión se define inductivamente como:

- (1) $\vartheta_L^\#(r) = v$ si y solo si $v \in L$ es la interpretación de $r \in \Sigma_L^T$;
- (2) $\vartheta_L^\#(v^\#) = v^\#$ si y solo si $v^\# \in \Sigma_L^{T^\#}$ es un valor simbólico;
- (3) $\vartheta_L^\#(\zeta^n(A_1, \dots, A_n)) = \vartheta_L(\zeta^n(v_1, \dots, v_n))$ si y solo si ζ^n es una conectiva de Σ_L^C interpretada mediante la función de verdad $F_{\zeta^n} : L^n \rightarrow L$ y, para todo $1 \leq i \leq n$, $\vartheta_L^\#(A_i) = v_i \in L$;
- (4) $\vartheta_L^\#(\zeta(A_1, \dots, A_n)) = \zeta(\vartheta_L^\#(A_1), \dots, \vartheta_L^\#(A_n))$ en cualquier otro caso, donde la conectiva –posiblemente simbólica– ζ no es evaluada.

4.2. Sustituciones simbólicas

Dado un retículo L y un lenguaje simbólico $\mathcal{L}^\#$, consideramos a continuación la noción de *sustitución simbólica*, que en esencia es una aplicación de valores y conectivas simbólicos de $\mathcal{L}^\#$ a valores y conectivas pertenecientes a L .

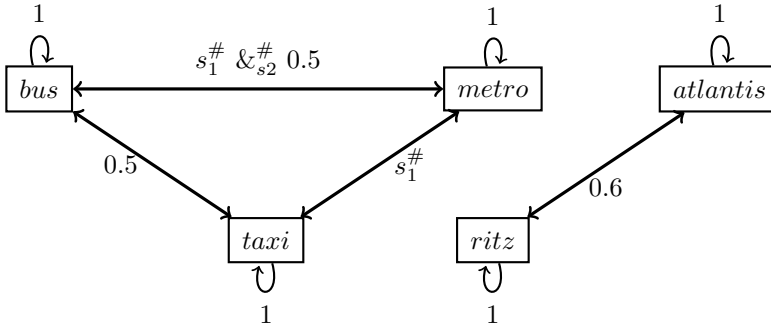
Definición 4.1 (Sustitución simbólica). Sea $\mathcal{L}^\#$ un lenguaje simbólico de primer orden. Una *sustitución simbólica* es una aplicación $\Theta : \Sigma_L^{T^\#} \cup \Sigma_L^{C^\#} \rightarrow \Sigma_L^T \cup \Sigma_L^C$ que asigna un elemento de Σ_L^T a cada valor simbólico de $\Sigma_L^{T^\#}$, y un símbolo de conectiva de Σ_L^C a cada conectiva simbólica de $\Sigma_L^{C^\#}$.

Denotamos por $\text{sym}(E^\#)$ al conjunto de valores y conectivas simbólicos en $E^\#$. Dada una sustitución simbólica Θ de $\text{sym}(E^\#)$, denotamos por $E^\#\Theta$ a la expresión resultante de reemplazar cada elemento simbólico $e^\# \in \text{sym}(E^\#)$ por $\Theta(e^\#)$ en $E^\#$.

4.3. Relaciones de similitud simbólicas

Definición 4.2 (Relación de similitud simbólica, [MR21]). Dado un dominio \mathcal{U} y un retículo completo L con una t-norma –posiblemente simbólica– fija \wedge , una *relación de similitud simbólica* es una aplicación $\mathcal{R}^\# : \mathcal{U} \times \mathcal{U} \rightarrow \exp_L^\#$ tal que, para cualquier sustitución simbólica Θ de $\text{sym}(\mathcal{R}^\#)$, el resultado de interpretar todas las L -expresiones en $\mathcal{R}^\#\Theta$, esto es, $\mathcal{R}(x, y) = \vartheta_L(\mathcal{R}^\#(x, y)\Theta)$ para todo $x, y \in \mathcal{U}$, genera una relación de similitud \mathcal{R} .

Ejemplo 4.1. Sea L el retículo completo $([0, 1], \leq)$ y \wedge una t-norma simbólica $\&_{s_2}^\#$. El siguiente grafo caracteriza la relación de similitud simbólica $\mathcal{R}^\#$ definida sobre el universo $\mathcal{U} = \{\text{ritz}, \text{atlantis}, \text{metro}, \text{taxi}, \text{bus}\}$.



Dado que la definición 4.2 puede parecer demasiado exigente a la hora de diseñar relaciones de similitud simbólicas válidas, presentamos a continuación un método para lograr este objetivo de forma fácil y segura, partiendo de un conjunto inicial de ecuaciones de similitud simbólicas.

Definición 4.3 (Esquema de similitud simbólico, [MR20]). Dado un dominio \mathcal{U} y un retículo completo L con una t-norma \wedge fija (posiblemente simbólica), un *esquema de similitud simbólico* $\mathcal{S}^\#$ es un conjunto de ecuaciones de la forma $x \sim y = v^\#$, donde $x, y \in \mathcal{U}$ y $v^\#$ es una $L^\#$ -expresión.

El cierre de un esquema de similitud simbólico se computa mediante el algoritmo 4.1, inspirado por [JI08, KY74, NDMDB02] (que es una adaptación del algoritmo 2.1) y produce una relación de similitud simbólica válida. Por simplicidad, asumimos que no hay ecuaciones conflictivas en el esquema de similitud simbólico inicial $\mathcal{S}^\#$; esto es, no hay dos ecuaciones distintas $(x \sim y = v_1) \in \mathcal{S}^\#$ y $(x \sim y = v_2) \in \mathcal{S}^\#$ (o equivalentemente $(y \sim x = v_2) \in \mathcal{S}^\#$) tal que $v_1 \neq v_2$.

Algoritmo 4.1: Cierre de un esquema de similitud simbólico

Datos: Un esquema de similitud simbólico $\mathcal{S}^\#$ en \mathcal{U} con una t-norma \wedge (posiblemente simbólica)

Resultado: Una relación de similitud simbólica $\mathcal{R}^\#$ en \mathcal{U}

Sea $\mathcal{R}^\#$ una relación binaria difusa en \mathcal{U} con todas sus entradas a \perp ;

para cada $x \in \mathcal{U}$ **hacer**

para cada $y \in \mathcal{U}$ **hacer**

si $(x \sim y = v) \in \mathcal{S}^\# \vee (y \sim x = v) \in \mathcal{S}^\#$ **entonces**

$\mathcal{R}^\#(x, y) \leftarrow v$;

en otro caso

$\mathcal{R}^\#(x, y) \leftarrow \perp$;

fin

fin

$\mathcal{R}^\#(x, x) \leftarrow \top$;

fin

para cada $x \in \mathcal{U}$ **hacer**

para cada $y \in \mathcal{U}$ **hacer**

para cada $z \in \mathcal{U}$ **hacer**

$\mathcal{R}^\#(x, y) \leftarrow \vartheta_L^\#(\sup\{\mathcal{R}^\#(x, y), \mathcal{R}^\#(x, z) \wedge \mathcal{R}^\#(z, y)\})$;

fin

fin

fin

devolver $\mathcal{R}^\#$;

Ejemplo 4.2. El cierre del siguiente esquema de similitud simbólico $\mathcal{S}^\#$ definido sobre el universo $\mathcal{U} = \{\text{ritz}, \text{atlantis}, \text{metro}, \text{taxi}, \text{bus}\}$ con una t-norma simbólica $\&_{s_2}^\#$:

$$\mathcal{S}^\# = \begin{cases} \text{metro} \sim \text{taxi} = s_1^\# \\ \text{taxi} \sim \text{bus} = 0.5 \\ \text{atlantis} \sim \text{ritz} = 0.6 \end{cases}$$

genera la relación de similitud simbólica $\mathcal{R}^\#$ mostrada en el ejemplo 4.1.

Definición 4.4 (Regla simbólica SFASILL). Dado un retículo completo L , una *regla simbólica* SFASILL es una fórmula bien formada de la forma $A \leftarrow B$, donde A (la cabeza) es un átomo de \mathcal{L}_L y B (el cuerpo) es un objetivo simbólico, es decir, una fórmula bien formada de $\mathcal{L}_L^\#$ que no contiene la conectiva \leftarrow .

Definición 4.5 (Programa simbólico SFASILL). Un *programa simbólico* SFASILL es una tupla $\langle \Pi^\#, \mathcal{R}^\#, L \rangle$, donde $\Pi^\#$ es un conjunto de reglas simbólicas construidas sobre un alfabeto Σ y $\Sigma_L^\#$, $\mathcal{R}^\#$ es una relación de similitud simbólica cuyo dominio es Σ , y L es un retículo completo.

Ejemplo 4.3. El retículo $L = ([0, 1], \leq)$, la relación de similitud simbólica $\mathcal{R}^\#$ mostrada en el ejemplo 4.1 y el siguiente conjunto de reglas $\Pi^\#$ forman un

programa sFASILL $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$:

$$\Pi^\# = \begin{cases} R_1 : \text{cheap}(\text{taxi}) & \leftarrow s_3^\# \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_4 : \text{good_hotel}(x) & \leftarrow @_{s_4}^\#(\text{@}_{\text{very}}(\text{close}(x, y)), \text{cheap}(y)) \end{cases}$$

Trivialmente, se observa que dado un programa sFASILL $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ y una sustitución simbólica Θ en $\text{sym}(\mathcal{P}^\#)$, el programa $\mathcal{P}^\#\Theta = \langle \Pi^\#\Theta, \mathcal{R}^\#\Theta, L \rangle$ resultante es un programa FASILL.

Ejemplo 4.4. Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ el programa simbólico mostrado en el ejemplo 4.3. Dada la sustitución simbólica

$$\Theta = \{s_1^\#/0.4, \&_{s_2}^\#/\&_{\text{godel}}, s_3^\#/0.8, @_{s_4}^\#/@_{\text{aver}}\},$$

entonces $\mathcal{P}^\#\Theta$ produce el programa FASILL mostrado en el ejemplo 3.1.

4.4. Semántica operacional

La semántica operacional de sFASILL se diseña de manera análoga a como se recoge en la sección 3.2 para el lenguaje FASILL. En primer lugar, se introduce un tipo de paso operacional que, a diferencia de la programación lógica difusa estándar, devuelve una $L^\#$ -expresión posiblemente simbólica. A partir de ahí, un paso interpretativo evalúa las conectivas no simbólicas tanto como sea posible y produce una respuesta final, que todavía puede contener valores y conectivas simbólicas. Además, sFASILL extiende el algoritmo de unificación débil para tratar con relaciones de similitud simbólicas introduciendo el concepto de *unificador simbólico débil más general* (o *s.w.m.g.u.*).

Ejemplo 4.5. Dada la relación de similitud $\mathcal{R}^\#$ del ejemplo 4.1 con una t-norma fija simbólica $\&_{s_2}^\#$, tenemos que:

$$\begin{aligned} \langle \{\text{close}(\text{ritz}, \text{taxi}) \sim \text{close}(\text{atlantis}, \text{metro})\}, \text{id}, \top \rangle & \Rightarrow_{\text{wmgu}} \\ \langle \{\text{ritz} \sim \text{atlantis}, \text{taxi} \sim \text{metro}\}, \text{id}, \top \rangle & \Rightarrow_{\text{wmgu}} \\ \langle \{\text{taxi} \sim \text{metro}\}, \text{id}, 0.6 \rangle & \Rightarrow_{\text{wmgu}} \\ \langle \{\}, \text{id}, (0.6 \&_{s_2}^\# s_1^\#) \rangle & \end{aligned}$$

y, por lo tanto, el unificador simbólico débil más general de $\text{close}(\text{ritz}, \text{taxi})$ y $\text{close}(\text{atlantis}, \text{metro})$ es la $L^\#$ -expresión $(0.6 \&_{s_2}^\# s_1^\#)$:

$$\text{wmgu}_{\mathcal{R}^\#}(\text{close}(\text{ritz}, \text{taxi}), \text{close}(\text{atlantis}, \text{metro})) = \langle \text{id}, (0.6 \&_{s_2}^\# s_1^\#) \rangle.$$

La semántica operacional de los programas FASILL y sFASILL está basada en un esquema similar. La principal diferencia es que, para programas FASILL,

el paso interpretativo siempre devuelve un valor $v \in L$, mientras que para programas sFASILL es posible obtener una expresión cuyas componentes simbólicas deben ser reemplazadas por elementos y conectivas del retículo antes de poder computar su valor. Llamamos a estas respuestas simbólicas evaluadas tanto como es posible *respuestas computadas difusas simbólicas* (o *s.f.c.a.* para abreviar).

Ejemplo 4.6. Sea $\mathcal{P}^\#$ el programa mostrado en el ejemplo 4.3. La siguiente es una derivación que lleva a una respuesta computada difusa simbólica para el objetivo $\mathcal{G} = \text{good_hotel}(x)$ en $\mathcal{P}^\#$:

$$\begin{aligned} \mathcal{D}^{\mathcal{P}^\#} : \quad & \langle \text{good_hotel}(x), id \rangle && \rightsquigarrow_{SS}^{R_4} \\ & \langle @_{s_4}^\# (@_{\text{very}}(\underline{\text{close}}(x_1, y_1)), \text{cheap}(y_1)), \{x/x_1, y/y_1\} \rangle && \rightsquigarrow_{SS}^{R_2} \\ & \langle @_{s_4}^\# (@_{\text{very}}(0.7), \underline{\text{cheap}}(\text{taxi})), \{x/\text{hydropolis}, y/\text{taxi}\} \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle @_{s_4}^\# (@_{\text{very}}(0.7), s_3^\#), \{x/\text{hydropolis}, y/\text{taxi}\} \rangle && \rightsquigarrow_{IS} \\ & \langle @_{s_4}^\# (0.49, s_3^\#), \{x/\text{hydropolis}, y/\text{taxi}\} \rangle && \end{aligned}$$

Entonces, $\langle @_{s_4}^\# (0.49, s_3^\#), \{x/\text{hydropolis}\} \rangle$ es una respuesta computada difusa simbólica para \mathcal{G} . Además, para este mismo objetivo existe otra derivación con respuesta computada difusa simbólica $\langle @_{s_4}^\# (0.81, (s_1^\# \&_2^\# s_3^\#)), \{x/\text{ritz}\} \rangle$ (véase la figura 4.1).

Tal y como demostramos originalmente en [MPRV17] para programas sMALP (una extensión simbólica de la programación lógica multi-adjunta), el siguiente teorema, que adaptamos aquí al lenguaje sFASILL –pero sin contemplar todavía relaciones de similitud simbólicas–, es un resultado fundamental para poder utilizar sFASILL en el proceso de calibrado que estudiaremos en el siguiente capítulo.

Teorema 4.1. *Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}, L \rangle$ un programa sFASILL con una relación de similitud ordinaria (que no contiene constantes simbólicas). Sea \mathcal{G} un objetivo (posiblemente simbólico). Entonces, para cualquier sustitución simbólica Θ en $\text{sym}(\mathcal{P}^\#) \cup \text{sym}(\mathcal{G})$, tenemos que $\langle v, \theta \rangle$ es una respuesta computada difusa para $\mathcal{G}\Theta$ en $\mathcal{P}^\#\Theta$ si y solo si existe una respuesta computada difusa simbólica $\langle \mathcal{G}', \theta' \rangle$ para \mathcal{G} en $\mathcal{P}^\#$ tal que $\langle \mathcal{G}'\Theta, \theta' \rangle \rightsquigarrow_{IS}^* \langle v, \theta \rangle$, donde θ' es un renombramiento de θ .*

Demostración. Considérense las siguientes derivaciones para un objetivo \mathcal{G} con respecto a los programas $\mathcal{P}^\#$ y $\mathcal{P}^\#\Theta$, respectivamente. (Por simplicidad, asumiremos que se utilizan las mismas variables frescas para renombrar las reglas de los dos programas en ambas derivaciones):

$$\begin{aligned} \mathcal{D}^{\mathcal{P}^\#} & : \quad \langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle \mathcal{G}'', \theta \rangle \rightsquigarrow_{IS}^* \langle \mathcal{G}', \theta \rangle; \\ \mathcal{D}^{\mathcal{P}^\#\Theta} & : \quad \langle \mathcal{G}\Theta, id \rangle \rightsquigarrow^* \langle \mathcal{G}''\Theta, \theta \rangle \rightsquigarrow_{IS}^* \langle \mathcal{G}'\Theta, \theta \rangle; \end{aligned}$$

donde $\langle \mathcal{G}'', \theta \rangle$ y $\langle \mathcal{G}''\Theta, \theta \rangle$ son los estados alcanzados tras explotar todos los átomos de ambos objetivos. La prueba procede en tres pasos:

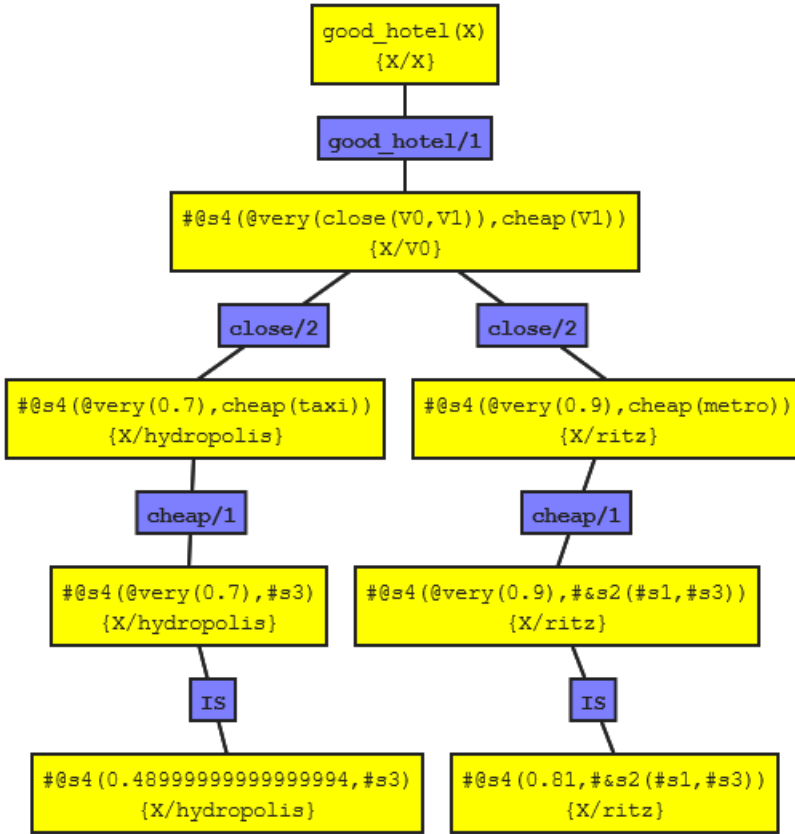


Figura 4.1: Árbol de derivación simbólico generado por el sistema FASILL en línea.

- (1) Primero se observa que la secuencia de pasos de éxito y de fallo en $\mathcal{D}^{\mathcal{P}^\#}$ y en $\mathcal{D}^{\mathcal{P}^\#\Theta}$ explotan los mismos átomos en ambas derivaciones, es decir, una regla $R \in \Pi^\#$ es utilizada en $\mathcal{D}^{\mathcal{P}^\#}$ si y solo si la regla correspondiente $R\Theta \in \Pi^\#\Theta$ es utilizada en $\mathcal{D}^{\mathcal{P}^\#\Theta}$.
- (2) Después aplicamos pasos interpretativos hasta alcanzar la respuesta computada difusa simbólica $\langle \mathcal{G}', \theta \rangle$ en la derivación $\mathcal{D}^{\mathcal{P}^\#}$. Por el teorema 3.1 de independencia de la regla de computación difusa con respecto a los pasos interpretativos, podemos aplicar los mismos pasos interpretativos en la derivación $\mathcal{D}^{\mathcal{P}^\#\Theta}$, llegando al estado $\langle \mathcal{G}'\Theta, \theta \rangle$, que no es necesariamente una respuesta computada difusa.
- (3) Por último, basta con aplicar la sustitución simbólica Θ a la respuesta computada difusa simbólica $\langle \mathcal{G}', \theta \rangle$ en la derivación $\mathcal{D}^{\mathcal{P}^\#}$ y completar las

derivaciones con los pasos interpretativos restantes hasta alcanzar la misma respuesta computada difusa $\langle v, \theta \rangle$ en ambas.

□

Ejemplo 4.7. Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ el programa SFASILL mostrado en el ejemplo 4.3. Dado el objetivo simbólico $\mathcal{G} = (\underline{\text{close}(x, \text{metro})} \mid_{\text{godel}} s_5^\#)$, la siguiente es una respuesta computada difusa simbólica para \mathcal{G} en $\mathcal{P}^\#$:

$$\begin{aligned} \mathcal{D}^{\mathcal{P}^\#} : & \langle (\underline{\text{close}(x, \text{metro})} \mid_{\text{godel}} s_5^\#), \text{id} \rangle && \rightsquigarrow_{SS}^{R_2} \\ & \langle ((s_1^\# \ \&_{s_2}^\# \ 0.7) \mid_{\text{godel}} s_5^\#), \{x/\text{hydropolis}\} \rangle \end{aligned}$$

Dada la sustitución simbólica $\Theta = \{s_1^\#/0.0, \&_{s_2}^\#/\&_{\text{godel}}, s_5^\#/0.5\}$, obtenemos entonces la siguiente respuesta computada difusa a partir de la s.f.c.a. anterior:

$$\begin{aligned} \mathcal{D}^{\mathcal{P}^\#} : & \langle ((s_1^\# \ \&_{s_2}^\# \ 0.7) \mid_{\text{godel}} s_5^\#)\Theta, \{x/\text{hydropolis}\} \rangle && \equiv \\ & \langle (\underline{(0.0 \ \&_{\text{godel}} \ 0.7)} \mid_{\text{godel}} 0.5), \{x/\text{hydropolis}\} \rangle && \rightsquigarrow_{IS} \\ & \langle (\underline{0.0} \mid_{\text{godel}} 0.5), \{x/\text{hydropolis}\} \rangle && \rightsquigarrow_{IS} \\ & \langle 0.5, \{x/\text{hydropolis}\} \rangle \end{aligned}$$

Además, existe otra f.c.a. para \mathcal{G} en $\mathcal{P}^\#$: $\langle 0.9, \{x/\text{ritz}\} \rangle$. Si consideramos el mismo objetivo \mathcal{G} en $\mathcal{P}^\#\Theta$, la f.c.a. para *ritz* sigue teniendo un grado de verdad asociado de 0.9, ya que en su derivación no intervienen constantes simbólicas de la relación de similitud. Sin embargo, la f.c.a. para *hydropolis* ya no existe, dado que los símbolos *metro* y *taxi* no son similares y, por lo tanto, el átomo $\text{close}(x, \text{metro})$ no unifica débilmente con la cabeza de la segunda regla, $\text{close}(\text{hydropolis}, \text{taxi})$.

Como se observa en el ejemplo anterior, el teorema 4.1 no es válido, en general, cuando el programa contiene una relación de similitud simbólica. Intuitivamente, el problema es que algunos pasos de éxito que pueden darse en el programa simbólico $\mathcal{P}^\#$ unificando débilmente el átomo seleccionado con la cabeza de la regla mediante una relación de similitud simbólica, no ocurren en el programa $\mathcal{P}^\#\Theta$ cuando la sustitución simbólica Θ provoca que ciertos símbolos ya no sean similares.

Para solucionar este problema, siguiendo la línea de nuestras propuestas [MR20, MR21], introducimos la noción de *sustitución simbólica segura*, que requeriremos para extender el teorema 4.1 a SFASILL con relaciones de similitud simbólicas.

Definición 4.6 (Sustitución simbólica segura). Dada una relación de similitud simbólica $\mathcal{R}^\#$ sobre un dominio \mathcal{U} , y una sustitución simbólica Θ en $\text{sym}(\mathcal{R}^\#)$, decimos que la sustitución simbólica Θ es *segura* respecto a $\mathcal{R}^\#$ si, para todo valor simbólico $v^\# \in \text{sym}(\mathcal{R}^\#)$, $\Theta(v^\#) > \perp$, y la t-norma \wedge asociada a $\mathcal{R}^\#\Theta$ es estrictamente positiva; es decir, para todo $x, y \in L$, si $x, y > \perp$, entonces $x \wedge y > \perp$.

Nótese que, además del caso trivial donde asignamos \perp a un valor simbólico de la relación de similitud $\mathcal{R}^\#$, también es posible generar sustituciones simbólicas inseguras ligando valores simbólicos a valores distintos de \perp si la t-norma no es estrictamente positiva. En el ejemplo 4.7, si aplicamos la sustitución simbólica $\Theta = \{s_1^\#/0.4, \&_{s_2}^\#/\&_{luka}, \dots\}$ a la entrada $\mathcal{R}^\#(taxi, metro) = (s_1^\# \&_{s_2}^\# 0.5)$, obtenemos $\vartheta_L(\mathcal{R}^\#(taxi, metro)\Theta) = \vartheta_L(0.4 \&_{luka} 0.5) = \text{máx}\{0.4+0.5-1, 0\} = 0$, lo que implica que Θ no es una sustitución simbólica segura con respecto a $\mathcal{R}^\#$.

Teorema 4.2. *Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ un programa sFASILL con una relación de similitud simbólica. Sea \mathcal{G} un objetivo (posiblemente simbólico). Entonces, para cualquier sustitución simbólica segura Θ en $\text{sym}(\mathcal{P}^\#) \cup \text{sym}(\mathcal{G})$, tenemos que $\langle v, \theta \rangle$ es una respuesta computada difusa para $\mathcal{G}\Theta$ en $\mathcal{P}^\#\Theta$ si y solo si existe una respuesta computada difusa simbólica $\langle \mathcal{G}', \theta' \rangle$ para \mathcal{G} en $\mathcal{P}^\#$ tal que $\langle \mathcal{G}'\Theta, \theta' \rangle \rightsquigarrow_{IS}^* \langle v, \theta' \rangle$, donde θ' es un renombramiento de θ .*

Demostración. Esta demostración es análoga a la demostración del teorema 4.1. La restricción de las sustituciones simbólicas seguras garantiza que un paso de éxito dado en el programa $\mathcal{P}^\#$ con una regla R es posible si y solo si el mismo paso de éxito en $\mathcal{P}^\#\Theta$ con la regla $R\Theta$ también puede darse. \square

En la práctica no siempre es posible comprobar si una conectiva es estrictamente positiva. Además, para el propósito principal de esta extensión simbólica, que es el calibrado automático de programas lógicos difusos (véase el capítulo 5), sólo es necesario garantizar que toda unificación débil que tiene éxito en la derivación simbólica mediante un paso \rightsquigarrow_{SS} sobre un átomo A del objetivo con una regla $H \leftarrow B$ del programa, $\text{wmgu}_{\mathcal{R}}(A, H) \neq \text{fallo}$, puede replicarse al aplicar previamente la sustitución simbólica al programa y al objetivo.

Definición 4.7 (Conjunto de grados de unificación de una derivación). Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ un programa sFASILL. Dada una derivación (genérica) en el programa $\mathcal{P}^\#$,

$$\mathcal{D} : \langle \mathcal{G}_1[A_1], \theta_1 \rangle \rightsquigarrow_1 \langle \mathcal{G}_2[A_2], \theta_2 \rangle \rightsquigarrow_2 \dots \rightsquigarrow_n \langle \mathcal{G}_{n+1}, \theta_{n+1} \rangle,$$

denotamos por $\text{deg}(\mathcal{D})$ al conjunto de grados de unificación $\{v : (H \leftarrow B) \in \Pi^\#, \text{wmgu}_{\mathcal{R}}(A_i, H) = \langle v, \sigma \rangle, \langle \mathcal{G}_i[A_i], \theta_i \rangle \rightsquigarrow_i \langle \mathcal{G}_i[A_i]/(v \wedge B)\sigma, \theta_i\sigma \rangle, 1 \leq i \leq n\}$ obtenidos en los pasos de éxito dados en \mathcal{D} al unificar átomos de los subobjetivos con las cabezas de reglas de $\Pi^\#$.

Con esto en mente, damos una versión más práctica del teorema 4.2, que utilizaremos en el siguiente capítulo para diseñar un algoritmo de calibrado de programas sFASILL con relaciones de similitud simbólicas.

Teorema 4.3. *Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ un programa sFASILL y \mathcal{G} un objetivo (posiblemente simbólico). Sea \mathcal{D} una derivación para el objetivo \mathcal{G} en $\mathcal{P}^\#$ que lleva a una respuesta computada difusa simbólica $\langle \mathcal{G}', \theta' \rangle$. Entonces, para cualquier sustitución simbólica Θ en $\text{sym}(\mathcal{P}^\#) \cup \text{sym}(\mathcal{G})$ tal que $\forall E^\# \in \text{deg}(\mathcal{D}), \vartheta_L(E^\#\Theta) > \perp$, tenemos que $\langle \mathcal{G}'\Theta, \theta' \rangle \rightsquigarrow_{IS}^* \langle v, \theta' \rangle$ y además $\langle v, \theta \rangle$ es una respuesta computada difusa para $\mathcal{G}\Theta$ en $\mathcal{P}^\#\Theta$, donde θ' es un renombramiento de θ .*

Demostración. Esta demostración es análoga a la demostración del teorema 4.1. La restricción impuesta sobre el conjunto de grados de unificación de la derivación simbólica garantiza que es posible un paso de éxito dado en el programa $\mathcal{P}^\#$ con una regla R si y solo si también puede aplicarse el mismo paso de éxito en $\mathcal{P}^\#\Theta$ con la regla $R\Theta$. \square

4.5. Detalles de implementación

```

fasill> consult('sym_good_hotel.fpl').
<1.0, {}>

fasill> consult_sim('sym_good_hotel.sim').
<1.0, {}>

fasill> bus ~ taxi.
<#&s2(#s1, 0.5), {}>

fasill> good_hotel(X).
<#s4(0.49, #s3), {X/hydropolis}> ;
<#s4(0.81, #&s2(#s1, #s3)), {X/ritz}>

```

Figura 4.2: Consola interactiva del sistema FASILL ejecutando un programa simbólico.

En esta sección se detallan los aspectos de implementación necesarios para dar soporte a la extensión simbólica en el sistema FASILL. Esta extensión puede ser activada o desactivada en tiempo de ejecución, estableciendo el valor de la bandera `symbolic` a `true` o `false` mediante la directiva `set_fasill_flag/3`. La figura 4.2 muestra la consola interactiva de FASILL ejecutando el programa simbólico del ejemplo 4.3. Las figuras 4.3 y 4.4 muestran la herramienta online ejecutando este mismo programa.

Sintaxis simbólica

La extensión simbólica requiere algunos cambios en la sintaxis del lenguaje con el fin de permitir el análisis de las constantes simbólicas de SFASILL. En primer lugar define nuevos tipos de operadores: conjunciones simbólicas $(\&)/2$, disyunciones simbólicas $(\#|)/2$, agregadores simbólicos $(\#@)/n$ y conectivas simbólicas $(\#?)/n$ en general. Este último tipo de conectivas simbólicas pueden ser reemplazadas en tiempo de calibrado por cualquier conectiva del retículo con la misma aridad. Todos los nuevos operadores deben ser etiquetados, siendo la etiqueta el identificador simbólico del operador. La tabla 4.1 muestra las nuevas entradas añadidas a la tabla 3.1 que contiene los operadores iniciales de FASILL.

</> Program

```

1 cheap(taxi) <- #s3.
2 close(hydropolis, taxi) <- 0.7.
3 close(ritz, metro) <- 0.9.
4 good_hotel(X) <- #s4(@very(close(X, Y)), cheap(Y)).

```

Linearize program

Extend program

Unfold program

● Lattice

```

1 % Elements
2 member(X) :- number(X), 0 <= X, X <= 1.
3 members([0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]).
4

```

bool

unit

real

= Similarity Relation

```

1 metro ~ taxi = #s1.
2 taxi ~ bus = 0.5.
3 atlantis ~ ritz = 0.6.
4 -tnorm = #s2.

```

ⓘ Depth limit**✂** Cut value

Running

Tuning

🚩 Goal

Run goal

Figura 4.3: Área de entrada del sistema FASILL en línea cargando un programa simbólico.

🔍 Fuzzy Computed Answers

```

1  <#@s4(0.48999999999999994,#s3), {X/hydropolis}>
2  <#@s4(0.81,#&s2(#s1,#s3)), {X/ritz}>
3  execution time: 14 milliseconds

```

🌲 Derivation tree

```

1  GOAL <good_hotel(X), {X/X}>
2  good_hotel/1 <#@s4(@very(close(V0,V1)),cheap(V1)), {X/V0}>
3  close/2 <#@s4(@very(0.7),cheap(taxi)), {X/hydropolis}>
4  cheap/1 <#@s4(@very(0.7),#s3), {X/hydropolis}>
5      IS <#@s4(0.48999999999999994,#s3), {X/hydropolis}>
6  close/2 <#@s4(@very(0.9),cheap(metro)), {X/ritz}>
7  cheap/1 <#@s4(@very(0.9),#&s2(#s1,#s3)), {X/ritz}>
8      IS <#@s4(0.81,#&s2(#s1,#s3)), {X/ritz}>

```

Draw derivation tree

Figura 4.4: Área de salida del sistema FASILL en línea ejecutando un programa simbólico.

Además, para analizar los valores simbólicos, es necesario añadir una nueva producción a la gramática de `grammar_expression_zero//2`, que analiza un término FASILL (átomos, números, términos compuestos, listas, etcétera). Aquí definimos un valor simbólico como un átomo “#” seguido de un átomo cuyo identificador tiene al menos un carácter de longitud.

```

1  grammar_expression_zero(T, P) -->
2      grammar_symbolic_constant(T, P), !.
3
4  grammar_symbolic_constant(term('#'(Name), []), P) -->
5      [ token(atom, '#', ['#'], _, false),
6        token(atom, Name, [_C|_], P, _) ].

```

Tabla 4.1: Operadores predefinidos de SFASILL.

Prioridad	Asociatividad	Etiquetado	Operadores
1100	xfy	✓	#
1000	xfy	✓	#&
0	f	✓	#@ #?

Relaciones de similitud simbólicas

Para facilitar el cómputo del cierre reflexivo, simétrico y transitivo de un esquema de similitud simbólico, se ha modificado la representación interna de las ecuaciones de similitud añadiendo un parámetro adicional que especifica si el grado de similitud de una ecuación es simbólico o no. Por lo tanto, una ecuación de similitud $x/n \sim y/n = \alpha$ se almacena ahora en el sistema FASILL como un hecho dinámico del predicado `fasill_similarity/4`, donde el último argumento es un átomo que especifica si α es un elemento simbólico (`true`) o no (`false`).

El cierre transitivo ha sido actualizado conforme a la especificación del algoritmo 4.1 que describe el cierre de un esquema de similitud simbólico, añadiendo algunas simplificaciones que no se contemplan en este. Por ejemplo, cuando el grado de similitud de alguno de los operandos de una t-norma es \top , aprovechamos la condición de frontera $x \wedge \top = \top \wedge x = x$ (para todo $x \in L$) para simplificar la expresión simbólica resultante.

```

1  similarity_closure_transitive(Dom, _, Tnorm, Bot, Top) :-
2  forall(
3      ( member(Y/Arity, Dom),
4        member(X/Arity, Dom), X \= Y,
5        member(Z/Arity, Dom), Z \= Y, X \= Z,
6        once( fasill_similarity(X/Arity,Z/Arity,TDxz,Sxz)
7              ; (TDxz = Bot, Sxz = false) ),
8        once(fasill_similarity(X/Arity,Y/Arity,TDxy,Sxy)),
9        once(fasill_similarity(Y/Arity,Z/Arity,TDyz,Syz))
10     ), (
11       (TDxy == Top -> TDy = TDyz, Sy = Syz ;
12         (TDyz == Top -> TDy = TDxy, Sy = Sxy ;
13           ((Sxy == false, Syz == false, Tnorm = '&'(_)) ->
14             ( lattice_call_connective(Tnorm, [TDxy,TDyz], TDy),
15               Sy = false)
16           ; ( TDy = term(Tnorm, [TDxy, TDyz]),
17             Sy = true )
18         )
19       )
20     ),
21     (TDxz == Bot -> TD = TDy, S = Sy ;
22       (TDy == Bot -> TD = TDxz, S = Sxz ;
23         ((Sxz == false, Sy == false, Tnorm = '&'(_)) ->
24           (lattice_call_supremum(TDxz, TDy, TD), S = false) ;
25           (TD = sup(TDxz, TDy), S = true)
26         )
27       )
28     ),
29     retractall(fasill_similarity(X/Arity,Z/Arity,_,_)),

```

```

30     assertz(fasill_similarity(X/Arity,Z/Arity,TD,S))
31     )
32 ).

```

Semántica operacional simbólica

El predicado `is_fuzzy_computed_answer/1` tiene éxito cuando el término FASILL recibido es una respuesta computada difusa (posiblemente simbólica). Cuando la extensión simbólica está desactivada, basta con comprobar que el término es un grado de verdad del retículo. En caso contrario, se comprueba que la expresión es interpretable (esto es, que no contiene ningún átomo), pero que no es posible seleccionar ninguna subexpresión evaluable.

```

1  is_fuzzy_computed_answer(X) :-
2      current_fasill_flag(symbolic, term(false, [])), !,
3      lattice_call_member(X).
4  is_fuzzy_computed_answer(X) :-
5      interpretable(X),
6      \+select_expression(X, _, _, _).

```

Por otro lado, es necesario modificar la función de selección para que también sea posible resolver las fórmulas atómicas que aparecen dentro de las nuevas conectivas simbólicas. El predicado `select_atom/4` implementa la función de selección de FASILL. El objetivo `select_atom(Expr, ExprVar, Var, Atom)` tiene éxito cuando `Atom` es el átomo seleccionado dentro de la expresión `Expr`, donde `ExprVar` es la expresión `Expr` en la que se ha reemplazado el átomo seleccionado por `Var`. En particular, la cláusula modificada es la siguiente.

```

1  select_atom(term(Term, Args), term(Term, ArgsVar), Var, Atom) :-
2      functor(Term, Op, _),
3      member(Op, ['@', '&', '|', '#@', '#&', '#|', '#?']), !,
4      select_atom(Args, ArgsVar, Var, Atom).

```

Por último, con el fin de permitir la aplicación de sustituciones simbólicas a programas SFASILL sin necesidad de modificar las reglas o la relación de similitud, se ha añadido una nueva bandera denominada `symbolic_substitution`, cuyo valor puede ser modificado mediante la directiva `set_fasill_flag`. Una sustitución simbólica se codifica como una lista de pares, donde las claves son constantes simbólicas y los valores son elementos o conectivas del retículo. Por defecto, la sustitución simbólica almacenada es la sustitución identidad, representada como una lista vacía.

```

1  select_expression(term('#'(S),[]), Var, Var, term('#',S)) :-
2      !,
3      current_fasill_flag(symbolic_substitution, Sub),

```

```

4     fasill_member(term('-', [term('#'(S),[]), _]), Sub).
5 select_expression(term(Term,Args), Var, Var, term(Term,Args)) :-
6     functor(Term, Op, _),
7     once(member(Op, ['#@', '#&', '#|', '#?'])),
8     maplist(lattice_call_member, Args),
9     current_fasill_flag(symbolic_substitution, Sub),
10    fasill_member(term('-', [term(Term,[]), term(_,[])]), Sub),
11    !.

```

Cuando FASILL selecciona una L -expresión a interpretar y se encuentra con una constante simbólica, comprueba si esta aparece en el dominio de la sustitución simbólica almacenada en la bandera `symbolic_substitution`. Si es así, selecciona dicha expresión y la interpreta; si no, la ignora y busca la siguiente expresión interpretable.

```

1 interpret(term('#'(S), []), Value) :-
2     current_fasill_flag(symbolic_substitution, Sub),
3     fasill_member(term('-', [term('#'(S), []), Value]), Sub),
4     !.
5 interpret(term(Term,Args), Result) :-
6     functor(Term, Op, _),
7     once(member(Op, ['#@', '#&', '#|', '#?'])),
8     current_fasill_flag(symbolic_substitution, Sub),
9     fasill_member(term('-', [term(Term,[]), term(Value,[])]), Sub),
10    interpret(term(Value,Args), Result),
11    !.

```

De esta forma, cuando se utiliza la directiva

```
set_fasill_flag(symbolic_substitution, Sub)
```

en un programa simbólico, el sistema FASILL es capaz de interpretar las constantes simbólicas que pertenecen al dominio de la sustitución simbólica `Sub` por medio de pasos interpretativos.

Ejemplo 4.8. Considérese el siguiente programa simbólico que define el predicado `short_list/1`, que recibe una lista y tiene éxito con un grado de verdad mayor cuanto más corta es la lista.

```

1 <- set_fasill_flag(
2     symbolic_substitution, [(#&s1)-(&prod), #s2-0.9]).
3
4 short_list([]).
5 short_list([_|T]) <- #&s1(#s2, short_list(T)).

```

Entonces, para un objetivo $\mathcal{G} = \text{short_list}([\dots])$ el sistema FASILL aplica automáticamente la sustitución simbólica $\Theta = \{\&_{s1}^{\#}/\&_{prod}, s_2^{\#}/0.9\}$ en la derivación

(tras resolver todos los átomos). Por ejemplo:

$$\begin{aligned}
\mathcal{D} : \langle \underline{short_list}([_, _]), id \rangle & \rightsquigarrow_{SS}^{R_2} \\
\langle (s_2^\# \&_{s_1}^\# \underline{short_list}([_])), id \rangle & \rightsquigarrow_{SS}^{R_2} \\
\langle (s_2^\# \&_{s_1}^\# (s_2^\# \&_{s_1}^\# \underline{short_list}([_])), id \rangle & \rightsquigarrow_{SS}^{R_1} \\
\langle (s_2^\# \&_{s_1}^\# (s_2^\# \&_{s_1}^\# 1.0)), id \rangle & \rightsquigarrow_{IS} \\
\langle (0.9 \&_{s_1}^\# (s_2^\# \&_{s_1}^\# 1.0)), id \rangle & \rightsquigarrow_{IS} \\
\langle (0.9 \&_{s_1}^\# \underline{(0.9 \&_{s_1}^\# 1.0)}), id \rangle & \rightsquigarrow_{IS} \\
\langle (0.9 \&_{s_1}^\# 0.9), id \rangle & \rightsquigarrow_{IS} \\
\langle 0.81, id \rangle &
\end{aligned}$$

4.6. Conclusiones

En este capítulo hemos introducido una extensión simbólica del lenguaje FASILL que es necesaria como preámbulo al diseño de técnicas automáticas de calibrado de programas lógicos difusos. En particular, las aportaciones de este capítulo son las siguientes.

- (1) Hemos descrito la extensión simbólica de FASILL, denominada sFASILL, que permite introducir en las reglas y relaciones de similitud de los programas, valores y conectivas simbólicas que no pertenecen al retículo asociado a los mismos.
- (2) Hemos introducido la noción de sustitución simbólica, que asigna a cada constante simbólica del programa un valor o conectiva del retículo. A partir de este concepto, hemos definido las relaciones de similitud simbólicas, que permiten que elementos del dominio se relacionen con un grado de similitud simbólico. Cuando se aplica una sustitución simbólica sobre una relación de similitud simbólica, esta se vuelve una relación de similitud clásica.
- (3) Hemos diseñado un algoritmo de cierre simbólico inspirado en [JI08] que, a partir de un esquema de similitud simbólico, computa una relación de similitud simbólica válida.
- (4) Hemos extendido el algoritmo de unificación débil para tratar con relaciones de similitud simbólicas, permitiendo que símbolos que se relacionan con un grado de similitud simbólico unifiquen débilmente. Además, hemos adaptado la semántica operacional de FASILL para postergar la interpretación de las constantes simbólicas hasta que su valor es conocido.
- (5) Hemos demostrado que, cuando un programa sFASILL no contiene constantes simbólicas en la relación de similitud, el programa simbólico lleva a las mismas respuestas computadas difusas si: *i*) aplicamos una sustitución

simbólica al programa y entonces calculamos la derivación; o si *ii*) calculamos la respuesta computada difusa simbólica, aplicamos la sustitución simbólica a dicha s.f.c.a. y calculamos los últimos pasos interpretativos.

- (6) Cuando el programa sFASILL contiene constantes simbólicas en la relación de similitud, hemos visto que no todas las sustituciones simbólicas satisfacen esta propiedad. Por lo tanto, hemos caracterizado las sustituciones simbólicas que son seguras respecto a una relación de similitud simbólica, y hemos demostrado que para estas sustituciones simbólicas sí se preservan las respuestas computadas difusas. Además, hemos simplificado esta condición de seguridad a nivel práctico mediante la noción de *conjunto de grados de unificación* de una derivación.
- (7) Hemos implementado la extensión simbólica en el sistema FASILL con el fin de introducir constantes simbólicas tanto en las reglas de los programas como en sus relaciones de similitud, calcular respuestas computadas difusas simbólicas para estos programas simbólicos y aplicar sustituciones simbólicas a los mismos.

Como veremos en el siguiente capítulo, las demostraciones sobre las propiedades de la semántica operacional de la extensión simbólica de FASILL resultan de vital importancia para el diseño de algoritmos eficientes de calibrado de programas sFASILL.

Capítulo 5

Calibrado de programas lógicos difusos

Introducir cambios en alguna de las componentes difusas de un programa (los grados de verdad, las conectivas difusas o la relación de similitud) puede afectar –a veces de manera inesperada– a las respuestas computadas difusas de un objetivo dado. Normalmente, el programador tiene un modelo en mente en el que el valor de algunos parámetros está claro. Por ejemplo, en ocasiones el grado de verdad de un hecho puede ser determinado estadística o experimentalmente. Sin embargo, en otros casos los valores de verdad o las conectivas más apropiadas dependen de nociones subjetivas y el programador no es capaz de deducir dichos valores. En un escenario típico se dispone de un extenso conjunto de respuestas esperadas (casos de prueba) con los que el programador puede seguir una estrategia de modificar valores y conectivas en el programa, y comparar los resultados de las respuestas computadas difusas hasta dar con el modelo que mejor se aproxime a los casos de prueba. Desafortunadamente, esta operación es tediosa y puede consumir mucho tiempo. Incluso, puede ser impracticable cuando el programa es complejo o debe modelar correctamente un largo número de casos de prueba.

En este capítulo discutimos diversas técnicas automáticas para calibrar programas lógicos difusos integrados utilizando la extensión simbólica del lenguaje FASILL, siguiendo la línea de nuestros trabajos previos sobre calibrado de programas MALP [MPRV17, MR17, RM20, MR20] y, más recientemente, programas FASILL [MR21]. Además, presentamos una serie de casos de uso que relacionan el calibrado con otros campos de aplicación de interés como la verificación de la equivalencia de circuitos combinatoriales, el aprendizaje automático o la web semántica.

5.1. Motivación y antecedentes

Si bien no encontramos en la literatura ninguna publicación previa a nuestro trabajo [MPRV17] que aborde el problema del calibrado de programas (y, en particular, de programas lógicos difusos) de forma general, sí podemos establecer algunas conexiones con otros campos afines como son el aprendizaje supervisado o la optimización matemática. En particular:

- El aprendizaje supervisado [Rus10] es una técnica de aprendizaje automático que permite deducir una función a partir de un conjunto de datos de entrenamiento. La función inferida puede ser utilizada posteriormente para predecir el valor ante nuevas entradas. Para ello, se espera que el modelo producido por el algoritmo de aprendizaje sea capaz de generalizar a partir del conjunto de datos utilizado en el proceso de entrenamiento. Algunos de los modelos de aprendizaje supervisado más utilizados son la regresión lineal [MPV21], los árboles de decisión [RM05], las redes neuronales [Bis94] o los clasificadores bayesianos ingenuos [HY01]. En esta línea, el calibrado de programas lógicos difusos puede ser visto como una técnica de aprendizaje supervisado, en el sentido de que permite deducir un programa a partir de un conjunto de datos de entrenamiento (los casos de prueba). De hecho, como veremos al final del capítulo, es posible codificar como programas sFASILL algunos modelos clásicos de aprendizaje supervisado y entrenar dichos modelos mediante las técnicas de calibrado desarrolladas en esta tesis.
- Por otro lado, la optimización matemática [SW18] es el proceso de formalización y búsqueda de la mejor solución (respecto a un criterio determinado) en un espacio de búsqueda dado. Esto es, dada una función objetivo $f : A \rightarrow \mathbb{R}$, un problema de optimización consiste en encontrar un $x_0 \in A$ tal que $f(x_0) \leq f(x)$ para todo $x \in A$. Aquí se incluyen campos como la programación lineal [V⁺20], la programación entera [Wol20] o la programación no lineal [Ber97]. En esencia, calibrar un programa simbólico $\mathcal{P}^\#$ consiste en buscar automáticamente una sustitución simbólica Θ que minimice la distancia entre (1) las respuestas computadas difusas producidas en $\mathcal{P}^\#\Theta$ para los objetivos de un conjunto de casos de prueba y (2) los valores esperados para dichos casos de prueba. Tal y como veremos en la sección 5.4, el problema de calibrado puede ser reducido a un problema de optimización una vez se han calculado las respuestas computadas difusas simbólicas para los objetivos de los casos de prueba.

Por lo tanto, las técnicas de calibrado de programas lógicos difusos que proponemos a continuación permiten al sistema FASILL actuar como una capa de abstracción en la formulación y resolución de problemas de satisficibilidad y optimización. Esto proporciona al programador un marco de programación de alto nivel mucho más rico y expresivo que otras herramientas específicas como LINGO [XX05] o Z3 [MB08] (del que hablaremos después con más detalle) para codificar este tipo de problemas.

5.2. Introducción

Comenzamos esta sección formalizando el concepto de espacio métrico, esto es, un conjunto sobre el que se define una noción de distancia entre todo par de elementos del mismo. Nótese que el dominio del conjunto no tiene por qué ser necesariamente numérico.

Definición 5.1 (Espacio métrico, [AP90]). Un *espacio métrico* es un par (M, d) donde M es un conjunto y d es una distancia sobre M , es decir, una función $d : M \times M \rightarrow \mathbb{R}$ tal que para todo $x, y, z \in M$ se verifican las siguientes propiedades:

- (1) $d(x, y) \geq 0$, y en particular $d(x, y) = 0$ si y solo si $x = y$;
- (2) $d(x, y) = d(y, x)$;
- (3) $d(x, z) \leq d(x, y) + d(y, z)$ (desigualdad triangular).

En adelante, exigiremos que todo retículo L asociado a un programa simbólico a calibrar conforme un espacio métrico mediante la definición de una noción de distancia sobre L . Denotaremos por $d(x, y)$ a la distancia entre dos elementos del retículo.

Ejemplo 5.1. La distancia más común en \mathbb{R} es dada por el valor absoluto de la resta entre dos números, es decir, por $d(x, y) = |x - y|$, y se denomina distancia usual. Esta es la noción de distancia que utilizaremos para el retículo $([0, 1], \leq)$ considerado a lo largo de la memoria. No obstante, hay infinidad de distancias posibles, como la distancia discreta, definida como:

$$d(x, y) = \begin{cases} 0 & \text{si } x = y \\ 1 & \text{si } x \neq y \end{cases}$$

Ahora estamos preparados para precisar el problema del calibrado de programas lógicos difusos. Por simplicidad, consideraremos únicamente la primera respuesta computada difusa de un objetivo. Nótese que esta no es una restricción significativa, dado que es posible codificar múltiples soluciones en una lista para la que el objetivo principal es siempre determinista y todas las llamadas no deterministas son ocultadas en la computación. Extender las siguientes definiciones y algoritmos para múltiples soluciones no es difícil, pero hace la formalización más incómoda.

Definición 5.2 (Caso de prueba). Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ un programa sFASILL. Un *caso de prueba* es un par $\langle \mathcal{G}, v \rangle$, donde \mathcal{G} es un objetivo y $v \in L$ es el grado de verdad que se desea obtener asociado a la respuesta computada difusa (simbólica) de \mathcal{G} en $\mathcal{P}^\#$.

Definición 5.3 (Calibrado de programas simbólicos). Sea $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ un programa sFASILL y $T = \{ \langle \mathcal{G}_1, v_1 \rangle, \dots, \langle \mathcal{G}_n, v_n \rangle \}$ un conjunto de casos de prueba. El *calibrado* del programa $\mathcal{P}^\#$ respecto a T consiste en la búsqueda de

una sustitución simbólica Θ de $\text{sym}(\mathcal{P}^\#) \cup \bigcup_{i=1}^n \text{sym}(\mathcal{G}_i)$ que minimiza la suma de las distancias entre las respuestas computadas difusas producidas en $\mathcal{P}^\#\Theta$ para los objetivos $\mathcal{G}_1\Theta, \dots, \mathcal{G}_n\Theta$ y los valores esperados v_1, \dots, v_n ; esto es:

$$\arg \min_{\Theta} \sum_{i=1}^n d(v_i, r_i) : \langle \mathcal{G}_i\Theta, id \rangle \rightsquigarrow^* \langle r_i, \sigma_i \rangle \text{ en } \mathcal{P}^\#\Theta, r_i \in L$$

Se observa aquí la necesidad de que el retículo asociado al programa simbólico conforme un espacio métrico que nos permita calcular la distancia entre (1) el grado de verdad asociado a la respuesta computada difusa del objetivo de un caso de prueba y (2) el grado de verdad esperado para el mismo.

Ejemplo 5.2. Sea $\mathcal{P}^\#$ el programa simbólico mostrado en el ejemplo 4.3. Supongamos que queremos calibrar el programa $\mathcal{P}^\#$ con respecto a los siguientes casos de prueba:

$$\begin{aligned} t_1 &= \langle \text{good_hotel}(\text{hydropolis}), 0.7 \rangle; \\ t_2 &= \langle \text{close}(\text{atlantis}, \text{bus}), 0.3 \rangle. \end{aligned}$$

Es decir, tras el proceso de calibrado, esperamos que el programa FASILL resultante produzca una respuesta computada difusa con un grado de verdad asociado lo más cercano posible a 0.7 para el objetivo $\text{good_hotel}(\text{hydropolis})$, y una respuesta computada difusa con un grado de verdad asociado próximo a 0.3 para el objetivo $\text{close}(\text{atlantis}, \text{bus})$.

Es importante destacar que la definición 5.3 no hace mención explícita a la generación del espacio de búsqueda de las sustituciones simbólicas en un problema de calibrado. Esto es así porque, como veremos a continuación, cada algoritmo de calibrado que proponemos –cuyo método de búsqueda puede estar basado en el sistema FASILL o en un resolutor de satisfacibilidad– genera las sustituciones simbólicas candidatas de forma diferente.

5.3. Calibrado de programas en el sistema FASILL

En esta sección introducimos los algoritmos de calibrado que delegan la búsqueda de la mejor sustitución simbólica al sistema FASILL. La precisión de estos algoritmos de calibrado puede ser parametrizada en función del conjunto de sustituciones simbólicas considerado. Por ejemplo, para el retículo $([0, 1], \leq)$ podemos reemplazar los valores simbólicos por el conjunto de valores $\{0.0, 0.5, 1.0\}$ o por el conjunto con más elementos $\{0.0, 0.1, 0.2, \dots, 0.8, 0.9, 1.0\}$. Lo mismo ocurre con las conectivas. El sistema FASILL genera automáticamente todas las posibles combinaciones de los valores y conectivas considerados para las constantes simbólicas que aparecen en el programa y en los objetivos de los casos de prueba. Obviamente, cuanto mayor es el dominio de valores y conectivas, más preciso es el resultado, pero más extensa y costosa es la búsqueda.

La versión más ingenua de la técnica de calibrado descrita en el algoritmo 5.1, a la que denominamos *calibrado básico*, aplica cada una de las posibles

sustituciones simbólicas Θ_i sobre el programa simbólico $\mathcal{P}^\#$. Entonces, para cada programa FASILL $\mathcal{P}^\#\Theta_i$ obtenido se calculan las respuestas computadas difusas para los objetivos de los casos de prueba, y se computa la distancia entre los valores de verdad obtenidos y los esperados. Así, el sistema es capaz de encontrar la sustitución simbólica que menos se desvía de los casos de prueba introducidos por el usuario.

Algoritmo 5.1: Calibrado básico de programas lógicos difusos.

Datos: Un programa simbólico $\mathcal{P}^\#$ y un conjunto de casos de prueba

$\langle \mathcal{G}_i, v_i \rangle$, con $1 \leq i \leq k$.

Resultado: Una sustitución simbólica Θ .

Considerar un número finito de sustituciones simbólicas para

$sym(\mathcal{P}^\#) \cup \bigcup_{i=1}^k sym(\mathcal{G}_i)$, denotadas por $\Theta_1, \dots, \Theta_n$;

$\tau \leftarrow +\infty$;

para cada $i \in \{1, \dots, n\}$ **hacer**

$\mathcal{P}_i \leftarrow \mathcal{P}^\#\Theta_i$;

$\epsilon \leftarrow 0$;

para cada $j \in \{1, \dots, k\}$ **hacer**

 Computar la f.c.a. $\langle \mathcal{G}_j\Theta_i, id \rangle \rightsquigarrow^* \langle v_{i,j}, \theta_{i,j} \rangle$ en \mathcal{P}_i ;

$\epsilon \leftarrow \epsilon + d(v_{i,j}, v_j)$;

si $\epsilon \geq \tau$ **entonces**

salir;

fin

fin

si $\epsilon < \tau$ **entonces**

$\tau \leftarrow \epsilon$;

$\Theta \leftarrow \Theta_i$;

fin

fin

devolver Θ ;

Este algoritmo, y todos los sucesivos, incluyen una técnica de umbralización (haciendo uso del filtro o umbral τ que se actualiza dinámicamente) para descartar prematuramente sustituciones simbólicas cuya desviación es mayor que la desviación de la mejor sustitución simbólica encontrada hasta el momento.

Ejemplo 5.3. Considérese el problema de calibrado mostrado en el ejemplo 5.2. El programa simbólico $\mathcal{P}^\#$ contiene 4 constantes simbólicas. Por simplicidad, tomaremos únicamente los valores $\{0.3, 0.8\}$ como posibles valores del retículo para $s_1^\#$ y $s_3^\#$, las conjunciones $\{\&_{godel}, \&_{luka}, \&_{prod}\}$ para $\&_{s_2}^\#$ y los agregadores $\{@_{aver}, @_{geom}\}$ para $@_{s_4}^\#$. En la siguiente tabla se muestran los grados de verdad asociados a las f.c.a. de los casos de prueba t_1 y t_2 , junto con sus desviaciones ϵ_1 y ϵ_2 con respecto de los valores esperados, para cada posible sustitución simbólica. Los valores anotados con asterisco (*) no son computados en tiempo

de calibrado debido a la umbralización.

Θ	$s_1^\#$	$\&_{s_2}^\#$	$s_3^\#$	$@_{s_4}^\#$	t_1	ϵ_1	t_2	ϵ_2	ϵ_{total}
Θ_1	0.3	$\&_{godel}$	0.3	$@_{aver}$	0.395	0.305	0.3	0.0	0.305
Θ_2	0.3	$\&_{godel}$	0.3	$@_{geom}$	0.3834	0.3166	0.3*	0.0*	0.3166*
Θ_3	0.3	$\&_{godel}$	0.8	$@_{aver}$	0.645	0.055	0.3	0.0	0.055
Θ_4	0.3	$\&_{godel}$	0.8	$@_{geom}$	0.626	0.074	0.3*	0.0*	0.074*
Θ_5	0.3	$\&_{luka}$	0.3	$@_{aver}$	0.395	0.305	0.0*	0.3*	0.605*
Θ_6	0.3	$\&_{luka}$	0.3	$@_{geom}$	0.3834	0.3166	0.0*	0.3*	0.6166*
Θ_7	0.3	$\&_{luka}$	0.8	$@_{aver}$	0.645	0.055	0.0*	0.3*	0.355*
Θ_8	0.3	$\&_{luka}$	0.8	$@_{geom}$	0.626	0.074	0.0*	0.3*	0.374*
Θ_9	0.3	$\&_{prod}$	0.3	$@_{aver}$	0.395	0.305	0.081*	0.219*	0.524*
Θ_{10}	0.3	$\&_{prod}$	0.3	$@_{geom}$	0.3834	0.3166	0.081*	0.219*	0.5356*
Θ_{11}	0.3	$\&_{prod}$	0.8	$@_{aver}$	0.645	0.055	0.081*	0.219*	0.274*
Θ_{12}	0.3	$\&_{prod}$	0.8	$@_{geom}$	0.626	0.074	0.081*	0.219*	0.293*
Θ_{13}	0.8	$\&_{godel}$	0.3	$@_{aver}$	0.395	0.305	0.5*	0.2*	0.505*
Θ_{14}	0.8	$\&_{godel}$	0.3	$@_{geom}$	0.3834	0.3166	0.5*	0.2*	0.5166*
Θ_{15}	0.8	$\&_{godel}$	0.8	$@_{aver}$	0.645	0.055	0.5*	0.2*	0.255*
Θ_{16}	0.8	$\&_{godel}$	0.8	$@_{geom}$	0.626	0.074	0.5*	0.2*	0.274*
Θ_{17}	0.8	$\&_{luka}$	0.3	$@_{aver}$	0.395	0.305	0.0*	0.3*	0.605*
Θ_{18}	0.8	$\&_{luka}$	0.3	$@_{geom}$	0.3834	0.3166	0.0*	0.3*	0.6166*
Θ_{19}	0.8	$\&_{luka}$	0.8	$@_{aver}$	0.645	0.055	0.0*	0.3*	0.355*
Θ_{20}	0.8	$\&_{luka}$	0.8	$@_{geom}$	0.626	0.074	0.0*	0.3*	0.374*
Θ_{21}	0.8	$\&_{prod}$	0.3	$@_{aver}$	0.395	0.305	0.216*	0.084*	0.389*
Θ_{22}	0.8	$\&_{prod}$	0.3	$@_{geom}$	0.3834	0.3166	0.216*	0.084*	0.4006*
Θ_{23}	0.8	$\&_{prod}$	0.8	$@_{aver}$	0.645	0.055	0.216*	0.084*	0.139*
Θ_{24}	0.8	$\&_{prod}$	0.8	$@_{geom}$	0.626	0.074	0.216*	0.084*	0.158*

Como se observa en la tabla, para este problema de calibrado el sistema FASILL genera $2^3 \cdot 3 = 24$ sustituciones simbólicas, y la mejor sustitución simbólica para los casos de prueba t_1 y t_2 es $\Theta_3 = \{s_1^\#/0.3, \&_{s_2}^\#/\&_{godel}, s_3^\#/0.8, @_{s_4}^\#/@_{aver}\}$ con un error absoluto de 0.055.

Esta aproximación no hace uso de la semántica operacional simbólica, ya que cada sustitución simbólica se aplica al programa y al objetivo antes de computar cualquier derivación. Esto tiene varios inconvenientes en el rendimiento del proceso de calibrado. El primero y más inmediato es que los pasos de éxito y de fallo realizados en todas las derivaciones para un mismo objetivo son idénticos, indistintamente de la sustitución simbólica aplicada (tal y como establecemos en los teoremas 4.1, 4.2 y 4.3). Así que si consideramos n sustituciones simbólicas para k casos de prueba, FASILL estaría computando $(n - 1)k$ derivaciones admisibles (es decir, basadas únicamente en pasos de éxito y fallo, que no interpretativos) repetidas que no son realmente necesarias. El algoritmo 5.2, al que denominamos *calibrado simbólico*, soluciona este problema al calcular previamente las respuestas computadas difusas simbólicas para cada objetivo de los casos de prueba. Entonces, para cada sustitución simbólica, esta se aplica a cada una de las s.f.c.a. y se computa la f.c.a. correspondiente aplicando los últimos pasos interpretativos. Como mencionamos en la sección 4.4, si la relación de similitud asociada al programa contiene constantes simbólicas, es necesario

asegurarnos de que las sustituciones simbólicas son seguras antes de proceder.

Algoritmo 5.2: Calibrado simbólico de programas lógicos difusos.

Datos: Un programa simbólico $\mathcal{P}^\#$ y un conjunto de casos de prueba $\langle \mathcal{G}_i, v_i \rangle$, con $1 \leq i \leq k$.

Resultado: Una sustitución simbólica Θ .

para cada $i \in \{1, \dots, k\}$ **hacer**

 | Computar la s.f.c.a. $\langle \mathcal{G}_i, id \rangle \rightsquigarrow^* \langle \mathcal{G}'_i, \theta_i \rangle$ en $\mathcal{P}^\#$;

fin

Considerar un número finito de sustituciones simbólicas para

$\bigcup_{i=1}^k \text{sym}(\mathcal{G}'_i)$, denotadas por $\Theta_1, \dots, \Theta_n$;

$\tau \leftarrow +\infty$;

para cada $i \in \{1, \dots, n\}$ **hacer**

si Θ_i es segura **entonces**

$\epsilon \leftarrow 0$;

para cada $j \in \{1, \dots, k\}$ **hacer**

 | Computar la f.c.a. $\langle \mathcal{G}'_j \Theta_i, \theta_j \rangle \rightsquigarrow_{IS}^* \langle v_{i,j}, \theta_j \rangle$ en $\mathcal{P}^\#$;

$\epsilon \leftarrow \epsilon + d(v_{i,j}, v_j)$;

si $\epsilon \geq \tau$ **entonces**

 | salir;

fin

fin

si $\epsilon < \tau$ **entonces**

$\tau \leftarrow \epsilon$;

$\Theta \leftarrow \Theta_i$;

fin

fin

fin

devolver Θ ;

En la práctica, por el teorema 4.3, para considerar que una sustitución simbólica es segura FASILL genera un conjunto de *precondiciones* a partir de los conjuntos de grados de unificación de las derivaciones de las s.f.c.a. de los casos de prueba. Básicamente, una precondición es una $L^\#$ -expresión (que también puede entenderse como un objetivo sin átomos) que debe ser evaluada a un grado de verdad mayor que \perp tras aplicarse una sustitución simbólica.¹ Si una sustitución simbólica no satisface todas las precondiciones, es descartada.

Ejemplo 5.4. Dado el problema de calibrado mostrado en el ejemplo 5.2, las respuestas computadas difusas simbólicas para los casos de prueba t_1 y t_2 son

¹Más generalmente, una precondición en el sistema FASILL puede ser un objetivo cualquiera (con o sin átomos) que debe llevar a una respuesta computada difusa con grado de verdad asociado mayor que \perp . Sin embargo, como veremos más adelante, el calibrado basado en SMT sí restringe las precondiciones a $L^\#$ -expresiones (que, obviamente, nunca contienen átomos).

las siguientes:

$$\begin{aligned}
 \mathcal{D}_1^{\#} : & \langle \underline{good_hotel(hydropolis)}, id \rangle && \rightsquigarrow_{SS}^{R_4} \\
 & \langle @_{s_4}^{\#}(\underline{@_{very}(close(hydropolis), y_1)}), cheap(y_1), \{x/hydropolis, y/y_1\} \rangle && \rightsquigarrow_{SS}^{R_2} \\
 & \langle @_{s_4}^{\#}(\underline{@_{very}(0.7), cheap(taxi)}), \{x/hydropolis, y/taxi\} \rangle && \rightsquigarrow_{SS}^{R_1} \\
 & \langle @_{s_4}^{\#}(\underline{@_{very}(0.7), s_3^{\#}}), \{x/hydropolis, y/taxi\} \rangle && \rightsquigarrow_{IS} \\
 & \langle @_{s_4}^{\#}(0.49, s_3^{\#}), \{x/hydropolis, y/taxi\} \rangle && \\
 \\
 \mathcal{D}_2^{\#} : & \langle \underline{close(atlantis, bus)}, id \rangle && \rightsquigarrow_{SS}^{R_3} \\
 & \langle ((0.6 \&_{s_2}^{\#} (0.5 \&_{s_2}^{\#} s_1^{\#})) \&_{s_2}^{\#} 0.9), id \rangle &&
 \end{aligned}$$

De estas derivaciones se extrae una única precondition a partir de los conjuntos de grados de unificación de $\mathcal{D}_1^{\#}$ y $\mathcal{D}_2^{\#}$, concretamente, al explotar el átomo $close(atlantis, bus)$ con R_3 en la segunda derivación:

$$\text{deg}(close(atlantis, bus), close(ritz, metro)) = (0.6 \&_{s_2}^{\#} (0.5 \&_{s_2}^{\#} s_1^{\#})).$$

Por lo tanto, para que una sustitución simbólica Θ sea considerada segura debe verificar que $\vartheta_L((0.6 \&_{s_2}^{\#} (0.5 \&_{s_2}^{\#} s_1^{\#}))\Theta) > 0$. En este caso, las sustituciones simbólicas que reemplazan la constante simbólica $\&_{s_2}^{\#}$ por la t-norma de Łukasiewicz ($\Theta_5, \Theta_6, \Theta_7, \Theta_8, \Theta_{17}, \Theta_{18}, \Theta_{19}$ y Θ_{20}) no satisfacen esta precondition y son descartadas. Por lo tanto, la mejor sustitución simbólica encontrada con el método simbólico sigue siendo $\Theta_3 = \{s_1^{\#}/0.3, \&_{s_2}^{\#}/\&_{godel}, s_3^{\#}/0.8, @_{s_4}^{\#}/@_{aver}\}$.

Además de calcular menos derivaciones, otra ventaja del algoritmo de calibrado simbólico es que este permite dividir automáticamente los casos de prueba y las constantes simbólicas en varios conjuntos disjuntos en función de las constantes simbólicas que ocurren en las s.f.c.a. de los casos de prueba, reduciendo el problema combinatorio de calibrado en varios sub-problemas de calibrado más pequeños (con menos casos de prueba y constantes simbólicas). En particular, reducir el número de constantes simbólicas disminuye exponencialmente el número de sustituciones simbólicas a considerar. Supongamos un problema de calibrado en el que hay m constantes simbólicas distintas en las respuestas computadas difusas simbólicas (calculadas a partir de los objetivos de k casos de prueba) y donde cada una de ellas puede ser reemplazada por p valores distintos. Entonces, FASILL genera p^m sustituciones simbólicas que debe comprobar para los k casos de prueba y, por lo tanto, la complejidad es $\mathcal{O}(kp^m)$. Supongamos ahora que podemos separar los k casos de prueba y las m constantes simbólicas en j conjuntos disjuntos, donde cada conjunto contiene $\frac{k}{j}$ casos de prueba cuyas s.f.c.a. solo contienen las $\frac{m}{j}$ constantes simbólicas de su propio conjunto. Entonces, la complejidad del calibrado se reduce a $\mathcal{O}(kp^{m/j})$.²

²Nótese que este es un caso hipotético en el que tanto los casos de prueba como las constantes simbólicas se reparten uniformemente en varios conjuntos disjuntos con el fin de simplificar los cálculos. En los problemas reales de calibrado, cada conjunto disjunto suele contener un número distinto y variable de casos de prueba y de constantes simbólicas.

El siguiente algoritmo de calibrado, al que denominamos *calibrado simbólico disjunto*, contempla esta simplificación con el fin de calibrar cada conjunto disjunto de constantes simbólicas por separado.

Algoritmo 5.3: Calibrado simbólico disjunto de programas lógicos difusos.

Datos: Un programa simbólico $\mathcal{P}^\#$ y un conjunto de casos de prueba $\langle \mathcal{G}_i, v_i \rangle$, con $1 \leq i \leq k$.

Resultado: Una sustitución simbólica Θ .

$\mathcal{C} \leftarrow \emptyset$;

para cada $i \in \{1, \dots, k\}$ **hacer**

 Computar la s.f.c.a. $\langle \mathcal{G}_i, id \rangle \rightsquigarrow^* \langle \mathcal{G}'_i, \theta_i \rangle$ en $\mathcal{P}^\#$;

 Obtener las constantes simbólicas $sym(\mathcal{G}'_i) = \{s_1^\#, \dots, s_l^\#\}$;

para cada $s^\# \in sym(\mathcal{G}'_i)$ **hacer**

si $\nexists (S, T) \in \mathcal{C} : s^\# \in S$ **entonces**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{(\{s^\#\}, \emptyset)\}$;

fin

fin

para cada $j \in \{1, \dots, l-1\}$ **hacer**

$A \leftarrow (S_a, T_a) \in \mathcal{C} : s_j^\# \in S_a$;

$B \leftarrow (S_b, T_b) \in \mathcal{C} : s_{j+1}^\# \in S_b$;

$\mathcal{C} \leftarrow ((\mathcal{C} - A) - B) \cup \{(S_a \cup S_b, T_a \cup T_b)\}$;

fin

$A \leftarrow (S, T) \in \mathcal{C} : s_l^\# \in S$;

$\mathcal{C} \leftarrow (\mathcal{C} - A) \cup \{(S, T \cup \{\langle \mathcal{G}_i, v_i \rangle\})\}$;

fin

$\Theta = \emptyset$;

para cada $(S, T) \in \mathcal{C}$ **hacer**

$\Theta' \leftarrow$ Calibrar el programa $\mathcal{P}^\#$ para los casos de prueba T
 aplicando el algoritmo 5.2;

$\Theta \leftarrow \Theta \cup \Theta'$;

fin

devolver Θ ;

Ejemplo 5.5. El problema de calibrado considerado en el ejemplo 5.1 puede dividirse en dos subproblemas. Por un lado, las constantes simbólicas $s_1^\#$ y $\&_{s_2}^\#$ que ocurren en la derivación del caso de prueba t_1 , y por otro lado las constantes simbólicas $s_3^\#$ y $@_{s_4}^\#$ que ocurren en la derivación del caso de prueba t_2 . En la siguiente tabla se muestra el grado de verdad asociado a la f.c.a. del caso de prueba t_1 junto con su desviación ϵ_1 con respecto del valor esperado, para cada

posible sustitución simbólica.

Θ_1	$s_3^\#$	$@_{s_4}^\#$	t_1	ϵ_1	ϵ_{total}
$\Theta_{1,1}$	0.3	$@_{aver}$	0.395	0.305	0.305
$\Theta_{1,2}$	0.3	$@_{geom}$	0.3834	0.3166	0.3166
$\Theta_{1,3}$	0.8	$@_{aver}$	0.645	0.055	0.055
$\Theta_{1,4}$	0.8	$@_{geom}$	0.626	0.074	0.074

Para este problema de calibrado, FASILL genera $2^2 = 4$ sustituciones simbólicas. No hay ninguna precondition, y por lo tanto todas las sustituciones son seguras. La mejor sustitución simbólica para el caso de prueba t_1 es $\Theta_{1,3} = \{s_3^\#/0.8, @_{s_4}^\#/@_{aver}\}$ con un error absoluto de 0.055. En la siguiente tabla se muestra el grado de verdad asociado a la f.c.a. del caso de prueba t_2 junto con su desviación ϵ_2 con respecto del valor esperado, para cada posible sustitución simbólica.

Θ_2	$s_1^\#$	$\&_{s_2}^\#$	t_2	ϵ_2	ϵ_{total}
$\Theta_{2,1}$	0.3	$\&_{godel}$	0.3	0.0	0.0
$\Theta_{2,2}$	0.3	$\&_{luka}$	0.0	0.3	0.3
$\Theta_{2,3}$	0.3	$\&_{prod}$	0.081	0.219	0.219
$\Theta_{2,4}$	0.8	$\&_{godel}$	0.5	0.2	0.2
$\Theta_{2,5}$	0.8	$\&_{luka}$	0.0	0.3	0.3
$\Theta_{2,6}$	0.8	$\&_{prod}$	0.216	0.084	0.084

Para este problema de calibrado, FASILL genera $2 \cdot 3 = 6$ sustituciones simbólicas. Hay una precondition, $(0.6 \&_{s_2}^\# (0.5 \&_{s_2}^\# s_1^\#))$, que descarta las sustituciones simbólicas $\Theta_{2,2}$ y $\Theta_{2,5}$. La mejor sustitución simbólica para el caso de prueba t_2 es $\Theta_{2,1} = \{s_1^\#/0.3, \&_{s_2}^\#/\&_{godel}\}$ con un error absoluto de 0.0. Así, la mejor sustitución simbólica es $\Theta = \Theta_{1,3} \cup \Theta_{2,1}$ con un error absoluto de 0.055, que coincide con la sustitución simbólica obtenida en el ejemplo 5.4. Además, hemos pasado de considerar 24 sustituciones simbólicas a tan solo 10.

Aunque el algoritmo de calibrado simbólico mejora la eficiencia del calibrado básico, es posible que en ocasiones este método no sea capaz de encontrar la mejor sustitución simbólica cuando hay constantes simbólicas en la relación de similitud asociada al programa, debido a que la mejor sustitución simbólica podría no satisfacer las preconditiones y ser descartada. En el siguiente ejemplo se muestra cómo el método de calibrado básico encuentra la mejor sustitución simbólica, mientras que el método simbólico encuentra una solución subóptima.

Ejemplo 5.6. Sea $\mathcal{P}^\#$ el programa simbólico mostrado en el ejemplo 4.3, que hemos calibrado en los ejemplos anteriores. Supongamos que queremos calibrar el programa $\mathcal{P}^\#$ con respecto a los siguientes casos de prueba:

$$t_1 = \langle good_hotel(hydropolis), 0.7 \rangle;$$

$$t_2 = \langle close(atlantis, bus), 0.0 \rangle.$$

Es decir, hemos cambiado el grado de verdad esperado para el caso de prueba t_2 , de 0.3 a 0.0. Vamos a considerar las mismas sustituciones simbólicas que en el

ejemplo 5.3. Para este conjunto de casos de prueba, el método de calibrado básico reporta la sustitución simbólica $\Theta_7 = \{s_1^\# / 0.3, \&_{s_2}^\# / \&_{luka}, s_3^\# / 0.8, @_{s_4}^\# / @_{aver}\}$ con una desviación de 0.055. Al igual que en los problemas anteriores, en el método simbólico hay una precondition, $(0.6 \&_{s_2}^\# (0.5 \&_{s_2}^\# s_1^\#))$, que no se satisface para la sustitución simbólica Θ_7 :

$$\vartheta_L((0.6 \&_{s_2}^\# (0.5 \&_{s_2}^\# s_1^\#))\Theta_7) = \vartheta_L(0.6 \&_{luka} (0.5 \&_{luka} 0.3)) = 0.$$

Por lo tanto, el método de calibrado simbólico descarta Θ_7 y en su lugar reporta la sustitución simbólica $\Theta_{11} = \{s_1^\# / 0.3, \&_{s_2}^\# / \&_{prod}, s_3^\# / 0.8, @_{s_4}^\# / @_{aver}\}$ con una desviación de 0.217.

Sin embargo, en la práctica este problema no es tan frecuente y su efecto se mitiga al considerar un espacio de búsqueda mayor a la hora de calibrar. Más aún, es posible modificar el algoritmo de calibrado simbólico para calcular correctamente la desviación de las sustituciones simbólicas inseguras aplicando el algoritmo de calibrado básico; esto es, ejecutando los objetivos originales de los casos de prueba en $\mathcal{P}^\#\Theta$ para estas sustituciones simbólicas inseguras en lugar de utilizar las respuestas computadas difusas simbólicas.

5.4. Calibrado basado en satisfacibilidad

El problema de satisfacibilidad booleana (SAT) consiste en determinar si una fórmula proposicional puede satisfacerse [MW12]. El problema de satisfacibilidad módulo teorías (SMT) generaliza este problema añadiendo razonamiento sobre igualdad, aritmética, cuantificadores y otras teorías de primer orden de interés. Los resolutores de satisfacibilidad son herramientas utilizadas para decidir la satisfacibilidad de fórmulas en dichas teorías, y constituyen una amplia línea de investigación [BT18, MW12] en el desarrollo de técnicas eficaces para la demostración automática de teoremas en la lógica clásica. También existen algunas aproximaciones [ABMV12, VBG12] para la lógica difusa.

En esta sección, siguiendo la línea de nuestra propuesta [RM20], mostramos cómo es posible expresar un problema de calibrado de un programa sFASILL como un problema de satisfacibilidad que puede ser resuelto por un resolutor SMT. Tal y como se indicó al principio del capítulo, una vez que se han obtenido las respuestas computadas difusas simbólicas para los casos de prueba, el problema de calibrado se reduce a un problema de optimización en el que únicamente hay que asignar valores y evaluar conectivas. Por lo tanto, gracias a la semántica operacional de la extensión simbólica de FASILL y a los resultados sobre la preservación de las respuestas computadas difusas (véanse los teoremas 4.1, 4.2 y 4.3), podemos calcular las s.f.c.a. en el sistema FASILL y delegar la búsqueda de la mejor sustitución simbólica a otro sistema.

Aquí utilizaremos uno de los resolutores SMT más populares, Z3 [MB08, BM09, MB12, BMNW19], junto al estándar SMT-LIB: una iniciativa internacional que proporciona una descripción de las teorías utilizadas en un sistema SMT y que define un lenguaje común para la entrada/salida en estos sistemas

[BST⁺10, BMR⁺11]. La funcionalidad principal de Z3 es comprobar la satisfacibilidad de fórmulas lógicas sobre una o más teorías y producir modelos para las fórmulas satisfacibles. No obstante, en muchos casos los modelos arbitrarios son insuficientes para aplicaciones que deben resolver problemas de optimización: uno o varios valores deben ser mínimos o máximos. Por lo tanto, Z3 extiende el estándar SMT-LIB para incorporar nuevos comandos que permiten expresar la optimización de objetivos [BPF15].

Una de las principales ventajas del calibrado en Z3 es que no es necesario considerar un subconjunto de grados de verdad cuando trabajamos con un retículo en un intervalo real con infinitos elementos y, por lo tanto, el resultado es más preciso. Además, como veremos más adelante, puede llegar a ser mucho más eficiente que el sistema FASILL para algunos programas. Los pasos a seguir son los siguientes:

- (1) Primero, el retículo completo asociado al programa sFASILL debe ser traducido (manualmente) a un lenguaje soportado por Z3, por ejemplo, SMT-LIB.
- (2) Después, el sistema FASILL calcula las respuestas computadas difusas simbólicas de los casos de prueba, y estas son traducidas automáticamente a fórmulas de Z3 (junto a las precondiciones necesarias para descartar sustituciones simbólicas inseguras).
- (3) Finalmente, el resolutor Z3 se ejecuta con el objetivo de minimizar la desviación de las s.f.c.a. con respecto de los grados de verdad esperados de los casos de prueba, comprobando la satisfacibilidad de las fórmulas.

Ejemplo 5.7. Considérese el problema de calibrado mostrado en el ejemplo 5.2. El método de calibrado basado en satisfacibilidad en Z3 produce el siguiente programa en SMT-LIB, Primero, se declaran las constantes simbólicas y la desviación, donde las conectivas simbólicas $\&_{s_2}^{\#}$ y $@_{s_4}^{\#}$ son representadas como cadenas de caracteres (`sym!and!2!s2` y `sym!agr!2!s4`), y los valores simbólicos $s_1^{\#}$ y $s_3^{\#}$ como valores reales (`sym!td!0!s1` y `sym!td!0!s3`).

```

1 (declare-const sym!td!0!s1 Real)
2 (declare-const sym!and!2!s2 String)
3 (declare-const sym!td!0!s3 Real)
4 (declare-const sym!agr!2!s4 String)
5 (declare-const deviation! Real)

```

A continuación, se especifica el dominio de las constantes simbólicas; es decir, el rango de valores que pueden tomar. En el caso de los valores simbólicos, para el retículo $([0, 1], \leq)$, el rango es cualquier número real entre 0 y 1. Para la conectiva $\&_{s_2}^{\#}$, es cualquiera de las t-normas definidas en el retículo. Y para la conectiva $@_{s_4}^{\#}$, es cualquier agregador de aridad 2 definido en el retículo.

```

1 (assert (lat!member sym!td!0!s1))
2 (assert (dom!sym!and!2 sym!and!2!s2))
3 (assert (lat!member sym!td!0!s3))
4 (assert (dom!sym!agr!2 sym!agr!2!s4))

```

Después se establecen las precondiciones en forma de $L^\#$ -expresiones que deben ser distintas de \perp . En este caso, como ya vimos en el ejemplo 5.4, hay una sola precondición: $(0.6 \&_{s_2}^\# (0.5 \&_{s_2}^\# s_1^\#))$.

```

1 (assert
2   (not
3     (=
4       (call!sym!and!2 sym!and!2!s2 0.6
5         (call!sym!and!2 sym!and!2!s2 0.5 sym!td!0!s1))
6       0.0)))

```

Entonces, se establece la desviación como la suma de las distancias entre las respuestas computadas difusas simbólicas y los valores esperados de los casos de prueba.

```

1 (assert
2   (= deviation!
3     (+
4       (lat!distance 0.7
5         (call!sym!agr!2 sym!agr!2!s4 0.49 sym!td!0!s3))
6       (lat!distance 0.3
7         (call!sym!and!2 sym!and!2!s2
8           (call!sym!and!2 sym!and!2!s2 0.6
9             (lat!supremum sym!td!0!s1
10              (call!sym!and!2 sym!and!2!s2
11                (lat!supremum 0.4
12                  (call!sym!and!2 sym!and!2!s2 sym!td!0!s1
13                    (call!sym!and!2 sym!and!2!s2 sym!td!0!s1 0.4))))
14                (call!sym!and!2 sym!and!2!s2 0.4 sym!td!0!s1))))
15             0.9))))))

```

Finalmente, se busca un modelo que satisface estas restricciones, minimizando la desviación.

```

1 (minimize deviation!)
2 (check-sat)
3 (get-model)

```

Para este programa, Z3 reporta el siguiente modelo:

```

1 (model
2   (define-fun sym!agr!2!s4 () String "agr_aver")
3   (define-fun sym!and!2!s2 () String "and_godel")
4   (define-fun sym!td!0!s1 () Real 0.3)
5   (define-fun sym!td!0!s3 () Real 0.91)
6   (define-fun deviation! () Real 0.0)
7 )

```

El modelo generado por Z3, que es transparente al usuario, es analizado por el sistema FASILL para construir la sustitución simbólica

$$\Theta = \{s_1^\# / 0.3, \&_{s_2}^\# / \&_{godel}, s_3^\# / 0.91, @_{s_4}^\# / @_{aver}\},$$

que produce una desviación de 0.0 respecto a los casos de prueba.

5.5. Detalles de implementación

En esta sección se presentan los detalles de implementación del calibrado de programas en el sistema FASILL. Los casos de prueba pueden ser introducidos en el entorno mediante el predicado incorporado `consult_testcases/1`, que recibe la ruta de un fichero que contiene los casos de prueba. Los programas pueden ser calibrados mediante el comando `:tuning`, que calibra el programa cargado en el entorno reportando la mejor sustitución simbólica encontrada, la desviación de dicha sustitución simbólica con respecto a los casos de prueba y el tiempo de calibrado. La figura 5.1 muestra el proceso de calibrado del programa simbólico del ejemplo 4.3 en la consola interactiva del sistema FASILL.

La herramienta web también ofrece la posibilidad de calibrar programas simbólicos [MR17]. Sobre el cuadro de texto del objetivo hay dos pestañas para alternar entre la ejecución de objetivos y el calibrado de programas, tal y como se muestra en la figura 5.2. En la pestaña de calibrado, el usuario puede introducir los casos de prueba además de precondiciones adicionales a las que puedan ser generadas automáticamente por el sistema, y seleccionar el método de calibrado (mediante FASILL o mediante Z3). Tras pulsar el botón de calibrado el resultado se muestra en el área de salida.

El módulo `fasill_tuning` del sistema FASILL contiene la implementación de las técnicas de calibrado descritas en los algoritmos 5.1 y 5.3, y exporta predicados para el calibrado y el razonamiento de programas simbólicos como, por ejemplo, la recolección de las constantes simbólicas que ocurren en una expresión, o la aplicación de una sustitución simbólica a una expresión. Además, el módulo `fasill_tuning_smt` contiene la implementación del método de calibrado basado en satisfacibilidad mediante el uso del resolutor Z3.

Calibrado básico

El predicado principal para calibrar un programa mediante el método básico es `fasill_basic_tuning/3`, que recibe un valor de tolerancia para el error

```

fasill> :listing.
(1) cheap(taxi) <- #s3.
(2) close(hydropolis,taxi) <- 0.7.
(3) close(ritz,metro) <- 0.9.
(4) good_hotel(X) <- #@s4(@very(close(X,Y)),cheap(Y)).

fasill> consult_testcases('good_hotel.test.fpl').
<1.0, {}>

fasill> :tuning.
best symbolic substitution:
    {#@s4/@aver,#s3/0.8,#&s2/&godel,#s1/0.3}
deviation:
    0.05499999999999994
execution time:
    17 milliseconds

```

Figura 5.1: Calibrado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el comando `:tuning`.

cometido por la sustitución simbólica y devuelve la primera sustitución simbólica cuya desviación es menor que dicho umbral. Cuando el umbral es 0, este predicado devuelve la sustitución simbólica que minimiza la desviación (entre las sustituciones simbólicas consideradas).

```

1  fasill_basic_tuning(Cut, S, D) :-
2      retractall(tuning_best_substitution(_, _)),
3      findall(Body, fasill_rule(_,body(Body),_), ExprSym, GoalSym),
4      findall(Goal, fasill_testcase(_, Goal), GoalSym, SimSym),
5      findall(TD, similarity_between(_, _, _, TD, true), SimSym),
6      fasill_findall_symbolic_cons(ExprSym, Sym),
7      findall(testcase(TD,Goal), fasill_testcase(TD,Goal), Tests),
8      ( fasill_symbolic_substitution(Sym, Sub),
9        fasill_basic_tuning_check_testcases(Tests, Sub),
10       tuning_best_substitution(S, D),
11       ( D =< Cut -> ! ; false )
12       ; tuning_best_substitution(S, D) ).

```

Primero, FASILL recoge todos los cuerpos de las reglas del programa, los objetivos de los casos de prueba y las entradas (simbólicas) de la relación de similitud. Entonces busca todas las constantes simbólicas que aparecen en estas expresiones mediante el predicado `fasill_findall_symbolic_cons/2`. A continuación, por reevaluación, el predicado `fasill_symbolic_substitution/2` genera todas las posibles sustituciones simbólicas para dichas constantes, mientras que el pre-

Running Tuning

✎ Test cases

```

1 0.7 -> good_hotel(hydropolis).
2 0.3 -> close(atlantis, bus).

```

FASILL ▼ Tune program

Cut value (optional)

Output

🔍 Symbolic substitution

```

1 best symbolic substitution: {#@s4/@aver,#s3/0.8,#&s2/&godel,#s1/0.3}
2 deviation: 0.05499999999999994
3 execution time: 17 milliseconds

```

Figura 5.2: Área de calibrado del sistema FASILL en línea.

dicado `fasill_basic_tuning_check_testcases/2` recibe el conjunto de casos de prueba y una sustitución simbólica para comprobar la desviación cometida por la misma respecto a los casos de prueba. Al realizar esta búsqueda, el sistema almacena en todo momento la mejor sustitución simbólica como un hecho del predicado dinámico `tuning_best_substitution/2`, que contiene la sustitución y el error cometido respecto a los casos de prueba. De este modo, se evita generar en memoria todas las posibles sustituciones simbólicas para luego iterarlas y en su lugar la búsqueda se realiza por reevaluación, generando y comprobando las sustituciones simbólicas una a una.

```

1 fasill_basic_tuning_check_testcases(Tests, S) :-
2     fasill_symbolic_substitution_flag(S, Flag),
3     set_fasill_flag(symbolic_substitution, Flag),
4     fasill_basic_tuning_check_testcases(Tests, S, 0.0).
5
6 fasill_basic_tuning_check_testcases([], S, D) :-
7     (tuning_best_substitution(_, Best) -> Best > D ; true),
8     retractall(tuning_best_substitution(_, _)),

```

```

9      asserta(tuning_best_substitution(S, D)).
10     fasill_basic_tuning_check_testcases([testcase(E,G)|Ts], S, DO) :-
11       (tuning_best_substitution(_, Best) -> Best > DO ; true),
12       once(query(G, state(TD, _))),
13       lattice_call_distance(E, TD, num(Distance)),
14       D1 is DO + Distance,
15       fasill_basic_tuning_check_testcases(Ts, S, D1).

```

Para calcular el error de cada una de las sustituciones simbólicas, el predicado `fasill_basic_tuning_check_testcases/2` establece la sustitución simbólica en el entorno mediante la bandera `symbolic_substitution`. Entonces, el sistema itera los casos de prueba, computando la primera respuesta computada difusa para el objetivo y calculando la distancia entre el valor obtenido y el valor esperado.

Calibrado simbólico

El predicado principal para calibrar un programa mediante el método simbólico es `fasill_tuning/3`, que recibe un valor de tolerancia para el error cometido por la sustitución simbólica, y devuelve la primera sustitución simbólica cuya desviación es menor que dicho umbral. Igual que en el calibrado básico, cuando el umbral es 0, este predicado devuelve la sustitución simbólica que minimiza la desviación (entre las sustituciones simbólicas consideradas).

```

1     fasill_tuning(Cut, S, Deviation) :-
2       fasill_tuning_disjoint_sets(Tests, Preconditions),
3       fasill_tuning_loop(Cut, Tests, Preconditions, Ss, Deviation),
4       append(Ss, S).

```

Primero, el predicado `fasill_tuning_disjoint_sets/2` separa los casos de prueba y las condiciones en conjuntos disjuntos, devolviendo una lista de casos de prueba y una lista de condiciones, ambas indexadas y ordenadas por las constantes simbólicas que contienen. Las componentes conexas se obtienen mediante una estructura de datos para conjuntos disjuntos que hemos publicado como un paquete independiente para SWI-Prolog³ y cuyo código fuente está disponible en GitHub.⁴ A partir de ahí, el predicado `fasill_tuning_loop/5` busca la mejor sustitución simbólica para cada uno de estos subproblemas de calibrado y las sustituciones parciales se componen para obtener la sustitución simbólica final.

```

1     fasill_tuning_loop(Cut, Tests, Preconditions, Ss, D) :-
2       fasill_tuning_loop(Cut, Tests, Preconditions, Ss, 0.0, D).
3
4     fasill_tuning_loop(_, [], [], [], D, D).

```

³https://www.swi-prolog.org/pack/list?p=union_find

⁴<https://github.com/jariazavalverde/prolog-union-find>

```

5  fasill_tuning_loop(Cut, [C-T], [C-P], [S], D0, D2) :-
6      Tolerance is Cut - D0,
7      fasill_tuning_best_substitution(Tolerance, C, T, P, S, D1),
8      D2 is D0 + D1.
9  fasill_tuning_loop(Cut, [C-T, T2|Ts], [C-P, P2|Ps], [S|Ss], D0, D3) :-
10     fasill_tuning_best_substitution(0.0, C, T, P, S, D1),
11     D2 is D0 + D1,
12     fasill_tuning_loop(Cut, [T2|Ts], [P2|Ps], Ss, D2, D3).

```

Con el fin de podar la búsqueda en función del valor de corte, cuando sólo queda un conjunto por calibrar, se calcula la tolerancia como la diferencia entre el valor de corte y el error cometido hasta ese momento. Este umbral se utiliza para buscar una sustitución simbólica con un error menor o igual que dicha tolerancia mediante el predicado `fasill_tuning_best_substitution/6`. Mientras haya más de un conjunto, esta tolerancia se establece a 0 para encontrar la mejor sustitución simbólica.

El predicado `fasill_tuning_best_substitution/6` recibe el valor de tolerancia, las constantes simbólicas, el conjunto de casos de prueba y el conjunto de precondiciones de un subproblema de calibrado para proceder a la búsqueda de la mejor sustitución simbólica para dicho problema. Para ello, genera por reevaluación todas las posibles sustituciones simbólicas para las constantes simbólicas indicadas mediante el predicado `fasill_symbolic_substitution/2`. Además, comprueba para cada una de ellas que se satisfacen todas las precondiciones y calcula la desviación que se produce con respecto a los casos de prueba.

```

1  fasill_tuning_best_substitution(
2      Tolerance, Sym, Tests, Preconditions, S, D) :-
3      retractall(tuning_best_substitution(_, _)),
4      ( fasill_symbolic_substitution(Sym, Sub),
5        fasill_tuning_check_preconditions(Preconditions, Sub),
6        fasill_tuning_check_testcases(Tests, Sub),
7        tuning_best_substitution(S, D),
8        ( D =< Tolerance -> ! ; false )
9        ; tuning_best_substitution(S, D) ).

```

El predicado `fasill_tuning_check_preconditions/2` recibe una lista de precondiciones y una sustitución simbólica para comprobar que esta es segura. Para ello ejecuta cada precondición como un objetivo y comprueba que el grado de verdad asociado al mismo es distinto de \perp . Además de las precondiciones generadas por el sistema FASILL, el usuario puede introducir junto a los casos de prueba objetivos arbitrarios que serán también considerados precondiciones. En el calibrado basado en Z3 esto no es posible, ya que únicamente podemos evaluar L -expresiones (lo cual es suficiente para comprobar las precondiciones generadas por FASILL).

```

1  fasill_tuning_check_preconditions([], _).
2  fasill_tuning_check_preconditions([precondition(GoalS)|Ps], S) :-
3      fasill_apply_symbolic_substitution(GoalS, S, Goal),
4      query(Goal, state(FCA, _)),
5      lattice_call_bot(Bot),
6      \+lattice_call_leq(FCA, Bot),
7      fasill_tuning_check_preconditions(Ps, S).

```

El predicado `fasill_tuning_check_testcases/2` se encarga de, dado un conjunto de casos de prueba y una sustitución simbólica, calcular el error cometido por la misma. Para ello aplica la sustitución simbólica a los objetivos de los casos de prueba, termina de calcular las respuestas computadas difusas asociadas a los caso de prueba y, finalmente, calcula la distancia entre los valores obtenidos y los valores esperados. Si en algún momento el error cometido por la sustitución simbólica es mayor que la desviación de la mejor sustitución simbólica encontrada hasta el momento, el objetivo falla y esa sustitución simbólica es descartada. Cuando se han comprobado todos los casos de prueba, si la desviación es menor que la desviación de la mejor sustitución simbólica encontrada hasta el momento, se elimina la anterior sustitución y se almacena la nueva.

```

1  fasill_tuning_check_testcases(Tests, S) :-
2      fasill_tuning_check_testcases(Tests, S, 0.0).
3
4  fasill_tuning_check_testcases([], S, Deviation) :-
5      ( tuning_best_substitution(_, Best) ->
6          Best > Deviation
7          ; true),
8      retractall(tuning_best_substitution(_, _)),
9      asserta(tuning_best_substitution(S, Deviation)).
10  fasill_tuning_check_testcases([testcase(E,SFCA)|Tests],S,D0) :-
11      (tuning_best_substitution(_, Best) -> Best > D0 ; true),
12      fasill_apply_symbolic_substitution(SFCA, S, FCA),
13      query(FCA, state(TD, _)),
14      lattice_call_distance(E, TD, num(Distance)),
15      D1 is D0 + Distance,
16      fasill_tuning_check_testcases(Tests, S, D1).

```

Calibrado basado en satisfacibilidad

El predicado principal para calibrar un programa mediante el método basado en satisfacibilidad es `tuning_smt/4`, que recibe un átomo que representa el tipo de dato de los elementos del retículo en SMT-LIB (`Real`, `Bool`, ...) y la ruta a un fichero que contiene la definición del mismo en formato SMT-LIB, y devuelve la sustitución simbólica y la desviación reportadas por Z3. La lectura

y escritura de expresiones y programas en formato SMT-LIB es gestionada por un paquete independiente que hemos publicado para SWI-Prolog⁵ y que utiliza listas anidadas para representar internamente dichas expresiones.

```

1 tuning_smt(Domain, LatFile, Substitution, Deviation) :-
2   smtlib_read_script(LatFile, list(Lat)),
3   tuning_smt_sfca(ListOfSFCA),
4   fasill_findall_symbolic_cons(ListOfSFCA, ListCons),
5   list_to_set(ListCons, Cons),
6   tuning_smt_decl_const(Domain, Cons, Dec),
7   ( member([reserved('define-fun'),symbol('lat!member')|_],Lat)
8   -> tuning_smt_members(Cons, Members)
9   ; Members = []
10  ),
11  tuning_smt_preconditions(Pre),
12  tuning_smt_minimize(ListOfSFCA, Minimize),
13  GetModel = [[reserved('check-sat')], [reserved('get-model')]],
14  append([Lat, Dec, Members, Pre, Minimize, GetModel], Script),
15  smtlib_write_to_file('.tuning.smt2', list(Script)),
16  shell('z3 -smt2 .tuning.smt2 > .result.tuning.smt2', _),
17  smtlib_read_expressions('.result.tuning.smt2', Z3answer),
18  tuning_smt_answer(Z3answer, Substitution, Deviation).

```

En esencia, este predicado calcula las respuestas computadas difusas simbólicas para los objetivos de los casos de prueba cargados en el entorno, busca las constantes simbólicas contenidas en las s.f.c.a., genera las declaraciones de las variables que representan las constantes simbólicas en Z3 y las precondiciones necesarias para descartar sustituciones simbólicas inseguras y conforma el programa final en formato SMT-LIB. Este programa se almacena en un fichero temporal y se ejecuta en Z3, cuya salida es analizada por FASILL para generar la sustitución simbólica resultante.

El predicado `sfca_to_smtlib/2` traduce un objetivo sFASILL a una expresión SMT-LIB en la representación interna utilizada por el paquete de SWI-Prolog.

```

1 sfca_to_smtlib(num(X), numeral(X)) :-
2   !,
3   integer(X).
4 sfca_to_smtlib(num(X), decimal(X)) :-
5   !,
6   float(X).
7 sfca_to_smtlib(term('#'(X),[]), symbol(Y)) :-
8   !,
9   atom_concat('sym!td!0!', X, Y).

```

⁵<https://eu.swi-prolog.org/pack/list?p=smtlib>

```

10 sfca_to_smtlib(term(X,[]), symbol(X)) :-
11     !,
12     atomic(X).
13 sfca_to_smtlib(sup(X,Y), [symbol('lat!supremum'), Ex, Ey]) :-
14     !,
15     sfca_to_smtlib(X, Ex),
16     sfca_to_smtlib(Y, Ey).
17 sfca_to_smtlib(term(X,Xs), [symbol(Con),symbol(Name2)|Xs2]) :-
18     X =.. [Op,Name],
19     member((Op,Pre), [( '#&', 'and!' ), ( '#|', 'or!' ), ( '#@', 'agr!' )]),
20     !,
21     length(Xs, Length),
22     atomic_list_concat(['call!sym!', Pre, Length], Con),
23     atomic_list_concat(['sym!', Pre, Length, '!', Name], Name2),
24     maplist(sfca_to_smtlib, Xs, Xs2).
25 sfca_to_smtlib(term(X,Xs), [symbol(Con)|Xs2]) :-
26     ( X = '&', Op = '&', lattice_tnorm('&'(Name)))
27     ; ( X = '|', Op = '|', lattice_tconorm('|' (Name)))
28     ; X =.. [Op,Name]
29     ), !,
30     member((Op,Pre), [( '&', 'lat!and!' ), ( '|', 'lat!or!' ),
31     ( '@', 'lat!agr!' ), ( '#', 'sym!td!' )]),
32     length(Xs, Length),
33     atomic_list_concat([Pre, Name, '!', Length], Con),
34     maplist(sfca_to_smtlib, Xs, Xs2).

```

El resto de predicados de este módulo simplemente utilizan los predicados descritos en las implementaciones de los algoritmos de calibrado anteriores para recoger las precondiciones y las respuestas computadas difusas simbólicas y traducirlas a SMT-LIB mediante el predicado `sfca_to_smtlib/2`. Por ejemplo, el predicado `tuning_smt_minimize/2` recibe la lista de las respuestas computadas difusas simbólicas de los objetivos de los casos de prueba (junto a los grados de verdad esperados para los mismos) y construye el objetivo a minimizar en SMT-LIB.

```

1 tuning_smt_minimize(ListOfSFCA, [Assert, Minimize]) :-
2     findall([symbol('lat!distance'), TD_, SMT],
3         ( member((SFCA,TD), ListOfSFCA),
4             sfca_to_smtlib(TD, TD_),
5             sfca_to_smtlib(SFCA, SMT)
6         ), Distances),
7     Assert = [reserved(assert),
8     [symbol(=), symbol('deviation!'), [symbol(+)|Distances]]],
9     Minimize = [reserved(minimize), symbol('deviation!')].

```

5.6. Pruebas y evaluación de rendimiento

Para determinar el rendimiento de los distintos métodos de calibrado de programas lógicos difusos en el entorno FASILL, hemos preparado una serie de pruebas y casos de uso con los que además ilustramos la utilidad de las distintas técnicas de calibrado en diversos dominios de aplicación de interés.

En nuestro primer experimento discutimos los resultados de calibrar un mismo programa FASILL tanto con el método de calibrado básico como con el método simbólico, variando tanto el número de pasos de computación necesarios para resolver todos los átomos, como el número de sustituciones simbólicas exploradas. Para ello, consideramos una regla general de la forma “ $p \leftarrow q \wedge \dots \wedge q \wedge s^\#$ ” (que contiene k instancias del átomo q y una constante simbólica $s^\#$), junto con un hecho “ $q \leftarrow \top$ ”; e introducimos un único caso de prueba: $\langle p, \top \rangle$. Además, en todas las ejecuciones, la única sustitución simbólica que produce una desviación de 0 es considerada la última en el espacio de búsqueda para asegurar que ambos métodos exploran exactamente el mismo número de sustituciones simbólicas.

Tabla 5.1: Tiempo medio de ejecución (en milisegundos) de los métodos de calibrado básico y simbólico tras 50 ejecuciones, en función del número de átomos en la derivación (k) y del número de sustituciones simbólicas consideradas, para una constante simbólica y un caso de prueba.

Calibrado	k	Tiempo (ms)					
		10_Θ	50_Θ	100_Θ	250_Θ	500_Θ	1000_Θ
Básico	1	1	15	15	17	35	70
	5	15	16	18	53	98	198
	10	16	23	39	98	212	414
	25	16	85	134	343	692	1373
	50	42	211	431	1029	2090	4250
	100	148	709	1337	4164	8100	13515
Simbólico	1	1	15	16	16	17	25
	5	15	16	16	16	31	68
	10	15	16	17	26	53	104
	25	15	21	31	87	151	232
	50	15	34	59	114	228	471
	100	32	79	104	250	481	967

La tabla 5.1 resume el tiempo medio de ejecución obtenido tras calibrar diferentes instancias del programa FASILL propuesto mediante el algoritmo básico, que no hace uso de la semántica operacional de la extensión simbólica, y mediante el algoritmo simbólico.⁶ Como es de esperar, el contraste de ambos algoritmos revela una mejora muy significativa del tiempo de ejecución del método de calibrado simbólico frente al método básico, que se hace más pro-

⁶Todas las pruebas de esta sección han sido ejecutadas en Swi-Prolog 8.0.4 (64 bits) utilizando un ordenador de sobremesa equipado con un procesador AMD Opteron™ @ 1593 MHz y 2.00 GB RAM.

nunciada conforme más átomos se explotan en la derivación y más sustituciones simbólicas se consideran, ya que el método simbólico sólo calcula la s.f.c.a. una única vez en todo el proceso, mientras que el método básico la recalcula para cada sustitución simbólica. En particular, para 1000 sustituciones simbólicas y 100 átomos, el método simbólico es más de 10 veces más rápido que el método básico. Las figuras 5.3 y 5.4 ilustran la comparación de los tiempos de ejecución (en escala logarítmica) de los métodos de calibrado básico y simbólico (1) en función del número de sustituciones simbólicas consideradas y (2) en función del número de átomos explotados en los objetivos de los casos de prueba, donde se aprecia que el método simbólico puede reducir el tiempo de ejecución varias magnitudes.

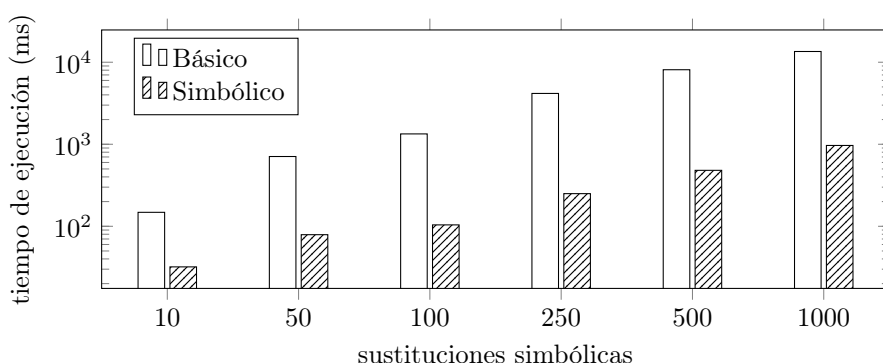


Figura 5.3: Comparación de los tiempos de ejecución de los algoritmos de calibrado básico y simbólico en función del número de sustituciones simbólicas consideradas (con 100 átomos).

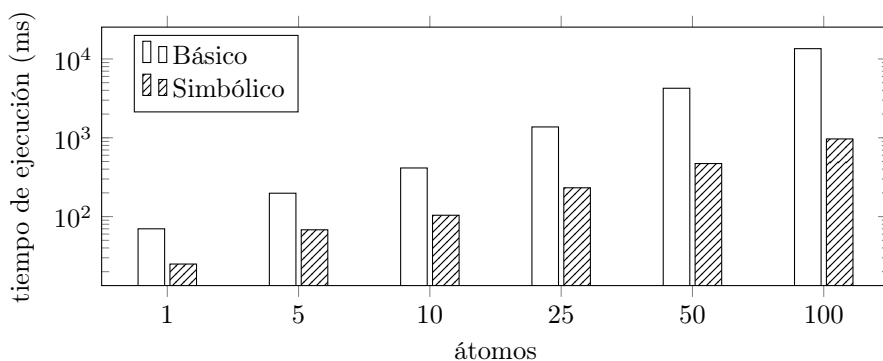


Figura 5.4: Comparación de los tiempos de ejecución de los algoritmos de calibrado básico y simbólico en función del número de átomos en los objetivos de los casos de prueba (con 1000 sustituciones simbólicas).

En lo que resta de sección presentamos algunas aplicaciones prácticas del

calibrado de programas lógicos difusos, como son la equivalencia de circuitos combinacionales [RM20], el aprendizaje automático [MPR17, RM20] o la web semántica [AJBTMR19]. Junto a estas aplicaciones presentamos algunos experimentos con el objetivo de medir la eficiencia del método de calibrado basado en satisfacibilidad.

5.6.1. Equivalencia de circuitos combinacionales

Un circuito combinacional es un circuito formado por funciones lógicas elementales que tiene un determinado número de entradas y salidas, y donde las salidas dependen únicamente de la combinación de sus entradas. En la figura 5.5 se muestran dos circuitos combinacionales con cuatro entradas (x_0, x_1, x_2, x_3) y dos salidas (y_0, y_1). El problema de verificar la equivalencia de dos circuitos combinacionales es de vital importancia en el campo de la verificación de circuitos digitales. Como consecuencia, han surgido numerosos enfoques para solucionar este problema [MSG99]. Aquí, proponemos un método basado en satisfacibilidad, expresando el problema como un programa lógico del cual se obtendrá la equivalencia mediante el calibrado de un objetivo.

Sean C_A y C_B dos circuitos combinacionales, ambos con n entradas x_1, \dots, x_n

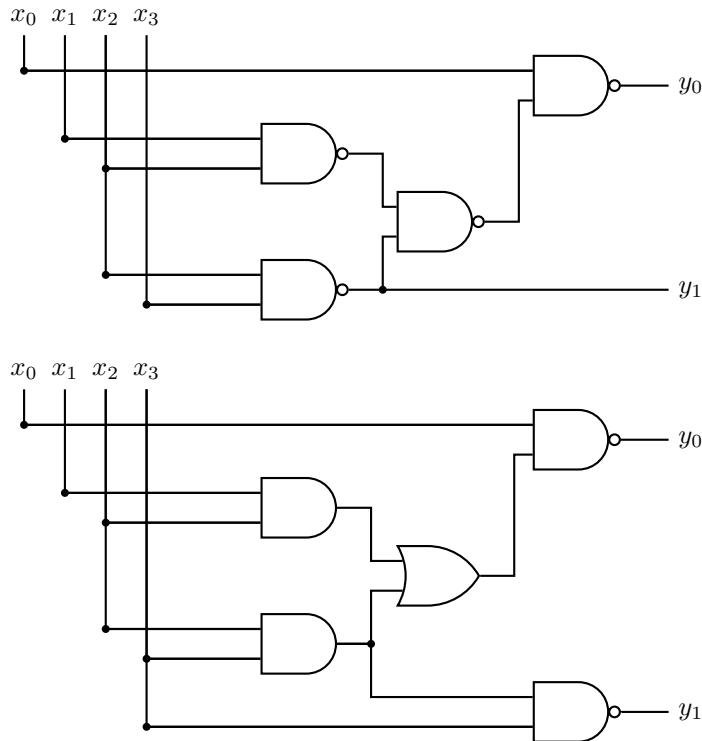


Figura 5.5: Dos circuitos combinacionales equivalentes.

y m salidas, C_A con salidas y_1, \dots, y_m y C_B con salidas w_1, \dots, w_m . La función implementada por cada uno de los circuitos se define como sigue: $f_A : \{0, 1\}^n \rightarrow \{0, 1\}^m$ y $f_B : \{0, 1\}^n \rightarrow \{0, 1\}^m$. Sea $x \in \{0, 1\}^n$, y definamos $f_A(x) = (f_{A,1}(x), \dots, f_{A,m}(x))$ y $f_B(x) = (f_{B,1}(x), \dots, f_{B,m}(x))$. Ambos circuitos no son equivalentes si se satisface la siguiente condición:

$$\exists x \in \{0, 1\}^n, \exists i \in [1, m], f_{A,i}(x) \neq f_{B,i}(x)$$

que puede ser expresada como el siguiente problema de satisfacibilidad, representado en la figura 5.6 como un circuito combinacional (conocido como *miter*) [MS08]:

$$\bigvee_{i=1}^n (f_{A,i}(x) \oplus f_{B,i}(x)) = 1 \quad (5.1)$$

A partir de estos resultados, es fácil codificar en forma normal clausal el problema de verificar la equivalencia de dos circuitos combinacionales y, por lo tanto, es susceptible de ser implementado y calibrado como un programa lógico.

Implementación

En este problema utilizaremos el retículo booleano $\mathcal{B} = \{0, 1\}$ descrito en el anexo A.2. Expresaremos cada uno de los circuitos combinacionales en FASILL como predicados de aridad 2, donde el primer argumento es una lista que contiene las entradas y el segundo argumento es una lista que contiene las salidas.

Ejemplo 5.8. Considérense los circuitos combinacionales representados en la figura 5.5. A continuación se muestra una posible codificación de estos circuitos en el lenguaje FASILL como los predicados $a/2$ y $b/2$:

```

1  a([X0,X1,X2,X3], [Y0,Y1]) :-
2      @not((X0 & @not((@not((X1 & X2)) & @not((X2 & X3)))))) on Y0,
3      @not((X2 & X3)) on Y1.
```

```

1  b([X0,X1,X2,X3], [Y0,Y1]) :-
2      @not((X0 & ((X1 & X2) | (X2 & X3)))) on Y0,
3      @not(((X2 & X3) & X3)) on Y1.
```

Podemos comprobar la salida de estos circuitos ante determinadas entradas. Por ejemplo, para la asignación $(0, 1, 1, 1)$, el objetivo $\mathcal{G} = (x = [0, 1, 1, 1] \& a(x, y) \& b(x, w))$ devuelve la respuesta computada difusa

$$\langle 1, \{y/[1, 1], w/[1, 1]\} \rangle.$$

Además, podemos pasar valores simbólicos como entradas a los circuitos y obtener las L^s -expresiones correspondientes en forma de términos. Por ejemplo, para

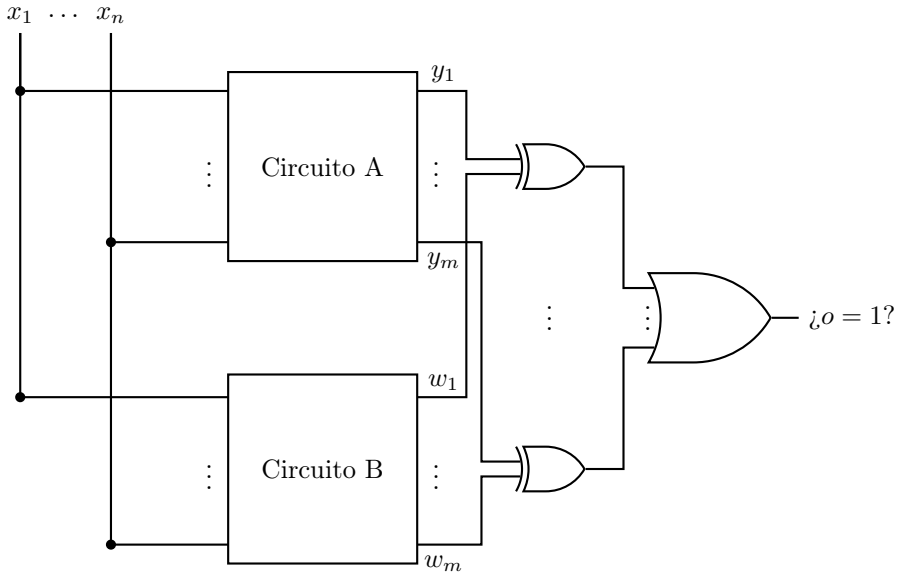


Figura 5.6: Circuito de comprobación de equivalencia (*miter*).

la asignación simbólica $(x_1^\#, x_2^\#, x_3^\#, x_4^\#)$, el objetivo $\mathcal{G} = a([x_1^\#, x_2^\#, x_3^\#, x_4^\#], y)$ devuelve la respuesta computada difusa

$$\langle 1, \{y/[\text{@not}(\&(x_1^\#, \text{@not}(\&(\text{@not}(\&(x_2^\#, x_3^\#))), \text{@not}(\&(x_3^\#, x_4^\#))))), \text{@not}(\&(x_3^\#, x_4^\#))]\} \rangle.$$

Para comprobar la equivalencia entre dos circuitos codificaremos el circuito de equivalencia representado en la figura 5.6. El predicado `miter/3` toma dos circuitos (dos átomos que representan el nombre de los predicados de cada circuito) y una lista de entradas, y comprueba si alguna de las salidas de ambos circuitos es distinta para la entrada proporcionada. Este predicado se evalúa con grado de verdad 0 cuando todas las salidas son iguales ante la entrada proporcionada, o con grado de verdad 1 en cualquier otro caso.

```

1 zip_xor([], [], []).
2 zip_xor([X|Xs], [Y|Ys], [@xor(X,Y)|Zs]) :- zip_xor(Xs,Ys,Zs).
3
4 fold_or([], false).
5 fold_or([X|Xs], '!'(X,Ys)) :- fold_or(Xs, Ys).
6
7 miter(Ca, Cb, Xs) :-
8     call(Ca, Xs, Ys),
9     call(Cb, Xs, Ws),

```

```

10 zip_xor(Ys, Ws, XOR),
11 fold_or(XOR, OR),
12 OR.
```

El predicado `zip_xor/2` recibe dos listas y devuelve una lista combinando elemento a elemento las dos primeras con el agregador `@_xor`. El predicado `fold_or/2` recibe una lista y la reduce a un único término en forma de disyunción, tomando 0 como valor inicial.

Ejemplo 5.9. Con el predicado `miter/3` podemos comprobar si los circuitos implementados en el ejemplo 5.8 proporcionan alguna salida distinta ante una entrada determinada. Por ejemplo, el objetivo $\mathcal{G} = \text{miter}(a, b, [0, 1, 1, 1])$ lleva a la respuesta computada difusa $\langle 0, \{\} \rangle$; es decir, todas las salidas de ambos circuitos son idénticas ante la entrada $(0, 1, 1, 1)$. Si evaluamos este predicado con una entrada simbólica, ahora obtendremos una respuesta computada difusa simbólica, ya que `miter/3` se evalúa con el grado de verdad que representa el término devuelto como disyunción de todas las disyunciones exclusivas de las salidas de ambos circuitos. Por ejemplo, para el objetivo $\mathcal{G} = \text{miter}(a, b, [x_1^\#, x_2^\#, x_3^\#, x_4^\#])$ obtenemos la s.f.c.a.

$$\langle \&(1, \&(1, \&(1, \&(1, |(@_xor(@_not(\&(x_1^\#, @_not(\&(@_not(\&(x_2^\#, x_3^\#), \\ @_not(\&(x_3^\#, x_4^\#)))))), @_not(\&(x_1^\#, |(\&(x_2^\#, x_3^\#), \&(x_3^\#, x_4^\#))))), \\ |(@_xor(@_not(\&(x_3^\#, x_4^\#)), @_not(\&(\&(x_3^\#, x_4^\#), x_4^\#))), 0))))), \{\}) \rangle.$$

Se observa que ahora la L^s -expresión forma parte del grado de verdad de la respuesta computada en lugar de formar parte de la sustitución.

Una vez implementado el predicado `miter/3` es fácil detectar la equivalencia entre dos circuitos cualesquiera mediante el calibrado de programas. Para ello solo es necesario un caso de prueba de la siguiente forma:

$$1 \rightarrow \text{miter}(C_A, C_B, [x_1^\#, \dots, x_m^\#]).$$

Este caso de prueba le indica al sistema que queremos encontrar una combinación de entradas (x_1, \dots, x_m) para los circuitos C_A y C_B tal que la salida del predicado `miter/3` sea 1, es decir, tal que alguna de las salidas y_i de ambos circuitos sea distinta. Si los circuitos son equivalentes el sistema no será capaz de encontrar tal asignación de valores y devolverá una sustitución simbólica arbitraria con una desviación de 1.0 (puesto que habrá fallado con el único caso de prueba que hemos introducido, que será evaluado a 0). En caso contrario, devolverá una sustitución simbólica para la que alguna de las salidas es distinta en ambos circuitos, con una desviación de 0.0.

Ejemplo 5.10. Comprobemos la equivalencia de los circuitos mostrados en la figura 5.5. Para ello, ejecutamos el proceso de calibrado con el programa FASILL mostrado en el ejemplo 5.8 introduciendo el siguiente caso de prueba:

```

1 true -> miter(a, b, [#x1, #x2, #x3, #x4]).
```

El entorno FASILL proporciona la siguiente salida que demuestra que los circuitos son equivalentes, ya que la desviación de la sustitución simbólica encontrada es de 1.0:

```

1 substitution: {#x1/false,#x2/false,#x3/false,#x4/false}
2 deviation: 1.0

```

Ejemplo 5.11. Introduzcamos ahora un tercer circuito combinacional, C_C , no equivalente a los dos anteriores, representado en FASILL por el siguiente predicado $c/2$:

```

1 c([X0,X1,X2,X3], [Y0,Y1]) :-
2   @not((X0 & X1)) on Y0,
3   @not((X2 | X3)) on Y1.

```

Lanzamos ahora el proceso de calibrado para comprobar la equivalencia de los circuitos $a/2$ y $c/2$. En este caso, el sistema encuentra una combinación de las entradas, $(0, 0, 0, 1)$, para la que ambos circuitos producen salidas diferentes, ya que la desviación es de 0.0:

```

1 substitution: {#x1/false,#x2/false,#x3/false,#x4/true}
2 deviation: 0.0

```

Para esta combinación, el circuito $a/2$ produce las salidas $(1, 1)$, mientras que el circuito $c/2$ produce las salidas $(1, 0)$.

En la tabla 5.2 se muestran todas las posibles combinaciones de entradas para los tres circuitos combinacionales ejemplificados, C_A , C_B y C_C , junto a sus salidas. En esta tabla se observa que, en efecto, los circuitos C_A y C_B son equivalentes entre sí, pero el circuito C_C produce salidas distintas para varias de las combinaciones de entradas (entre ellas para la combinación que reporta el proceso de calibrado).

Resultados experimentales

En la tabla 5.3 se resumen las medias de los tiempos de ejecución (en milisegundos) del algoritmo de calibrado simbólico en FASILL y del algoritmo de calibrado basado en satisfacibilidad en Z3, tras 50 ejecuciones, al comprobar la equivalencia de diversos circuitos combinacionales variando el número de entradas de los mismos. En esta tabla se observa que el aumento del número de constantes simbólicas (que se corresponde con el número de entradas de los circuitos combinacionales) supone un incremento exponencial en el tiempo de ejecución del método de calibrado del entorno FASILL, mientras que utilizando Z3 es mucho más eficiente.

Tabla 5.2: Tabla de verdad de las funciones lógicas implementadas por los circuitos combinacionales C_A , C_B y C_C .

Entradas				C_A		C_B		C_C	
x_0	x_1	x_2	x_3	y_0	y_1	w_0	w_1	v_0	v_1
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	1	1	1	0
0	0	1	1	1	0	1	0	1	0
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0
0	1	1	0	1	1	1	1	1	0
0	1	1	1	1	0	1	0	1	0
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	0
1	0	1	1	0	0	0	0	1	0
1	1	0	0	1	1	1	1	0	1
1	1	0	1	1	1	1	1	0	0
1	1	1	0	0	1	0	1	0	0
1	1	1	1	0	0	0	0	0	0

Tabla 5.3: Tiempo medio de ejecución (en milisegundos) de los algoritmos de calibrado simbólico y basado en satisfacibilidad, tras 50 ejecuciones, al comprobar la equivalencia de circuitos combinacionales en función del número de entradas.

# entradas	Tiempo (ms)	
	Simbólico	Satisfacibilidad
4	45	36
5	930	37
6	2260	40
7	5670	41
8	12200	42
9	29340	43
10	65480	45

5.6.2. Regresión lineal

En estadística, los métodos de regresión estudian la construcción de modelos para representar la relación entre una variable dependiente y un conjunto de variables explicativas. Un modelo de regresión lineal [MPV21] puede ser descrito mediante la siguiente ecuación:

$$y = \beta_0 + \beta_1 x_1 + \dots + \beta_n x_n + \epsilon \quad (5.2)$$

donde β_0, \dots, β_n son parámetros fijos desconocidos, x_1, \dots, x_n son variables explicativas (no estocásticas) y ϵ es una variable aleatoria inobservable.

En aplicaciones prácticas disponemos de una muestra de observaciones de estas variables, y el modelo anterior sugiere que la relación entre estas se satisfaga para cada una de las observaciones correspondientes [NKNW96]. La ecuación 5.2 indica que la variable aleatoria y se genera como combinación lineal de las variables explicativas, salvo en una perturbación aleatoria ϵ . El problema que abordamos es el de estimar los parámetros desconocidos β_0, \dots, β_n . Para ello contamos con una muestra de k observaciones de la variable aleatoria y y de los correspondientes valores de las variables explicativas x_i . Debemos encontrar entonces aquellos valores de β_0, \dots, β_n que hagan mínimos los errores de estimación.

Ejemplo 5.12. Un viejo estudio [Pie46] midió la frecuencia de chirridos (pulsos por segundo) de los grillos 15 veces, a diferentes temperaturas. En la tabla 5.4 se muestran los datos resultantes, donde la primera columna representa la temperatura en grados centígrados y la segunda columna representa el nivel de ruido en decibelios. Se quiere estudiar la relación existente entre la temperatura

Tabla 5.4: Conjunto de datos del estudio [Pie46].

Temp.	Ruido	Temp.	Ruido	Temp.	Ruido
20.00	88.59	15.50	75.19	15.00	79.59
16.00	71.59	14.69	69.69	17.20	82.59
19.79	93.30	17.10	82.00	16.00	80.59
18.39	84.30	15.39	69.40	17.00	83.50
17.10	80.59	16.20	83.30	14.39	76.30

y el nivel de ruido generado por los grillos. Utilizando estos datos para crear un modelo de regresión, obtenemos –mediante el método de los mínimos cuadrados⁷– la recta $y = 25.23 + 3.29x$ mostrada en la figura 5.7 junto al diagrama de dispersión de los datos.

Implementación

Expresaremos el modelo de regresión como un programa `SFASILL` con una única regla que codifica la ecuación 5.2 mediante la combinación de las conectivas `|add` y `@mul` del retículo conformado por la recta real extendida $\overline{\mathbb{R}}$ (véase el apéndice A.3), donde los parámetros β_i serán originalmente constantes simbólicas:

$$y(x_1, \dots, x_n) \leftarrow \beta_0^\# \mid_{add} @_{mul}(\beta_1^\#, x_1) \mid_{add} \dots \mid_{add} @_{mul}(\beta_n^\#, x_n).$$

Este predicado recibe como entradas n variables explicativas y se evalúa con un grado de verdad que es combinación lineal de estas entradas. Para encontrar los parámetros β_i que mejor se ajustan a los datos que tenemos, calibraremos este programa introduciendo un caso de prueba por cada muestra del conjunto de

⁷Técnica numérica que minimiza la suma de las diferencias al cuadrado de un conjunto de pares.

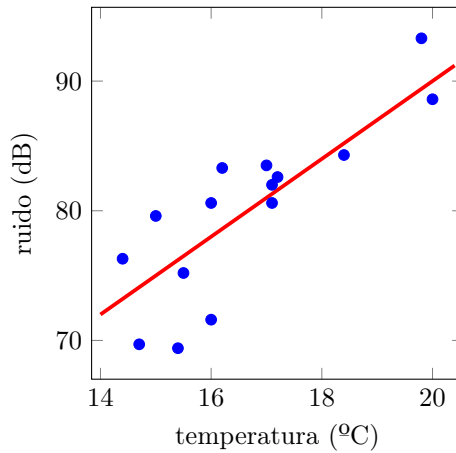


Figura 5.7: Diagrama de dispersión del conjunto de datos del estudio [Pie46].

datos, donde el valor de la variable dependiente será el grado de verdad esperado del caso de prueba.

Ejemplo 5.13. Calibremos el modelo de regresión mostrado en el ejemplo 5.12. Dado que el conjunto de datos tiene únicamente una variable explicativa (la temperatura) el programa tendrá dos constantes simbólicas:

```
1 chirps(Temperature) <- #b0 |add @mul(#b1, Temperature).
```

Cada muestra del conjunto de datos (véase la tabla 5.4) se convierte en un caso de prueba, obteniéndose así el siguiente conjunto de casos de prueba:

```
1 88.6 -> chirps(20.0). 71.6 -> chirps(16.0). 93.3 -> chirps(19.8).
2 84.3 -> chirps(18.4). 80.6 -> chirps(17.1). 75.2 -> chirps(15.5).
3 69.7 -> chirps(14.7). 82.0 -> chirps(17.1). 69.4 -> chirps(15.4).
4 83.3 -> chirps(16.2). 79.6 -> chirps(15.0). 82.6 -> chirps(17.2).
5 80.6 -> chirps(16.0). 83.5 -> chirps(17.0). 76.3 -> chirps(14.4).
```

Tras ejecutar el proceso de calibrado basado en satisfacibilidad, el sistema FASILL reporta la sustitución simbólica $\Theta = \{\beta_0^\# / 42.92, \beta_1^\# / 2.28\}$ con una desviación de 43.23:

```
1 substitution: {#b0/42.92, #b1/2.28}
2 deviation: 43.23
```

Podemos observar que los valores $\beta_0 = 42.92$ y $\beta_1 = 2.28$ no son los mismos que los obtenidos en el ejemplo 5.12. Esto es debido a que en el ejemplo anterior se ha minimizado la distancia utilizando el error cuadrático medio, mientras que aquí se ha utilizado el error absoluto medio, que es la noción de distancia

que hemos definido en el retículo asociado a este programa. En la figura 5.8 se muestra la recta $y = 42.92 + 2.28x$ junto al diagrama de dispersión de los datos, donde la línea discontinua representa el modelo actual y la línea continua representa el modelo del ejemplo 5.12.

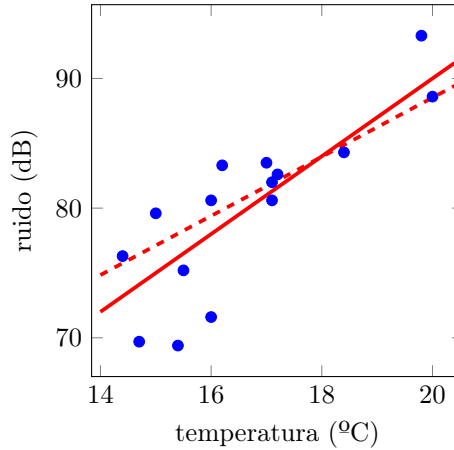


Figura 5.8: Modelo de regresión lineal del ejemplo 5.13.

Resultados experimentales

Dado que los elementos del retículo real no están acotados en un intervalo, resulta complicado escoger una muestra representativa de los mismos adecuada para un problema de regresión y, aunque fuese posible, el algoritmo de calibrado simbólico de FASILL no sería capaz de resolver estos problemas en un tiempo razonable (si se desea un resultado suficientemente preciso), puesto que tendría que evaluar una gran cantidad de sustituciones simbólicas. Por lo tanto, en el siguiente experimento únicamente medimos el tiempo de ejecución de la técnica de calibrado basada en satisfacibilidad con Z3 al calibrar diversos problemas de regresión, variando el número de casos de prueba y el número de variables explicativas.

En la tabla 5.5 se resumen las medias de los tiempos de ejecución (en segundos) del algoritmo de calibrado basado en satisfacibilidad tras 10 ejecuciones, al ajustar modelos de regresión variando el número de variables explicativas de los mismos y el número de casos de prueba. La figura 5.9 ilustra la comparación de estos tiempos de ejecución. Observamos que tanto el incremento de variables explicativas como el incremento de casos de prueba suponen un crecimiento exponencial del tiempo de ejecución del algoritmo de calibrado en Z3, siendo más determinante el número de variables explicativas del modelo que el número de casos de prueba.

Tabla 5.5: Tiempo medio de ejecución (en segundos) del algoritmo de calibrado basado en satisfacibilidad en Z3 para regresión lineal en función del número de variables explicativas y del número de casos de prueba (k).

k	Tiempo (s)				
	1_x	2_x	3_x	4_x	5_x
20	0.35	1.55	6.27	11.38	33.48
30	2.22	6.45	27.05	49.40	72.34
40	6.01	25.32	107.68	165.52	378.15
50	14.04	43.60	184.35	583.45	1547.56
60	29.80	150.85	619.22	1094.20	3319.37
70	40.30	213.32	794.55	2452.84	7532.63
80	51.08	311.07	1058.35	4261.00	17317.25
90	106.15	625.05	1658.60	7688.81	40313.90
100	215.10	874.47	3189.72	11204.84	85873.49

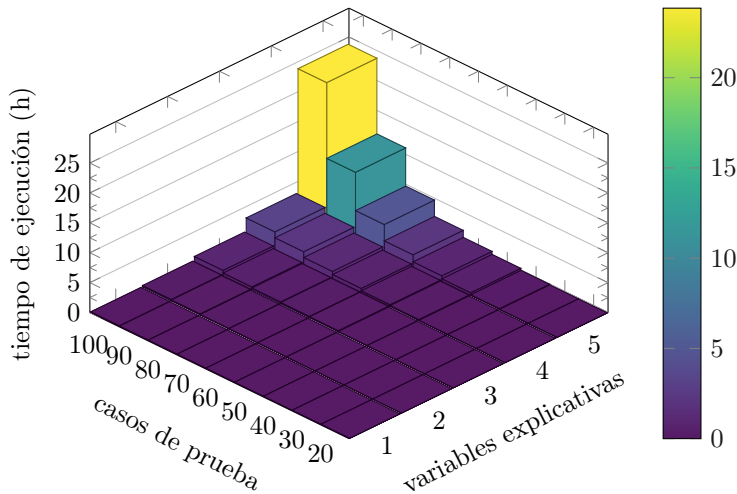


Figura 5.9: Comparación de los tiempos de ejecución del algoritmo de calibrado basado en satisfacibilidad ajustando modelos de regresión lineal.

5.6.3. Redes neuronales

Una red neuronal artificial [Bis94] es una función matemática concebida como un conjunto de neuronas, usualmente organizadas por capas, donde cada neurona recibe entradas y produce una única salida que puede ser enviada a múltiples neuronas. La salida de la k -ésima neurona se define como una suma ponderada de sus entradas x_1, \dots, x_n :

$$y_k = f \left(\sum_{i=0}^n w_{ki} x_i \right) \quad (5.3)$$

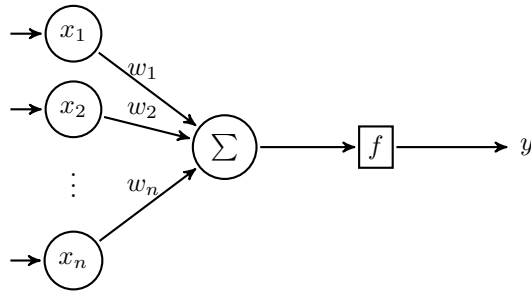
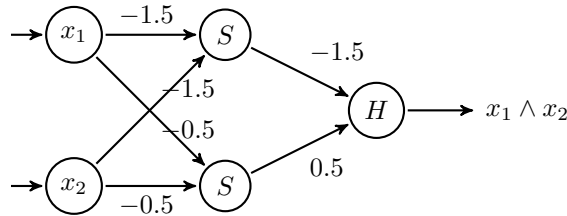


Figura 5.10: Diagrama de un modelo de neurona artificial.

donde f es una función de activación (que normalmente no es lineal). La figura 5.10 ilustra el modelo de una neurona artificial. Al igual que en el problema anterior, en la práctica disponemos de una muestra de observaciones con las entradas y el resultado esperado para las mismas, donde el objetivo es ajustar los pesos y las funciones de activación de la red para mejorar la precisión de sus resultados.

Ejemplo 5.14. La siguiente red neuronal, que toma dos entradas y se compone de tres nodos organizados en dos capas, implementa la función de verdad de la conjunción lógica.



Aquí, los nodos de la primera capa utilizan la función de activación logística $S : \mathbb{R} \rightarrow [0, 1]$ y el nodo de la segunda capa utiliza la función de activación de escalón unitario $H : \mathbb{R} \rightarrow \{0, 1\}$, donde:

$$S(x) = \frac{1}{1 + e^{-x}}; \quad H(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}.$$

Por ejemplo, para las entradas $x_1 = x_2 = 1$, la red neuronal produce la salida $y = 1$:

$$\begin{aligned} & H(-1.5 \cdot S(1 \cdot (-1.5) + 1 \cdot (-1.5)) + 0.5 \cdot S(1 \cdot (-0.5) + 1 \cdot (-0.5))) = \\ & H(-1.5 \cdot S(-3) + 0.5 \cdot S(-1)) \approx \\ & H(-1.5 \cdot 0.0474 + 0.5 \cdot 0.2689) \approx \\ & H(0.0633) = 1; \end{aligned}$$

mientras que para cualquier otra combinación de las entradas $x_1, x_2 \in \{0, 1\}$ produce la salida $y = 0$.

Implementación

Expresaremos la red neuronal como un programa SFASILL, donde el nodo k -ésimo de la capa j se codifica como una regla $y_{(j,k)}$ que modela la ecuación 5.3 mediante la combinación de las conectivas $|_{add}$ y $@_{mul}$ del retículo conformado por la recta real extendida $\overline{\mathbb{R}}$ (véase el apéndice A.3), donde los pesos $w_{(j,k,i)}$ y las funciones de activación f_j serán originalmente constantes simbólicas. Los nodos de la primera capa utilizan las entradas como grados de verdad:

$$y_{(1,k)}(x_1, \dots, x_n) \leftarrow @_{f_1}^{\#} (@_{mul}(w_{(1,k,1)}^{\#}, x_1) |_{add} \dots |_{add} @_{mul}(w_{(1,k,n)}^{\#}, x_n))$$

mientras que los nodos del resto de capas invocan a los predicados que codifican los nodos de la capa anterior, pasando las entradas como argumentos hasta los nodos de la primera capa:

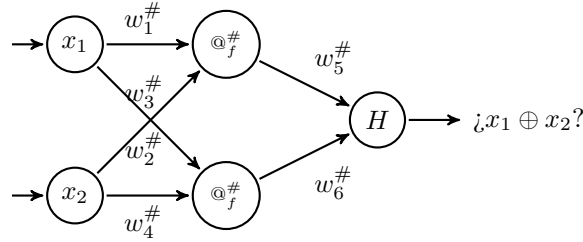
$$y_{(j,k)}(x_1, \dots, x_n) \leftarrow @_{f_j}^{\#} (@_{mul}(w_{(j,k,1)}^{\#}, y_{(j-1,1)}(x_1, \dots, x_n)) |_{add} \dots |_{add} @_{mul}(w_{(j,k,m)}^{\#}, y_{(j-1,m)}(x_1, \dots, x_n)))$$

Además de las conectivas $|_{add}$ y $@_{mul}$, equipamos el retículo con un amplio repertorio de agregadores que actúan como funciones de activación, cuyas funciones de verdad se resumen en la figura 5.11. Para encontrar los parámetros $w_{(j,k,i)}$ y las funciones de activación f_j que mejor se ajustan a los datos, calibraremos este programa introduciendo un caso de prueba por cada muestra del conjunto de datos.

$F_{@sigmoid}(x) = \frac{1}{1 + e^{-x}}$	$F_{@softplus}(x) = \ln(1 + e^x)$
$F_{@relu}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$F_{@leaky_relu}(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
$F_{@softsign}(x) = \frac{1}{1 + x }$	$F_{@sinusoid}(x) = \begin{cases} 1 & \text{if } x = 0 \\ \frac{\sin(x)}{x} & \text{if } x \neq 0 \end{cases}$
$F_{@arctan}(x) = \arctan(x)$	$F_{@tanh}(x) = \tanh(x)$
$F_{@binary}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$	$F_{@identity}(x) = x$

Figura 5.11: Funciones de verdad asociadas a las funciones de activación utilizadas en redes neuronales.

Ejemplo 5.15. Considérese la red neuronal mostrada en el ejemplo 5.14 que modela la conjunción lógica. Supongamos que queremos modificar su comportamiento para modelar otra puerta lógica, por ejemplo, la disyunción exclusiva.



Dado que la red neuronal tiene tres nodos organizados en dos capas, el programa sFASILL se compone de tres reglas:

```

1 node_1_1(X1, X2) <- #@f((@mul(#w1, X1) |add @mul(#w2, X2))).
2 node_1_2(X1, X2) <- #@f((@mul(#w3, X1) |add @mul(#w4, X2))).
3 node_2_1(X1, X2) <- @binary((
4   @mul(#w5, node_1_1(X1, X2)) |add
5   @mul(#w6, node_1_2(X1, X2))))).

```

Para especificar el comportamiento deseado introducimos un caso de prueba para cada posible combinación de entradas de esta puerta lógica:

```

1 0.0 -> node_2_1(0.0, 0.0). 1.0 -> node_2_1(0.0, 1.0).
2 1.0 -> node_2_1(1.0, 0.0). 0.0 -> node_2_1(1.0, 1.0).

```

Tras ejecutar el proceso de calibrado simbólico, considerando los grados de verdad $\{-1.5, -0.5, 0.5, 1.5\}$ como posibles valores de los elementos simbólicos, el sistema FASILL reporta la sustitución simbólica $\Theta = \{w_1^\# / -1.5, w_2^\# / 1.5, w_3^\# / -0.5, w_4^\# / -0.5, w_5^\# / 0.5, w_6^\# / -1.5, @f^\# / @gaussian\}$ con un error de 0.0.

```

1 substitution: {#w1/-1.5, #w2/1.5, #w3/-0.5,
2               #w4/-0.5, #w5/0.5, #w6/-1.5, #@f1/@gaussian}
3 deviation: 0.0

```

En [MPR19b] presentamos un ejemplo más amplio de calibrado de redes neuronales, donde estudiamos el proceso de calibrado de una red neuronal previamente entrenada para la clasificación de flores de Iris [Fis36].

5.6.4. La web semántica

La web semántica [BLHL01] es una extensión de la web enfocada al desarrollo de estándares y tecnologías para la publicación y consulta de datos legibles por aplicaciones informáticas. RDF (acrónimo de «*marco de descripción de recursos*») proporciona un modelo de datos simple para expresar declaraciones utilizando tuplas de la forma (*sujeto, predicado, valor*). Un conjunto de sentencias RDF utiliza un vocabulario particular que define las propiedades y los tipos de datos que son significativos para la aplicación en cuestión. Por otro

lado, SPARQL es un lenguaje de consulta de RDF –esto es, un lenguaje de consulta semántica para bases de datos– capaz de recuperar y manipular datos almacenados en formato RDF [SET09].

Ejemplo 5.16. SPARQL permite el acceso a información disponible en la web a través de diversas plataformas, como DBpedia,⁸ que proporciona acceso a toda la información de Wikipedia. La siguiente consulta SPARQL extrae el nombre y las estadísticas de los jugadores de béisbol pertenecientes a los *Toronto Blue Jays* (excluyendo a los lanzadores) ordenados ascendentemente por nombre:

```

1 SELECT DISTINCT ?name ?batting ?homeruns ?runs WHERE {
2     ?player a dbo:BaseballPlayer ;
3         foaf:name ?name ; dbo:team ?team;
4         dbo:position ?position ; dbp:stat1value ?batting ;
5         dbp:stat2value ?homeruns ; dbp:stat3value ?runs .
6     FILTER(?team = dbr:Toronto_Blue_Jays
7         && ?position != dbr:Pitcher
8         && ?position != dbr:Coach_(baseball)) .
9 } ORDER BY ASC(?name)

```

En la tabla 5.6 se muestran los jugadores recuperados de DBpedia mediante esta consulta.

Tanto RDF como SPARQL han sido diseñados para consultar información clara y precisa. No obstante, algunos contextos requieren tratar con incertidumbre. Con este objetivo se diseñó el lenguaje de consulta FSA-SPARQL [ABM17, AJBTM18], una extensión de SPARQL que permite trabajar con predicados difusos y agregar grados de verdad mediante conectivas difusas [AJBTMR21]. En [AJBTMR19] diseñamos una herramienta capaz de compilar las consultas FSA-SPARQL a FASILL con el fin de aplicar las técnicas de calibrado de FASILL sobre las consultas FSA-SPARQL. Posteriormente, en [AJBTMR22] hemos extendido esta herramienta para calibrar también automáticamente los conjuntos difusos de los atributos difusos asociados a los conjuntos de datos consultados.

Ejemplo 5.17. Podemos exportar los resultados extraídos de la consulta mostrada en el ejemplo 5.16 en formato JSON e importarlos en FSA-SPARQL difuminando los datos de las estadísticas de los jugadores. La siguiente consulta FSA-SPARQL extrae los jugadores de béisbol mejor considerados por encima de 0.6, donde este criterio es computado como la media aritmética entre tener una media de bateo muy alta y haber impulsado un alto número de carreras:

```

1 PREFIX sn:<http://sn.org#>
2 PREFIX f:<http://www.fuzzy.org#>
3 PREFIX l:<http://www.lattice.org#>
4 SELECT ?name ?rank WHERE {
5     sn:root sn:list ?player .
6     ?player sn:name ?name .

```

⁸<http://wiki.dbpedia.org>

```

7   ?player f:type (sn:batting f:high ?batting) .
8   ?player f:type (sn:runs f:high ?runs) .
9   BIND(1:WMEAN(0.5, 1:VERY(?batting), ?runs) as ?rank) .
10  FILTER(?rank > 0.6) .
11 }

```

En esta consulta, la media de bateo y las carreras impulsadas son atributos difusos, mientras que el nombre es un atributo clásico. En la tabla 5.7 se muestran los jugadores recuperados mediante esta consulta. Por ejemplo, bajo este criterio, Bo Bichette es un buen jugador con un grado de verdad de 0.69.

El objetivo ahora es calibrar este tipo de consultas difusas para seleccionar los grados de verdad y las conectivas más adecuadas en base a información previa proporcionada por el usuario, como podría ser el valor de verdad esperado para determinados jugadores ante la consulta mostrada en el ejemplo 5.17.

Tabla 5.6: Respuesta a la consulta SPARQL del ejemplo 5.16.

Nombre	Media de bateo	Jonrones	Carreras impulsadas
Aledmys Díaz	0.271	62	231
Alejandro Kirk	0.259	9	27
Billy McKinney	0.215	27	68
Bo Bichette	0.301	45	146
Casey Candaele	0.25	11	139
Cavan Biggio	0.235	31	103
Charlie Montoyo	0.4	0	3
Danny Jansen	0.212	33	99
Dante Bichette	0.299	274	1141
Dave Hudgens	0.143	0	0
Derek Fisher	0.195	17	53
George Springer	0.269	196	508
Jesús Figueroa	0.253	1	11
Joshua Palacios	0.2	0	4
Lourdes Gurriel Jr.	0.282	63	202
Luis Rivera	0.233	28	209
Mallex Smith	0.255	13	114
Mark Budzinski	0	0	0
Mike Ohlman	0.231	3	0
Nevin Ashley	0.1	0	1
Otto Lopez	0	0	0
Randal Grichuk	0.245	156	439
Reese McGuire	0.248	9	26
Sal Butera	0.227	8	76
Santiago Espinal	0.301	2	23
Teoscar Hernández	0.26	108	303
Vladimir Guerrero Jr.	0.289	72	213

Tabla 5.7: Respuesta a la consulta FSA-SPARQL del ejemplo 5.17.

Nombre	Rango
Dante Bichette	0.9950125
George Springer	0.8570125
Randal Grichuk	0.7628125
Vladimir Guerrero Jr.	0.7265300
Teoscar Hernández	0.7183347
Bo Bichette	0.6919368
Lourdes Gurriel Jr.	0.6796065
Aledmys Díaz	0.6691934

Implementación

En la extensión difusa de RDF, los conjuntos difusos son representados mediante clases RDF y el grado de pertenencia de un elemento al conjunto difuso es anotado mediante una propiedad RDF y un grado de pertenencia. Por ejemplo, `?player f:type (sn:batting f:high ?batting)` se refiere al grado de pertenencia del jugador al conjunto difuso “alto” de la propiedad RDF “media de bateo”. En general, un elemento dado puede pertenecer a un mismo conjunto difuso con diferentes grados de pertenencia respecto a diferentes atributos. En esencia, FSA-SPARQL es una extensión de SPARQL que permite consultar conjuntos de datos RDF difusos. Las tuplas RDF difusas pueden ser traducidas a tuplas RDF estándares siguiendo la transformación definida en [AJBTMR22]. En la práctica los datos carecen de una componente (explícita) difusa. Por esta razón proveemos un mecanismo que nos permite difuminar los conjuntos de datos clásicos. En términos generales, un valor numérico puede considerarse “alto”, “medio” o “bajo” para una propiedad determinada. En [AJBTMR22], proponemos el uso de funciones trapezoidales para modelar las funciones de pertenencia debido a su alta generalidad y simplicidad. Tal y como se muestra en la figura 5.12, la función de pertenencia de un conjunto difuso A definido como un trapecoide depende de cuatro parámetros (a , b , c y d), y verifica la siguiente fórmula:

$$\mu_A(x) = \begin{cases} 0 & \text{si } x \leq a \\ (x - a)/(b - a) & \text{si } a \leq x \leq b \\ 1 & \text{si } b \leq x \leq c \\ (d - x)/(d - c) & \text{si } c \leq x \leq d \\ 0 & \text{si } d \leq x \end{cases} \quad (5.4)$$

Esta función es codificada en Prolog como un agregador del retículo asociado a los programas FASILL de la siguiente manera.

```

1 agr_trapezoidal(A^^T, _B^^T, _C^^T, D^^T, X^^T, 0^^T) :-
2   X < A ; X > D.
3 agr_trapezoidal(A^^T, B^^T, _C^^T, _D^^T, X^^T, Y^^T) :-
4   A =< X, X =< B, (B-A == 0 -> Y = 1 ; Y is (X-A)/(B-A)).
5 agr_trapezoidal(_A^^T, B^^T, C^^T, _D^^T, X^^T, 1^^T) :-

```

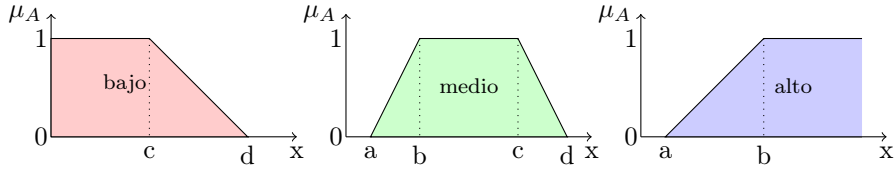


Figura 5.12: Conjuntos difusos basados en una función de pertenencia trapezoidal.

```

6   B =< X, X =< C.
7   agr_trapezoidal(_A^^T, _B^^T, C^^T, D^^T, X^^T, Y^^T) :-
8   C =< X, X =< D, (D-C ::= 0 -> Y = 1 ; Y is (D-X)/(D-C)).

```

Como indicamos anteriormente, consideraremos tres conjuntos difusos (“alto”, “medio” y “bajo”) para cada atributo numérico. El problema de calibrado consistirá en buscar los 4 parámetros de las correspondientes funciones de pertenencia trapezoidales que definen los límites superiores e inferiores de cada trapezoido, que representaremos aquí como valores simbólicos etiquetados como $ll^\#$, $ls^\#$, $us^\#$ y $ul^\#$. El proceso de calibrado de FASILL se ejecuta manipulando directamente la representación de las tuplas RDF. Concretamente, para cada atributo numérico y para cada conjunto difuso (por ejemplo, para la media de bateo alta del jugador Vladimir Guerrero Jr. que es el jugador número 26 considerado en los datos extraídos de DBpedia) se genera un hecho FASILL de la siguiente forma:

```

1   rdf('http://sn.org#root&26http://sn.org#batting_high',
2     'http://www.fuzzy.org#truth', TD) :-
3     @trapezoidal(#ll_batting_high, #ls_batting_high,
4     #us_batting_high, #ul_batting_high) on TD.

```

Así, el proceso de calibrado de un conjunto difuso consiste en seleccionar algunos elementos del conjunto y establecer un grado de pertenencia esperado para los mismos.

Ejemplo 5.18. Consideremos la consulta FSA-SPARQL mostrada en el ejemplo 5.17. Para poder ejecutar dicha consulta, antes ha sido necesario difuminar los atributos numéricos “media de bateo”, “jonrones” y “carreras impulsadas” del conjunto de datos extraído de DBpedia. Centrémonos en el atributo “jonrones”. Observando los datos vemos que Dante Bichette ha anotado 274 jonrones, lo que nos parece una cantidad alta con un grado de pertenencia de 0.9, media con un grado de pertenencia de 0.2 y baja con un grado de pertenencia de 0. Esto genera los siguientes tres casos de prueba:

```

1   0.0^^_ -> @trapezoidal(0^^_, 0^^_, #ul_homeruns_low,
2   #us_homeruns_low, 274^^_).

```

```

3 0.2^_^ -> @trapezoidal(#ll_homeruns_medium, #ls_homeruns_medium,
4  #ul_homeruns_medium, #us_homeruns_medium, 274^_^).
5 0.9^_^ -> @trapezoidal(#ll_homeruns_high, #ls_homeruns_high,
6  274^_^, 274^_^, 274^_^).

```

Tras introducir algunos casos de prueba más (basados en los datos de la tabla 5.6) establecemos cinco elementos equidistantes en el intervalo $[0, 274]$ como posibles valores para las funciones de pertenencia de los conjuntos difusos y ejecutamos el proceso de calibrado en FASILL. Esto genera los conjuntos difusos mostrados en la figura 5.13 para el atributo “jonrones”.

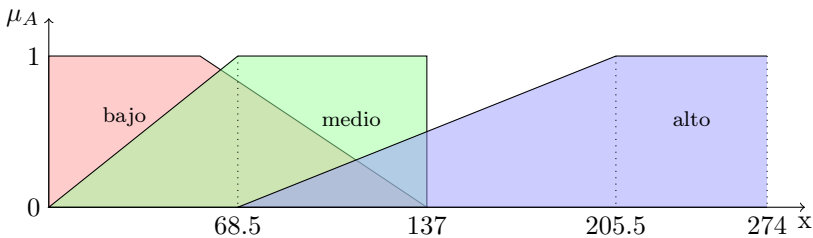


Figura 5.13: Conjuntos difusos para el atributo “jonrones”.

Las consultas FSA-SPARQL pueden ser compiladas a reglas FASILL, y las tuplas RDF difusas –que se compilan a tuplas RDF estándares– se expresan trivialmente como hechos del predicado `rdf/3`. Los detalles concretos de la transformación se describen en [AJBTMR19]. Esto permite calibrar las consultas FSA-SPARQL para encontrar los valores y conectivas más apropiados para una consulta.

Ejemplo 5.19. La consulta FSA-SPARQL mostrada en el ejemplo 5.17 se traduce al siguiente programa FASILL.

```

1 query(Name, Rank) :-
2   rdf('http://sn.org#root', 'http://sn.org#list', A),
3   rdf(A, 'http://sn.org#name', Name),
4   rdf(A, 'http://www.fuzzy.org#type', C),
5   rdf(C, 'http://www.fuzzy.org#onProperty',
6     'http://sn.org#batting'),
7   rdf(C, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
8     'http://www.fuzzy.org#high'),
9   rdf(C, 'http://www.fuzzy.org#truth', E),
10  rdf(A, 'http://www.fuzzy.org#type', D),
11  rdf(D, 'http://www.fuzzy.org#onProperty',
12    'http://sn.org#runs'),
13  rdf(D, 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
14    'http://www.fuzzy.org#high'),
15  rdf(D, 'http://www.fuzzy.org#truth', H),

```

```

16 J = 0.5^^'http://www.w3.org/2001/XMLSchema#decimal',
17 'http://www.lattice.org#VERY'(E, G),
18 'http://www.lattice.org#WMEAN'(J, G, H, Rank),
19 { Rank > 0.6^^'http://www.w3.org/2001/XMLSchema#decimal' }.

```

A esta regla que codifica la consulta original hay que añadirle las tuplas RDF difusas y las definiciones de las conectivas difusas y los operadores de FSA-SPARQL. Las conectivas y los operadores se traducen como reglas que evalúan una conectiva (utilizando la correspondiente definición del retículo descrito en el anexo A.4) y devuelven el resultado como un término mediante el predicado incorporado `on/2`.

```

1 'http://www.lattice.org#WMEAN'(W,X,Y,Z) :- @wmean(W,X,Y) on Z.

```

Además, por cuestiones de eficiencia, es conveniente desactivar los pasos de fallo de la semántica operacional de FASILL mediante la directiva

```
:- set_fasill_flag(failure_steps, false).
```

Si ejecutamos el objetivo difuso $\mathcal{G} = (query(x, v) \& v)$ en el sistema FASILL, obtenemos las respuestas computadas difusas que se corresponden con las respuestas de la tabla 5.7. Supongamos que no estamos conformes con los resultados de la consulta, ya que consideramos que Bo Bichette y Vladimir Guerrero Jr. deberían tener un grado de verdad más alto. Podemos modificar la consulta original para introducir un peso simbólico $w^\#$ en la media ponderada y una conectiva simbólica $@^\#_{op}$ en lugar del modificador lingüístico $@_{very}$:

```

1 BIND(1:WMEAN('#w', 1:APP('#@op', ?batting), ?runs) as ?rank) .

```

Ahora introducimos los casos de prueba para los jugadores mencionados:

```

1 0.95^^_ -> query('Bo Bichette', TD) & TD.
2 0.90^^_ -> query('Vladimir Guerrero Jr.', TD) & TD.

```

Considerando los valores $\{0.0, 0.1, \dots, 0.9, 1.0\}$ para los valores simbólicos, el calibrado reporta la siguiente sustitución simbólica.

```

1 substitution: {#w/0.9, #@op/@more_or_less}
2 deviation: 0.04

```

Para facilitar el proceso de calibrado de los conjuntos difusos, así como la ejecución y el calibrado de las consultas FSA-SPARQL, en [AJBTMR22] hemos desarrollado una aplicación en línea accesible desde la URL <https://dectau.uclm.es/floper/tuning-social-networks>, que permite cargar conjuntos de datos en formato JSON y realizar de forma transparente al usuario todas estas acciones. La herramienta ha sido especialmente diseñada para el análisis y la gestión flexible de información recuperada de redes sociales en tiempo real.

5.7. Conclusiones

En este capítulo hemos presentado la técnica de calibrado automático de programas lógicos difusos y hemos diseñado diversos algoritmos para calibrar programas simbólicos, que hemos implementado en nuestro sistema FASILL. Además, hemos mostrado su aplicación práctica en distintos dominios de interés. Los propósitos de este capítulo se enumeran a continuación.

- (1) Hemos introducido el concepto de calibrado de programas lógicos difusos, que consiste en la búsqueda automática de la mejor sustitución simbólica para un programa en función de un conjunto de casos de prueba introducidos por el usuario y que están formados por un objetivo y un grado de verdad esperado para el mismo.
- (2) Hemos descrito un algoritmo de calibrado básico que no hace uso de la semántica operacional de la extensión simbólica de FASILL. Este método de calibrado aplica cada sustitución simbólica sobre el programa SFASILL para obtener un programa FASILL ordinario, y después calcula las respuestas computadas difusas para los objetivos de los casos de prueba.
- (3) En el capítulo anterior vimos que, bajo determinadas condiciones, los pasos de éxito y de fallo realizados en una derivación para un objetivo simbólico son siempre los mismos, con independencia de las sustituciones simbólicas aplicadas. A partir de esta observación hemos diseñado un algoritmo de calibrado simbólico que mejora la eficiencia del método de calibrado básico, al calcular primero las respuestas computadas difusas simbólicas para los objetivos de los casos de prueba, y aplicar después cada una de las sustituciones simbólicas para realizar los últimos pasos interpretativos. De este modo, el método de calibrado simbólico realiza menos pasos de computación que el método de calibrado básico, mejorando notablemente su eficiencia.
- (4) A partir del método de calibrado simbólico hemos diseñado una mejora del mismo que permite agrupar automáticamente las constantes simbólicas del programa en conjuntos disjuntos en función de sus ocurrencias en las respuestas computadas difusas simbólicas para los casos de prueba. Esto permite dividir el proceso de calibrado en varios subproblemas de calibrado más pequeños, que reportan los mismos resultados que el problema original, pero que mejoran significativamente la eficiencia al considerar un menor número de sustituciones simbólicas.
- (5) Aunque el método de calibrado simbólico permite diseñar algoritmos más eficientes al realizar menos pasos de computación y al considerar menos sustituciones simbólicas, hemos visto que cuando el programa contiene constantes simbólicas en la relación de similitud es posible que el método de calibrado simbólico descarte la mejor sustitución simbólica al considerarla insegura.

- (6) Haciendo uso de la semántica operacional de la extensión simbólica de FASILL, hemos descrito un algoritmo de calibrado basado en satisfacibilidad, que delega la búsqueda de la mejor sustitución simbólica a una herramienta externa de satisfacibilidad módulo teorías (en concreto, el popular resolutor Z3). Para ello, el sistema FASILL calcula las respuestas computadas difusas simbólicas de los casos de prueba y expresa el problema de calibrado como un problema de optimización en el que únicamente hay que evaluar L -expresiones y minimizar la desviación de las mismas con respecto a los valores esperados para los casos de prueba.
- (7) Hemos implementado las técnicas de calibrado en el sistema FASILL y descrito su implementación de alto nivel en Prolog. En particular, hemos incorporado el método de calibrado básico, el método de calibrado simbólico (disjunto) y el método de calibrado basado en satisfacibilidad (con Z3). Además, todas las técnicas de calibrado implementadas pueden ejecutarse en la herramienta web.
- (8) Hemos diseñado una serie de experimentos para comparar los distintos métodos de calibrado. Como era de esperar, hemos observado que cuanto más largas son las derivaciones de los objetivos de los casos de prueba, y cuantas más sustituciones simbólicas consideremos, más ganancia se obtiene en el método de calibrado simbólico frente al método de calibrado básico.
- (9) Hemos aplicado las técnicas de calibrado a problemas no triviales, como la comprobación de la equivalencia de circuitos combinatoriales, el aprendizaje automático (regresión lineal y redes neuronales) y la web semántica (con aplicaciones en la gestión flexible de bases de datos, redes sociales, etcétera). Hemos observado que en función de la naturaleza del problema, es más conveniente un método de calibrado u otro. Por ejemplo, el método de calibrado basado en satisfacibilidad es mucho más eficiente a la hora de comprobar la equivalencia de circuitos combinatoriales, y es más conveniente para regresión lineal ya que no requiere considerar una muestra de los valores del retículo. Sin embargo, no es aconsejable para problemas como el calibrado de consultas FSA-SPARQL, ya que Z3 no es completo para la optimización de objetivos no lineales.
- (10) Respecto a las consultas FSA-SPARQL, hemos mostrado cómo es posible utilizar el calibrado a dos niveles. En primer lugar hemos difuminado los atributos numéricos de los conjuntos de datos RDF, calibrando los conjuntos difusos de dichos atributos. En segundo lugar hemos calibrado las propias consultas FSA-SPARQL compilándolas previamente a FASILL, y calibrando los predicados resultantes.

Es importante destacar que los resultados sobre la extensión simbólica de FASILL dados en el capítulo anterior no solo han favorecido el diseño de algoritmos de calibrado más eficientes en el propio sistema FASILL, sino que además nos

han permitido que el proceso de optimización (la búsqueda de la mejor sustitución simbólica) se desvincule totalmente de la semántica operacional de FASILL una vez calculadas las respuestas computadas difusas simbólicas. Gracias a esto hemos sido capaces de combinar las técnicas de calibrado con los resolutores de satisfacibilidad.

Capítulo 6

Desplegado de programas lógicos difusos

El despliegado es una técnica de transformación automática de programas, que ha sido ampliamente utilizada en varios marcos declarativos para mejorar la eficiencia de los mismos. La transformación de despliegado clásica se basa en la aplicación de pasos de computación sobre los cuerpos de las reglas del programa y en la aplicación de los unificadores sobre sus cabezas. Sin embargo, cuando consideramos relaciones de similitud, la generación y aplicación prematura de unificadores débiles en tiempo de despliegado puede destruir la corrección y la completitud de la transformación, tal y como mostramos en [MPR17, MR19a]. En este capítulo, siguiendo la línea de nuestra propuesta [JIMR22a], estudiamos cómo evitar el riesgo a la hora de desplegar programas FASILL imponiendo algunas condiciones sobre los programas que pueden ser desplegados de forma segura. Además, demostramos que estas condiciones son suficientes para garantizar la corrección de la transformación de despliegado y presentamos algunos experimentos realizados para medir la mejora de eficiencia producida por los programas desplegados.

6.1. Motivación y antecedentes

La transformación de programas es un técnica útil para derivar programas correctos y eficientes por medio de la aplicación de reglas de transformación elementales que mejoran la eficiencia del programa original en algún aspecto, preservando su significado. Las transformaciones de plegado y despliegado, introducidas por primera vez en [BD77] para programas funcionales, son las técnicas más básicas y poderosas para un marco de transformación de programas.

La regla de despliegado considerada originalmente en programación lógica clásica [Tam84] consiste en el reemplazamiento de una cláusula C del programa por el conjunto de cláusulas obtenidas tras la aplicación de un paso de computación, en todas sus posibles formas, sobre el cuerpo de C . En es-

te contexto se han propuesto diversas aproximaciones a la regla de desplegado [PP94, PP96, PP98], todas ellas similares a la original de H. Tamaki, aunque en ocasiones se permite que la regla a desplegar y la regla desplecante pertenezcan a programas distintos (dentro de la misma secuencia de transformaciones) [PP94] o que varias reglas puedan ser desplegadas simultáneamente [LM88].

Ejemplo 6.1. Sea Π el programa lógico definido mostrado en el ejemplo 2.11 que replicamos aquí por conveniencia utilizando la notación usual de listas:

$$\Pi = \begin{cases} C_1 : & \text{append}([], x_1, x_1) & \leftarrow \\ C_2 : & \text{append}([x_1|x_2], x_3, [x_1|x_4]) & \leftarrow \text{append}(x_2, x_3, x_4) \end{cases}$$

El desplegado clásico de la cláusula C_2 (respecto a las cláusulas C_1 y C_2) en el programa Π produce el siguiente programa lógico definido:

$$\Pi' = \begin{cases} C_1 : & \text{append}([], x_1, x_1) & \leftarrow \\ C_{2.1} : & \text{append}([x_1], x_2, [x_1|x_2]) & \leftarrow \\ C_{2.2} : & \text{append}([x_1, x_2|x_3], x_4, [x_1, x_2|x_5]) & \leftarrow \text{append}(x_3, x_4, x_5) \end{cases}$$

Aunque la regla de desplegado aplicada en programación lógica pura es simple, aparecen problemas cuando se consideran programas Prolog reales en los que intervienen estructuras de control y otros predicados extralógicos. En [Pre93, Sah93, PAH05] se proponen distintas aproximaciones para eliminar la mayoría de estos problemas y simplificar el proceso de desplegado en tales programas.

En el contexto de la programación lógica difusa, la regla de desplegado clásica fue adaptada a una variante difusa de la programación lógica en [JIMP05b]. Además, también ha sido adaptada al marco de la programación lógica multi-adjunta en [JIMP05a, JIMP06, JIMM⁺13], que cuenta con retículos de grados de verdad y unificación sintáctica pero carece de relaciones de similitud.

En cuanto a la fundamentación teórica de la transformación de programas, el problema principal es determinar las condiciones bajo las cuales la técnica de transformación empleada es correcta y completa. Desde el punto de vista de la semántica operacional, *corrección* significa que las respuestas computadas por el programa transformado son respuestas computadas también por el programa original. La *completitud* se corresponde con el concepto inverso complementario del anterior, es decir, que las respuestas computadas por el programa original son respuestas computadas también por el programa desplegado. A lo largo de este capítulo estableceremos los resultados de corrección y completitud de la transformación de desplegado para programas FASILL.

6.2. Transformación de desplegado para programas FASILL

Comenzamos esta sección adaptando de forma ingenua a FASILL la regla de desplegado difuso originalmente definida en [JIMP05a] para MALP.

Definición 6.1 (Desplegado difuso basado en similitud, [JIMR22a]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL. Dada una regla $R : (H \leftarrow B) \in \Pi$ con un cuerpo no vacío, el *desplegado* de R en el programa \mathcal{P} es el nuevo programa $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$, donde $\Pi' = (\Pi - \{R\}) \cup \{H\sigma \leftarrow B' \mid \langle B, id \rangle \rightsquigarrow \langle B', \sigma \rangle\}$.

El siguiente ejemplo ilustra cómo desplegar programas FASILL en base a la definición previa.

Ejemplo 6.2. Sea $\mathcal{P}_0 = \langle \Pi_0, \mathcal{R}_0, L \rangle$ el programa FASILL mostrado en el ejemplo 3.1 cuyas reglas se enumeran a continuación:

$$\Pi_0 = \left\{ \begin{array}{ll} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_4 : \text{good_hotel}(x) & \leftarrow @_{\text{aver}}(@_{\text{very}}(\text{close}(x, y)), \text{cheap}(y)) \end{array} \right.$$

El desplegado de la regla R_4 (respecto a las reglas R_2 y R_3) en \mathcal{P}_0 lleva al programa transformado $\mathcal{P}'_0 = \langle \Pi'_0, \mathcal{R}_0, L \rangle$, donde

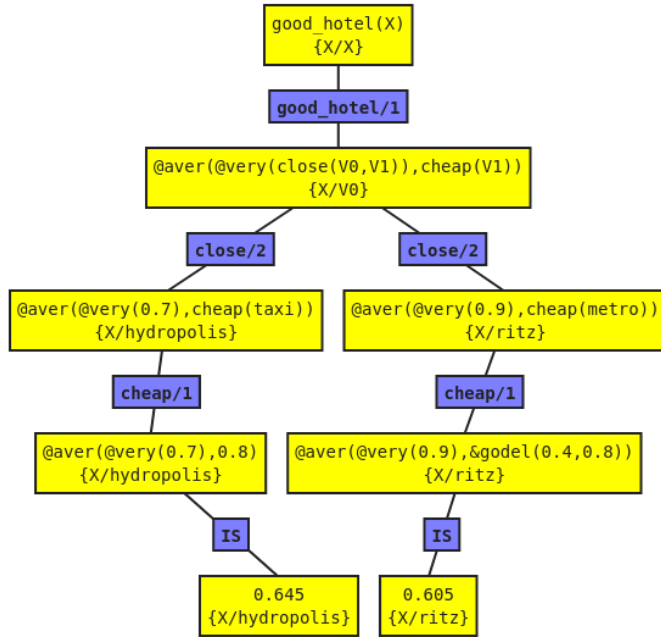
$$\Pi'_0 = \left\{ \begin{array}{ll} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_{4.2} : \text{good_hotel}(\text{hydropolis}) & \leftarrow @_{\text{aver}}(@_{\text{very}}(0.7), \text{cheap}(\text{taxi})) \\ R_{4.3} : \text{good_hotel}(\text{ritz}) & \leftarrow @_{\text{aver}}(@_{\text{very}}(0.9), \text{cheap}(\text{metro})) \end{array} \right.$$

Tras un paso de desplegado podemos observar que el programa \mathcal{P}'_0 lleva a las mismas respuestas computadas difusas que el programa \mathcal{P}_0 para un objetivo como $\mathcal{G}_0 = \text{good_hotel}(x)$:

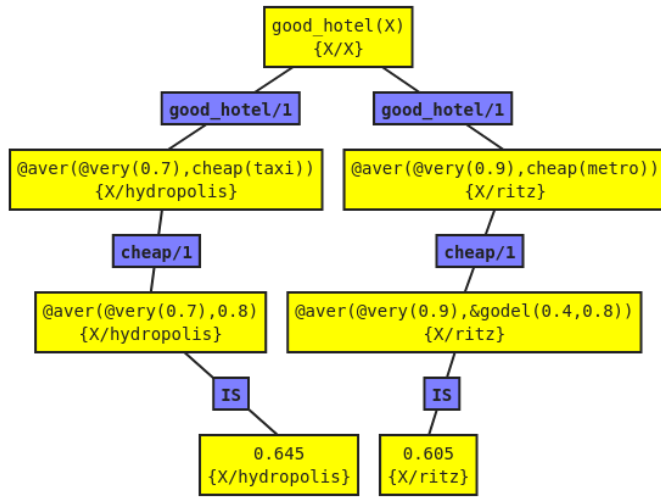
$$\begin{aligned} \mathcal{D}_0^{\mathcal{P}'_0} : \langle \underline{\text{good_hotel}(x)}, id \rangle & \rightsquigarrow_{SS}^{R_{4.2}} \\ & \langle @_{\text{aver}}(@_{\text{very}}(0.7), \underline{\text{cheap}(\text{taxi})}), \{x/\text{hydropolis}\} \rangle \rightsquigarrow_{SS}^{R_1} \\ & \langle @_{\text{aver}}(@_{\text{very}}(0.7), 0.8), \{x/\text{hydropolis}\} \rangle \rightsquigarrow_{IS}^* \\ & \langle 0.645, \{x/\text{hydropolis}\} \rangle \end{aligned}$$

Aparte de esta, existe otra derivación para el objetivo \mathcal{G}_0 con respuesta computada difusa $\langle 0.605, \{x/\text{ritz}\} \rangle$. La figura 6.1 muestra el árbol de derivación para el objetivo \mathcal{G}_0 ejecutado en el programa original y en su versión desplegada.

El ejemplo 6.2 ilustra que los programas desplegados producen derivaciones más cortas a la vez que preservan las respuestas computadas difusas, como buscamos. De hecho, partiendo de un programa inicial podemos construir una secuencia de transformaciones donde cada programa FASILL en la secuencia se obtiene por desplegado de una regla a partir del programa anterior.



(a) Árbol de derivación para \mathcal{G}_0 en \mathcal{P}_0 .



(b) Árbol de derivación para \mathcal{G}_0 en \mathcal{P}'_0 .

Figura 6.1: Comparación de los árboles de derivación para la ejecución de un mismo objetivo en el programa original y en el programa desplegado.

Ejemplo 6.3. Podemos seguir aplicando pasos de desplegado sobre el programa \mathcal{P}'_0 obtenido por desplegado en el ejemplo 6.2. El desplegado de la regla $R_{4.2}$ (respecto a la regla R_1) en \mathcal{P}'_0 lleva al programa transformado $\mathcal{P}_0^{(2)} = \langle \Pi_0^{(2)}, \mathcal{R}_0, L \rangle$:

$$\Pi_0^{(2)} = \left\{ \begin{array}{ll} R_1 : & \textit{cheap}(\textit{taxi}) \quad \leftarrow \quad 0.8 \\ R_2 : & \textit{close}(\textit{hydropolis}, \textit{taxi}) \quad \leftarrow \quad 0.7 \\ R_3 : & \textit{close}(\textit{ritz}, \textit{metro}) \quad \leftarrow \quad 0.9 \\ R_{4.2.1} : & \textit{good_hotel}(\textit{hydropolis}) \quad \leftarrow \quad @_{aver}(@_{very}(0.7), 0.8) \\ R_{4.3} : & \textit{good_hotel}(\textit{ritz}) \quad \leftarrow \quad @_{aver}(@_{very}(0.9), \textit{cheap}(\textit{metro})) \end{array} \right.$$

Ahora, el desplegado de la regla $R_{4.3}$ (respecto a la regla R_1) en $\mathcal{P}_0^{(2)}$ lleva al programa transformado $\mathcal{P}_0^{(3)} = \langle \Pi_0^{(3)}, \mathcal{R}_0, L \rangle$:

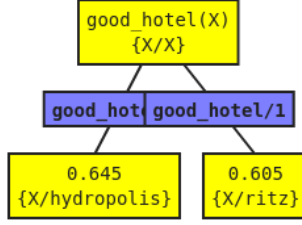
$$\Pi_0^{(3)} = \left\{ \begin{array}{ll} R_1 : & \textit{cheap}(\textit{taxi}) \quad \leftarrow \quad 0.8 \\ R_2 : & \textit{close}(\textit{hydropolis}, \textit{taxi}) \quad \leftarrow \quad 0.7 \\ R_3 : & \textit{close}(\textit{ritz}, \textit{metro}) \quad \leftarrow \quad 0.9 \\ R_{4.2.1} : & \textit{good_hotel}(\textit{hydropolis}) \quad \leftarrow \quad @_{aver}(@_{very}(0.7), 0.8) \\ R_{4.3.1} : & \textit{good_hotel}(\textit{ritz}) \quad \leftarrow \quad @_{aver}(@_{very}(0.9), \&_{godel}(0.8, 0.4)) \end{array} \right.$$

Tras aplicar varios pasos de desplegado basados en pasos interpretativos, obtenemos el programa transformado $\mathcal{P}_0^{(8)} = \langle \Pi_0^{(8)}, \mathcal{R}_0, L \rangle$:

$$\Pi_0^{(8)} = \left\{ \begin{array}{ll} R_1 : & \textit{cheap}(\textit{taxi}) \quad \leftarrow \quad 0.8 \\ R_2 : & \textit{close}(\textit{hydropolis}, \textit{taxi}) \quad \leftarrow \quad 0.7 \\ R_3 : & \textit{close}(\textit{ritz}, \textit{metro}) \quad \leftarrow \quad 0.9 \\ R_{4.2.1.is} : & \textit{good_hotel}(\textit{hydropolis}) \quad \leftarrow \quad 0.645 \\ R_{4.3.1.is} : & \textit{good_hotel}(\textit{ritz}) \quad \leftarrow \quad 0.605 \end{array} \right.$$

En este punto no es posible aplicar más pasos de desplegado sobre el programa $\mathcal{P}_0^{(8)}$, ya que todas las reglas se han transformado en hechos. Es fácil ver que este programa lleva a las mismas respuestas computadas difusas que \mathcal{P}_0 para el objetivo $\mathcal{G}_0 = \textit{good_hotel}(x)$ en un solo paso de computación (véase la figura 6.2).

Sin embargo, la transformación de desplegado descrita en la definición 6.1 para programas FASILL no es segura en general, en el sentido de que el programa desplegado no siempre lleva a las mismas respuestas computadas difusas que el programa original, tal y como mostramos a continuación. En particular, encontramos dos fuentes de problemas de corrección y completitud: una debida a las relaciones de similitud asociadas al programa y otra debida a la semántica de los pasos de fallo.


 Figura 6.2: Árbol de derivación para el objetivo \mathcal{G}_0 en $\mathcal{P}_0^{(8)}$.

6.2.1. Problemas de corrección debidos a relaciones de similitud

El siguiente ejemplo, que considera un objetivo básico (sin variables), revela un problema de corrección en la transformación de desplegado al reportar diferentes grados de verdad en las respuestas computadas difusas para el objetivo ejecutado en el programa original y en el programa desplegado.

Ejemplo 6.4. Dado el objetivo $\mathcal{G}_1 = \text{good_hotel}(\text{atlantis})$, las siguientes derivaciones son posibles en los programas \mathcal{P}_0 y \mathcal{P}'_0 mostrados en el ejemplo 6.2:

$$\begin{aligned}
 \mathcal{D}_1^{\mathcal{P}_0} : & \langle \text{good_hotel}(\text{atlantis}), id \rangle && \rightsquigarrow_{SS}^{R_4} \\
 & \langle @_{aver}(@_{very}(\underline{\text{close}}(\text{atlantis}, y_1)), \underline{\text{cheap}}(y_1)), \{y/y_1\} \rangle && \rightsquigarrow_{SS}^{R_3} \\
 & \langle @_{aver}(@_{very}(\&_{godel}(0.6, 0.9)), \underline{\text{cheap}}(\text{metro})), \{y/\text{metro}\} \rangle && \rightsquigarrow_{SS}^{R_1} \\
 & \langle @_{aver}(@_{very}(\&_{godel}(0.6, 0.9)), \&_{godel}(0.4, 0.8)), \{y/\text{metro}\} \rangle && \rightsquigarrow_{IS}^* \\
 & \langle 0.38, \{y/\text{metro}\} \rangle && \\
 \\
 \mathcal{D}_1^{\mathcal{P}'_0} : & \langle \text{good_hotel}(\text{atlantis}), id \rangle && \rightsquigarrow_{SS}^{R_4, 2} \\
 & \langle \&_{godel}(0.6, @_{aver}(@_{very}(0.9), \underline{\text{cheap}}(\text{metro}))), id \rangle && \rightsquigarrow_{SS}^{R_3} \\
 & \langle \&_{godel}(0.6, @_{aver}(@_{very}(0.9), \&_{godel}(0.4, 0.8))), id \rangle && \rightsquigarrow_{IS}^* \\
 & \langle 0.6, id \rangle &&
 \end{aligned}$$

Aquí observamos que el programa original lleva a la respuesta computada difusa $\langle 0.38, id \rangle$ para el objetivo \mathcal{G}_1 , mientras que el programa desplegado lleva a la f.c.a. $\langle 0.6, id \rangle$.

Intuitivamente, este problema surge cuando la transformación de desplegado liga una variable de la cabeza de la regla desplegada a un término que “contiene similitudes”,¹ ya que el grado de similitud de dicho término no queda reflejado en el mismo lugar en el programa original y en el programa desplegado cuando

¹Esto es, el término contiene algún símbolo que se relaciona con otros –con grado de similitud distinto de \perp – en la relación de similitud asociada al programa.

la regla es invocada con un término similar. Más aún, si la variable instanciada aparece más de una vez en el cuerpo de la regla desplegada, la similitud se explota más de una vez en el programa original, pero sólo una vez en el programa desplegado, como ilustra el siguiente ejemplo.

Ejemplo 6.5. Sea $\mathcal{P}_0 = \langle \Pi_0, \mathcal{R}_0, L \rangle$ el programa FASILL mostrado en el ejemplo 3.1. Considérese un nuevo programa $\mathcal{P}_1 = \langle \Pi_1, \mathcal{R}_1, L \rangle$, donde $\mathcal{R}_1 = \mathcal{R}_0$ y

$$\Pi_1 = \Pi_0 \cup \{R_5 : \text{very_cheap}(x) \leftarrow \text{cheap}(x) \ \&_{\text{prod}} \ \text{cheap}(x)\}.$$

El despliegado de la regla R_5 (respecto a la regla R_1) en el programa \mathcal{P}_1 produce el programa $\mathcal{P}'_1 = \langle \Pi'_1, \mathcal{R}_1, L \rangle$, donde

$$\Pi'_1 = \Pi_0 \cup \{R_{5.1} : \text{very_cheap}(taxi) \leftarrow 0.8 \ \&_{\text{prod}} \ \text{cheap}(taxi)\}.$$

Entonces, las siguientes derivaciones son posibles en \mathcal{P}_1 y \mathcal{P}'_1 para el objetivo $\mathcal{G}_2 = \text{very_cheap}(bus)$:

$$\begin{aligned} \mathcal{D}_2^{\mathcal{P}_1} : \quad & \langle \text{very_cheap}(bus), id \rangle && \rightsquigarrow_{SS}^{R_5} \\ & \langle \&_{\text{prod}}(\text{cheap}(bus), \text{cheap}(bus)), id \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle \&_{\text{prod}}(\&_{\text{godel}}(0.8, 0.4), \text{cheap}(bus)), id \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle \&_{\text{prod}}(\&_{\text{godel}}(0.8, 0.4), \&_{\text{godel}}(0.8, 0.4)), id \rangle && \rightsquigarrow_{IS}^* \\ & \langle 0.16, id \rangle && \\ \\ \mathcal{D}_2^{\mathcal{P}'_1} : \quad & \langle \text{very_cheap}(bus), id \rangle && \rightsquigarrow_{SS}^{R_{5.1}} \\ & \langle \&_{\text{godel}}(0.4, \&_{\text{prod}}(0.8, \text{cheap}(taxi))), id \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle \&_{\text{godel}}(0.4, \&_{\text{prod}}(0.8, 0.8)), id \rangle && \rightsquigarrow_{IS}^* \\ & \langle 0.4, id \rangle && \end{aligned}$$

Como se observa en la derivación del programa original, la similitud entre los símbolos *taxi* y *bus* es explotada dos veces, mientras que en el programa desplegado solo se explota una vez al unificar el átomo del objetivo con la cabeza de la regla desplegada.

Este ejemplo refleja claramente que la transformación de despliegado modifica la semántica del programa original cuando intervienen relaciones de similitud. Con el fin de definir una condición suficiente para evitar este problema, introducimos algunos conceptos auxiliares.

Definición 6.2 (Término límite, [JIMR22a]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL sobre un alfabeto Σ . Un *término límite* se define inductivamente conforme a las siguientes reglas:

- (1) Toda variable es un término límite.
- (2) Si a es una constante, tal que para cualquier símbolo $b \in \Sigma$, $\mathcal{R}(a, b) = \perp$ o $\mathcal{R}(a, b) = \top$, entonces a es un término límite.

- (3) Si $f^n(t_1, \dots, t_n)$ es un término, tal que para cualquier símbolo n -ario $g^n \in \Sigma$, $\mathcal{R}(f^n, g^n) = \perp$ o $\mathcal{R}(f^n, g^n) = \top$, y t_1, \dots, t_n son términos límite, entonces $f^n(t_1, \dots, t_n)$ también lo es.

Definición 6.3 (Sustitución límite, [JIMR22a]). Sea θ una sustitución. θ es una *sustitución límite* si y solo si para todo término $t \in \text{ran}(\theta)$, t es un término límite.

Ahora, podemos formular una condición para desplegar programas FASILL de forma segura con el fin de preservar la corrección de la transformación al garantizar que sólo los términos límite se pueden involucrar en tiempo de desplegado a la hora de instanciar la cabeza de una regla desplegada.

Definición 6.4 (Condición de similitud límite, [JIMR22a]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL, $R : (H \leftarrow B) \in \mathcal{P}$ una regla del programa, y $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$ el programa FASILL transformado tras el desplegado de la regla R en el programa \mathcal{P} : $\Pi' = (\Pi - \{R\}) \cup \{H\sigma \leftarrow B' \mid \langle B, id \rangle \rightsquigarrow \langle B', \sigma \rangle\}$. Decimos que la regla R verifica la *condición de similitud límite* si y solo si toda sustitución σ verifica que $\sigma[\text{vars}(H)]$ es una sustitución límite.

Ejemplo 6.6. Sea $\mathcal{P}_0 = \langle \Pi_0, \mathcal{R}_0, L \rangle$ el programa FASILL mostrado en el ejemplo 3.1, y $\mathcal{P}'_0 = \langle \Pi'_0, \mathcal{R}_0, L \rangle$ su versión desplegada en el ejemplo 6.2. Considérese un nuevo programa $\mathcal{P}_2 = \langle \Pi_2, \mathcal{R}_2, L \rangle$, donde $\Pi_2 = \Pi_0$ y \mathcal{R}_2 es una modificación de \mathcal{R}_0 , donde los términos no-límite que causan problemas en el ejemplo 6.4 (*ritz* y *atlantis*) pasan a ser términos límite, $\mathcal{R}_2(\text{ritz}, \text{atlantis}) = 1.0$. Nótese que \mathcal{P}_2 difiere ligeramente de \mathcal{P}_0 , pero ahora la regla R_4 preserva la condición de similitud límite. Sea $\mathcal{P}'_2 = \langle \Pi'_0, \mathcal{R}_2, L \rangle$ el programa obtenido por desplegado de la regla R_4 en \mathcal{P}_2 . Ahora, las siguientes derivaciones –que llevan a la misma respuesta computada difusa– son posibles en \mathcal{P}_2 y en \mathcal{P}'_2 para el objetivo $\mathcal{G}_1 = \text{good_hotel}(\text{atlantis})$:

$$\begin{aligned}
 \mathcal{D}_1^{\mathcal{P}_2} : & \langle \text{good_hotel}(\text{atlantis}), id \rangle && \rightsquigarrow^{R_4} \\
 & \langle @\text{aver}(@\text{very}(\underline{\text{close}(\text{atlantis}, y_1)}), \text{cheap}(y_1)), \{y/y_1\} \rangle && \rightsquigarrow^{R_3} \\
 & \langle @\text{aver}(@\text{very}(\&\text{godel}(1.0, 0.9)), \underline{\text{cheap}(\text{metro})}), \{y/\text{metro}\} \rangle && \rightsquigarrow^{R_1} \\
 & \langle @\text{aver}(@\text{very}(\&\text{godel}(1.0, 0.9)), \&\text{godel}(0.4, 0.8)), \{y/\text{metro}\} \rangle && \rightsquigarrow^*_{IS} \\
 & (0.605, \{y/\text{metro}\})
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{D}'_1^{\mathcal{P}'_2} : & \langle \text{good_hotel}(\text{atlantis}), id \rangle && \rightsquigarrow^{R_{4,2}} \\
 & \langle \&\text{godel}(1.0, @\text{aver}(@\text{very}(0.9), \underline{\text{cheap}(\text{metro})})), id \rangle && \rightsquigarrow^{R_3} \\
 & \langle \&\text{godel}(1.0, @\text{aver}(@\text{very}(0.9), \underline{\&\text{godel}(0.4, 0.8)})), id \rangle && \rightsquigarrow^*_{IS} \\
 & (0.605, id)
 \end{aligned}$$

Como se esperaba, cuando *ritz* y *atlantis* se convierten en términos límite, los programas \mathcal{P}_2 y \mathcal{P}'_2 proporcionan las mismas respuestas computadas difusas para el objetivo \mathcal{G}_1 .

6.2.2. Problemas de completitud debidos a pasos de fallo

El siguiente ejemplo revela un problema de completitud en la transformación de desplegado, al reportar diferentes grados de verdad en las respuestas computadas difusas para un objetivo ejecutado en el programa original y en el programa desplegado, cuando intervienen pasos de fallo en las derivaciones.

Ejemplo 6.7. Sean \mathcal{P}_0 y \mathcal{P}'_0 los programas FASILL mostrados en el ejemplo 6.2. Considérese el objetivo $\mathcal{G}_3 = \text{good_hotel}(\text{senator})$ en \mathcal{P}_0 y \mathcal{P}'_0 :

$$\begin{array}{ll}
 \mathcal{D}_3^{\mathcal{P}_0} : & \langle \text{good_hotel}(\text{senator}), id \rangle \quad \rightsquigarrow_{SS}^{R_4} \\
 & \langle @_{aver}(@_{very}(\text{close}(\text{senator}, y_1)), \text{cheap}(y_1)), \{y/y_1\} \rangle \quad \rightsquigarrow_{FS} \\
 & \langle @_{aver}(@_{very}(0.0), \text{cheap}(y_1)), \{y/y_1\} \rangle \quad \rightsquigarrow_{SS}^{R_1} \\
 & \langle @_{aver}(@_{very}(0.0), 0.8), \{y/taxi\} \rangle \quad \rightsquigarrow_{IS}^* \\
 & \langle 0.4, id \rangle \\
 \\
 \mathcal{D}_3^{\mathcal{P}'_0} : & \langle \text{good_hotel}(\text{senator}), id \rangle \quad \rightsquigarrow_{FS} \\
 & \langle 0.0, id \rangle
 \end{array}$$

Aquí observamos que el programa original lleva a la respuesta computada difusa $\langle 0.4, id \rangle$ para el objetivo \mathcal{G}_3 , mientras que en el programa desplegado falla.

Este ejemplo pone de relieve que instanciar las cabezas de las reglas desplegadas, en general, puede provocar pasos de fallo prematuros en tiempo de ejecución. La siguiente condición ayuda a prevenir esta situación.

Definición 6.5 (Condición de preservación de cabezas, [JIMR22a]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL, $R : (H \leftarrow B) \in \mathcal{P}$ una regla del programa, y $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$ el programa FASILL transformado tras el desplegado de la regla R en el programa \mathcal{P} : $\Pi' = (\Pi - \{R\}) \cup \{H\sigma \leftarrow B' \mid \langle B, id \rangle \rightsquigarrow \langle B', \sigma \rangle\}$. Decimos que la regla R verifica la *condición de preservación de cabezas* si y solo si toda cabeza $H\sigma$ coincide (hasta renombramiento de variables) con H .

Ejemplo 6.8. Sea $\mathcal{P}_0 = \langle \Pi_0, \mathcal{R}_0, L \rangle$ el programa FASILL mostrado en el ejemplo 3.1. Considérese un nuevo programa $\mathcal{P}_4 = \langle \Pi_4, \mathcal{R}_4, L \rangle$, donde $\mathcal{R}_4 = \mathcal{R}_0$ y Π_4 es similar a Π_0 cambiando el orden de los átomos en la regla R_4 :

$$\Pi_4 = \left\{ \begin{array}{ll}
 R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\
 R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\
 R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\
 R_4 : \text{good_hotel}(x) & \leftarrow @_{aver}(\text{cheap}(y), @_{very}(\text{close}(x, y)))
 \end{array} \right.$$

Entonces, el desplegado de la regla R_4 (respecto a la regla R_1) en el programa

\mathcal{P}_4 lleva al programa transformado $\mathcal{P}'_4 = \langle \Pi'_4, \mathcal{R}_4, L \rangle$, donde:

$$\Pi'_4 = \begin{cases} R_1 : & \text{cheap}(taxi) & \leftarrow & 0.8 \\ R_2 : & \text{close}(\text{hydropolis}, taxi) & \leftarrow & 0.7 \\ R_3 : & \text{close}(\text{ritz}, metro) & \leftarrow & 0.9 \\ R_{4.1} : & \text{good_hotel}(x) & \leftarrow & @_{aver}(0.8, @_{very}(\text{close}(x, taxi))) \end{cases}$$

Dado que ahora la regla R_4 sí verifica la condición de preservación de cabezas, la transformación de desplegado es segura respecto al objetivo anterior $\mathcal{G}_3 = \text{good_hotel}(\text{senator})$ y las siguientes derivaciones (que producen las mismas respuestas computadas difusas) son posibles en \mathcal{P}_4 y \mathcal{P}'_4 para el objetivo \mathcal{G}_3 :

$$\begin{aligned} \mathcal{D}_3^{\mathcal{P}_4} : & \langle \underline{\text{good_hotel}(\text{senator})}, id \rangle && \rightsquigarrow_{SS}^{R_4} \\ & \langle @_{aver}(\text{cheap}(y_1), @_{very}(\text{close}(\text{senator}, y_1))), \{y/y_1\} \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle @_{aver}(0.8, @_{very}(\text{close}(\text{senator}, taxi))), \{y/taxi\} \rangle && \rightsquigarrow_{FS} \\ & \langle @_{aver}(0.8, @_{very}(0.0)), \{y/taxi\} \rangle && \rightsquigarrow_{IS}^* \\ & \langle 0.4, \{y/taxi\} \rangle && \\ \mathcal{D}_3^{\mathcal{P}'_4} : & \langle \underline{\text{good_hotel}(\text{senator})}, id \rangle && \rightsquigarrow_{SS}^{R_{4.1}} \\ & \langle @_{aver}(0.8, @_{very}(\text{close}(\text{senator}, taxi))), id \rangle && \rightsquigarrow_{FS} \\ & \langle @_{aver}(0.8, @_{very}(0.0)), id \rangle && \rightsquigarrow_{IS}^* \\ & \langle 0.4, id \rangle && \end{aligned}$$

La definición 6.5 puede parecer muy restrictiva incluso para reglas estándar como las mostradas en el ejemplo 6.1, que pueden ser desplegadas sin riesgo en lenguajes lógicos más simples como Prolog. Con el fin de relajar la definición previa, definimos una condición alternativa que habilita la instanciación de las cabezas de las reglas transformadas. En esencia, la siguiente definición comprueba que dada una regla $H \leftarrow B$, si eventualmente aplicamos un paso de fallo en el cuerpo de B , la expresión transformada B' se reduce a \perp sin instanciar las variables de H .

Definición 6.6 (Condición de anulación del cuerpo). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y $R : (H \leftarrow B) \in \Pi$ una regla del programa. Decimos que R satisface la *condición de anulación del cuerpo* si, para toda derivación \mathcal{D} posible para el objetivo $\mathcal{G} = B[A/\perp]$ en \mathcal{P} , tenemos que $\mathcal{D} : \langle B[A/\perp], id \rangle \rightsquigarrow^* \langle \perp, \sigma \rangle$ y $H\sigma$ coincide (hasta renombramiento de variables) con H .

Ejemplo 6.9. Sea $\mathcal{P}_0 = \langle \Pi_0, \mathcal{R}_0, L \rangle$ el programa FASILL mostrado en el ejemplo 3.1. Considérese un nuevo programa $\mathcal{P}_5 = \langle \Pi_5, \mathcal{R}_5, L \rangle$, donde $\mathcal{R}_5 = \mathcal{R}_0$ y Π_5 es similar a Π_0 , cambiando el agregador $@_{aver}$ por la t-norma $\&_{godel}$ en el cuerpo

de la regla R_4 :

$$\Pi_5 = \begin{cases} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_4 : \text{good_hotel}(x) & \leftarrow \&_{\text{godel}}(\text{@}_{\text{very}}(\text{close}(x, y)), \text{cheap}(y)) \end{cases}$$

Entonces, el desplegado de la regla R_4 (respecto a las reglas R_2 y R_3) en el programa \mathcal{P}_5 lleva al programa transformado $\mathcal{P}'_5 = \langle \Pi'_5, \mathcal{R}_4, L \rangle$, donde:

$$\Pi'_5 = \begin{cases} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_{4.2} : \text{good_hotel}(\text{hydropolis}) & \leftarrow \&_{\text{godel}}(\text{@}_{\text{very}}(0.7), \text{cheap}(\text{taxi})) \\ R_{4.3} : \text{good_hotel}(\text{ritz}) & \leftarrow \&_{\text{godel}}(\text{@}_{\text{very}}(0.9), \text{cheap}(\text{metro})) \end{cases}$$

Dado que ahora la regla R_4 sí verifica la condición de anulación del cuerpo, la transformación de desplegado es segura respecto al objetivo anterior $\mathcal{G}_3 = \text{good_hotel}(\text{senator})$ y las siguientes derivaciones (que producen las mismas respuestas computadas difusas) son posibles en \mathcal{P}_5 y \mathcal{P}'_5 para el objetivo \mathcal{G}_3 :

$$\begin{aligned} \mathcal{D}_3^{\mathcal{P}_5} : & \langle \text{good_hotel}(\text{senator}), \text{id} \rangle && \rightsquigarrow_{SS}^{R_4} \\ & \langle \&_{\text{godel}}(\text{@}_{\text{very}}(\text{close}(\text{senator}, y_1)), \text{cheap}(y_1)), \{y/y_1\} \rangle && \rightsquigarrow_{FS} \\ & \langle \&_{\text{godel}}(\text{@}_{\text{very}}(0.0), \text{cheap}(y_1)), \{y/y_1\} \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle \&_{\text{godel}}(\text{@}_{\text{very}}(0.0), 0.8), \{y/\text{taxi}\} \rangle && \rightsquigarrow_{IS}^* \\ & \langle 0.0, \{y/\text{taxi}\} \rangle && \\ \mathcal{D}_3^{\mathcal{P}'_5} : & \langle \text{good_hotel}(\text{senator}), \text{id} \rangle && \rightsquigarrow_{FS} \\ & \langle 0.0, \text{id} \rangle && \end{aligned}$$

Nótese que la segunda cláusula definida del programa mostrado en el ejemplo 6.1 también satisface esta última condición y, por lo tanto, puede ser desplegada de forma segura como buscábamos. Pero es importante remarcar que el poder expresivo de este código es mayor en nuestro entorno basado en similitudes que en lenguajes precedentes como Prolog o MALP. Por ejemplo, asumiendo que las constantes a , b y c son términos similares, un objetivo como $\text{append}([a, a], [b], [a, b, c])$ fallaría en Prolog o MALP, pero tendría éxito en el sistema FASILL.

6.3. Corrección y completitud de la transformación de desplegado

En esta sección demostramos formalmente las propiedades de corrección, completitud y eficiencia de los programas FASILL desplegados. Para ello introducimos primero la noción de desplegado seguro, que caracteriza las reglas del programa que pueden ser desplegadas sin modificar la semántica original del programa.

Definición 6.7 (Desplegado difuso seguro basado en similitud, [JIMR22a]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL, $R \in \Pi$ una regla del programa y \mathcal{P}' el desplegado basado en similitud de la regla R en el programa \mathcal{P} . Entonces, la transformación de desplegado es *segura* si R verifica la condición de similitud límite y al menos una de las dos condiciones: la preservación de cabezas o la anulación del cuerpo.

Es importante destacar que un mismo programa puede contener al mismo tiempo reglas que pueden ser desplegadas de forma segura y otras que no, como ilustra el siguiente ejemplo. Es decir, la noción de seguridad de la transformación de desplegado se da a nivel de regla, no de programa (aunque el resto de reglas del programa pueden influir en la seguridad del desplegado de una regla cuando este se basa en un paso de éxito \rightsquigarrow_{SS}).

Ejemplo 6.10. Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL sobre el retículo $([0, 1], \leq)$ donde:

$$\Pi = \left\{ \begin{array}{ll} R_1 : \text{cheap}(\text{taxi}) & \leftarrow 0.8 \\ R_2 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7 \\ R_3 : \text{close}(\text{ritz}, \text{metro}) & \leftarrow 0.9 \\ R_4 : \text{good_hotel}(x) & \leftarrow @_{\text{very}}(\text{close}(x, y)) \&_{\text{godel}} \text{cheap}(y) \\ R_5 : \text{very_cheap}(x) & \leftarrow \text{cheap}(x) \&_{\text{prod}} \text{cheap}(x) \end{array} \right.$$

y cuya relación de similitud \mathcal{R} queda determinada por la siguiente matriz (con la t-norma fija de Gödel):

\mathcal{R}	ritz	atlantis	metro	taxi	bus
ritz	1	1	0	0	0
atlantis	1	1	0	0	0
metro	0	0	1	0.4	0.5
taxi	0	0	0.4	1	0.4
bus	0	0	0.5	0.4	1

Entonces, la regla R_4 puede ser desplegada de forma segura (respecto a las reglas R_2 y R_3) ya que verifica las condiciones de similitud límite (en particular, los términos *hydropolis* y *ritz* son límite) y de anulación del cuerpo, aunque no la

de preservación de cabezas:

$$\Pi' = \begin{cases} R_1 : & \text{cheap}(taxi) & \leftarrow & 0.8 \\ R_2 : & \text{close}(hydropolis, taxi) & \leftarrow & 0.7 \\ R_3 : & \text{close}(ritz, metro) & \leftarrow & 0.9 \\ R_{4.2} : & \text{good_hotel}(hydropolis) & \leftarrow & @_{very}(0.7) \ \&_{godel} \ \text{cheap}(taxi) \\ R_{4.3} : & \text{good_hotel}(ritz) & \leftarrow & @_{very}(0.9) \ \&_{godel} \ \text{cheap}(metro) \\ R_5 : & \text{very_cheap}(x) & \leftarrow & \text{cheap}(x) \ \&_{prod} \ \text{cheap}(x) \end{cases}$$

Sin embargo, la regla R_5 no puede ser desplegada de forma segura (respecto a la regla R_1) ya que no verifica ninguna de estas condiciones.

Antes de demostrar las propiedades del desplegado, necesitamos algunos resultados intermedios sobre la aplicación de sustituciones límite.

Proposición 6.1 ([JIMR22a]). *Sean A y A' dos átomos tales que $\text{wmgu}_{\mathcal{R}}(A, A') \neq \text{fallo}$. Sea θ una sustitución límite idempotente. Si $\text{wmgu}_{\mathcal{R}}(A, A'\theta) \neq \text{fallo}$, entonces*

$$\text{deg}(A, A'\theta) = \text{deg}(A, A') = v.$$

Demostración. Para que A y A' sean débilmente unificables, para la misma posición p en A y en A' , si no hay ningún símbolo de variable debe haber dos átomos similares. Los grados de similitud de los símbolos de aquellas posiciones p en las que hay dos átomos son los únicos que contribuyen al grado de unificación v de A y A' . Llamamos a estas posiciones la *parte estable* de A y A' respecto a la unificación débil. Para otras posiciones $q \neq p$, hay un símbolo de variable en la posición q de A o en la posición q de A' , que genera un enlace utilizando la regla de transición **(3)** de la relación de unificación débil (figura 2.3), que no contribuye al grado de unificación.

Por otra parte, si A y $A'\theta$ son débilmente unificables, si en una posición p de A' hay una variable $x \in \text{dom}(\theta)$, en la misma posición p de A debe existir:

- (1) Una variable y que en tiempo de unificación genera un enlace $\{y/x\theta\}$ con grado de unificación \top . Nótese que $y \notin \text{vars}(x\theta)$ porque de otro modo A y $A'\theta$ no serían débilmente unificables, en contra de nuestra suposición inicial.
- (2) Un término t que debe unificar débilmente con $x\theta$. Dado que θ es una sustitución límite, el término $x\theta$ también es límite. Entonces, el grado de unificación de t y $x\theta$ debe ser $\text{deg}(t, x\theta) = \top$.

En cualquier caso, el paso de unificación siempre contribuye con grado de unificación \top , que es un elemento neutro para cualquier t-norma. Por lo tanto, el grado de unificación permanece intacto y queda únicamente determinado por la parte estable de A y A' . En consecuencia, el grado de unificación de $\text{wmgu}_{\mathcal{R}}(A, A'\theta)$ es v , que es a su vez el grado de unificación de $\text{wmgu}_{\mathcal{R}}(A, A')$. \square

Proposición 6.2 ([JIMR22a]). *Sea θ una sustitución idempotente (no necesariamente límite). Sean A y A' dos átomos tales que $\text{wmgur}_{\mathcal{R}}(A, A') = \langle \sigma, v \rangle$, donde $\sigma[\text{dom}(\theta)]$ es una sustitución límite idempotente. Si $\text{wmgur}_{\mathcal{R}}(A, A'\theta) \neq \text{fallo}$, entonces*

$$\text{deg}(A, A'\theta) = \text{deg}(A, A') = v.$$

Demostración. Esta proposición puede ser demostrada de manera análoga a la proposición 6.1, dado que $\sigma[\text{dom}(\theta)]$ es límite y, por lo tanto, para cada variable $x \in \text{dom}(\theta)$, donde $x\theta = t$, hay dos posibilidades:

- (1) t es un término límite y, por la proposición 6.1, $\{x/t\}$ no contribuye al grado de unificación de $\text{wmgur}_{\mathcal{R}}(A, A'\theta)$.
- (2) t es un término no-límite y, por lo tanto, la unificación débil de A y $A'\theta$ sólo tiene éxito si $x\sigma = s$ es una variable (dado que s es límite por hipótesis y un término límite no unifica con ningún término no-límite), en cuyo caso se genera un enlace en tiempo de unificación sin afectar al grado de unificación v .

□

Proposición 6.3 ([JIMR22a]). *Sean A, A', B y B' cuatro átomos tales que $\text{wmgur}_{\mathcal{R}}(A, A') = \alpha$ y $\text{wmgur}_{\mathcal{R}}(B, B') = \beta$, donde $\text{vars}(A) \cap \text{vars}(B) = \emptyset$. Entonces,*

$$\text{wmgur}_{\mathcal{R}}(B\alpha, B') = \text{fallo} \iff \text{wmgur}_{\mathcal{R}}(A, A'\beta) = \text{fallo}.$$

Demostración.

(\Rightarrow) Para que B y B' sean débilmente unificables, y no lo sean $B\alpha$ y B' , debe existir una posición p tal que en la posición p de B hay un símbolo de variable $x \in \text{dom}(\alpha)$, donde $x\alpha = t$ es un término y $\text{wmgur}_{\mathcal{R}}(B\{x/t\}, B') = \text{fallo}$. Entonces, $x\beta = s$ debe verificar que $\text{wmgur}_{\mathcal{R}}(t, s) = \text{fallo}$. Dado que $\text{vars}(A) \cap \text{vars}(B) = \emptyset$ y $x \in \text{dom}(\alpha)$, x debe aparecer en A' en alguna posición, y $\text{wmgur}_{\mathcal{R}}(A, A'\{x/s\}) = \text{fallo}$. Por lo tanto, $\text{wmgur}_{\mathcal{R}}(A, A'\beta) = \text{fallo}$.

(\Leftarrow) El caso recíproco puede ser demostrado de forma análoga.

□

Intuitivamente, el siguiente lema muestra que, incluso en el caso de que dos pasos de éxito no puedan ser intercambiados, como el segundo paso explota un átomo introducido tras el primer paso, su efecto (respecto a las sustituciones de las respuestas computadas difusas) puede ser simulado por un único paso utilizando una regla transformada obtenida por desplegado.

Lema 6.1 ([JIMR22a]). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL con una t -norma fija \wedge , \mathcal{G}_0 un objetivo y $R_1, R_2 \in \Pi$ dos reglas del programa. Entonces,*

$$\begin{aligned} \mathcal{D} : \langle \mathcal{G}_0, \theta_0 \rangle &\rightsquigarrow_{SS}^{R_1} \langle \mathcal{G}_1, \theta_0 \theta_1 \rangle \rightsquigarrow_{SS}^{R_2} \langle \mathcal{G}_2, \theta_0 \theta_1 \theta_2 \rangle \\ &\text{si y solo si,} \\ \mathcal{D}' : \langle \mathcal{G}_0, \theta_0 \rangle &\rightsquigarrow_{SS}^{R_3} \langle \mathcal{G}_3, \theta_0 \theta_3 \rangle \end{aligned}$$

y, además, $\theta_0 \theta_1 \theta_2 = \theta_0 \theta_3 [\text{vars}(\mathcal{G}_0)]$, donde el segundo paso en \mathcal{D} explota un átomo introducido en \mathcal{G}_1 tras el primer paso, y siendo R_3 una regla obtenida por desplegado (del átomo A_1 en el cuerpo de) R_1 utilizando R_2 .

Demostración.

(\Rightarrow) Sea $R_1 : H_1 \leftarrow B_1[A_1]$. Sea H_2 la cabeza de la regla R_2 , y supongamos que A es el átomo seleccionado en \mathcal{G}_0 . Entonces, en la primera derivación \mathcal{D} tenemos que: $\theta_1 = \text{wmgu}_{\mathcal{R}}(A, H_1)$ y $\mathcal{G}_1 = \mathcal{G}_0[A/(v_1 \wedge B_1[A_1])]\theta_1$. Además, si $A_1 \theta_1$ es el átomo seleccionado en \mathcal{G}_1 , $\theta_2 = \text{wmgu}_{\mathcal{R}}(A_1 \theta_1, H_2)$. Ahora, considérese $\sigma = \text{wmgu}_{\mathcal{R}}(A_1, H_2)$. Entonces, se dan las siguientes igualdades:

$$\begin{aligned} \theta_1 \theta_2 &= \\ \theta_1 \text{wmgu}_{\mathcal{R}}(A_1 \theta_1, H_2) &= \text{(dado que } \text{dom}(\theta_1) \cap \text{vars}(R_2) = \emptyset\text{)} \\ \theta_1 \text{wmgu}_{\mathcal{R}}(\widehat{\text{wmgu}}_{\mathcal{R}}(A_1, H_2) \theta_1) &= \\ \theta_1 \text{wmgu}_{\mathcal{R}}(\widehat{\sigma} \theta_1) &= \text{(por la proposición 2.5)} \\ \theta_1 \uparrow \sigma &= \text{(por la proposición 2.5)} \\ \sigma \text{wmgu}_{\mathcal{R}}(\widehat{\theta}_1 \sigma) &= \\ \sigma \text{wmgu}_{\mathcal{R}}(\widehat{\text{wmgu}}_{\mathcal{R}}(A, H_1) \sigma) &= \text{(dado que } \text{dom}(\sigma) \cap \text{vars}(\mathcal{G}_0) = \emptyset\text{)} \\ \sigma \text{wmgu}_{\mathcal{R}}(A, H_1 \sigma) & \end{aligned}$$

Más aún, dado que $\theta_1 \theta_2$ puede ser computado sin que se produzca un fallo, entonces σ también podrá computarse (es decir, $\text{wmgu}_{\mathcal{R}}(A_1, H_2) \neq \text{fallo}$) y existe una regla R_3 obtenida por desplegado (del átomo A_1 en el cuerpo de) R_1 utilizando R_2 , tal que la cabeza de R_3 es el átomo $H_1 \sigma$. Ahora, dado que $\text{wmgu}_{\mathcal{R}}(A, H_1 \sigma) \neq \text{fallo}$, puede darse el siguiente paso de éxito sobre el átomo seleccionado A en \mathcal{G}_0 : $\langle \mathcal{G}_0, \theta_0 \rangle \rightsquigarrow_{SS}^{R_3} \langle \mathcal{G}_3, \theta_0 \theta_3 \rangle$, donde $\theta_3 = \text{wmgu}_{\mathcal{R}}(A, H_1 \sigma)$. Finalmente, dado que $\theta_1 \theta_2 = \sigma \theta_3$, entonces $\theta_0 \theta_1 \theta_2 = \theta_0 \sigma \theta_3$, y como $\text{dom}(\sigma) \cap \text{vars}(\mathcal{G}_0) = \emptyset$ y $\text{dom}(\sigma) \cap \text{dom}(\theta_0) = \emptyset$, tenemos que $\theta_0 \theta_1 \theta_2 = \theta_0 \theta_3 [\text{vars}(\mathcal{G}_0)]$, como queríamos demostrar.

(\Leftarrow) Este caso puede ser demostrado de forma similar al caso anterior, explotando la equivalencia entre $\theta_1 \theta_2$ y $\sigma \theta_3$.

□

En lo que sigue, establecemos los resultados de corrección y completitud para el desplegado difuso basado en similitud que, en este contexto, implican que la transformación preserva la igualdad sintáctica entre las respuestas computadas difusas de los programas involucrados.

Teorema 6.1 (Corrección del desplegado difuso basado en similitud, [JIMR22a]). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y \mathcal{G} un objetivo. Si $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$ es el programa obtenido por desplegado seguro (véase la definición 6.7) de \mathcal{P} , entonces:*

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta \rangle \text{ en } \mathcal{P}$$

si

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta' \rangle \text{ en } \mathcal{P}'$$

donde $v \in L$ y $\theta = \theta'[vars(\mathcal{G})]$.

Demostración. Sea $\mathcal{D}' : \langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta \rangle$ la derivación (genérica) para el objetivo \mathcal{G} en el programa \mathcal{P}' que pretendemos simular construyendo una nueva derivación para \mathcal{G} en \mathcal{P} . La construcción de \mathcal{D} se hace por inducción sobre la longitud k de \mathcal{D}' . Dado que el caso base ($k = 0$) es trivial, procedemos con el caso general cuando $k > 0$. Entonces, $\mathcal{D}' : \langle \mathcal{G}, id \rangle \rightsquigarrow \langle \mathcal{G}', \vartheta \rangle \rightsquigarrow^* \langle v, \theta' \rangle$. Si el primer paso de \mathcal{D}' ha sido realizado con un paso interpretativo de la definición 3.3 o incluso si ha sido realizado con un paso de éxito utilizando una regla $R \in \Pi$, entonces se sigue el resultado por la hipótesis de inducción. En otro caso, el paso inicial ha sido realizado con un paso de fallo o con un paso de éxito utilizando una regla R' que ha sido obtenida por desplegado de otra regla $R \in \Pi$. Dado que el paso de desplegado ha sido realizado con uno de los tres tipos de pasos de la definición 3.3, consideramos cada caso por separado.

(1) Desplegado basado en un paso de éxito.

Sean $R_1 : (H_1 \leftarrow B_1[A_1]) \in \Pi$ y $R_2 : (H_2 \leftarrow B_2) \in \Pi$ tales que, por desplegado de R_1 respecto a R_2 utilizando un paso de éxito de la definición 3.3, obtenemos $R' : (H_1\sigma \leftarrow B_1[A_1/(v'_2 \wedge B_2)]\sigma) \in \Pi'$, donde $wmgu_{\mathcal{R}}(A_1, H_2) = \langle \sigma, v'_2 \rangle$. Entonces, existen dos posibles derivaciones:

a) El primer paso en \mathcal{D}' es un paso de éxito.

Asumimos que A es el átomo seleccionado en \mathcal{G} , y

$$\begin{aligned} \mathcal{D}' : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^{R'} \\ & \langle (\mathcal{G}[A/(v'_1 \wedge B_1[A_1/(v'_2 \wedge B_2)])]\sigma\gamma, \sigma\gamma) \rangle \rightsquigarrow^* \\ & \langle v, \theta' \rangle \end{aligned}$$

donde $wmgu_{\mathcal{R}}(A, H_1\sigma) = \langle \gamma, v'_1 \rangle$. Ahora, el primer paso de \mathcal{D}' puede ser simulado en la derivación \mathcal{D} utilizando las reglas $R_1, R_2 \in \Pi$ como sigue:

$$\begin{aligned} \mathcal{D} : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^{R_1} \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1]])\alpha, \alpha) \rangle \rightsquigarrow_{SS}^{R_2} \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1/(v_2 \wedge B_2)])]\alpha\beta, \alpha\beta) \rangle \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$ y $\text{wmgur}_{\mathcal{R}}(A_1\alpha, H_2) = \langle \beta, v_2 \rangle$. Por las proposiciones 6.1 y 6.2 se dan respectivamente las igualdades $v_1 = v'_1$ y $v_2 = v'_2$, y por el lema 6.1 concluimos que $\alpha\beta = \sigma\gamma[\text{vars}(\mathcal{G})]$, y por lo tanto el tercer estado en \mathcal{D} coincide sintácticamente con el segundo estado en \mathcal{D}' . Además, por la hipótesis de inducción $\theta = \theta'[\text{vars}(\mathcal{G})]$ y las derivaciones completas \mathcal{D} y \mathcal{D}' son equivalentes, como queríamos demostrar.

b) El primer paso en \mathcal{D}' es un paso de fallo.

Asumimos que A es el átomo seleccionado en \mathcal{G} y

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{FS} \langle \mathcal{G}[A/\perp], id \rangle \rightsquigarrow^* \langle v, \theta \rangle$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1\sigma) = \text{fallo}$; entonces:

- Si $\text{wmgur}_{\mathcal{R}}(A, H_1) = \text{fallo}$ o la condición de preservación de cabezas (véase la definición 6.5) se cumple, se puede dar un paso de fallo en \mathcal{D} si es dado en \mathcal{D}' , dado que la cabeza de cualquier regla desplegada es la misma (hasta renombramiento de variables) que la cabeza de la regla R_1 y, por lo tanto, $\text{wmgur}_{\mathcal{R}}(A, H_1\sigma) = \text{wmgur}_{\mathcal{R}}(A, H_1) = \text{fallo}$. Entonces es posible la siguiente derivación \mathcal{D} en \mathcal{P} :

$$\mathcal{D} : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{FS} \langle \mathcal{G}[A/\perp], id \rangle \rightsquigarrow^* \langle v, \theta \rangle$$

Además, por la hipótesis de inducción, $\theta = \theta'[\text{vars}(\mathcal{G})]$ y las derivaciones completas \mathcal{D} y \mathcal{D}' son equivalentes, como queríamos demostrar.

- Si $\text{wmgur}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$ y la condición de anulación del cuerpo (véase la definición 6.6) se cumple, entonces $B_1[A_1/\perp]$ lleva a \perp en un número finito de pasos. Además, por la proposición 6.3 tenemos que $\text{wmgur}_{\mathcal{R}}(A_1\alpha, H_2) = \text{fallo}$. Ahora, el primer paso de \mathcal{D}' puede simularse en la derivación \mathcal{D} utilizando la regla $R_1 \in \Pi$ como sigue:

$$\begin{array}{ll} \mathcal{D} : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^{R_1} \\ \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1])])\alpha, \alpha \rangle & \rightsquigarrow_{FS} \\ \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])])\alpha, \alpha \rangle & \rightsquigarrow^* \\ \langle (\mathcal{G}[A/\perp])\alpha\beta, \alpha\beta \rangle & \rightsquigarrow^* \\ \langle v, \theta \rangle & \end{array}$$

Además, dado que la condición de anulación del cuerpo requiere que $(\text{dom}(\alpha) \cup \text{dom}(\beta)) \cap \text{vars}(\mathcal{G}) = \emptyset$, y por la hipótesis de inducción $\theta = \theta'[\text{vars}(\mathcal{G})]$, ambas derivaciones \mathcal{D} y \mathcal{D}' son equivalentes, como queríamos demostrar.

(2) Desplegado basado en un paso de fallo.

Sea $R : (H_1 \leftarrow B_1[A_1]) \in \Pi$ tal que el átomo seleccionado en B_1 no unifica con la cabeza de ninguna regla en \mathcal{P} y, por lo tanto, el desplegado de R utilizando la regla de fallo de la definición 3.3 produce la nueva regla $R' : (H_1 \leftarrow B_1[A_1/\perp]) \in \Pi'$. Si el primer paso de \mathcal{D}' es un paso de fallo, entonces el resultado sigue de la hipótesis de inducción, dado que el desplegado basado en un paso de fallo no modifica la cabeza de la regla R' y, por lo tanto, si no hay ninguna regla $R'_i : (H'_i \leftarrow B'_i) \in \Pi'$ tal que $\text{wmg}_{\mathcal{R}}(A, H'_i) \neq \text{fallo}$, entonces no hay ninguna regla $R_i : H_i \leftarrow B_i \in \Pi$ tal que $\text{wmg}_{\mathcal{R}}(A, H_i) \neq \text{fallo}$. En otro caso, el primer paso de \mathcal{D}' ha sido dado con un paso de éxito utilizando la nueva regla $R' \in \Pi'$:

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{SS}^{R'} \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])]\alpha, \alpha \rangle \rightsquigarrow^* \langle v, \theta' \rangle$$

donde $\text{wmg}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$. Ahora, el primer paso de \mathcal{D}' puede simularse en la derivación \mathcal{D} utilizando la regla $R \in \Pi$ y un paso de fallo:

$$\begin{aligned} \mathcal{D} : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^R \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1])]\alpha, \alpha \rangle \rightsquigarrow_{FS} \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])]\alpha, \alpha \rangle \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

Nótese que en el segundo paso de \mathcal{D} , el objetivo $A_1\alpha$ debe fallar dado que el objetivo A_1 –que subsume al objetivo $A_1\alpha$ – también falla al desplegar la regla R . Dado que el tercer estado de \mathcal{D} coincide sintácticamente con el segundo estado de \mathcal{D}' , el resultado sigue por la hipótesis de inducción.

(3) Desplegado basado en un paso interpretativo.

Sea $R : (H_1 \leftarrow B_1[\zeta(r_1, \dots, r_n)]) \in \Pi$ y, por lo tanto, por desplegado de R utilizando un paso interpretativo de la definición 3.3, obtenemos la nueva regla desplegada $R' : (H_1 \leftarrow B_1[\zeta(r_1, \dots, r_n)/r_{n+1}]) \in \Pi'$, tal que $\vartheta_L(\zeta(r_1, \dots, r_n)) = r_{n+1}$. Entonces, la derivación \mathcal{D}' es como sigue:

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{SS}^{R'} \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)/r_{n+1}])]\alpha, \alpha \rangle \rightsquigarrow^* \langle v, \theta' \rangle$$

donde $\text{wmg}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$. Ahora, el primer paso de \mathcal{D}' puede simularse en la derivación \mathcal{D} utilizando la regla R y un paso interpretativo (reordenando el paso interpretativo por el teorema 3.1):

$$\begin{aligned} \mathcal{D} : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^R \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)])]\alpha, \alpha \rangle \rightsquigarrow_{IS} \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)/r_{n+1}])]\alpha, \alpha \rangle \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

Dado que el tercer estado de \mathcal{D} coincide sintácticamente con el segundo estado de \mathcal{D}' , el resultado sigue de la hipótesis de inducción.

□

Teorema 6.2 (Completitud del desplegado difuso basado en similitud, [JIMR22a]).
 Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y \mathcal{G} un objetivo. Si $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$ es el programa obtenido por desplegado seguro (véase la definición 6.7) de \mathcal{P} , entonces:

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta' \rangle \text{ en } \mathcal{P}'$$

si

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta \rangle \text{ en } \mathcal{P}$$

donde $v \in L$ y $\theta' = \theta[\text{vars}(\mathcal{G})]$.

Demostración. Sea $\mathcal{D} : \langle \mathcal{G}, id \rangle \rightsquigarrow^* \langle v, \theta \rangle$ la derivación (genérica) para el objetivo \mathcal{G} en el programa \mathcal{P} que pretendemos simular construyendo una nueva derivación \mathcal{D}' para \mathcal{G} en \mathcal{P}' . La construcción de \mathcal{D}' se hace por inducción sobre la longitud k de \mathcal{D} . Dado que el caso base ($k = 0$) es trivial, procedemos con el caso general cuando $k > 0$. Entonces, $\mathcal{D} : \langle \mathcal{G}, id \rangle \rightsquigarrow \langle \mathcal{G}', \vartheta \rangle \rightsquigarrow^* \langle v, \theta \rangle$. Si el primer paso de \mathcal{D} ha sido realizado con un paso interpretativo o un paso de fallo de la definición 3.3 o incluso si ha sido realizado con un paso de éxito utilizando una regla $R \in \Pi$ diferente de la regla desplegada, entonces se sigue el resultado por la hipótesis de inducción. En otro caso, el paso inicial ha sido realizado con un paso de éxito utilizando una regla $R' \in \Pi'$ que ha sido obtenida por desplegado de otra regla $R_1 \in \Pi$. Dado que el paso de desplegado ha sido realizado con uno de los tres tipos de pasos de la definición 3.3, consideramos cada caso por separado.

(1) Desplegado basado en un paso de éxito.

Sean $R_1 : (H_1 \leftarrow B_1[A_1]) \in \Pi$ y $R_2 : (H_2 \leftarrow B_2) \in \Pi$ tales que, por desplegado de R_1 con respecto de la regla R_2 utilizando un paso de éxito, obtenemos la regla $R' : (H_1\sigma \leftarrow B_1[A_1/(v'_2 \wedge B_2)]\sigma) \in \Pi'$, donde $\text{wmg}_{\mathcal{R}}(A_1, H_2) = \langle \sigma, v'_2 \rangle$. Entonces, existen dos posibles derivaciones:

a) El segundo paso en \mathcal{D} es un paso de éxito.

Asumimos que A es el átomo seleccionado en \mathcal{G} , y

$$\begin{aligned} \mathcal{D} : \quad & \langle \mathcal{G}[A], id \rangle && \rightsquigarrow_{SS}^{R_1} \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1])])\alpha, \alpha \rangle && \rightsquigarrow_{SS}^{R_2} \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1/(v_2 \wedge B_2)])])\alpha\beta, \alpha\beta \rangle && \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

donde $\text{wmg}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$ y $\text{wmg}_{\mathcal{R}}(A_1\alpha, H_2) = \langle \beta, v_2 \rangle$. Entonces, los primeros dos pasos de \mathcal{D} pueden ser simulados en \mathcal{D}' utilizando la regla transformada $R' \in \Pi'$ como sigue:

$$\begin{aligned} \mathcal{D}' : \quad & \langle \mathcal{G}[A], id \rangle && \rightsquigarrow_{SS}^{R'} \\ & \langle (\mathcal{G}[A/(v'_1 \wedge B_1[A_1/(v'_2 \wedge B_2)])])\sigma\gamma, \sigma\gamma \rangle && \rightsquigarrow^* \\ & \langle v, \theta' \rangle \end{aligned}$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1\sigma) = \langle \gamma, v'_1 \rangle$. Por las proposiciones 6.1 y 6.2 tenemos que $v_1 = v'_1$ y $v_2 = v'_2$, respectivamente, y por el lema 6.1 concluimos que $\alpha\beta = \sigma\gamma[\text{vars}(\mathcal{G})]$, y por lo tanto el tercer estado en \mathcal{D} coincide sintácticamente con el segundo estado en \mathcal{D}' . Además, por la hipótesis de inducción $\theta = \theta'[\text{vars}(\mathcal{G})]$ y las derivaciones completas \mathcal{D} y \mathcal{D}' son equivalentes, como queríamos demostrar.

b) El segundo paso en \mathcal{D} es un paso de fallo.

Asumimos que A es el átomo seleccionado en \mathcal{G} al construir \mathcal{D} :

$$\begin{aligned} \mathcal{D} : \quad & \langle \mathcal{G}[A], id \rangle && \rightsquigarrow_{SS}^R \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1])])\alpha, \alpha \rangle && \rightsquigarrow_{FS} \\ & \langle (\mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])])\alpha, \alpha \rangle && \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$. Entonces, hay dos posibilidades:

- Si se verifica la condición de preservación de cabezas (véase la definición 6.5), la derivación \mathcal{D} es imposible en \mathcal{P} , dado que se ha realizado una transformación de despliegado con un paso de éxito sobre $B_1[A_1]$ con R_2 , cuyas variables no se ven afectadas por α (hasta renombramiento de variables) y por lo tanto, si $\text{wmgur}_{\mathcal{R}}(A_1, H_2) \neq \text{fallo}$ entonces $\text{wmgur}_{\mathcal{R}}(A_1\alpha, H_2) \neq \text{fallo}$.
- Si se verifica la condición de anulación del cuerpo (véase la definición 6.6), por la proposición 6.3 $\text{wmgur}_{\mathcal{R}}(A, H_1\sigma) = \text{fallo}$ y los primeros dos pasos de \mathcal{D} pueden simularse en \mathcal{D}' utilizando un paso de fallo como sigue:

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{FS} \langle (\mathcal{G}[A/\perp]), id \rangle \rightsquigarrow^* \langle v, \theta' \rangle$$

dado que $B_1[A_1/\perp]$ lleva a \perp en un número finito de pasos y $\text{dom}(\alpha) \cap \text{vars}(\mathcal{G}) = \emptyset$, lo que implica que el tercer estado en \mathcal{D} es equivalente al segundo estado en \mathcal{D}' (restringido a $\text{vars}(\mathcal{G})$). Además, por la hipótesis de inducción $\theta = \theta'[\text{vars}(\mathcal{G})]$ y las derivaciones completas \mathcal{D} y \mathcal{D}' son equivalentes, como queríamos demostrar.

(2) Desplegado basado en un paso de fallo.

Sea $R : H_1 \leftarrow B_1[A_1] \in \Pi$ tal que el átomo seleccionado en B_1 no unifica con la cabeza de ninguna regla en \mathcal{P} y, por lo tanto, por despliegado de la regla R aplicando un paso de fallo, obtenemos una nueva regla desplegada $R' : H_1 \leftarrow B_1[A_1/\perp] \in \Pi'$. Entonces, la derivación \mathcal{D} tiene la siguiente forma:

$$\begin{aligned} \mathcal{D} : \quad & \langle \mathcal{G}[A], id \rangle && \rightsquigarrow_{SS}^R \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1])]\alpha, \alpha \rangle && \rightsquigarrow_{FS} \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])]\alpha, \alpha \rangle && \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1) = v_1$. Ahora, los dos primeros pasos de \mathcal{D} pueden simularse en \mathcal{D}' utilizando la regla $R' \in \Pi'$ como sigue:

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{SS}^{R'} \langle \mathcal{G}[A/(v_1 \wedge B_1[A_1/\perp])], \alpha, \alpha \rangle \rightsquigarrow^* \langle v, \theta' \rangle$$

Dado que el tercer estado de \mathcal{D} coincide sintácticamente con el segundo estado en \mathcal{D}' , el resultado sigue de la hipótesis de inducción.

(3) Desplegado basado en un paso interpretativo.

Sea $R : H_1 \leftarrow B_1[\zeta(r_1, \dots, r_n)] \in \Pi$ y, por lo tanto, por desplegado de la regla R utilizando un paso interpretativo, obtenemos una nueva regla $R' : H_1 \leftarrow B_1[\zeta(r_1, \dots, r_n)/r_{n+1}] \in \Pi'$, tal que $\vartheta_L(\zeta(r_1, \dots, r_n)) = r_{n+1}$. Entonces, la derivación \mathcal{D} tiene la siguiente forma (tras reordenar los pasos interpretativos por el teorema 3.1):

$$\begin{aligned} \mathcal{D} : \langle \mathcal{G}[A], id \rangle & \rightsquigarrow_{SS}^R \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)])], \alpha, \alpha \rangle \rightsquigarrow_{IS} \\ & \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)/r_{n+1}])], \alpha, \alpha \rangle \rightsquigarrow^* \\ & \langle v, \theta \rangle \end{aligned}$$

donde $\text{wmgur}_{\mathcal{R}}(A, H_1) = \langle \alpha, v_1 \rangle$. Ahora, los dos primeros pasos de \mathcal{D} pueden simularse en \mathcal{D}' utilizando la regla $R' \in \Pi'$ como sigue:

$$\mathcal{D}' : \langle \mathcal{G}[A], id \rangle \rightsquigarrow_{SS}^{R'} \langle \mathcal{G}[A/(v_1 \wedge B_1[\zeta(r_1, \dots, r_n)/r_{n+1}])], \alpha, \alpha \rangle \rightsquigarrow^* \langle v, \theta' \rangle$$

Dado que el tercer estado de \mathcal{D} coincide sintácticamente con el segundo estado de \mathcal{D}' , el resultado sigue por la hipótesis de inducción. □

Teorema 6.3 (Corrección, completitud y eficiencia del desplegado difuso basado en similitud, [JIMR22a]). *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y \mathcal{G} un objetivo. Si $\mathcal{P}' = \langle \Pi', \mathcal{R}, L \rangle$ es el programa obtenido por desplegado seguro (véase la definición 6.7) de \mathcal{P} , entonces:*

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^m \langle v, \theta \rangle \text{ en } \mathcal{P}$$

si y solo si

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^n \langle v, \theta' \rangle \text{ en } \mathcal{P}'$$

donde $v \in L$, $\theta = \theta'[\text{vars}(\mathcal{G})]$ y $n \leq m$.

Demostración. La corrección y la completitud de la transformación, es decir, la equivalencia de las respuestas computadas difusas obtenidas tras ejecutar un objetivo en el programa original y en el programa desplegado, se sigue inmediatamente de los teoremas 6.1 y 6.2. Sobre la reducción de la longitud de las derivaciones en el programa transformado, podemos ver en las demostraciones

de los teoremas 6.1 y 6.2 que cualquier paso de éxito realizado con la nueva regla transformada, obtenida tras la aplicación de un paso de desplegado seguro, subsume dos pasos de computación realizados con reglas del programa original, que confirma que $n \leq m$, como queríamos demostrar. \square

Nótese que bajo la definición 6.7 de desplegado seguro cualquier desplegado basado en un paso de fallo \rightsquigarrow_{FS} o en un paso interpretativo \rightsquigarrow_{IS} es considerado seguro, ya que estos pasos de computación no modifican la cabeza de la regla desplegada, lo cual verifica trivialmente las condiciones de similitud límite y de preservación de cabezas. Además, por la misma razón, cualquier desplegado basado en un paso de éxito \rightsquigarrow_{SS} explotando un átomo que no contiene variables de la cabeza (en particular, cualquier átomo básico) también es siempre seguro. Formalizamos este resultado en el siguiente corolario.

Corolario 6.1. *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y \mathcal{G} un objetivo. Si \mathcal{P}' es el programa obtenido por desplegado (véase la definición 6.1) de \mathcal{P} mediante la aplicación de un paso de fallo \rightsquigarrow_{FS} , un paso interpretativo \rightsquigarrow_{IS} , o un paso de éxito \rightsquigarrow_{SS} explotando un átomo que no contiene variables de la cabeza, entonces:*

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^m \langle v, \theta \rangle \text{ en } \mathcal{P}$$

si y solo si

$$\langle \mathcal{G}, id \rangle \rightsquigarrow^n \langle v, \theta' \rangle \text{ en } \mathcal{P}'$$

donde $v \in L$, $\theta = \theta'[vars(\mathcal{G})]$ y $n \leq m$.

6.4. Detalles de implementación

En esta sección se presentan los detalles de implementación de la transformación de desplegado en el sistema FASILL, donde las reglas pueden ser desplegadas mediante el comando `:undoId(Id)`, que recibe el identificador² de una regla cargada actualmente en el entorno y aplica un paso de desplegado sobre ella, o mediante el predicado incorporado `unfold/1`, que recibe un término y despliega la primera regla que unifica con dicho término. La figura 6.3 muestra un paso de desplegado sobre el programa del ejemplo 3.1 en la consola interactiva del sistema FASILL utilizando el comando `:unfold`, mientras que la figura 6.4 muestra el mismo paso de desplegado utilizando el predicado incorporado `unfold/1`.

La herramienta en línea también ofrece la posibilidad de aplicar pasos de desplegado sobre los programas FASILL [MR19a]. Bajo el cuadro de texto que contiene las reglas del programa hay un botón que permite iniciar el proceso de desplegado, enumerando cada una de las reglas introducidas, tal y como se muestra en la figura 6.5. Cuando el usuario selecciona la regla a desplegar, el sistema FASILL aplica un paso de desplegado sobre esta y actualiza el programa

²Como se puede ver en las figuras 6.3 y 6.4, el usuario puede consultar el identificador de las reglas mediante el comando `:listing`, que enumera las reglas junto a sus identificadores.

```

fasill> :listing.
(1) cheap(taxi) <- 0.8.
(2) close(hydropolis,taxi) <- 0.7.
(3) close(ritz,metro) <- 0.9.
(4) good_hotel(X) <- @aver(@very(close(X,Y)),cheap(Y)).

fasill> :unfold('4').
<1.0, {}>

fasill> :listing.
(1) cheap(taxi) <- 0.8.
(2) close(hydropolis,taxi) <- 0.7.
(3) close(ritz,metro) <- 0.9.
(4-1) good_hotel(hydropolis) <- @aver(@very(0.7),cheap(taxi)).
(4-2) good_hotel(ritz) <- @aver(@very(0.9),cheap(metro)).

```

Figura 6.3: Despliegado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el comando `:unfold`.

```

fasill> :listing.
(1) cheap(taxi) <- 0.8.
(2) close(hydropolis,taxi) <- 0.7.
(3) close(ritz,metro) <- 0.9.
(4) good_hotel(X) <- @aver(@very(close(X,Y)),cheap(Y)).

fasill> unfold((good_hotel(_) <- Body)).
<1.0, {Body/@aver(@very(close(V1,V2)),cheap(V2))}>

fasill> :listing.
(1) cheap(taxi) <- 0.8.
(2) close(hydropolis,taxi) <- 0.7.
(3) close(ritz,metro) <- 0.9.
(4-1) good_hotel(hydropolis) <- @aver(@very(0.7),cheap(taxi)).
(4-2) good_hotel(ritz) <- @aver(@very(0.9),cheap(metro)).

```

Figura 6.4: Despliegado de programas lógicos difusos en la consola interactiva del sistema FASILL mediante el predicado incorporado `unfold/1`.

en el cuadro de texto, mostrando información sobre las condiciones de seguridad del despliegado: la condición de similitud límite, la condición de preservación de cabezas y la condición de anulación del cuerpo. La figura 6.6 muestra el resultado de desplegar la regla R_4 del programa del ejemplo 3.1 en la herramienta en

```

1 cheap(taxi) <- 0.8.
2 close(hydropolis, taxi) <- 0.7.
3 close(ritz, metro) <- 0.9.
4 good_hotel(X) <- @aver(@very(close(X, Y)), cheap(Y)).

```

[Linearize program](#)
[Extend program](#)
[Unfold program](#)

```

1 cheap(taxi) <- 0.8.

```

```

2 close(hydropolis,taxi) <- 0.7.

```

```

3 close(ritz,metro) <- 0.9.

```

```

4 good_hotel(X) <- @aver(@very(close(X,Y)),cheap(Y)).

```

Figura 6.5: Área de despliegado del sistema FASILL en línea.

```

1 cheap(taxi) <- 0.8.
2 close(hydropolis,taxi) <- 0.7.
3 close(ritz,metro) <- 0.9.
4 good_hotel(hydropolis) <- @aver(@very(0.7),cheap(taxi)).
5 good_hotel(ritz) <- @aver(@very(0.9),cheap(metro)).
6 % bound-similarity condition: false
7 % head-preserving condition: false
8 % body-overriding condition: false

```

Unfolding is **not safe**: ✘ Bound-similarity ✘ Head-preserving ✘ Body-overriding [Unfold program](#)

Figura 6.6: Despliegado de programas en el sistema FASILL en línea.

línea, que informa de que esta regla no cumple ninguna de las tres condiciones de seguridad y, por lo tanto, el despliegado no es seguro (como ya vimos al inicio del capítulo).

El módulo `fasill_unfolding` del sistema FASILL contiene la implementación de la técnica de despliegado, y exporta predicados para el despliegado y la verificación de las condiciones de seguridad. El predicado principal para desplegar un programa es `classic_unfold_by_id/1`, que recibe el identificador de una regla del programa y aplica un paso de despliegado sobre esta, modificando el programa actual de la sesión. Este predicado simplemente busca la regla asociada al identificador indicado e invoca al predicado `classic_unfold/1`, que se encarga de realizar la transformación de despliegado sobre la regla.

```

1 classic_unfold_by_id(Id) :-
2     fasill_rule(Head, Body, Info),
3     member(id(Id), Info), !,

```

```
4 classic_unfold(fasill_rule(Head, Body, Info)).
```

El predicado `classic_unfold/1` recibe una regla en su representación básica y aplica un paso de desplegado en todas sus posibles formas sobre esta mediante el predicado `classic_unfold/2`. Entonces elimina la regla del programa, inserta las nuevas reglas obtenidas por desplegado y reordena las reglas del programa en base a los identificadores para que las nuevas reglas aparezcan en el orden correcto (reemplazando a la regla desplegada).

```
1 classic_unfold(R1) :-
2     findall(R, classic_unfold(R1, R), Rules),
3     Rules \= [],
4     once(retract(R1)),
5     forall(member(Rule, Rules), assertz(Rule)),
6     sort_rules_by_id.
```

Finalmente, el predicado `classic_unfold/2` recibe una regla y devuelve una regla desplegada sin modificar el programa original. Para ello realiza un paso de inferencia sobre el cuerpo de la regla y aplica la sustitución obtenida a la cabeza de la misma. Por reevaluación, el paso de inferencia se realiza en todas sus posibles formas.

```
1 classic_unfold(R1, R2) :-
2     R1 = fasill_rule(head(Head1), body(Body1), Info1),
3     select(id(Id1), Info1, Info2),
4     init_substitution(Body1, Vars),
5     inference(unfolding/0,
6         state(Body1, Vars), state(Body2, Sub), _),
7     Body1 \= Body2,
8     apply(Sub, Head1, Head2),
9     next_unfolding_id(N),
10    atomic_list_concat([Id1, -, N], Id2),
11    R2 = fasill_rule(head(Head2), body(Body2), [id(Id2)|Info2]).
```

La condición de similitud límite descrita en la definición 6.4 se comprueba mediante el predicado `bound_similarity/1`, que recibe una regla en su representación básica y verifica que, para todo paso de computación posible realizado sobre el cuerpo de la regla, la sustitución restringida a las variables de la cabeza es límite.

```
1 bound_similarity(R) :-
2     R = fasill_rule(head(Head), body(Body), _),
3     init_substitution(Head, Sub0),
4     forall(
5         inference(unfolding/0,
```

```

6         state(Body, Sub0), state(_, Sub1), _),
7         is_bound_substitution(Sub1)).

```

El predicado `is_bound_substitution/1` recibe una sustitución y comprueba que cada término que aparece en los enlaces de las variables es un término límite. Para ello, cuando el predicado `is_bound/1` recibe un término, comprueba su grado de similitud con el resto de símbolos para verificar que es \perp o \top .

```

1  is_bound_substitution(Sub) :-
2      substitution_to_list(Sub, List),
3      forall(member(_Term, List), is_bound(Term)).
4
5  is_bound(term(Name, Args)) :- !,
6      length(Args, Arity),
7      lattice_call_top(Top),
8      lattice_call_bot(Bot),
9      forall(
10         similarity_between(Name, _, Arity, TD, _),
11         (TD == Top ; TD == Bot)),
12         is_bound(Args).
13  is_bound([]) :- !.
14  is_bound([X|Xs]) :- !,
15         is_bound(X),
16         is_bound(Xs).
17  is_bound(_).

```

La condición de preservación de cabezas descrita en la definición 6.5 se comprueba mediante el predicado `head_preserving/1`, que recibe una regla en su representación básica y verifica que para todo paso de computación posible realizado sobre el cuerpo de la regla, tras aplicar la sustitución obtenida a la cabeza de la regla, esta cabeza es un renombramiento de la cabeza de la regla original.

```

1  head_preserving(R) :-
2      R = fasill_rule(head(Head), body(Body), _),
3      init_substitution(Head, S0),
4      forall(
5          ( inference(unfolding/0, state(Body,S0), state(_,S1), _),
6              apply(S1, Head, Head1)
7          ),
8          is_renaming(Head, Head1)
9      ).

```

Por último, la condición de anulación del cuerpo descrita en la definición 6.6 se comprueba mediante el predicado `body_overriding/1`, que recibe una regla en su representación básica y verifica que para toda derivación posible

realizada sobre el cuerpo de la regla (tras reemplazar el átomo más a la izquierda por \perp) lleva a una f.c.a. con grado de verdad asociado de \perp . Además, al aplicar la sustitución generada en la derivación sobre la cabeza, esta debe ser un renombramiento de la cabeza de la regla original.

```

1 body_overriding(R) :-
2   R = fasill_rule(head(Head), body(Body), _),
3   init_substitution(Head, S0),
4   lattice_call_bot(Bot),
5   ( select_atom(Body, BodyBot, Bot, _) ->
6     forall(
7       derivation(unfolding/0, state(BodyBot,S0), State, _),
8       ( State = exception(_)
9       ; State = state(TD, S1),
10        TD = Bot,
11        apply(S1, Head, Head1),
12        is_renaming(Head, Head1) )
13    ) ; true ).

```

6.5. Pruebas y evaluación de rendimiento

Para determinar la ganancia de eficiencia producida por la transformación de desplegado sobre programas FASILL proponemos en [JIMR22a] los siguientes experimentos donde medimos el tiempo de ejecución y el número de inferencias de los programas descritos a continuación, tanto para su versión original como para su versión desplegada.³

En el primer experimento nos proponemos medir la ganancia de eficiencia en algunos programas FASILL simples tras aplicar un único paso de desplegado (seguro) sobre una regla del programa. En la tabla 6.1 se resumen los resultados de la ejecución de los programas de prueba listados en la figura 6.7. La tabla muestra el tiempo de ejecución medio y el número de inferencias tras ejecutar un objetivo en el programa original y en el programa desplegado, donde las reglas marcadas con un asterisco (*) en la figura 6.7 son desplegadas de forma segura una única vez (un solo paso de desplegado). La ganancia del programa desplegado con respecto al programa original ha sido computada utilizando el tiempo y el número de inferencias realizadas por SWI-Prolog ejecutando el sistema FASILL. Todos los objetivos explotan similitudes y han sido elegidos para proporcionar un tiempo de ejecución razonablemente largo. Por ejemplo, el predicado `palindrome/1` es invocado con una lista $[a, \dots, a, c, b, \dots, b]$, donde $\mathcal{R}(a, b) = 0.5$ (véase también la explicación del predicado `append/3` al final de la sección 6.2). Como se observa en la tabla 6.1, tras un paso de desplegado todos los programas muestran una ganancia significativa de eficiencia, llegando

³Todas las pruebas de esta sección han sido ejecutadas en SWI-Prolog 8.0.6 (64 bits) utilizando un ordenador de sobremesa equipado con un procesador AMD Opteron™ @1593 MHz y 2.00 GB RAM.

<pre>append([], X, X). append([H T], X, [H S]) <- append(T, X, S). % (*) double_append(X, Y, Z, W) <- append(X, Y, U) & append(U, Z, W).</pre>	<pre>flip(tree(X,Xs), tree(X,Ys)) <- flip(Xs, [], Ys). % (*) flip([], Ys, Ys). flip([X Xs], Ys, Zs) <- flip(X, Y) & flip(Xs, [Y Ys], Zs).</pre>
(a) double_append	(d) flip
<pre>reverse(Xs, Ys) <- reverse(Xs, [], Ys). reverse([], Ys, Ys). reverse([X Xs], Ys, Zs) <- reverse(Xs, [X Ys], Zs). % (*) palindrome(Xs) <- reverse(Xs,Xs).</pre>	<pre>add(z, N, N). add(s(M), N, s(P)) <- add(M, N, P). % (*) mul(z, _, z). mul(s(M), N, Y) <- mul(M, N, X) & add(N, X, Y).</pre>
(b) palindrome	(e) peano
<pre>add(z, N, N). add(s(M), N, s(P)) <- add(M, N, P). % (*) fibonacci(z, z). fibonacci(s(z), s(z)). fibonacci(s(s(N)), Z) <- fibonacci(N, X) & fibonacci(s(N), Y) & add(X, Y, Z).</pre>	<pre>append([], X, X). append([H T], X, [H S]) <- append(T, X, S). % (*) inorder(empty, []). inorder(tree(X, L, R), Xs) <- inorder(L, Ls) & inorder(R, Rs) & append(Ls, [X Rs], Xs).</pre>
(c) fibonacci	(f) inorder
<pre>minlist([X], X, []). minlist([X,H Xs],Z,[W Ys]) <- minlist([H Xs],Y,Ys) & minmax(X,Y,Z,W). minmax(z, s(N), z, s(N)). minmax(s(M), z, z, s(M)). minmax(s(M), s(N), s(P), s(Q)) <- minmax(M, N, P, Q). % (*) selsort([], []). selsort([X Xs], [Y Zs]) <- minlist([X Xs],Y,Ys) & selsort(Ys,Zs).</pre>	
(g) selsort	

Figura 6.7: Programas FASILL de prueba.

a reducir casi a la mitad el tiempo de ejecución en algunos casos. La figura 6.8 ilustra la comparación de los tiempos de ejecución del programa original frente al programa desplegado.

En el segundo experimento nos proponemos medir la ganancia de eficiencia lograda por la transformación de desplegado al aplicar cada vez más pasos de desplegado sobre un mismo programa. La tabla 6.2 muestra el tiempo medio de ejecución y el número de inferencias al buscar todas las respuestas computadas

Tabla 6.1: Tiempo medio de ejecución (en milisegundos) y número de inferencias al ejecutar los programas FASILL de prueba mostrados en la figura 6.7 tras 50 ejecuciones.

Prueba	Programa original		Programa desplegado		Ganancia	
	Tiempo	Inferencias	Tiempo	Inferencias	Tiempo	Inf.
double_append	9756	119609091	6342	67128927	1.54	1.78
palindrome	11461	139254118	6965	81913808	1.65	1.70
fibonacci	15528	198805804	11004	141097136	1.41	1.40
flip	13040	130521853	9139	87785578	1.43	1.48
peano	8345	105121757	4315	54293704	1.93	1.93
inorder	27343	302048850	19578	206022142	1.40	1.47
selsort	23976	312062385	14037	175726911	1.71	1.77

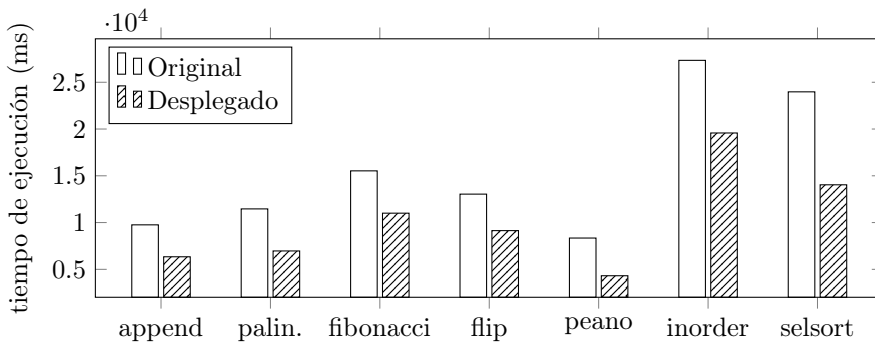


Figura 6.8: Comparación de los tiempos de ejecución de los programas originales frente a los programas desplegados.

difusas para el predicado `mul/3` utilizando el programa mostrado en la figura 6.7e, en función del número de pasos de desplegado aplicados sobre el programa original. Tal y como sucede en otros marcos de programación declarativos, se observa que tras un determinado número de pasos de desplegado sobre el programa, los resultados empiezan a perder eficiencia debido al sobre coste introducido al tratar de unificar el átomo seleccionado con las cabezas de las crecientes reglas transformadas. La figura 6.9 muestra el tiempo medio de ejecución (en milisegundos) de los programas de prueba de la figura 6.7 en función del número de pasos de desplegado (seguros) aplicados sobre el programa original.

6.5.1. Desplegado difuso de programas simbólicos

Hasta ahora, hemos considerado las operaciones de desplegado difuso y de calibrado de programas simbólicos como dos técnicas independientes con objetivos claramente diferenciados: el desplegado mejora la eficiencia de los programas y el calibrado ajusta sus reglas. Sin embargo, la transformación de desplegado también puede utilizarse para optimizar el proceso de calibrado, acercando los

Tabla 6.2: Tiempo medio de ejecución (en milisegundos) y número de inferencias al buscar todas las f.c.a. para el predicado `mul/3` utilizando el programa de prueba mostrado en la figura 6.7e tras 50 ejecuciones, dependiendo del número de pasos de despliegado realizados sobre el programa original.

Desplegados	Tiempo	Inferencias	Ganancia
0	8345	105121757	1.00
1	4315	54293704	1.93
2	2134	27797061	3.78
3	1226	15665907	6.71
4	800	9818130	10.70
5	501	6307196	16.67
6	454	5396529	19.48
7	748	10062129	10.44

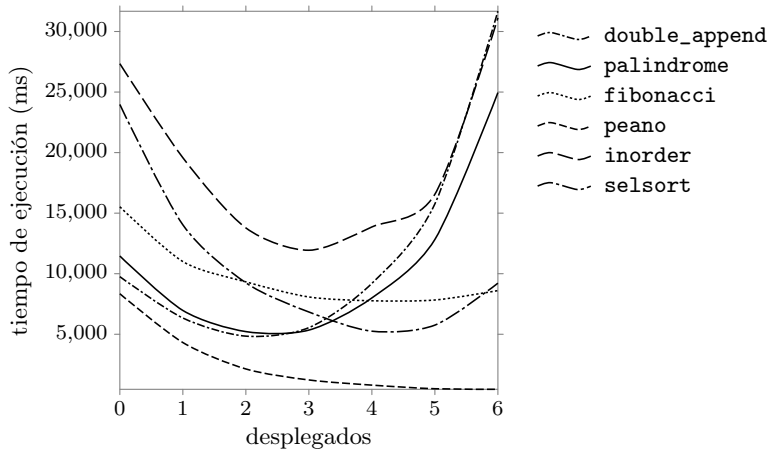


Figura 6.9: Comparación de los tiempos de ejecución de varios programas FASILL en función del número de pasos de despliegado.

tiempos de ejecución del método de calibrado básico al del simbólico.

Siguiendo la línea de nuestras propuestas [MPR19a, MR19b], hemos diseñado un último experimento para contrastar los tiempos de ejecución de los algoritmos de calibrado utilizando el método de calibrado básico y el método de calibrado simbólico tras aplicar varios pasos de despliegado. La tabla 6.3 resume el tiempo medio de ejecución del proceso de calibrado del siguiente programa difuso:

$$\Pi^{\#} = \begin{cases} R_1 : p(z) & \leftarrow 1.0 \\ R_2 : p(s(x)) & \leftarrow p(x) \&_{prod} s^{\#} \end{cases}$$

en función del número de desplegados y del número de sustituciones simbólicas consideradas. Para ello, utilizamos un único caso de prueba de la forma $\langle 1.0, p(s(s(s(\dots z \dots)))) \rangle$ para el que la única sustitución simbólica que produce

una desviación de 0.0 es visitada la última en el espacio de búsqueda. En esta tabla observamos que, tras aplicar las transformaciones de desplegado, el tiempo de calibrado del método básico mejora notoriamente, aunque no es capaz de alcanzar el tiempo de ejecución del método de calibrado simbólico. Más aún, como es de esperar, la diferencia de tiempos se acentúa cuantas más sustituciones simbólicas consideramos. Además, es importante destacar que aunque el desplegado también puede mejorar el tiempo de ejecución del método de calibrado simbólico, la ganancia es menos significativa, ya que la fase admisible para el objetivo del caso de prueba se computa una sola vez. La figura 6.10 ilustra la comparación de los tiempos de ejecución del algoritmo de calibrado básico en función del número de pasos de desplegado aplicados y del número de sustituciones simbólicas consideradas.

Tabla 6.3: Tiempo medio de ejecución (en milisegundos) al calibrar un programa tras 50 ejecuciones, dependiendo del número de pasos de desplegado realizados sobre el programa original y del número de sustituciones simbólicas consideradas.

Calibrado	Desplegados	Tiempo (ms)					
		10 _θ	50 _θ	100 _θ	250 _θ	500 _θ	1000 _θ
Básico	0	500	2942	4901	12765	25088	50958
	1	302	1380	2776	7078	14239	27854
	2	172	776	1541	3896	7713	15427
	3	104	489	958	2369	4797	9500
	4	72	338	672	1656	3323	6630
Simbólico	0	68	128	217	446	861	1743
	1	51	109	195	445	825	1623
	2	32	104	176	434	821	1609
	3	29	90	168	410	814	1594
	4	26	87	165	409	796	1593

6.6. Conclusiones

En este capítulo hemos adaptado al lenguaje FASILL la transformación de desplegado clásica de otros paradigmas y hemos demostrado sus propiedades de corrección y completitud para un subconjunto de programas FASILL que pueden ser desplegados de forma segura. A continuación detallamos las contribuciones de este capítulo.

- (1) Hemos adaptado la transformación de desplegado (clásica de otros paradigmas como el lógico y el funcional) al lenguaje lógico difuso FASILL. Para esta operación de desplegado difuso hemos observado que no todos los programas FASILL preservan las respuestas computadas difusas tras aplicar un paso de desplegado en determinadas reglas.

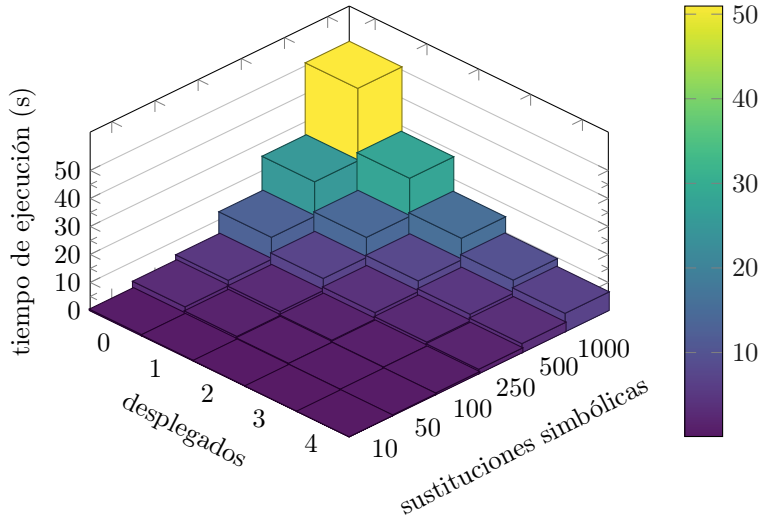


Figura 6.10: Comparación de los tiempos de ejecución del algoritmo de calibrado básico en función del número de pasos de desplegado aplicados.

- (2) Cuando la transformación de desplegado liga una variable de la cabeza de la regla desplegada a un término “con similitudes”, hemos visto que pueden surgir problemas de corrección en el programa desplegado, ya que los grados de similitud no se componen de la misma forma en el programa original y en el programa desplegado. Al usar el programa transformado, esos grados de verdad pueden introducirse en diferentes posiciones de expresiones constituidas por operadores difusos que no son asociativos y conmutativos y, por tanto, influye el orden en el que se computan. Así pues, para un mismo objetivo, el programa original y el programa desplegado pueden llevar a respuestas computadas difusas distintas.
- (3) Cuando la transformación de desplegado instancia la cabeza de la regla desplegada, hemos visto que pueden aparecer pasos de fallo prematuros en las derivaciones del programa desplegado, generando problemas de completitud. Es decir, para un mismo objetivo, el programa desplegado puede “perder” respuestas computadas difusas que sí están en el programa original.
- (4) Analizando estos problemas de corrección y completitud, hemos caracterizado los programas que pueden ser desplegados de forma segura. Para ello hemos definido tres condiciones de seguridad: la condición de similitud límite, la preservación de cabezas y la anulación del cuerpo. Para que un programa desplegado preserve las respuestas computadas difusas del programa original, la regla desplegada debe satisfacer la condición de similitud límite y al menos una de las otras dos condiciones.

- (5) Hemos demostrado las propiedades de corrección, completitud y eficiencia de la transformación de desplegado para programas FASILL cuando la regla desplegada satisface las condiciones mencionadas.
- (6) Hemos implementado la transformación de desplegado en el sistema FASILL. Además, los programas pueden ser desplegados en la herramienta web. Cuando se aplica un paso de desplegado, FASILL comprueba las condiciones de seguridad y reporta al usuario si cada una de ellas se verifica o no, y si el desplegado es seguro o inseguro.
- (7) Hemos diseñado una serie de experimentos para medir la ganancia de eficiencia de los programas desplegados frente a los programas originales. Como es habitual en este tipo de transformaciones, tras un número determinado de pasos de desplegado, el programa empieza a perder eficiencia debido al tiempo que conlleva unificar un átomo con las cabezas del creciente número de reglas del programa desplegado.
- (8) Hemos combinado la transformación de desplegado difuso con las técnicas de calibrado de programas simbólicos para tratar de mejorar la eficiencia del método de calibrado básico y acercar su rendimiento al del método de calibrado simbólico.

Hay que tener en cuenta que el lenguaje FASILL combina un algoritmo de unificación débil basado en relaciones de similitud con una gran variedad de conectivas difusas definidas sobre un retículo completo. En este contexto, el precio a pagar por disponer de un marco de programación tan amplio y flexible como este es que la transformación de desplegado no puede aplicarse (de forma segura) sobre cualquier regla de un programa. Sin embargo, hemos identificado un amplio subconjunto de programas FASILL sobre los que es posible aplicar esta transformación preservando la semántica original, para los que hemos dado los resultados de corrección y completitud de la transformación.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo presentamos un resumen sobre el trabajo desarrollado en esta tesis y detallamos algunas líneas de investigación que están ya en marcha o que podrían iniciarse en un futuro. Nuestros objetivos generales, que presentamos al principio de la memoria (véase la sección 1.1) han sido abordados con éxito en este trabajo. Mediante nuestra investigación hemos diseñado diversos algoritmos de calibrado de programas lógicos difusos y hemos sido capaces de definir una transformación de desplegado difuso, sobre la que hemos podido caracterizar los programas que pueden ser desplegados de forma segura estableciendo una serie de condiciones que las reglas desplegadas deben satisfacer. Además de proporcionar las demostraciones sobre las propiedades fundamentales de estas técnicas de calibrado y desplegado, hemos incluido una descripción de la implementación de alto nivel de las mismas en el entorno de programación FASILL, que integra en un mismo lenguaje las características de diversos lenguajes difusos. Por último, hemos ilustrado nuestros desarrollos con ejemplos y aplicaciones de mayor o menor envergadura al tiempo que los contrastamos con otros trabajos relacionados.

7.1. Conclusiones

A continuación detallamos las principales contribuciones que hemos aportado durante el desarrollo de esta tesis, las cuales hemos dividido en varias categorías: las aportaciones teóricas al lenguaje FASILL, el desarrollo práctico del sistema FASILL, el diseño de la extensión simbólica del lenguaje FASILL, las técnicas de calibrado de programas simbólicos y la transformación de desplegado difuso.

Aportaciones teóricas al lenguaje FASILL

Las primeras contribuciones de esta tesis se centran en aspectos teóricos del lenguaje FASILL. Aunque en este trabajo no aportamos ninguna novedad en lo referente a la semántica operacional o declarativa de FASILL, originalmente

descritas en trabajos preliminares [JIMP17, JIMP18], la transformación de desplegado difuso ha requerido la formulación y demostración de algunos resultados teóricos previos que hemos presentado en [JIMR22a, JIMR22b]. En particular:

- (1) hemos demostrado que, aunque la regla de computación de FASILL no es independiente en general, sí lo es respecto de los pasos interpretativos (véase el teorema 3.1);
- (2) hemos demostrado que el algoritmo de unificación débil que utiliza FASILL genera sustituciones idempotentes (véase la proposición 2.4);
- (3) hemos definido el operador de composición paralela débil (véase la definición 2.41), y hemos estudiado su relación con la composición estándar de sustituciones (véase la proposición 2.41).

De forma más general, en [JIMR22b] hemos analizado las propiedades algebraicas de las sustituciones en el contexto de las relaciones de similitud, donde el conjunto de sustituciones idempotentes conforma una estructura de retículo, cuya cota mínima superior se define como el operador de composición paralela débil.

Desarrollo e implementación del sistema FASILL

Una de las contribuciones transversales (pero no por ello menos importante) de este trabajo ha sido la implementación y el mantenimiento de un entorno de programación lógico difuso que soporte todas las características del lenguaje FASILL, y que nos permita incorporar todas las técnicas y transformaciones que estudiamos en esta tesis. El sistema FASILL es una implementación de alto nivel escrita en Prolog que permite activar y desactivar las diferentes componentes difusas del lenguaje, y que dispone de un amplio conjunto de predicados incorporados para el manejo de excepciones, la comparación de términos, la evaluación y comparación aritmética, el procesamiento de átomos, etcétera. En [JIMR20] proporcionamos una descripción detallada tanto del lenguaje FASILL como de su implementación, sentando las bases sobre las que hemos llevado a la práctica las técnicas de calibrado y desplegado de programas lógicos difusos integrados. El código fuente de FASILL está disponible en GitHub: <https://github.com/jariazavalverde/fasill>. Además, hemos desarrollado una aplicación web que permite ejecutar, calibrar y desplegar programas lógicos difusos en línea [MR17, MR19a], que está disponible a través de la URL <https://dectau.uclm.es/fasill/sandbox>. En resumen:

- (1) hemos desarrollado el sistema FASILL, que es un entorno de programación lógico difuso que integra distintas componentes difusas en una misma herramienta;
- (2) hemos implementado las técnicas de calibrado y desplegado difuso sobre el sistema FASILL, lo que nos ha permitido estudiar y medir el rendimiento de las distintas técnicas propuestas en la tesis;

- (3) hemos dispuesto una aplicación web que permite probar todas las funcionalidades del sistema FASILL de forma cómoda y segura, sin necesidad de instalación.

Por otra parte, en [GMM⁺17, GMRS18] hemos avanzado en el desarrollo de herramientas gráficas para el diseño y análisis de retículos y relaciones de similitud que permiten cargar, exportar y visualizar de forma cómoda retículos completos y esquemas de similitud compatibles con el sistema FASILL.

La extensión simbólica sFASILL

Como preámbulo a la introducción de las técnicas de calibrado de programas lógicos difusos, ha sido necesario diseñar una extensión simbólica de FASILL que permita anotar valores y conectivas simbólicas (que no pertenecen al retículo asociado al programa) tanto en las reglas como en las relaciones de similitud. En [MPRV17] introducimos por primera vez una extensión simbólica para programas multi-adjuntos, que pueden verse como programas FASILL sin relaciones de similitud. Posteriormente, en [MR20, MR21] adaptamos esta extensión simbólica a FASILL considerando también las relaciones de similitud simbólicas, que suponen un reto adicional al no garantizarse en general la existencia de las mismas respuestas computadas difusas antes y después de aplicar una sustitución simbólica al programa. Por lo tanto:

- (1) hemos elaborado una extensión simbólica de FASILL, llamada sFASILL, que nos permite introducir constantes simbólicas en las reglas y en las relaciones de similitud de los programas lógicos difusos, postergando la evaluación de las mismas hasta que los valores son conocidos;
- (2) hemos diseñado un algoritmo de cierre simbólico que a partir de un esquema de similitud simbólico computa una relación de similitud simbólica válida (véase el algoritmo 4.1);
- (3) hemos demostrado que, cuando el programa no contiene constantes simbólicas en la relación de similitud asociada, el programa simbólico lleva a las mismas respuestas computadas difusas si: *i*) aplicamos una sustitución simbólica al programa y después calculamos la derivación; o si *ii*) calculamos la respuesta computada difusa simbólica, aplicamos la sustitución simbólica a dicha s.f.c.a. y calculamos los últimos pasos interpretativos. (véase el teorema 4.1);
- (4) cuando el programa contiene constantes simbólicas en la relación de similitud, hemos caracterizado el tipo de programas y sustituciones simbólicas que son seguras (véase la definición 4.6) en el sentido de que preservan las respuestas computadas difusas independientemente de si se aplica la sustitución simbólica al programa antes de calcular la derivación, o a la respuesta computada difusa simbólica correspondiente (véanse los teoremas 4.2 y 4.3);

- (5) hemos implementado la extensión simbólica en el sistema FASILL, que puede ser deshabilitada mediante una directiva del lenguaje.

Calibrado de programas lógicos difusos

El primero de los objetivos fundamentales de esta tesis es el diseño de técnicas eficientes de calibrado de programas lógicos difusos integrados. Dado un programa simbólico, su calibrado consiste en la búsqueda automática de la mejor sustitución simbólica que minimiza la desviación de sus posteriores respuestas con respecto a una serie de casos de prueba –esto es, una serie de objetivos junto a los grados de verdad esperados para los mismos– introducidos por el usuario. En [MPRV17] definimos las primeras técnicas de calibrado para programas multi-adjuntos, que posteriormente adaptamos a FASILL en [MR20, MR21]. Después, en [RM20] integramos las técnicas de calibrado con resolutores de satisfacibilidad y mostramos algunas de sus aplicaciones, como la equivalencia de circuitos combinatoriales o la regresión lineal, al tiempo que utilizamos el calibrado en el proceso de entrenamiento de redes neuronales [MPR19b] y en la flexibilización de consultas FSA-SPARQL [AJBTMR19, AJBTMR21, AJBTMR22]. De manera sintetizada:

- (1) hemos diseñado un algoritmo de calibrado básico que únicamente hace uso de la semántica operacional de FASILL (véase el algoritmo 5.1) y que, de alguna manera, sienta las bases del resto de técnicas de calibrado más eficientes;
- (2) hemos diseñado un algoritmo de calibrado simbólico basado en la semántica operacional de la extensión simbólica sFASILL (véase el algoritmo 5.2) que calcula menos derivaciones y permite reducir automáticamente el espacio de búsqueda (véase el algoritmo 5.3);
- (3) hemos combinado el algoritmo de calibrado simbólico con resolutores de satisfacibilidad para realizar la búsqueda de la mejor sustitución simbólica mediante un resolutor SMT como Z3, lo que supone un nuevo incremento drástico de eficiencia en algunas aplicaciones;
- (4) hemos implementado en el sistema FASILL los algoritmos de calibrado básico, calibrado simbólico (disjunto) y calibrado basado en satisfacibilidad, que además pueden ser ejecutados en la aplicación web;
- (5) hemos utilizado las técnicas de calibrado en diversos dominios de aplicación de interés, como la verificación de la equivalencia de circuitos combinatoriales, el entrenamiento de redes neuronales y modelos de regresión lineal y la web semántica.

Despliegado de programas lógicos difusos

El segundo de los objetivos fundamentales de esta tesis es la adaptación de la transformación de despliegado, clásica de otros paradigmas declarativos como

el lógico y el funcional, a los programas lógicos difusos integrados, que permite generar código más eficiente al aplicar pasos de computación sobre los cuerpos de las reglas. En [MPR17] adaptamos la transformación de desplegado a los programas FASILL y planteamos algunos de los problemas de corrección y completitud que pueden aparecer. Posteriormente, en [JIMR22a] caracterizamos los programas que pueden ser desplegados con seguridad, preservando las respuestas computadas difusas y damos las demostraciones de corrección y completitud de la transformación. Concretamente:

- (1) hemos adaptado la transformación de desplegado a nuestro entorno difuso (véase la definición 6.1);
- (2) hemos detectado las principales fuentes de problemas que pueden surgir a la hora de aplicar pasos de desplegado difuso sobre los programas FASILL y hemos establecido unas condiciones de seguridad para garantizar la corrección de la transformación (véanse las definiciones 6.4, 6.5 y 6.6);
- (3) hemos demostrado las propiedades de corrección, completitud y eficiencia de la transformación de desplegado difuso para programas FASILL (véanse los teoremas 6.1, 6.2 y 6.3);
- (4) hemos implementado en el sistema FASILL la transformación de desplegado difuso y, además de poder ser ejecutada en la aplicación web, también es capaz de reportar si el desplegado verifica las preceptivas condiciones de seguridad;
- (5) hemos medido la ganancia de eficiencia de la transformación de desplegado difuso, comparando los tiempos de ejecución y el número de inferencias de los programas FASILL desplegados respecto a los programas originales.

Además, en [MPR19a, MR19b] relacionamos las técnicas de desplegado difuso con la extensión simbólica del lenguaje FASILL y las sucesivas técnicas de calibrado.

7.2. Trabajo futuro

En esta sección detallamos las líneas de investigación en las que estamos trabajando actualmente en nuestro grupo DEC-TAU, como es el caso del desplegado de programas lógicos difusos basado en guardas, además de otras posibles líneas de trabajo que podrían iniciarse en un futuro, como el calibrado de programas simbólicos basado en programación lógica con restricciones o el desplegado de programas simbólicos.

Calibrado basado en programación lógica con restricciones

En [RM20] hemos estudiado cómo es posible integrar las técnicas de calibrado de programas lógicos difusos con resolutores de satisfacibilidad módulo

teorías y hemos comprobado su viabilidad mediante una serie de casos prácticos. Como ya comentamos en la sección 5.4, una de las principales desventajas del algoritmo de calibrado basado en satisfacibilidad que hemos implementado con Z3 es que requiere una traducción manual del retículo asociado al programa, de Prolog a SMT-LIB.

Por su parte, la programación lógica con restricciones (CLP) es una extensión de la programación lógica que incluye conceptos sobre satisfacibilidad de restricciones [JL87]. Muchos sistemas Prolog modernos [KLB⁺22] extienden la noción de variable lógica para permitir que una variable se ligue a un dominio en lugar de a un valor en concreto, o para establecer restricciones sobre los valores a los que se puede ligar [Boi86, BS91]. Por ejemplo, SWI-Prolog permite extender el mecanismo de unificación [Neu90] mediante variables con atributos, originalmente definidas en [Hol92]. Esto ha permitido implementar en Prolog diversas bibliotecas de CLP sobre distintos dominios [Tri12, Tri16, Tri18].

Como línea de trabajo futuro nos proponemos integrar las técnicas de calibrado de programas simbólicos con estas bibliotecas de programación lógica con restricciones con el fin de mejorar la eficiencia del calibrado sin la necesidad de tener que expresar el retículo en dos lenguajes diferentes.

Desplegado difuso basado en guardas

En [JIMR22a] hemos definido una transformación de desplegado difuso que, en general, no siempre preserva las respuestas computadas difusas del programa original. Por lo tanto, hemos caracterizado los programas que pueden ser desplegados de forma segura imponiendo una serie de condiciones de seguridad que las reglas deben satisfacer. Dado que la función de selección de FASILL no es independiente, cuando no es posible desplegar de forma segura una regla resolviendo el átomo más a la izquierda, entonces no es posible seguir desplegando dicha regla, ya que resolver cualquier otro átomo podría afectar a la corrección de la transformación.

En esta línea, estamos trabajando en un paso de desplegado alternativo que, dada una regla que no pueda ser desplegada de forma segura en base a la definición 6.7, permita dar un paso de computación sobre su cuerpo sin aplicar las partes inseguras de los w.m.g.u. generados. Para ello, el átomo explotado es reemplazado por una serie de guardas cuyas condiciones son la representación ecuacional de las partes inseguras de los unificadores débiles y cuyos cuerpos son el cuerpo de las reglas con cuyas cabezas unificó el átomo explotado (a los que se les aplica la parte segura de las sustituciones). Para llevar a cabo esta transformación de desplegado difuso basado en guardas debemos introducir una nueva estructura de control en la semántica operacional de FASILL, que se comportaría como una combinación de los operadores $(->)/2$ (condicional), $(;)/2$ (disyunción) y $(*->)/2$ (corte blando) de Prolog, de forma que:

- el operador condicional nos permite expresar una guarda que ejecuta el cuerpo si el problema de unificación tiene éxito;

- la disyunción nos permite expresar varias guardas que simulan por reevaluación todas las posibles formas de dar un paso de computación explotando un átomo (comportándose como un paso de éxito en la semántica operacional de FASILL);
- y el corte blando, en combinación con la disyunción, nos permite definir una rama de fallo que se ejecuta únicamente si todas las guardas anteriores fallan (comportándose como un paso de fallo en la semántica operacional de FASILL).

Aunque este paso de desplegado basado en guardas no genera código más eficiente por sí mismo (en el sentido de que no reduce la longitud de las derivaciones) puede ser útil para habilitar futuros pasos de desplegado (clásico) en los cuerpos de las guardas que sí mejoran la eficiencia del programa.

Desplegado difuso de programas simbólicos

En [MPR19a, MR19b] hemos aplicado la transformación de desplegado difuso sobre programas simbólicos para tratar de reconciliar los tiempos de ejecución de los algoritmos de calibrado básico y simbólico. La idea es conseguir una mejora significativa en el tiempo de ejecución del método básico sin volver a incurrir en el principal inconveniente del método simbólico: descartar sustituciones simbólicas inseguras que, en ocasiones, resultan ser las más óptimas. Una solución al problema de las sustituciones simbólicas inseguras podría pasar por diseñar un nuevo algoritmo de calibrado que combine los métodos de calibrado básico y simbólico para que, en lugar de descartar las sustituciones simbólicas inseguras, estas sean comprobadas en $\mathcal{P}^{\#}\Theta$ a partir de los objetivos originales de los casos de prueba (como en el método básico) en lugar de a partir de las s.f.c.a. de los objetivos de los casos de prueba (como en el método simbólico).

Aunque podemos aplicar la transformación de desplegado sobre programas simbólicos (ya que la implementación de la semántica operacional simbólica es transparente a la del desplegado) todavía no hemos estudiado la corrección de esta transformación en el marco de sFASILL. Como línea de trabajo futuro nos proponemos estudiar las propiedades fundamentales de la transformación de desplegado difuso sobre sFASILL con el fin de mejorar la eficiencia de los algoritmos de calibrado que no hacen uso de la semántica operacional de la extensión simbólica. En esta línea, el desplegado difuso basado en guardas podría ser crucial para gestionar de forma segura las similitudes simbólicas entre (los átomos de) los cuerpos de las reglas desplegadas y las cabezas de las reglas desplegadas.

Anexo A

Retículos completos en FASILL

El lenguaje FASILL permite que cada programa lógico difuso describa su propia noción de verdad y sus propias conectivas difusas mediante la definición de un retículo completo asociado al programa. En este anexo describimos los retículos utilizados a lo largo de la memoria, incluyendo las conectivas definidas sobre los mismos, y algunos aspectos prácticos relacionados con el sistema FASILL. Además, proporcionamos el código completo de su implementación, tanto en Prolog como en SMT-LIB.

A.1. Retículo en el intervalo unitario

El *intervalo unitario* I (véase la figura A.1) es el intervalo cerrado $[0, 1]$, compuesto por todos los números reales mayores o iguales que 0 y menores o iguales que 1. Este intervalo es un conjunto totalmente ordenado y un retículo completo con la relación de orden usual de los números reales: (I, \leq) . En particular, el ínfimo y el supremo de I son respectivamente los valores 0 y 1.

```
1 member(X) :- number(X), X >= 0.0, X <= 1.0.  
2 leq(X, Y) :- X <= Y.  
3 bot(0.0).  
4 top(1.0).  
5 supremum(X,Y,Z) :- Z is max(X,Y).  
6 distance(X,Y,Z) :- Z is abs(X-Y).
```

Es importante remarcar que, debido a la naturaleza simbólica tanto de Prolog como de FASILL, en estos lenguajes los términos 0.0 y 1.0 son distintos



Figura A.1: Intervalo unitario como subconjunto de la recta real.

$F_{\&godel}(x, y) = \min\{x, y\}$	$F_{ godel}(x, y) = \max\{x, y\}$
$F_{\&luka}(x, y) = \max\{x + y - 1, 0\}$	$F_{ luka}(x, y) = \min\{x + y, 1\}$
$F_{\&prod}(x, y) = xy$	$F_{ prod}(x, y) = x + y - xy$
$F_{@aver}(x, y) = (x + y)/2$	$F_{@geom}(x, y) = \sqrt{xy}$
$F_{@very}(x) = x^2$	

Figura A.2: Funciones de verdad de las conectivas difusas definidas en el retículo unitario (I, \leq) .

a los términos 0 y 1, aunque su evaluación aritmética sí produce los mismos resultados. Es decir, el objetivo “0 = 0.0” falla (los términos no unifican), pero el objetivo “0 =:= 0.0” tiene éxito (los términos se evalúan al mismo valor). Por lo tanto, cuando se utiliza el retículo unitario en un programa FASILL, es aconsejable denotar los valores 0 y 1 mediante los términos 0.0 y 1.0. Si queremos forzar el uso de estos términos, podemos reemplazar la descripción anterior del predicado `member/1` por la siguiente definición que comprueba si el término es un número real.

```
1 member(X) :- float(X), X >= 0.0, X <= 1.0.
```

Este retículo viene cargado por defecto en el sistema FASILL y equipado con las conectivas procedentes de las lógicas de Gödel, de Łukasiewicz y del producto, junto a los agregadores de la media aritmética, la media geométrica y el modificador lingüístico “muy”, cuyas funciones de verdad se resumen en la figura A.2. Estas conectivas se codifican en Prolog como sigue:

```
1 and_prod(X,Y,Z) :- Z is X*Y.
2 and_godel(X,Y,Z) :- Z is min(X,Y).
3 and_luka(X,Y,Z) :- Z is max(X+Y-1.0,0.0).
4 or_prod(X,Y,Z) :- Z is (X+Y)-(X*Y).
5 or_godel(X,Y,Z) :- Z is max(X,Y).
6 or_luka(X,Y,Z) :- Z is min(X+Y,1.0).
7 agr_aver(X,Y,Z) :- Z is (X+Y)/2.
8 agr_geom(X,Y,Z) :- Z is sqrt(X*Y).
9 agr_very(X,Y) :- Y is X*X.
```

El código completo de este retículo codificado en Prolog¹ y en SMT-LIB² puede consultarse en el repositorio de FASILL.

¹<https://github.com/jariazavalverde/fasill/blob/master/lattices/real.lat.pl>

²<https://github.com/jariazavalverde/fasill/blob/master/lattices/real.lat.smt2>



Figura A.3: Diagrama de Hasse del conjunto ordenado $\mathcal{B} = \{0, 1\}$.

A.2. Retículo booleano

En álgebra, un retículo booleano [MO97] es un retículo (L, \leq) acotado y distributivo en el que para cada elemento $x \in L$ existe un único elemento complementario $\neg x \in L$. Un ejemplo canónico de retículo booleano completo es el conjunto $\mathcal{B} = \{0, 1\}$, donde 0 y 1 representan respectivamente los valores de verdad “falso” y “verdadero” y cuya relación de orden \leq se define de la siguiente forma:

$$x \leq y \text{ si y solo si } x = 0 \text{ o } y = 1,$$

que se ilustra como un diagrama de Hasse en la figura A.3. En esta memoria nos referimos por *retículo booleano* a este retículo de dos elementos (\mathcal{B}, \leq) , no a su definición más general.

```

1 member(false). member(true).
2 leq(false, _). leq(_, true).
3 bot(false).
4 top(true).
5 supremum(false,Y,Y). supremum(true,_,true).
6 distance(X,X,0). distance(_,_,1).

```

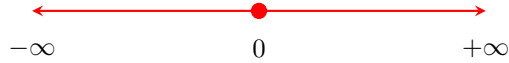
En FASILL representamos los elementos de este retículo mediante los átomos `false` y `true`, y utilizamos la noción de distancia discreta a la hora de calibrar programas lógicos (véase el ejemplo 5.1). Además, equipamos este retículos con las conectivas lógicas usuales: la conjunción, la disyunción, la disyunción exclusiva y la negación.

```

1 and_bool(true,true,true). and_bool(_,_,false).
2 or_bool(false,false,false). or_bool(_,_,true).
3 agr_xor(X,X,false). agr_xor(_,_,true).
4 agr_not(true, false). agr_not(false,true).

```

Nótese que los agregadores $@_{xor}$ y $@_{not}$ no satisfacen las condiciones que exigimos en la definición 2.36. En particular, no son monótonos crecientes ni cumplen las condiciones de frontera. Sin embargo, en la práctica resultan de utilidad y son convenientes para algunos programas como, por ejemplo, en la verificación

Figura A.4: Recta real extendida $\overline{\mathbb{R}}$.

de la equivalencia de circuitos combinatoriales descrita en la sección 5.6.1. El código completo de este retículo codificado en Prolog³ y en SMT-LIB⁴ puede consultarse en el repositorio de FASILL.

A.3. Retículo en la recta real extendida

La *recta real extendida* $\overline{\mathbb{R}}$ (véase la figura A.4) se obtiene a partir de los números reales \mathbb{R} al añadir dos nuevos elementos: $+\infty$ (infinito positivo) y $-\infty$ (infinito negativo). La recta real extendida puede convertirse en un conjunto totalmente ordenado al definir $-\infty \leq x \leq +\infty$ para todo $x \in \mathbb{R}$, y conforma un retículo completo, donde los elementos $-\infty$ y $+\infty$ son respectivamente el ínfimo y el supremo de $\overline{\mathbb{R}}$.

```

1 member(+inf). member(-inf). member(X) :- number(X).
2 leq(X, Y) :- X =< Y.
3 bot(-inf).
4 top(+inf).
5 supremum(X,Y,Z) :- Z is max(X,Y).
6 distance(X,Y,Z) :- Z is abs(X-Y).

```

En FASILL representamos los elementos de \mathbb{R} como números y los elementos $+\infty$ y $-\infty$ como los términos compuestos `+inf` y `-inf`. En SWI-Prolog, el átomo `inf` representa una función evaluable y, por lo tanto, podemos utilizar los predicados de comparación y evaluación aritmética usuales sobre ellos. En un sistema Prolog que no soporte tal función podemos definir explícitamente el comportamiento de estos elementos, por ejemplo:

```

1 leq(-inf, _). leq(_, +inf).
2 leq(X, Y) :- number(X), number(Y), X =< Y.

```

Además, equipamos este retículo con las siguientes conectivas, cuyas funciones de verdad se describen en la figura A.5.

```

1 and_min(X,Y,Z) :- Z is min(X,Y).
2 or_max(X,Y,Z) :- Z is max(X,Y).
3 or_add(+inf,_,+inf). or_add(_,+inf,+inf).
4 or_add(-inf,_,-inf). or_add(-,-,-inf).

```

³<https://github.com/jariazavalverde/fasill/blob/master/lattices/bool.lat.pl>

⁴<https://github.com/jariazavalverde/fasill/blob/master/lattices/bool.lat.smt2>

```

5 or_add(X,Y,Z) :- Z is X+Y.
6 agr_mul(X,Y,Z) :- Z is X*Y.

```

Se observa que, al igual que algunos agregadores definidos en el retículo booleano, el agregador $@_{mul}$ no satisface las condiciones que exigimos en la definición 2.36, pero resulta de utilidad en algunos problemas, como por ejemplo el calibrado de modelos de regresión lineal que se describe en la sección 5.6.2. De hecho, dejamos como indefinido su comportamiento cuando alguna de las entradas es $+\infty$ o $-\infty$. El código completo de este retículo codificado en Prolog⁵ y en SMT-LIB⁶ puede consultarse en el repositorio de FASILL.

$$\begin{array}{ll}
 F_{\&min}(x, y) = \text{mín}\{x, y\} & F_{|max}(x, y) = \text{máx}\{x, y\} \\
 F_{|add}(x, y) = \begin{cases} +\infty & \text{si } x = +\infty \text{ o } y = +\infty \\ -\infty & \text{si } x = -\infty \text{ o } y = -\infty \\ x + y & \text{si } x, y \in \mathbb{R} \end{cases} & F_{@mul}(x, y) = xy
 \end{array}$$

Figura A.5: Funciones de verdad de las conectivas difusas definidas en el retículo de la recta real extendida (\mathbb{R}, \leq) .

A.4. Retículo para FSA-SPARQL

Las consultas FSA-SPARQL actúan sobre grados de verdad en el intervalo unitario $[0, 1]$. Sin embargo, los conjuntos difusos se representan como funciones trapezoidales (véase la sección 5.6.4) que toman valores en \mathbb{R} . Por lo tanto, el retículo asociado a los programas FSA-SPARQL es una variante del retículo unitario donde se relaja la noción de grado de verdad para aceptar cualquier valor real, aunque el ínfimo y el supremo siguen siendo respectivamente los elementos 0 y 1.

```

1 :- op(200, xfx, ^^).
2 member(X^^'http://www.w3.org/2001/XMLSchema#decimal') :- number(X).
3 leq(X^^T, Y^^T) :- X =< Y.
4 bot(0.0^^'http://www.w3.org/2001/XMLSchema#decimal').
5 top(1.0^^'http://www.w3.org/2001/XMLSchema#decimal').
6 supremum(X^^T, Y^^T, Z^^T) :- Z is max(X, Y).
7 distance(X^^T, Y^^T, Z^^T) :- Z is abs(X - Y).

```

Las consultas FSA-SPARQL devuelven el tipo de dato junto al resultado, así que hemos declarado el operador infijo $(^^)/2$ en el retículo para anotar el valor decimal en los grados de verdad. Aunque bajo esta definición del predicado

⁵<https://github.com/jariazavalverde/fasill/blob/master/lattices/real.lat.pl>

⁶<https://github.com/jariazavalverde/fasill/blob/master/lattices/real.lat.smt2>

$$\begin{aligned}
F_{@mean}(x, y) &= (x + y)/2 \\
F_{@wmean}(w, x, y) &= wx + (1 - w)y \\
F_{@wsum}(u, x, v, y) &= \begin{cases} ux + vy & \text{si } u + v \leq 1 \\ 0 & \text{si } u + v > 1 \end{cases} \\
F_{@wmax}(u, x, v, y) &= \text{máx}\{\text{mín}\{u, x\}, \text{mín}\{v, y\}\} \\
F_{@wmin}(u, x, v, y) &= \text{mín}\{\text{máx}\{1 - u, x\}, \text{máx}\{1 - v, y\}\} \\
F_{@very}(x) &= x^2 \\
F_{@more_or_less}(x) &= \sqrt{x} \\
F_{@close_to}(x, l, a) &= 1/(1 + ((x - l)/a)^2) \\
F_{@at_least}(x, l, a) &= \begin{cases} 0 & \text{si } x \leq a \\ (x - a)/(l - a) & \text{si } a < x < l \\ 1 & \text{si } l \leq x \end{cases} \\
F_{@at_most}(x, l, a) &= \begin{cases} 0 & \text{si } x \geq a \\ (a - x)/(a - l) & \text{si } a > x > l \\ 1 & \text{si } x \leq l \end{cases}
\end{aligned}$$

Figura A.6: Funciones de verdad de las conectivas difusas definidas en el retículo de FSA-SPARQL.

`member/1` FASILL puede interpretar cualquier valor de \mathbb{R} como un grado de verdad, los valores fuera del intervalo $[0, 1]$ sólo deberían utilizarse dentro del agregador `@trapezoidal` que modela la función trapezoidal descrita en la ecuación 5.4, que a su vez genera un valor en $[0, 1]$.

```

1 agr_trapezoidal(A^^T, _B^^T, _C^^T, D^^T, X^^T, 0^^T) :-
2   X < A ; X > D.
3 agr_trapezoidal(A^^T, B^^T, _C^^T, _D^^T, X^^T, Y^^T) :-
4   A =< X, X =< B, (B-A == 0 -> Y = 1 ; Y is (X-A)/(B-A)).
5 agr_trapezoidal(_A^^T, B^^T, C^^T, _D^^T, X^^T, 1^^T) :-
6   B =< X, X =< C.
7 agr_trapezoidal(_A^^T, _B^^T, C^^T, D^^T, X^^T, Y^^T) :-
8   C =< X, X =< D, (D-C == 0 -> Y = 1 ; Y is (D-X)/(D-C)).

```

Además de este agregador, el retículo está equipado con todos los operadores difusos definidos en FSA-SPARQL [AJBTMR22], entre los que se incluyen las conectivas procedentes de las lógicas de Gödel, de Łukasiewicz y del producto, junto al resto de conectivas que se describen en la figura A.6, y que se codifican en Prolog como sigue:

```

1 agr_mean(X^^T, Y^^T, Z^^T) :- Z is (X+Y)/2.
2 agr_wmean(W^^T, X^^T, Y^^T, Z^^T) :- Z is W*X+(1-W)*Y.

```

```

3 agr_wsum(U^^T,X^^T,V^^T,Y^^T,Z^^T):- U+V =< 1.0, Z is U*X+V*Y.
4 agr_wsum(U^^T,X^^T,V^^T,Y^^T,0.0^^T).
5 agr_wmax(U^^T,X^^T,V^^T,Y^^T,Z^^T) :-
6     Z is max(min(U,X),min(V,Y)).
7 agr_wmin(U^^T,X^^T,V^^T,Y^^T,Z^^T) :-
8     Z is min(max(1-U,X),max(1-V,Y)).
9 agr_very(X^^T,Z^^T) :- Z is X*X.
10 agr_more_or_less(X^^T,Z^^T) :- Z is sqrt(X).
11 agr_close_to(X^^T,L^^T,A^^T,Z^^T) :- Z is 1.0/(1.0+((X-L)/A)^2).
12 agr_at_least(X^^T,L^^T,A^^T,Z^^T) :- X =<= A, Z is 0.0.
13 agr_at_least(X^^T,L^^T,A^^T,Z^^T) :-
14     A < X, X < L, Z is (X-A)/(L-A).
15 agr_at_least(X^^T,L^^T,A^^T,Z^^T) :- L =<= X, Z is 1.0.
16 agr_at_most(X^^T,L^^T,A^^T,Z^^T) :- X >= A, Z is 0.0.
17 agr_at_most(X^^T,L^^T,A^^T,Z^^T) :-
18     A > X, X > L, Z is (A-X)/(A-L).
19 agr_at_most(X^^T,L^^T,A^^T,Z^^T) :- X =<= L, Z is 1.0.

```

El código completo de este retículo codificado en Prolog⁷ puede consultarse en el repositorio de FASILL.

⁷<https://github.com/jariazavalverde/fasill/blob/master/lattices/fsa.lat.pl>

Bibliografía

- [ABM17] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón y Ginés Moreno. «A fuzzy extension of SPARQL based on fuzzy sets and aggregators». En «2017 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2017, Naples, Italy, July 9-12, 2017», páginas 1–6. IEEE, 2017.
- [ABMV12] Carlos Ansótegui, Miquel Bofill, Felip Manyà y Mateu Villaret. «Building automated theorem provers for infinitely-valued logics with satisfiability modulo theory solvers». En «2012 IEEE 42nd International Symposium on Multiple-Valued Logic», páginas 25–30. IEEE, 2012.
- [AG98] Teresa Alsinet y Lluís Godo. «Fuzzy unification degree». En «Logic Programming and Soft Computing-Theory and Applications, A Post-conference Workshop of JICSLP'98», 1998.
- [AJBTM18] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón y Ginés Moreno. «Fuzzy queries of social networks with FSA-SPARQL». *Expert Systems with Applications*, vol. 113, páginas 128–146, 2018. ISSN 0957-4174.
- [AJBTMR19] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón, Ginés Moreno y José A. Riaza. «Tuning Fuzzy SPARQL Queries in a Fuzzy Logic Programming Environment». En «2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)», páginas 1–7. IEEE, 2019. URL: <https://doi.org/10.1109/FUZZ-IEEE.2019.8858958>.
- [AJBTMR21] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón, Ginés Moreno y José A. Riaza. «Flexible Aggregation in FSA-SPARQL». En «2021 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)», páginas 1–7. IEEE, 2021. URL: <https://doi.org/10.1109/FUZZ45933.2021.9494499>.
- [AJBTMR22] Jesús M. Almendros-Jiménez, Antonio Becerra-Terón, Ginés Moreno y José A. Riaza. «Tuning Fuzzy Sets and FSA-SPARQL

- queries for the Flexible Retrieval of Social Networks Data», 2022. Artículo en proceso de publicación.
- [AP90] Alexander V. Arkhangel'skiĭ y Lev S. Pontryagin. «General Topology I: Basic Concepts and Constructions, Dimension Theory». Springer-Verlag, 1990.
- [BD77] Rod M. Burstall y John Darlington. «A Transformation System for Developing Recursive Programs». *Journal of the ACM (JACM)*, vol. 24, páginas 44 – 67, 1977.
- [Ber97] Dimitri P. Bertsekas. «Nonlinear programming». *Journal of the Operational Research Society*, vol. 48 n^o 3, página 334, 1997.
- [Bis94] Chris M. Bishop. «Neural networks and their applications». *Review of scientific instruments*, vol. 65 n^o 6, páginas 1803–1832, 1994.
- [BLHL01] Tim Berners-Lee, James Hendler y Ora Lassila. «The semantic web». *Scientific american*, vol. 284 n^o 5, páginas 34–43, 2001.
- [BM09] Nikolaj Bjørner y Leonardo de Moura. «Z310: Applications, enablers, challenges and directions». En «Sixth international workshop on constraints in formal verification», volumen 16, 2009.
- [BMNW19] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson y Christoph M. Wintersteiger. «Programming Z3», páginas 148–201. Springer International Publishing, Cham, 2019. ISBN: 9783030176013.
- [BMP95] James F. Baldwin, Trevor P. Martin y B. W. Pilsworth. «Fuzzy and evidential reasoning in artificial intelligence». John Wiley & Sons, Inc., USA, 1995. ISBN: 047195523X.
- [BMR⁺11] Clark Barrett, Leonardo de Moura, Silvio Ranise, Aaron Stump y Cesare Tinelli. «The SMT-LIB Initiative and the Rise of SMT». En Sharon Barner, Ian Harris, Daniel Kroening y Orna Raz (Eds.), «Hardware and Software: Verification and Testing», página 3. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN: 9783642195839.
- [Boi86] P. Boizumault. «A general model to implement DIF and FREEZE». En Ehud Shapiro (Ed.), «Third International Conference on Logic Programming», páginas 585–592. Springer Berlin Heidelberg, 1986. ISBN: 9783540398318.
- [BPF15] Nikolaj Bjørner, Anh-Dung Phan y Lars Fleckenstein. « ν Z - An Optimizing SMT Solver». En Christel Baier y Cesare Tinelli (Eds.), «Tools and Algorithms for the Construction and Analysis

- of Systems», páginas 194–199. Springer Berlin Heidelberg, 2015. ISBN: 9783662466810.
- [BS91] E. Börger y Peter H. Schmitt. «A formal operational semantics for languages of type Prolog III». En Egon Börger, Hans Kleine Büning, Michael M. Richter y Wolfgang Schönfeld (Eds.), «Computer Science Logic», páginas 67–79. Springer Berlin Heidelberg, 1991. ISBN: 9783540384014.
- [BST⁺10] Clark Barrett, Aaron Stump, Cesare Tinelli et al. «The SMT-LIB Standard: Version 2.0». En «Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)», volumen 13, páginas 1–14, 2010.
- [BT18] Clark Barrett y Cesare Tinelli. «Satisfiability modulo theories». En «Handbook of model checking», páginas 305–343. Springer, 2018.
- [Car42] Rudolf Carnap. «Introduction to Semantics». Harvard University Press, 1942.
- [CFC⁺58] Haskell B. Curry, Robert Feys, William Craig, J. Roger Hindley y Jonathan P. Seldin. «Combinatory logic», volumen 1. North-Holland Amsterdam, 1958.
- [CFF97] Christer Carlsson, Robert Fullér y Szeptelana Fullér. «Possibility and necessity in weighted aggregation». En «The Ordered Weighted Averaging Operators», páginas 18–28. Springer, 1997.
- [CL14] Chin-Liang Chang y Richard Char-Tung Lee. «Symbolic logic and mechanical theorem proving». Academic press, 2014.
- [CR96] Alain Colmerauer y Philippe Roussel. «The birth of Prolog». En «History of programming languages—II», páginas 331–367. Association for Computing Machinery, 1996.
- [CRARD08] Rafael Caballero, Mario Rodríguez-Artalejo y Carlos A. Romero-Díaz. «Similarity-Based Reasoning in Qualified Logic Programming». En «Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming», PPDP '08, página 185–194. Association for Computing Machinery, New York, NY, USA, 2008. ISBN: 9781605581170.
- [DKMS07] Fabrizio Durante, Erich P. Klement, Radko Mesiar y Carlo Sempi. «Conjunctors and their Residual Implicators: Characterizations and Construction Methods». *Mediterranean Journal of Mathematics*, vol. 4, páginas 343–356, 2007.

- [DP85] Didier Dubois y Henri Prade. «A review of fuzzy set aggregation connectives». *Information Sciences*, vol. 36 n^o 1, páginas 85–121, 1985. ISSN 0020-0255.
- [Ede85] Elmar Eder. «Properties of Substitutions and Unifications». *J. Symb. Comput.*, vol. 1 n^o 1, páginas 31–46, 1985.
- [FC98] János Fodor y Tomasa Calvo. «Aggregation functions defined by t-norms and t-conorms», páginas 36–48. Physica-Verlag HD, Heidelberg, 1998. ISBN: 9783790818895.
- [FGS00] Ferrante Formato, Giangiacomo Gerla y Maria I. Sessa. «Similarity-Based Unification». *Fundam. Inf.*, vol. 41 n^o 4, página 393–414, 2000. ISSN 0169-2968.
- [Fis36] Ronald A. Fisher. «The use of multiple measurements in taxonomic problems». *Annals of eugenics*, vol. 7 n^o 2, páginas 179–188, 1936.
- [GMM⁺17] Juan A. Guerrero, Félix Mendieta, Ginés Moreno, Jaime Penabad y José A. Ríaza. «Testing properties of fuzzy connectives and truth degrees with the latticemaker tool». En «2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA, November 27 - Dec. 1, 2017», páginas 1–8. IEEE, 2017. URL: <https://doi.org/10.1109/SSCI.2017.8280961>.
- [GMRS18] Juan A. Guerrero, Ginés Moreno, José A. Ríaza y Javier Sánchez. «Smart Design of Similarity Relations for Fuzzy Logic Programming Environments». En «2018 IEEE Symposium Series on Computational Intelligence (SSCI)», páginas 220–227. IEEE, 2018. URL: <https://doi.org/10.1109/SSCI.2018.8628871>.
- [GS00] David Gilbert y Michael Schroeder. «FURY: Fuzzy unification and resolution based on edit distance». En «Proceedings IEEE International Symposium on Bio-Informatics and Biomedical Engineering», páginas 330–336. IEEE, 2000.
- [Her30] Jacques Herbrand. «Recherches sur la théorie de la démonstration». Tesis doctoral, J. Dziewulski, 1930.
- [Hod13] Richard E. Hodel. «An introduction to mathematical logic». Courier Corporation, 2013.
- [Hof79] Douglas R. Hofstadter. «Gödel, Escher, Bach». Harvester press London, 1979.
- [Hol92] Christian Holzbaur. «Metastructures vs. attributed variables in the context of extensible unification». En «International Symposium on Programming Language Implementation and Logic Programming», páginas 260–268. Springer, 1992.

- [HV09] Kryštof Hoder y Andrei Voronkov. «Comparing Unification Algorithms in First-Order Theorem Proving». En Bärbel Mertsching, Marcus Hund y Zaheer Aziz (Eds.), «KI 2009: Advances in Artificial Intelligence», páginas 435–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN: 9783642046179.
- [HY01] David J. Hand y Keming Yu. «Idiot’s Bayes—not so stupid after all?». *International statistical review*, vol. 69 n^o 3, páginas 385–398, 2001.
- [II65] Yasuyuki Imai y Kiyoshi Iséki. «On axiom systems of propositional calculi. I». *Proceedings of the Japan Academy*, vol. 41 n^o 6, páginas 436–439, 1965.
- [IK85] Mitsuru Ishizuka y Naoki Kanai. «Prolog-ELF incorporating fuzzy logic». *New Generation Computing*, vol. 3 n^o 4, páginas 479–486, 1985.
- [ISO95] «Information technology – Programming languages – Prolog – Part 1: General core», 1995.
- [JI04] Pascual Julián-Iranzo. «Lógica simbólica para informáticos». Editorial RA-MA, 2004.
- [JI08] Pascual Julián-Iranzo. «A procedure for the construction of a similarity relation». En J.L. Verdegay L. Magdalena, M. Ojeda-Aciego (Ed.), «Proceedings of the 12th International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems (IPMU 2008), June 22-27, Torremolinos (Málaga), Spain», páginas 489–496. U. Málaga (ISBN 978-84-612-3061-7), 2008.
- [JIA07] Pascual Julián-Iranzo y María Alpuente. «Programación lógica: teoría y práctica», volumen 1. PEARSON, 2007.
- [JIMM⁺13] Pascual Julián-Iranzo, Jesús Medina, Pedro J. Morcillo, Ginés Moreno y Manuel Ojeda-Aciego. «An Unfolding-Based Preprocess for Reinforcing Thresholds in Fuzzy Tabulation». En Ignacio Rojas, Gonzalo Joya y Joan Gabestany (Eds.), «Advances in Computational Intelligence», páginas 647–655. Springer Berlin Heidelberg, 2013. ISBN: 9783642386794.
- [JIMOA17] Pascual Julián-Iranzo, Jesús Medina y Manuel Ojeda-Aciego. «On reductants in the framework of multi-adjoint logic programming». *Fuzzy Sets and Systems*, vol. 317, páginas 27–43, 2017. ISSN 0165-0114. Theme: Logic and Computer Science.
- [JIMP05a] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «On fuzzy unfolding: A multi-adjoint approach». *Fuzzy Sets and Systems*, vol. 154 n^o 1, páginas 16–33, 2005. ISSN 0165-0114.

- [JIMP05b] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «Unfolding-based Improvements on Fuzzy Logic Programs». *Electronic Notes in Theoretical Computer Science*, vol. 137 n^o 1, páginas 69–103, 2005. ISSN 1571-0661. Proceedings of the Fourth Spanish Conference on Programming and Computer Languages (PROLE 2004).
- [JIMP06] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «Operational/Interpretive Unfolding of Multi-adjoint Logic Programs». *J. Univers. Comput. Sci.*, vol. 12 n^o 11, páginas 1679–1699, 2006.
- [JIMP09] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «On the Declarative Semantics of Multi-Adjoint Logic Programs». En Joan Cabestany, Francisco Sandoval, Alberto Prieto y Juan M. Corchado (Eds.), «Bio-Inspired Systems: Computational and Ambient Intelligence», páginas 253–260. Springer Berlin Heidelberg, 2009. ISBN: 9783642024788.
- [JIMP17] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «Thresholed semantic framework for a fully integrated fuzzy logic language». *Journal of Logical and Algebraic Methods in Programming*, vol. 93, páginas 42–67, 2017. ISSN 2352-2208.
- [JIMP18] Pascual Julián-Iranzo, Ginés Moreno y Jaime Penabad. «FASILL: Fuzzy Correct Answers and Soundness». En «2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)», página 1–8. IEEE, 2018.
- [JIMR20] Pascual Julián-Iranzo, Ginés Moreno y José A. Riaza. «The Fuzzy Logic Programming language FASILL: Design and implementation». *International Journal of Approximate Reasoning*, vol. 125, páginas 139–168, 2020. ISSN 0888-613X. URL: <https://www.sciencedirect.com/science/article/pii/S0888613X2030181X>.
- [JIMR22a] Pascual Julián-Iranzo, Ginés Moreno y José A. Riaza. «Seeking a Safe and Efficient Similarity-based Unfolding Rule», 2022. Artículo en proceso de publicación.
- [JIMR22b] Pascual Julián-Iranzo, Ginés Moreno y José A. Riaza. «Some Properties of Substitutions in the Framework of Similarity Relations», 2022. Artículo en proceso de publicación (en segunda revisión).
- [JIRM09a] Pascual Julián-Iranzo y Clemente Rubio-Manzano. «A declarative semantics for Bousi~Prolog». En «Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming», PPDP'09, página 149–160. Association

- for Computing Machinery, New York, NY, USA, 2009. ISBN: 9781605585680.
- [JIRM09b] Pascual Julián-Iranzo y Clemente Rubio-Manzano. «A Similarity-Based WAM for Bousi~Prolog». En Joan Cabestany, Francisco Sandoval, Alberto Prieto y Juan M. Corchado (Eds.), «Bio-Inspired Systems: Computational and Ambient Intelligence», páginas 245–252. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN: 9783642024788.
- [JIRM10] Pascual Julián-Iranzo y Clemente Rubio-Manzano. «An efficient fuzzy unification method and its implementation into the Bousi~Prolog system». En «International Conference on Fuzzy Systems», páginas 1–8. IEEE, 2010.
- [JIRM17] Pascual Julián-Iranzo y Clemente Rubio-Manzano. «A sound and complete semantics for a similarity-based logic programming language». *Fuzzy Sets and Systems*, vol. 317, páginas 1–26, 2017. ISSN 0165-0114. Theme: Logic and Computer Science.
- [JIRMG09] Pascual Julián-Iranzo, Clemente Rubio-Manzano y Juan Gallardo-Casero. «Bousi~Prolog: a Prolog extension language for flexible query answering». *Electronic Notes in Theoretical Computer Science*, vol. 248, páginas 131–147, 2009.
- [JISP20] Pascual Julián-Iranzo y Fernando Sáenz-Pérez. «Proximity-based unification: an efficient implementation method». *IEEE Transactions on Fuzzy Systems*, vol. 29 n^o 5, páginas 1238–1251, 2020.
- [JL87] Joxan Jaffar y Jean-Louis Lassez. «Constraint logic programming». En «Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages», POPL'87, página 111–119. Association for Computing Machinery, New York, NY, USA, 1987. ISBN: 0897912152.
- [KK99] Anna Kolesárová y Magda Komorníková. «Triangular norm-based iterative compensatory operators». *Fuzzy Sets and Systems*, vol. 104 n^o 1, páginas 109–120, 1999.
- [KLB⁺22] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu et al. «Fifty Years of Prolog and Beyond». *Theory and Practice of Logic Programming*, páginas 1-83, 2022.
- [KMP04] Erich P. Klement, Radko Mesiar y Endre Pap. «Triangular norms. Position paper II: general constructions and parameterized families». *Fuzzy Sets and Systems*, vol. 145 n^o 3, páginas 411–438, 2004. ISSN 0165-0114.

- [Kow74] Robert Kowalski. «Predicate logic as programming language». En «IFIP congress», volumen 74, páginas 569–544. North-Holland, 1974.
- [KY74] Abraham Kandel y Lawrence Yelowitz. «Fuzzy chains». *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-4 n^o 5, páginas 472–475, 1974.
- [Lee72] Richard C. T. Lee. «Fuzzy Logic and the Resolution Principle». *J. ACM*, vol. 19 n^o 1, página 109–119, jan de 1972. ISSN 0004-5411.
- [LL90] Deyi Li y Dongbo Liu. «A fuzzy prolog database system». John Wiley & Sons, Inc., USA, 1990. ISBN: 0471927627.
- [Llo12] John W. Lloyd. «Foundations of logic programming». Springer Science & Business Media, 2012.
- [LM88] Giorgio Levi y Paolo Mancarella. «The unfolding semantics of logic programs». Università degli studi di Pisa, Dipartimento di informatica, 1988.
- [LSS01] Vincenzo Loia, Sabrina Senatore y Maria I. Sessa. «Similarity-based SLD resolution and its implementation in an extended Prolog system». En «10th IEEE International Conference on Fuzzy Systems. (Cat. No.01CH37297)», volumen 2, páginas 650–653 vol.3. IEEE, 2001.
- [MB08] Leonardo de Moura y Nikolaj Bjørner. «Z3: An efficient SMT solver». En «International conference on Tools and Algorithms for the Construction and Analysis of Systems», páginas 337–340. Springer, 2008.
- [MB12] Leonardo de Moura y Nikolaj Bjørner. «Z3-a tutorial». Microsoft, Albuquerque, NM, USA, Tech. Rep, 2012.
- [Miz89a] Masaharu Mizumoto. «Pictorial representations of fuzzy connectives, Part I: Cases of t-norms, t-conorms and averaging operators». *Fuzzy Sets and Systems*, vol. 31 n^o 2, páginas 217–242, 1989. ISSN 0165-0114.
- [Miz89b] Masaharu Mizumoto. «Pictorial representations of fuzzy connectives, Part II: Cases of compensatory operators and self-dual operators». *Fuzzy Sets and Systems*, vol. 32 n^o 1, páginas 45–79, 1989. ISSN 0165-0114.
- [MM82] Alberto Martelli y Ugo Montanari. «An efficient unification algorithm». *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4 n^o 2, páginas 258–282, 1982.

- [MO97] Markus Müller-Olm. «Complete boolean lattices», páginas 9–14. Springer Berlin Heidelberg, 1997. ISBN: 9783540695394.
- [MOAV01a] Jesús Medina, Manuel Ojeda-Aciego y Peter Vojtaš. «Multi-adjoint Logic Programming with Continuous Semantics». En Thomas Eiter, Wolfgang Faber y Mirosław Truszczyński (Eds.), «Logic Programming and Nonmonotonic Reasoning», páginas 351–364. Springer Berlin Heidelberg, 2001. ISBN: 9783540454021.
- [MOAV01b] Jesús Medina, Manuel Ojeda-Aciego y Peter Vojtaš. «A procedural semantics for multi-adjoint logic programming». En «Portuguese Conference on Artificial Intelligence», páginas 290–297. Springer, 2001.
- [MOAV04] Jesús Medina, Manuel Ojeda-Aciego y Peter Vojtaš. «Similarity-based unification: a multi-adjoint approach». *Fuzzy Sets and Systems*, vol. 146 n^o 1, páginas 43–62, 2004. ISSN 0165-0114. Selected Papers from EUSFLAT 2001.
- [MPR17] Ginés Moreno, Jaime Penabad y José A. Riaza. «On Similarity-Based Unfolding». En Serafín Moral, Olivier Pivert, Daniel Sánchez y Nicolás Marín (Eds.), «Scalable Uncertainty Management», páginas 420–426. Springer International Publishing, Cham, 2017. ISBN: 9783319675824. URL: https://doi.org/10.1007/978-3-319-67582-4_32.
- [MPR19a] Ginés Moreno, Jaime Penabad y José A. Riaza. «Symbolic unfolding of multi-adjoint logic programs», páginas 43–51. Springer International Publishing, Cham, 2019. ISBN: 9783030004859. URL: https://doi.org/10.1007/978-3-030-00485-9_5.
- [MPR19b] Ginés Moreno, Jesús Pérez y José A. Riaza. «Fuzzy Logic Programming for Tuning Neural Networks». En Paul Fodor, Marco Montali, Diego Calvanese y Dumitru Roman (Eds.), «Rules and Reasoning», páginas 190–197. Springer International Publishing, Cham, 2019. ISBN: 9783030310950. URL: https://doi.org/10.1007/978-3-030-31095-0_14.
- [MPRV17] Ginés Moreno, Jaime Penabad, José A. Riaza y Germán Vidal. «Symbolic Execution and Thresholding for Efficiently Tuning Fuzzy Logic Programs». En Manuel V. Hermenegildo y Pedro Lopez-García (Eds.), «Logic-Based Program Synthesis and Transformation», páginas 131–147. Springer International Publishing, Cham, 2017. ISBN: 9783319631394. URL: https://doi.org/10.1007/978-3-319-63139-4_8.
- [MPV21] Douglas C. Montgomery, Elizabeth A. Peck y G. Geoffrey Vining. «Introduction to linear regression analysis». John Wiley & Sons, 2021.

- [MR17] Ginés Moreno y José A. Riaza. «An Online Tool for Tuning Fuzzy Logic Programs». En Stefania Costantini, Enrico Franconi, William Van Woensel, Roman Kontchakov, Fariba Sadri y Dumitru Roman (Eds.), «Rules and Reasoning», páginas 184–198. Springer International Publishing, Cham, 2017. ISBN: 9783319612522. URL: https://doi.org/10.1007/978-3-319-61252-2_13.
- [MR19a] Ginés Moreno y José A. Riaza. «An Online Tool for Unfolding Symbolic Fuzzy Logic Programs». En Ignacio Rojas, Gonzalo Joya y Andreu Catala (Eds.), «Advances in Computational Intelligence», páginas 475–487. Springer International Publishing, Cham, 2019. ISBN: 9783030205188. URL: https://doi.org/10.1007/978-3-030-20518-8_40.
- [MR19b] Ginés Moreno y José A. Riaza. «Combining Symbolic Unfolding and Tuning Techniques for Fuzzy Logic Programs». En «11th European Symposium on Computational Intelligence and Mathematics, ESCIM'19», páginas 1–12. Toledo, España, 2019. URL: http://escim2019.uca.es/wp-content/uploads/2019/12/BA_ESCIM2019.pdf.
- [MR20] Ginés Moreno y José A. Riaza. «Symbolic Similarity Relations for Tuning Fully Integrated Fuzzy Logic Programs». En Víctor Gutiérrez-Basulto, Tomáš Kliegr, Ahmet Soylu, Martin Giese y Dumitru Roman (Eds.), «Rules and Reasoning», páginas 150–158. Springer International Publishing, Cham, 2020. ISBN: 9783030579777. URL: https://doi.org/10.1007/978-3-030-57977-7_11.
- [MR21] Ginés Moreno y José A. Riaza. «A Safe and Effective Tuning Technique for Similarity-Based Fuzzy Logic Programs». En Ignacio Rojas, Gonzalo Joya y Andreu Català (Eds.), «Advances in Computational Intelligence», páginas 190–201. Springer International Publishing, Cham, 2021. ISBN: 9783030850302. URL: https://doi.org/10.1007/978-3-030-85030-2_16.
- [MS08] João Marques-Silva. «Practical applications of Boolean Satisfiability». En «2008 9th International Workshop on Discrete Event Systems», páginas 74–80. IEEE, 2008.
- [MSD89] Masao Mukaidono, Zuliang Shen y Liya Ding. «Fundamentals of fuzzy prolog». *International Journal of Approximate Reasoning*, vol. 3 n^o 2, páginas 179–193, 1989.
- [MSG99] João Marques-Silva y Thomas Glass. «Combinational Equivalence Checking Using Satisfiability and Recursive Learning». En «Proceedings of the Conference on Design, Automation and Test in Europe», DATE '99, páginas 145–149. Association for Computing Machinery, New York, NY, USA, 1999. ISBN: 1581131216.

- [MW12] Sharad Malik y Georg Weissenbacher. «Boolean satisfiability solvers: techniques and extensions». En «Software Safety and Security—Tools for Analysis and Verification». IOS Press, 2012.
- [NDMDB02] Helga Naessens, Hans De Meyer y Bernard De Baets. «Algorithms for the computation of T-transitive closures». *IEEE Transactions on Fuzzy Systems*, vol. 10 n^o 4, páginas 541–551, 2002.
- [Neu90] Ulrich Neumerkel. «Extensible unification by metastructures». *Proceeding of the META*, vol. 90, 1990.
- [NKNW96] John Neter, Michael H. Kutner, Christopher J. Nachtsheim y William Wasserman. «Applied linear statistical models», volumen 4. Irwin Chicago, 1996.
- [PAH05] Germán Puebla, Elvira Albert y Manuel V. Hermenegildo. «Efficient Local Unfolding with Ancestor Stacks for Full Prolog». En Sandro Etalle (Ed.), «Logic Based Program Synthesis and Transformation», páginas 149–165. Springer Berlin Heidelberg, 2005. ISBN: 9783540316831.
- [Pal90] Catuscia Palamidessi. «Algebraic properties of idempotent substitutions». En «International Colloquium on Automata, Languages, and Programming», páginas 386–399. Springer, 1990.
- [Pie46] George W. Pierce. «The songs of insects; with related material on the production, propagation, detection, and measurement of sonic and supersonic vibrations». Harvard University Press, 1946.
- [PP94] Alberto Pettorossi y Maurizio Proietti. «Transformation of logic programs: Foundations and techniques». *The Journal of Logic Programming*, vol. 19-20, páginas 261–320, 1994. ISSN 0743-1066. Special Issue: Ten Years of Logic Programming.
- [PP96] Alberto Pettorossi y Maurizio Proietti. «Rules and Strategies for Transforming Functional and Logic Programs». *ACM Comput. Surv.*, vol. 28 n^o 2, página 360–414, jun de 1996. ISSN 0360-0300.
- [PP98] Alberto Pettorossi y Maurizio Proietti. «Transformation of logic programs». *Handbook of logic in artificial intelligence and logic programming*, vol. 5, páginas 697–787, 1998.
- [Pre93] Steven Prestwich. «An unfold rule for full prolog», páginas 199–213. Springer London, London, 1993. ISBN: 9781447135609.
- [Ria22a] José A. Rianza. «Tau Prolog: A Prolog interpreter for the Web», 2022. Artículo en proceso de publicación (en tercera revisión).

- [Ria22b] José A. Riaza. «Web development with Tau Prolog». En Pascual Julián Iranzo (Ed.), «Actas de las XXI Jornadas sobre Programación y Lenguajes, PROLE'22», páginas 1–14. SISTEDES, Santiago de Compostela, España, 2022. URL: <https://biblioteca.sistedes.es/articulo/web-development-with-tau-prolog/>.
- [RM05] Lior Rokach y Oded Maimon. «Decision trees». En «Data mining and knowledge discovery handbook», páginas 165–192. Springer, 2005.
- [RM20] José A. Riaza y Ginés Moreno. «Using SAT/SMT Solvers for Efficiently Tuning Fuzzy Logic Programs». En «2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)», página 1–8. IEEE, 2020. URL: <https://doi.org/10.1109/FUZZ48607.2020.9177798>.
- [Rob65] John A. Robinson. «A machine-oriented logic based on the resolution principle». *Journal of the ACM (JACM)*, vol. 12 n^o 1, páginas 23–41, 1965.
- [Rus10] Stuart J. Russell. «Artificial intelligence a modern approach». Pearson Education, Inc., 2010.
- [Sah93] Dan Sahlin. «Mixtus: An automatic partial evaluator for full Prolog». *New Generation Computing*, vol. 12 n^o 1, páginas 7–51, 1993.
- [Ses02] Maria I. Sessa. «Approximate reasoning by similarity-based SLD resolution». *Theoretical Computer Science*, vol. 275 n^o 1, páginas 389–426, 2002. ISSN 0304-3975.
- [SET09] Toby Segaran, Colin Evans y Jamie Taylor. «Programming the semantic web: Build flexible applications with graph data». O'Reilly Media, Inc., 2009.
- [SS83] Berthold Schweizer y Abe Sklar. «Probabilistic metric spaces». North Holland series in probability and applied mathematics. North Holland, 1983. ISBN: 9780444006660. LCCN 81022428.
- [SS94] Leon Sterling y Ehud Y. Shapiro. «The Art of Prolog: Advanced Programming Techniques». MIT press, 1994.
- [SW18] Jan A. Snyman y Daniel N. Wilke. «Practical mathematical optimization: Basic optimization theory and gradient-based algorithms», páginas 3–40. Springer International Publishing, Cham, 2018. ISBN: 9783319775869.

- [Tam84] Hisao Tamaki. «Unfold/fold transformation of logic programs». En «Proceedings of 2nd Int. Logic Programming Conf.», páginas 127–138. Uppsala, 1984.
- [TdCC00] Enric Trillas, Cristina del Campo y Susana Cubillo. «When QM-operators are implication functions and conditional fuzzy relations». *International Journal of Intelligent Systems*, vol. 15 n^o 7, páginas 647–655, 2000.
- [Tri12] Markus Triska. «The finite domain constraint solver of SWI-Prolog». En «International Symposium on Functional and Logic Programming», páginas 307–316. Springer, 2012.
- [Tri16] Markus Triska. «The boolean constraint solver of SWI-Prolog (system description)». En «International Symposium on Functional and Logic Programming», páginas 45–61. Springer, 2016.
- [Tri18] Markus Triska. «Boolean constraints in SWI-Prolog: A comprehensive system description». *Science of Computer Programming*, vol. 164, páginas 98–115, 2018. ISSN 0167-6423. Special issue of selected papers from FLOPS 2016.
- [Tur92] Burhan I. Turksen. «Interval-valued fuzzy sets and ‘compensatory AND’». *Fuzzy Sets and Systems*, vol. 51 n^o 3, páginas 295–307, 1992. ISSN 0165-0114.
- [V⁺20] Robert J. Vanderbei et al. «Linear programming». Springer, 2020.
- [VBG12] Amanda Vidal, Félix Bou y Lluís Godo. «An SMT-based solver for continuous t-norm based logics». En «International Conference on Scalable Uncertainty Management», páginas 633–640. Springer, 2012.
- [War62] Stephen Warshall. «A Theorem on Boolean Matrices». *J. ACM*, vol. 9 n^o 1, página 11–12, jan de 1962. ISSN 0004-5411.
- [Wol20] Laurence A. Wolsey. «Integer programming». John Wiley & Sons, 2020.
- [WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska y Torbjörn Lager. «SWI-Prolog». *Theory and Practice of Logic Programming*, vol. 12 n^o 1-2, páginas 67–96, 2012.
- [XX05] Jin X. Xie y Yi Xue. «Optimization modeling and LINDO/LINGO software». *Beijing: Tsinghua University Press.*, vol. 18 n^o 4, páginas 67–73, 2005.
- [Yag94] Ronald R. Yager. «Aggregation operators and fuzzy systems modeling». *Fuzzy Sets and Systems*, vol. 67 n^o 2, páginas 129–145, 1994. ISSN 0165-0114.

- [Yin94] Mingsheng Ying. «A logic for approximate reasoning». *Journal of Symbolic Logic*, vol. 59 n^o 3, página 830–837, 1994.
- [Zad65] Lotfi A. Zadeh. «Fuzzy sets». *Information and Control*, vol. 8 n^o 3, páginas 338–353, 1965. ISSN 0019-9958.
- [Zad71] Lotfi A. Zadeh. «Similarity relations and fuzzy orderings». *Information Sciences*, vol. 3 n^o 2, páginas 177–200, 1971. ISSN 0020-0255.
- [Zad75] Lotfi A. Zadeh. «Fuzzy logic and approximate reasoning». *Synthese*, vol. 30 n^o 3, páginas 407–428, 1975.