



UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

**TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN**

TRABAJO FIN DE GRADO

Implementación de técnicas de despliegado difuso
sobre el entorno FLOPER

José Antonio Riaza Valverde

Junio de 2017





UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

**TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN**

TRABAJO FIN DE GRADO

Implementación de técnicas de despliegado difuso
sobre el entorno FLOPER

Autor: José Antonio Riaza Valverde

Directores: Ginés Damián Moreno Valverde
Jaime Penabad Vázquez

Junio de 2017

Declaración de Autoría

Yo, José Antonio Riaza Valverde con DNI 77578756D, declaro que soy el único autor del Trabajo Fin de Grado titulado *Implementación de técnicas de desplegado difuso sobre el entorno FLOPER*, y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en el mismo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 2 de Abril de 2017

Fdo.: José Antonio Riaza Valverde

Resumen

El objetivo fundamental de este Trabajo Fin de Grado es la introducción, en el entorno FLOPER, de un conjunto de técnicas de transformación y calibrado de programas lógicos difusos, que han sido desarrolladas en los últimos años por el grupo de investigación DEC-TAU en la Universidad de Castilla-La Mancha. En primer lugar, se implementan técnicas de despliegado difuso (que son clásicas en paradigmas declarativos precedentes como el lógico, funcional y el lógico-funcional), para generar programas más eficientes al aplicar pasos de computación sobre las reglas del programa original. En segundo lugar, se desarrollan técnicas de calibrado de programas difusos simbólicos, para elegir los pesos y conectivos adecuados en las reglas de un programa simbólico. Para ello, se introduce una extensión simbólica de los lenguajes MALP y FASILL que permite utilizar valores y conectivos desconocidos que no pertenecen al retículo asociado al programa.

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 presente trabajo da un paso más en el desarrollo del entorno FLOPER, centrándonos en la implementación de las técnicas de transformación y calibrado mencionadas. Asimismo, se ha desarrollado una nueva plataforma online donde poder desplegar y calibrar programas lógicos difusos.

Queremos destacar que la descripción de estas técnicas de transformación y calibrado automático de programas lógicos difusos, incluyendo los detalles de su implementación abordados en este trabajo, han originado recientemente cuatro publicaciones científicas presentadas en congresos internacionales (LOPSTR'16, RuleML+RR'17, SUM'17 y ESCIM'17).

Agradecimientos

Deseo agradecer la colaboración y el apoyo de las personas que han hecho posible este Trabajo Fin de Grado. Especialmente, a los directores del trabajo:

- A Ginés Moreno, por ofrecerme la inestimable oportunidad de incorporarme a su grupo de investigación, y permitirme colaborar junto a ellos. Le agradezco la amabilidad y la amistad que me ha demostrado en este breve período de tiempo. Su experiencia y su inquietud profesional han sido determinantes en el desarrollo del trabajo.
- A Jaime Penabad, por su atención y disponibilidad en todo momento para revisar y corregir la memoria, así como para resolver cualquier duda. Su experiencia en este campo y la exigencia que siempre procura en la formulación de conceptos y en la presentación de trabajos han sido de gran ayuda para la redacción de la memoria.

Índice general

Índice de figuras	XI
Índice de tablas	XIII
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Estructura de la memoria	3
2. Estado del arte	5
2.1. Lógica	5
2.1.1. Lógica de proposiciones	6
2.1.2. Lógica de predicados	7
2.2. Programación declarativa	9
2.3. Programación lógica	11
2.3.1. Sustitución	11
2.3.2. Unificación	12
2.3.3. Principio de resolución de la lógica de predicados	13
2.3.4. Resolución SLD	14
2.4. Prolog	15
2.4.1. Sintaxis	15
2.4.2. Mecanismo operacional	16
2.4.3. Predicados extralógicos	17
2.4.4. Gramáticas de cláusulas definidas	21
2.5. Lógica difusa	22
2.5.1. Conjuntos difusos	22
2.5.2. t-normas, t-conormas y agregadores	23
2.5.3. Programación lógica difusa	25
2.6. Lenguajes lógicos difusos	26
2.6.1. MALP	26
2.6.2. FASILL	33

2.6.3. El entorno FLOPER	38
3. Despliegado de programas lógicos difusos	43
3.1. Despliegado de programas FASILL	43
3.2. Corrección del desplegado	48
3.3. Implementación del desplegado	49
3.3.1. Reglas	49
3.3.2. Transformaciones	51
3.3.3. Desplegado	52
3.3.4. Herramienta online para el desplegado	54
4. Análisis y ejecución de programas simbólicos	57
4.1. Programación lógica difusa simbólica	57
4.1.1. sMALP	57
4.1.2. sFASILL	60
4.2. Despliegado de programas simbólicos	61
4.3. Implementación de sMALP y sFASILL	63
4.3.1. Análisis léxico y sintáctico	63
4.3.2. Ejecución parcial	68
4.3.3. Herramienta online para programas simbólicos	69
5. Calibrado de programas simbólicos	71
5.1. Sustituciones simbólicas	71
5.2. Distancia entre elementos del retículo	72
5.3. Algoritmos de calibrado	73
5.3.1. Calibrado básico	75
5.3.2. Calibrado simbólico	76
5.3.3. Calibrado umbralizado	78
5.4. Resultados experimentales	79
5.4.1. Sustituciones simbólicas	80
5.4.2. Casos de prueba	80
5.5. Implementación del calibrado	82
5.5.1. Análisis de los casos de prueba	82
5.5.2. Generación de sustituciones simbólicas	83
5.5.3. Algoritmos de calibrado	84
5.5.4. Herramienta online para el calibrado	88
6. Conclusiones y trabajo futuro	93
6.1. Conclusiones	93
6.2. Trabajo futuro	94

Bibliografía	97
Apéndices	101
A. Casos de prueba del conjunto de datos Iris	101
B. Predicados de desplegado en FLOPER	105
C. Predicados de calibrado en FLOPER	109

Índice de figuras

2.1. Pares adjuntos de las lógicas de Gödel, Łukasiewicz y producto sobre $([0, 1], \leq)$	28
2.2. Introducción de programas en el entorno FLOPER online	41
2.3. Respuestas computadas difusas en el entorno FLOPER online	41
2.4. Nuevo entorno gráfico de FLOPER en SWI-Prolog	42
3.1. Árbol de derivación del programa $\mathcal{P}_0$ del Ejemplo 3.2	47
3.2. Árbol de derivación del programa $\mathcal{P}_{10}$ del Ejemplo 3.2	47
3.3. Menú de desplegado del entorno FLOPER online	54
3.4. Selección de la regla a desplegar en el entorno FLOPER online	55
3.5. Programa desplegado en el entorno FLOPER online	55
4.1. Árbol de derivación del programa $\mathcal{P}_0$ del Ejemplo 4.4	64
4.2. Árbol de derivación del programa $\mathcal{P}_7$ del Ejemplo 4.4	65
4.3. Programa simbólico en el entorno FLOPER online	69
4.4. Desplegado simbólico en el entorno FLOPER online	69
4.5. Programa simbólico desplegado en el entorno FLOPER online	69
4.6. Ejecución simbólica en el entorno FLOPER online	70
5.1. Pestaña de ejecución en el entorno FLOPER online	89
5.2. Pestaña de calibrado en el entorno FLOPER online	89
5.3. Casos de prueba en el entorno FLOPER online	90
5.4. Proceso de calibrado en el entorno FLOPER online	90
5.5. Aplicación de una sustitución simbólica en el entorno FLOPER online .	91
5.6. Casos de prueba erróneos en el entorno FLOPER online	91

Índice de tablas

2.1. Sintaxis de la programación lógica y sintaxis Prolog	16
2.2. Grados de similaridad de la relación del Ejemplo 2.14	34
4.1. Elementos simbólicos de sMALP	64
5.1. Resumen de pesos al calibrar el programa del Ejemplo 5.3	76
5.2. Resumen de pesos al calibrar el programa del Ejemplo 5.3 con poda . .	79
5.3. Tiempo de ejecución del calibrado (en milisegundos)	80

Capítulo 1

Introducción

1.1. Motivación

El entorno de programación lógica difusa FLOPER ha sido desarrollado en la Universidad de Castilla-La Mancha a partir de otros seis proyectos fin de carrera o trabajos fin de grado que preceden a este. En su estado actual, la herramienta es capaz de ejecutar y depurar programas difusos (mediante un pre-proceso de los mismos que los traduce a código Prolog estándar) sobre cualquier plataforma Prolog, y está preparada para incorporar diversas técnicas para la transformación, optimización, especialización, entre otros, diseñadas en el grupo de investigación DEC-TAU en los últimos años. Además, el mismo incorpora un entorno gráfico escrito en Java que permite interactuar con el sistema de forma ágil y flexible sobre plataformas estandarizadas (como Windows, Linux e incluso Android), y también es posible ejecutar su última versión online a través de <http://dectau.uclm.es/floper/?q=sim/test>.

FASILL (*Fuzzy Aggregators and Similarity Into a Logic Language*) es un lenguaje de programación lógica difusa con anotaciones de grados de verdad implícitas/explicitas, una gran variedad de conectivos y unificación por similaridad. FASILL integra y extiende las características de MALP (*Multi-Adjoint Logic Programming*, un lenguaje de programación lógica difusa con reglas explícitamente anotadas) y Bousi~Prolog (que utiliza un algoritmo de unificación débil y está bien adaptado para la respuesta de consultas flexibles).

En este Trabajo Fin de Grado se pretende incorporar al entorno FLOPER dos nuevos mecanismos para la transformación automática de programas lógicos difusos, tanto para programas con sintaxis MALP como FASILL. En primer lugar, técnicas de desplegado de reglas, con el objetivo de generar programas más eficientes al aplicar pasos de computación sobre las reglas del programa original. En segundo lugar, mecanismos

de calibrado y ejecución simbólica de programas difusos, para facilitar la elección de los conectivos y pesos adecuados de las reglas del programa.

El desplegado es una técnica de transformación automática de programas ampliamente conocida y utilizada, que preserva la semántica de las reglas del programa original. Se basa en la aplicación de pasos de computación sobre el cuerpo de las reglas del programa. La transformación de desplegado permite optimizar los programas, generando código más eficiente, y es esencial para desarrollar sofisticadas y poderosas herramientas de programación, tales como sistemas de transformación basados en plegado/desplegado o evaluadores parciales.

Una de las tareas más difíciles a la hora de especificar las reglas de un programa lógico difuso es determinar los pesos adecuados de cada regla, así como los conectivos y operadores difusos más convenientes. Para superar estas dificultades, se introducen unas extensiones simbólicas de los programas MALP y FASILL, llamadas sMALP (*symbolic Multi-Adjoint Logic Programming*) y sFASILL (*symbolic Fuzzy Aggregators and Similarity Into a Logic Language*), con el objetivo de poder escribir reglas con grados de verdad y conectivos simbólicos, es decir, grados de verdad y conectivos que no están definidos en el retículo asociado al programa. Para evaluar estos programas, se hace necesario definir una nueva semántica operacional que retrase la evaluación de las expresiones simbólicas. Por lo tanto, una respuesta simbólica puede incluir grados de verdad y conectivos simbólicos (desconocidos).

1.2. Objetivos

En este Trabajo Fin de Grado nos centraremos en extender las funcionalidades del entorno de programación lógica difusa FLOPER, equipándolo con poderosas herramientas y mecanismos de transformación automática y calibrado de programas. Este trabajo se centra en tres objetivos fundamentales.

- En primer lugar, adaptaremos las técnicas de desplegado difuso vistas en [9, 10, 27] para programas MALP, extendiéndolas para el lenguaje FASILL, e implementaremos estas transformaciones sobre ambos lenguajes lógicos difusos en el entorno FLOPER. Este objetivo introduce la necesidad de incorporar al entorno un mecanismo para gestionar simultáneamente diferentes versiones de un programa, que serán generadas tras realizar una transformación de desplegado.
- En segundo lugar, introduciremos unas extensiones simbólicas de los lenguajes MALP y FASILL, que permiten utilizar constantes y conectivos simbólicos en los programas lógicos difusos, es decir, valores y conectivos que no pertenecen

al retículo asociado al programa. Para ello, describimos la nueva sintaxis y semántica operacional de estos lenguajes, así como las modificaciones realizadas en el proceso de análisis léxico y sintáctico en el entorno FLOPER para trabajar con estas extensiones simbólicas, permitiendo ejecutar parcialmente objetivos sin necesidad de reemplazar las constantes y conectivos simbólicos por elementos del retículo.

- Por último, haciendo uso de las nuevas extensiones simbólicas, describiremos e incorporaremos al entorno FLOPER una serie de métodos para el calibrado automático de programas simbólicos, esto es, técnicas de búsqueda que nos proporcionen la mejor sustitución simbólica frente a unos casos de prueba esperados. Como consecuencia, se introduce otra fase de análisis léxico y sintáctico de los casos de prueba. En el entorno FLOPER se introduce, además, la medición y comparación del tiempo de ejecución necesario para calibrar un programa simbólico con los diversos métodos de calibrado incorporados.

1.3. Estructura de la memoria

Tras un primer capítulo donde introducimos la motivación y los objetivos fundamentales del Trabajo Fin de Grado, la estructura de esta memoria es la que sigue.

- **Capítulo 2. Estado del arte.**

En el Capítulo 2 se revisan los formalismos que han sido necesarios para la realización del trabajo, sobre lógica, lógica difusa y programación lógica, así como la sintaxis y la semántica operacional de los lenguajes involucrados: Prolog, MALP y FASILL. Además, se muestran las últimas herramientas implementadas para el entorno FLOPER: un nuevo entorno de desarrollo implementado en SWI-Prolog con su paquete gráfico XPCE, y una nueva versión online en la que se incorporarán las nuevas técnicas expuestas en este trabajo.

- **Capítulo 3. Desplegado de programas lógicos difusos.**

En el Capítulo 3 se introduce la idea de desplegado difuso, y se formaliza su definición para el lenguaje FASILL, adaptándola de las técnicas de desplegado estudiadas para otros lenguajes lógicos. Se contemplan también los problemas asociados a la corrección y la completitud de la semántica de los programas lógicos difusos tras ser desplegados, que, como veremos, no siempre es posible conservar. Además, se muestra su implementación en el entorno FLOPER y su integración en la versión online.

- **Capítulo 4. Análisis y ejecución de programas simbólicos.**

En el Capítulo 4 se presentan las extensiones simbólicas de MALP y FASILL

que permitirán la introducción de grados de verdad y conectivos simbólicos en ambos lenguajes. También, aprovechando la nueva semántica operacional de los lenguajes simbólicos, se muestra que es posible aplicar las técnicas de desplegado estudiadas en el capítulo anterior, a los lenguajes simbólicos, sin realizar cambio alguno en el mecanismo del desplegado.

■ **Capítulo 5. Calibrado de programas simbólicos.**

En el capítulo 5 se contemplan una serie de métodos para el calibrado automático de programas simbólicos, esto es, métodos para buscar la mejor sustitución simbólica frente a un conjunto de casos de prueba; midiendo y comparando el rendimiento de estos métodos de búsqueda. Se muestra además su implementación sobre el entorno FLOPER y su integración en la versión online.

■ **Capítulo 6. Conclusiones y trabajo futuro.**

Finalmente, en el Capítulo 6 se recogen las conclusiones del trabajo, destacando las técnicas de desplegado y calibrado de programas lógicos difusos implementadas sobre el entorno FLOPER. En dicho capítulo se detallan, además, algunas líneas de trabajo futuro que pueden seguirse para ampliar la potencia de estas técnicas.

Capítulo 2

Estado del arte

En el desarrollo de este capítulo se hace mención de los conceptos elementales de lógica, programación lógica y Prolog, reproduciéndolos conforme se contemplan en [7] y [8]. Asimismo, la introducción o motivación de dichos conceptos se toman de las referencias citadas.

2.1. Lógica

La lógica es una disciplina que tiene sus orígenes en los trabajos de Aristóteles (siglo VI a. n. e.) que abordan, con el estudio de los silogismos, un conjunto de reglas para razonar y que se ha constituido en disciplina formal, es decir, en lógica simbólica, a partir de la segunda mitad del siglo XIX, con las aportaciones de A. De Morgan y G. Boole.

Desde el comienzo de su relación con la informática, la lógica ha jugado diversos papeles en el campo de la informática. Como una fuente de lenguajes y sistemas para el razonamiento, debido a su capacidad deductiva, se han empleado diferentes tipos de lógica para describir e implementar sistemas que razonan sobre un dominio en particular.

Desde el punto de vista de la fundamentación, la lógica se ha utilizado para proporcionar un modelo de cómputo. El λ -cálculo y la reducción de λ -expresiones a formas normales, o bien la lógica de *cláusulas de Horn* y el principio de resolución SLD representan visiones idealizadas de la idea de cómputo. La lógica también se ha empleado para establecer una descripción formal del significado (semántica) de los lenguajes de programación y en la especificación y verificación formal de programas.

Los diferentes sistemas lógicos elementales tienen en común, en su presentación, una etapa previa de formalización simbólica que suele hacerse a dos niveles [7]:

- *Lógica de proposiciones*: Se ocupa de enunciados declarativos simples o proposi-

ciones que se contemplan como un todo indivisible y que pueden ser combinados mediante *conectivas* (como ‘no’, ‘y’, ‘o’, ‘si ... entonces ...’).

- *Lógica de predicados*: En ella las proposiciones ya no son elementos indivisibles, porque se aborda un análisis interno de estos para observar qué afirman y los objetos de quienes se realiza la afirmación correspondiente.

2.1.1. Lógica de proposiciones

Las *proposiciones* son los componentes del discurso que tienen un significado completo (es decir, con verdad conocida), y pueden formar enunciados compuestos por medio de *conectivas*. La parte de la lógica encargada del estudio de las propiedades de estos enlaces es la *lógica de proposiciones* o *lógica de conectivas*.

La lógica de proposiciones no analiza la estructura de los enunciados: las relaciones entre sujeto y predicado verbal, entre otras. Por lo tanto, la lógica de proposiciones es la parte más simple de la lógica simbólica y se ocupa de los enunciados, como un todo, y las posibles combinaciones con otros enunciados.

Para formalizar un enunciado compuesto, podemos acudir al contexto de *fórmula bien formada*, que se construye haciendo uso de:

- Una cantidad finita de variables proposicionales: $p, q, r, \dots$
- Un conjunto de símbolos para las conectivas: $\neg, \wedge, \vee, \Rightarrow$ y $\Leftrightarrow$.
- Otros símbolos impropios.

Utilizando las conectivas se pueden construir enunciados compuestos a partir de enunciados más simples. También usando variables proposicionales y símbolos de conectiva podremos construir fórmulas bien formadas de cualquier longitud. Delimitamos el concepto de fórmula bien formada mediante la siguiente definición.

Definición 2.1. (Fórmula bien formada, [7]). *Una fórmula bien formada es una expresión, en la que intervienen variables proposicionales y símbolos de conectiva, conforme a las siguientes reglas:*

1. Toda variable proposicional $p, q, r, \dots$ es una fórmula bien formada.
2. Si $\mathcal{A}$ y $\mathcal{B}$ son fórmulas bien formadas, también lo son: $\neg\mathcal{A}$, $\neg\mathcal{B}$, $\mathcal{A} \wedge \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \Rightarrow \mathcal{B}$ y $\mathcal{A} \Leftrightarrow \mathcal{B}$.

Ejemplo 2.1. El razonamiento formalmente válido (es decir, lógicamente verdadero) que sigue

$$\begin{array}{ccc} \text{Todos los hombres son mortales} & & \\ \wedge & \Rightarrow & \text{Sócrates es mortal} \\ \text{Sócrates es hombre} & & \end{array}$$

reproduce un ejemplo clásico de *silogismo*, debido a Aristóteles, que se expresa en la lógica de proposiciones por la fórmula $p \wedge q \Rightarrow r$, donde p y q son las *premisas* (hipótesis) del silogismo y r es la *conclusión*. Ahora bien, en la lógica referida no puede justificarse la tesis r : no existen propiedades de las conectivas $\wedge, \Rightarrow$ que garanticen dicha tesis.

El ejemplo considerado revela las limitaciones de la lógica de conectivas, que usa, exclusivamente, propiedades de los enlaces sin ocuparse del análisis interno de las proposiciones. Para concluir que *Sócrates es mortal* nos vemos obligados a intervenir en lo que se dice (mediante el predicado verbal) en la proposición (*ser hombre, ser mortal*) y los objetos de quienes se realiza la afirmación.

La lógica de predicados, que abordamos 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 al lenguaje lógico y mayor capacidad deductiva.

2.1.2. Lógica de predicados

La *lógica de predicados* (que para nosotros será sinónimo de *lógica 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.

El alfabeto del lenguaje $\mathcal{L}$ de primer orden está constituido por los símbolos:

- Constantes: $a, b, c, \dots$ El conjunto infinito numerable de símbolos de constante lo denotaremos por $\mathcal{C}$.
- Símbolos de variable: $x, y, z, \dots$ El conjunto infinito numerable de símbolos de variable lo denotaremos por $\mathcal{V}$.
- Símbolos de función: $f, g, h, \dots$ El conjunto infinito numerable de símbolos de función lo denotaremos por $\mathcal{F}$.

- Símbolos de predicado: $P, Q, R, \dots$
- Conectivas y cuantificadores: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall$ y $\exists$.
- Otros símbolos impropios.

Con los símbolos del alfabeto anterior podemos formar cadenas de símbolos, de las cuales solo nos interesan aquellas que constituyen términos y fórmulas bien formadas. Nos referiremos a estas con el nombre de *expresiones* de $\mathcal{L}$.

Definición 2.2. (Término de $\mathcal{L}$, [7]). *Un término de $\mathcal{L}$ es una expresión, en la que intervienen símbolos de constante y símbolos de función, que se define de acuerdo a las siguientes reglas:*

1. Si $t \in \mathcal{V} \cup \mathcal{C}$ entonces t es un término. Esto es, toda variable o constante de $\mathcal{L}$ es un término de $\mathcal{L}$.
2. Si $t_1, t_2, \dots, t_n$ son términos de $\mathcal{L}$ y f es un símbolo de función n -ario de $\mathcal{L}$, entonces $f(t_1, t_2, \dots, t_n)$ es un término de $\mathcal{L}$.

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

Definición 2.3. (Fórmula atómica). *Una fórmula atómica de $\mathcal{L}$ es una expresión $P(t_1, t_2, \dots, t_n)$, en la que $t_1, t_2, \dots, t_n$ son términos de $\mathcal{L}$, y P es un símbolo de predicado n -ario de $\mathcal{L}$.*

Además, por ‘atómica’ entendemos indivisible. Las fórmulas atómicas son las partes más simples constitutivas de las fórmulas bien formadas. Las fórmulas atómicas serán las expresiones del lenguaje $\mathcal{L}$ que se interpretarán como enunciados, por ejemplo, significando que un cierto objeto verifica una determinada propiedad.

Definición 2.4. (Fórmula bien formada, [7]). *Una fórmula bien formada de $\mathcal{L}$ es una expresión, en la que intervienen fórmulas atómicas, conectivas o cuantificadores, que cumplen las siguientes reglas:*

1. Toda fórmula atómica de $\mathcal{L}$ es una fórmula bien formada.
2. Si $\mathcal{A}$ y $\mathcal{B}$ son fórmulas bien formadas de $\mathcal{L}$, también lo son: $\neg\mathcal{A}$, $\neg\mathcal{B}$, $\mathcal{A} \wedge \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \Rightarrow \mathcal{B}$ y $\mathcal{A} \Leftrightarrow \mathcal{B}$.
3. Si $\mathcal{A}$ es una fórmula bien formada de $\mathcal{L}$ que incorpora la variable $x \in \mathcal{V}$ entonces $\forall x, \mathcal{A}$ y $\exists x, \mathcal{A}$ son fórmulas bien formadas.

Semántica

La *semántica* estudia la adscripción de significado a los lenguajes de los sistemas formales. La semántica de un sistema lógico, suele abordarse desde la llamada *teoría de modelos*. En la teoría de modelos el significado se formaliza mediante la noción de *modelo*, que consiste en una entidad matemática, junto con las propiedades y relaciones que se dan entre sus elementos.

Para poder discutir sobre la verdad o falsedad de una fórmula es necesario asignar significado a los símbolos que la componen. *Interpretar* un formalismo consiste en seleccionar un modelo, esto es [7]:

1. Elegir un *dominio* o *universo del discurso*; es decir, un conjunto no vacío de individuos al que se referirán las variables.
2. Asignar significados a los símbolos peculiares del formalismo: asignar a cada constante un individuo, a cada símbolo de función una función en el dominio, y a cada símbolo de predicado una relación en el dominio.

Definición 2.5. (Interpretación, [7]). Una interpretación $\mathcal{I}$ de $\mathcal{L}$ es un par $(\mathcal{D}_{\mathcal{I}}, \mathcal{J})$ que consiste en:

1. Un conjunto no vacío $\mathcal{D}_{\mathcal{I}}$, el dominio de $\mathcal{I}$.
2. Una aplicación $\mathcal{J}$ que asigna:
 - a) A cada símbolo de constante de $\mathcal{L}$, a , un elemento distinguido de $\mathcal{D}_{\mathcal{I}}$; $\mathcal{J}(a) = \bar{a}$.
 - b) A cada símbolo de función f de $\mathcal{L}$ una función $\mathcal{J}(f) = \bar{f}$, tal que $\bar{f} : \mathcal{D}_{\mathcal{I}}^n \rightarrow \mathcal{D}_{\mathcal{I}}$.
 - c) A cada símbolo de predicado P de $\mathcal{L}$ una relación $\mathcal{J}(P) = R$, tal que $R \subset \mathcal{D}_{\mathcal{I}}^n$.

2.2. Programación declarativa

La *programación declarativa* es un paradigma de programación en el que el programador especifica *qué* debe computarse en lugar de *cómo* deben realizarse los cálculos. En este paradigma de programación, de acuerdo con el famoso aserto de Kowalski [15, 16, 17], un *programa* = *lógica* + *control*, y la tarea de programar consiste en centrar la atención en la *lógica* dejando de lado el *control* al sistema. El componente lógico

determina el significado del programa mientras que el componente de control solamente afecta a su eficiencia.

La característica fundamental de la programación declarativa es el uso de la lógica como lenguaje de programación, lo cual puede conceptualizarse como sigue (véase en [8]):

- Un *programa* es una teoría formal en una cierta lógica, esto es, un conjunto de fórmulas lógicas que resultan ser la especificación del problema que se pretende resolver.
- La *computación* se entiende como una forma de inferencia o deducción en dicha lógica.

Los principales requisitos que debe cumplir la lógica empleada son:

1. Disponer de un lenguaje que sea suficientemente expresivo para cubrir un campo de aplicación interesante.
2. Disponer de una semántica operacional, esto es, un mecanismo de cómputo que permita ejecutar los programas.
3. Disponer de una semántica declarativa que permita dar un significado a los programas de forma independiente a su posible ejecución.
4. Resultados de corrección y completitud que aseguren que lo que se computa coincide con aquello que es considerado como verdadero.

El tercero de los requisitos es tal vez el más importante desde el punto de vista de la declaratividad, ya que es el que permite especificar qué estamos computando. Concretamente, la semántica declarativa especifica el significado de los objetos sintácticos del lenguaje por medio de su traducción en elementos y estructuras de un dominio conocido.

La programación declarativa incluye como paradigmas más representativos la programación lógica, basada en lógica clausal, y la programación funcional, basada en lógica ecuacional¹. En esta memoria nos centraremos únicamente en el paradigma lógico, cuyas características se explican en la siguiente sección.

¹Estas dos clases de lógica son restricciones de la lógica de primer orden.

2.3. Programación lógica

La *programación lógica* se basa en fragmentos de la lógica de predicados, siendo el más popular la lógica de *cláusulas de Horn* (*HCL*), que pueden emplearse como base para un lenguaje de programación al poseer una semántica operacional susceptible de una implementación eficiente: la *resolución SLD*.

Definición 2.6. (Cláusula de Horn con cabeza). *Una cláusula de Horn con cabeza (o cláusula definida) es una disyunción de literales todos negados, salvo uno que se denomina cabeza. Así pues, una cláusula de Horn con cabeza es una cláusula de la forma $A \leftarrow B_1 \wedge \dots \wedge B_n$, $n \geq 0$, donde A , B_i , $i = 1, \dots, n$, representan átomos.*

Definición 2.7. (Programa lógico definido, [8]). *Un programa definido es un conjunto de cláusulas definidas Π . La definición de un símbolo de predicado² p que aparece en Π es el subconjunto de cláusulas de Π cuyas cabezas tienen como símbolo de predicado el predicado p .*

Como semántica declarativa se utiliza un método de prueba por refutación que emplea el algoritmo de *unificación* como mecanismo de base y permite la extracción de respuestas (esto es, el enlace de un valor a una variable lógica). La resolución SLD es un método de prueba correcto y completo para la lógica de cláusulas de Horn.

2.3.1. Sustitución

Para formalizar adecuadamente los conceptos de *unificación*, *resolución* y *respuesta computada*, es imprescindible introducir el concepto de sustitución.

Definición 2.8. (Sustitución, [8]). *Una sustitución σ es una aplicación que asigna, a cada variable X del conjunto de variables $\mathcal{V}$ de un lenguaje de primer orden $\mathcal{L}$, un término $\sigma(X)$ del conjunto de los términos $\mathcal{T}$ de $\mathcal{L}$.*

$$\begin{aligned} \sigma : \mathcal{V} &\rightarrow \mathcal{T} \\ X &\rightarrow \sigma(X) \end{aligned}$$

Es habitual representar las sustituciones como conjuntos finitos de la forma

$$\{X_1/t_1, X_2/t_2, \dots, X_n/t_n\}$$

²A diferencia de la lógica de predicados, en la programación lógica los predicados se designan por letras minúsculas p , q , r , ...

donde, para cada i , t_i es un término diferente de X_i . Los elementos X_i/t_i de la sustitución reciben el nombre de *enlaces*. El conjunto $\{X_1, X_2, \dots, X_n\}$ se denomina *dominio* de la sustitución y el conjunto $\{t_1, t_2, \dots, t_n\}$ se denomina *rango* de la sustitución.

Esta forma de representar una sustitución puede considerarse como una definición por extensión, en la que a cada variable X_i se le asocia su término t_i (mientras el resto de variables que no están en el conjunto no se modifican). La *sustitución identidad*, id , se representa mediante el conjunto vacío de elementos: $\{\}$.

Definición 2.9. (Instancia, [8]). La aplicación de una sustitución $\sigma = \{X_1/t_1, X_2/t_2, \dots, X_n/t_n\}$ a una expresión E , denotado $\sigma(E)$ o $E\sigma$, se obtiene reemplazando simultáneamente cada ocurrencia de X_i en la expresión E por el correspondiente término t_i . Se dice que $\sigma(E)$ es una instancia de E .

La relación “ser instancia de” introduce un orden de máxima generalidad entre las expresiones del lenguaje de primer orden $\mathcal{L}$. Sean E_1 y E_2 expresiones de $\mathcal{L}$, E_1 es más general que E_2 , denotado $E_1 \leq E_2$, si, y solo si, existe una sustitución σ tal que $\sigma(E_1) = E_2$.

Dadas las sustituciones σ y θ , la composición de σ y θ es la aplicación $\sigma \circ \theta$ ³ tal que $(\sigma \circ \theta)(E) = \sigma(\theta(E))$.

Ejemplo 2.2. Sea la expresión $E \equiv p(X, f(Y), Z)$ y la sustitución $\sigma = \{Y/b, X/f(a)\}$. La aplicación de σ a E es $\sigma(E) = p(f(a), f(b), Z)$. $p(f(a), f(b), Z)$ es una instancia de $p(X, f(Y), Z)$. Además, $p(X, f(Y), Z)$ es más general que $p(f(a), f(b), Z)$. Dada la sustitución $\theta = \{Z/g(X)\}$, la composición de σ y θ es la sustitución $\sigma \circ \theta = \{Z/g(f(a)), Y/b, X/f(a)\}$, y la aplicación de $\sigma \circ \theta$ a la expresión E es $(\sigma \circ \theta)(E) = \sigma(\theta(E)) = p(f(a), f(b), g(f(a)))$.

2.3.2. Unificación

La unificación de expresiones es un concepto fundamental para la programación lógica. Informalmente, *unificar* es el proceso por el cual dos o más expresiones se convierten en sintácticamente iguales mediante la aplicación de una sustitución, denominada unificadora. Este tipo de sustituciones sintetiza la noción del cómputo en el contexto de la programación lógica.

³En ocasiones escribiremos $\sigma\theta$.

Definición 2.10. (Unificador, [8]). Una sustitución θ es un unificador del conjunto de expresiones $\{E_1, E_2, \dots, E_k\}$ si, y solo si, $\theta(E_1) = \theta(E_2) = \dots = \theta(E_k)$. Se dice que el conjunto $E_1, E_2, \dots, E_k$ es unificable si existe un unificador para él.

Definición 2.11. (Unificador más general, [8]). Un unificador σ de un conjunto de expresiones $\mathcal{S}$ es un unificador más general (m.g.u.⁴) para $\mathcal{S}$ si, y solo si, cualquier unificador θ de $\mathcal{S}$ cumple que $\sigma \leq \theta$.

Ejemplo 2.3. Sea $\mathcal{S}$ el conjunto formado por las expresiones $E_1 \equiv p(X, g(b))$ y $E_2 \equiv p(f(Z), Y)$. La sustitución $\sigma = \{Z/a, Y/g(b), X/f(a)\}$ es un unificador de las expresiones E_1 y E_2 , ya que $\sigma(E_1) = \sigma(E_2) = p(f(a), g(b))$. No obstante, existe una sustitución más general que σ , $\theta = \{Y/g(b), X/f(Z)\}$ que también es un unificador de las expresiones E_1 y E_2 , $\theta(E_1) = \theta(E_2) = p(f(Z), g(b))$. Dado que θ es más general que cualquier otro unificador de las expresiones E_1 y E_2 , el unificador θ es un unificador más general para el conjunto $\mathcal{S}$.

2.3.3. Principio de resolución de la lógica de predicados

Una vez definidos los conceptos de unificador y unificador más general, podemos formular el principio de resolución para la lógica de predicados. Definiremos primero el concepto de *factor de una cláusula*, que nos permite obtener una versión simplificada de la misma.

Si dos o más literales⁵ (con el mismo signo) de una cláusula $\mathcal{C}$ tienen un unificador más general σ , entonces se dice que $\sigma(\mathcal{C})$ es un factor de $\mathcal{C}$.

Definición 2.12. (Resolvente binario, [8]). Sean $\mathcal{C}_1$ y $\mathcal{C}_2$ dos cláusulas sin variables en común. Sean L_1 y L_2 dos literales de $\mathcal{C}_1$ y $\mathcal{C}_2$, respectivamente. Si L_1 y $\neg L_2$ tienen un m.g.u. σ , entonces la cláusula $(\sigma(\mathcal{C}_1) \setminus \sigma(L_1)) \cup (\sigma(\mathcal{C}_2) \setminus \sigma(L_2))$ se conoce como resolvente binario de $\mathcal{C}_1$ y $\mathcal{C}_2$.

Definición 2.13. (Resolvente, [8]). Un resolvente de las cláusulas $\mathcal{C}_1$ y $\mathcal{C}_2$ es uno de los siguientes resolventes binarios:

⁴En adelante, denotaremos por $mgu(x, y)$ a la función que devuelve el unificador más general de las expresiones x e y .

⁵Un literal es una fórmula atómica o su negación.

1. Un resolvente binario de $\mathcal{C}_1$ y $\mathcal{C}_2$.
2. Un resolvente binario de $\mathcal{C}_1$ y un factor de $\mathcal{C}_2$.
3. Un resolvente binario de un factor de $\mathcal{C}_1$ y $\mathcal{C}_2$.
4. Un resolvente binario de un factor de $\mathcal{C}_1$ y un factor de $\mathcal{C}_2$.

Ejemplo 2.4. Sea $\mathcal{C}_1 \equiv p(X) \vee p(f(Y)) \vee r(g(Y))$ y $\mathcal{C}_2 \equiv \neg p(f(g(a))) \vee q(b)$, un factor de $\mathcal{C}_1$ es la cláusula $\mathcal{C}'_1 \equiv p(f(Y)) \vee r(g(Y))$. Un resolvente binario de $\mathcal{C}'_1$ y $\mathcal{C}_2$ es $r(f(g(a))) \vee q(b)$. Por tanto $r(f(g(a))) \vee q(b)$ es un resolvente de $\mathcal{C}_1$ y $\mathcal{C}_2$.

A la hora de aplicar un paso de resolución, un hecho importante es la existencia de un algoritmo de unificación que permita obtener el m.g.u. de un conjunto de expresiones unificables, o de notificar su inexistencia en caso contrario. Si no existe el m.g.u. entonces no podrá darse el paso de resolución.

2.3.4. Resolución SLD

La resolución SLD es una estrategia de resolución en la que, dado un programa Π y una cláusula A tal que A' es un objetivo $\mathcal{G}$, para probar la inconsistencia de $\Pi \cup \{\mathcal{G}\}$ (equivalentemente, que A es una consecuencia lógica de Π) partimos del objetivo $\mathcal{G}$, que tomamos como cláusula inicial, y resolvemos esta con alguna cláusula del programa Π para obtener un nuevo objetivo al unificar algún literal de $\mathcal{G}$ con la cabeza negada de alguna regla de Π .

Definición 2.14. (Regla de computación, [8]). Llamamos regla de computación (o función de selección) φ a una función que, cuando se aplica a un objetivo $\mathcal{G}$, selecciona uno, y solo uno, de los átomos de $\mathcal{G}$.

Ahora podemos definir la semántica operacional en términos de un sistema de transición de estados. Con este enfoque, si identificamos un estado E como un par $\langle \mathcal{G}; \theta \rangle$ formado por una cláusula objetivo $\mathcal{G}$ y una sustitución θ , definimos la resolución SLD (utilizando la función de selección φ) como un sistema de transición cuya relación de transición $\Rightarrow_{SLD} \subseteq (E \times E)$ es la menor relación binaria que satisface:

$$\frac{(\mathcal{G} \equiv \leftarrow Q_1 \wedge A' \wedge Q_2), \varphi(\mathcal{G}) = A', \mathcal{C} \equiv (A \leftarrow Q) \ll \Pi, \sigma = mgu(A, A')}{\langle \mathcal{G}; \theta \rangle \Rightarrow_{SLD} \langle \leftarrow \sigma(Q_1 \wedge Q \wedge Q_2); \sigma\theta \rangle}$$

donde $\mathcal{Q}$, $\mathcal{Q}_1$ y $\mathcal{Q}_2$ representan conjunciones de átomos cualesquiera y el símbolo “ $\ll$ ” se ha introducido para representar que $\mathcal{C}$ es una variante de una cláusula de Π . Una derivación SLD es una secuencia $\langle \mathcal{G}_0; id \rangle \Rightarrow_{SLD} \langle \mathcal{G}_1; \theta_1 \rangle \Rightarrow_{SLD} \cdots \Rightarrow_{SLD} \langle \mathcal{G}_n; \theta_n \rangle$. Una *refutación SLD* es una derivación de éxito (es decir, que conduce a la cláusula vacía) $\langle \mathcal{G}_0; id \rangle \Rightarrow_{SLD}^* \langle \square; \sigma \rangle$, donde σ es la *respuesta computada* en la derivación.

2.4. Prolog

El lenguaje de programación *Prolog* (*PROgrammation en LOGique*) es un lenguaje enmarcado en el paradigma lógico y declarativo, de propósito general, diseñado en 1972 por Alain Colmerauer y Philippe Roussel en la Universidad de Aix-Marseille.

En esencia, un programa Prolog es un conjunto de cláusulas de Horn que se tratan como la secuencia en que las fórmulas aparecen escritas textualmente en el programa. Dado un objetivo, el programa se ejecuta utilizando el mecanismo operacional de la resolución SLD. Para usar la programación lógica como un lenguaje de programación útil, es necesario alejarse de algunos de los supuestos formales de la programación lógica, tal y como veremos en esta sección.

2.4.1. Sintaxis

La sintaxis de Prolog es sencilla. Las construcciones más simples del lenguaje son los *términos* (que se utilizan para representar los datos) y los *átomos* (que se utilizan para expresar propiedades y relaciones entre los datos). Las instrucciones se codifican como cláusulas definidas. La única peculiaridad es la notación que utiliza Prolog para designar las conectivas lógicas y el hecho de que las cláusulas deben terminar con un punto “.” [8].

En la Tabla 2.1 se muestra la correspondencia entre la notación empleada en programación lógica y en Prolog, teniendo en cuenta que en Prolog estas conectivas se implementan como predicados predefinidos (*built-in*). Por ejemplo, la disyunción $,/2$ es un predicado de aridad 2 llamado ‘,’.

Por otro lado, se relaja la rigidez con la que la lógica de predicados designa los símbolos de constante, función, predicado y los símbolos de variable, permitiéndose la relación de identificadores. Un identificador es cualquier cadena resultante de la combinación de caracteres alfanuméricos ($\mathbf{a}, \dots, \mathbf{z}$, $\mathbf{A}, \dots, \mathbf{Z}$, $0, \dots, 9$) y el subrayado “_”, con la única restricción de que los identificadores para las variables⁶ deben comenzar

⁶Cuando se utiliza únicamente el símbolo “_” como identificador de una variable, decimos que la variable es *anónima*. Las variables anónimas reciben un tratamiento especial en Prolog, ya que los

Conectiva lógica	Sintaxis PL	Sintaxis Prolog
Conjunción	$\wedge$	<code>;/2</code>
Disyunción	$\vee$	<code>,/2</code>
Implicación (regla)	$\leftarrow$	<code>:-/2</code>
Negación	$\neg$	<code>\+/1</code>
Implicación (objetivo)	$\leftarrow$	<code>?-/2</code>

Tabla 2.1: Sintaxis de la programación lógica y sintaxis Prolog

por una letra mayúscula o subrayado, y los de símbolos de constante, función y relación por una letra minúscula. Prolog permite constantes numéricas, en este caso se utiliza la notación habitual para representar números enteros y reales (con y sin signo) [8].

Ejemplo 2.5. Sea $\mathcal{P}$ el programa Prolog que sigue, formado por el predicado `append/3`, que es cierto cuando la tercera lista es la concatenación de las dos primeras.

R_1 : `append([], Xs, Xs).`

R_2 : `append([H|T], Xs, [H|Ys]) :- append(T, Xs, Ys).`

El programa está formado por dos reglas. La primera regla es un hecho, y establece que la concatenación de la lista vacía `[]` con cualquier otra lista X_S , es la propia lista X_S . La segunda regla especifica que la concatenación de una lista (con cabeza H y cola T) con otra lista X_S , es una lista con cabeza H y cola Y_S , siendo Y_S la concatenación de la cola T con la lista X_S .

2.4.2. Mecanismo operacional

El mecanismo operacional adoptado por los sistemas que implementan la versión pura⁷ del lenguaje Prolog es la estrategia de resolución SLD modificada con las siguientes peculiaridades [8]:

1. La *regla de computación* de Prolog trata los objetivos como secuencias de átomos y siempre selecciona para resolver el átomo situado más a la izquierda dentro del objetivo.

valores enlazados a estas variables mediante el proceso de unificación no se muestran como resultado de un cómputo.

⁷El Prolog puro es un subconjunto de Prolog que esencialmente se corresponde con la visión de la programación lógica introducida en la Sección 2.3. Pero Prolog aumenta su expresividad con predicados no lógicos para la meta-programación y para la manipulación de cláusulas en tiempo de ejecución, como se muestra en la Sección 2.4.3.

2. Prolog intenta unificar el átomo seleccionado por la regla de computación con las cabezas de las cláusulas del programa, considerándolas en el orden textual en el que aparecen dentro del programa. Esto supone que la *regla de ordenación* toma las cláusulas de arriba hacia abajo.
3. Prolog utiliza una *regla de búsqueda* en profundidad con vuelta atrás (*backtracking*). La principal característica de este tipo de búsqueda es que se visitan los descendientes de un nodo del árbol de búsqueda antes de visitar cualquier otro nodo.
4. Prolog omite la comprobación de ocurrencia de variables (*occur check*) cuando realiza la unificación de dos expresiones, dado que es escaso el número de ocasiones en las que se producen.
5. Prolog permite sintaxis *ambivalente*. Un mismo símbolo de función o predicado puede utilizarse con diferentes aridades en un mismo programa.

Ejemplo 2.6. Sea $\mathcal{P}$ el programa Prolog del Ejemplo 2.5. En la siguiente derivación para el programa $\mathcal{P}$ y el objetivo `append([1,2],[3,4],X)`, subrayamos la expresión seleccionada en cada paso:

$$\begin{array}{ll}
\langle \text{append}([1,2],[3,4],X); id \rangle & \Rightarrow^{R_2} \\
\langle \text{append}([2],[3,4],Ys1); \{X/[1|Ys1], Xs1/[3,4], T_1/[2], H_1/1\} \rangle & \Rightarrow^{R_2} \\
\langle \text{append}([], [3,4], Ys2); \{Ys1/[2|Ys2], Xs2/[3,4], T_2/[], H_2/2, X/[1,2|Ys2], \dots\} \rangle & \Rightarrow^{R_1} \\
\langle \square; \{Ys2/[3,4], Xs3/[3,4], Ys2/[2,3,4], Xs2/[2,3,4], T_2/[], H_2/2, X/[1,2,3,4], \dots\} \rangle &
\end{array}$$

Finalmente, tras tres derivaciones obtenemos la respuesta computada $\{X/[1,2,3,4]\}$, es decir, la concatenación de la lista `[1,2]` con la lista `[3,4]` es la lista `[1,2,3,4]`.

2.4.3. Predicados extralógicos

En Prolog, el control es fijo y viene dado de forma automática por las peculiares características e implementación de este lenguaje: i) recursividad, como única forma de propiciar la repetición; ii) unificación, como único medio de enlazar valores a las variables; y iii) vueltas atrás automática, después de producirse un fallo [8]. En esta sección introducimos algunos de los mecanismos extralógicos de control de los que dispone Prolog.

Corte

Prolog proporciona un predicado denominado *corte*, denotado como `!/0`, que permite podar el espacio de búsqueda impidiendo la vuelta atrás automática. Este predicado

siempre tiene éxito y falla al intentar resatisfacerlo. Cuando un corte aparece en una cláusula su función es impedir la reevaluación de todos los subobjetivos que quedan a su izquierda en la cláusula de la que forma parte. Esto incluye la cabeza de la cláusula, que ya no intentará resatisfacerse.

Su forma de actuar es la siguiente: cuando al retroceder por una rama del árbol de búsqueda se alcanza el corte, todas las posibles alternativas de los puntos de elección entre el *objetivo padre* (el objetivo que lanzó la cláusula que contiene el corte) y la posición en la que se ejecutó el corte son eliminadas. Por lo tanto, cuando fracasa un subobjetivo que se encuentra a la derecha del corte y, con la vuelta atrás se retrocede hasta él, su efecto es que fracasa el nodo padre.

Se identifican dos tipos distintos de cortes: i) los denominados *cortes verdes*, que no afectan al significado declarativo del programa, de modo que el programa sigue computando las mismas respuestas que computaba cuando no había cortes; y ii) los *cortes rojos*, que sí afectan al significado declarativo del programa original, de modo que el uso del corte puede hacer que se pierdan respuestas, o que se den respuestas incorrectas. En el Ejemplo 2.7 se ilustra el uso del corte.

Ejemplo 2.7. Sea $\mathcal{P}$ el programa Prolog que sigue, formado por el predicado `parity/2`, que es cierto cuando el número N es par y el átomo es `even`, o cuando el número N es impar y el átomo es `odd`.

R_1 : `parity(N, even) :- 0 is N mod 2.`

R_2 : `parity(N, odd) :- 1 is N mod 2.`

Dado que las condiciones son excluyentes (si el número es par no puede ser impar, y si no es par debe ser impar), podemos utilizar el corte en la regla R_1 y omitir la comprobación en la regla R_2 .

R_1 : `parity(N, even) :- 0 is N mod 2, !.`

R_2 : `parity(_, odd).`

Si preguntamos si un número es par o impar, se comprobará la regla R_1 . Si el número es par el módulo dará 0 y el corte parará la reevaluación. Si el número es impar el módulo no dará 0, saltando así a la regla R_2 , que siempre es cierta.

No obstante, hay que tener cuidado, ya que el corte ahora resulta fundamental para la corrección del programa. Es decir, si queremos comprobar si un determinado número par es impar, por ejemplo `parity(2, odd)`, el segundo programa nos dirá erróneamente que sí, debido a que la evaluación del objetivo no pasa por el corte de R_1 (el término `even` en la cabeza de R_1 no unifica con el término `odd` del objetivo).

Negación por fallo

Para un conjunto de cláusulas de Horn definidas es imposible extraer información negativa utilizando como única regla de deducción la estrategia de resolución SLD. Informalmente, esto ocurre porque ninguna cláusula del programa $A \leftarrow B$ tiene como cabeza A un literal negativo; por tanto, la resolución de cualquier premisa B de una cláusula solo permite inferir como conclusión el correspondiente átomo positivo en la cabeza de esta [8].

El lenguaje Prolog proporciona una forma de negación basada en el principio de negación como fallo. Para representar la negación utiliza el operador unario $\backslash+/1$. El objetivo $\backslash+ \text{ ilegal}(X)$ en la regla $\text{legal}(X) :- \backslash+ \text{ ilegal}(X).$ se evalúa como sigue: Prolog trata de probar $\text{ilegal}(X)$. Si se puede encontrar una solución para $\text{ilegal}(X)$, el objetivo original falla. De lo contrario, el objetivo original es aceptado. El hecho de que $\backslash+ \text{ ilegal}(X)$ sea aceptado no significa que $\text{ilegal}(X)$ sea falso, sino que no se ha podido demostrar que es verdadero. En el Ejemplo 2.8 se muestra un uso de la negación.

Ejemplo 2.8. Los axiomas de Peano permiten representar los números naturales utilizando únicamente el cero (representado por el término **zero**) y la función sucesor (representada por el átomo **succ/1**). Sea $\mathcal{P}$ el programa Prolog que sigue, formado por el predicado **even/1**, que es cierto cuando un número de Peano N es par.

```
 $R_1 : \text{even}(\text{zero}).$ 
 $R_2 : \text{even}(\text{succ}(N)) :- \backslash+ \text{even}(N).$ 
```

El programa está formado por dos reglas. La primera regla es un hecho, y establece que el término **zero** es un número de Peano par. La segunda regla especifica que un número de Peano **succ(N)** es par si N no lo es. Si preguntamos si un número determinado es par, en efecto, obtenemos la respuesta esperada:

```
?- even(zero).           ?- even(succ(succ(zero))).
true.                    true.

?- even(succ(zero)).      ?- even(succ(succ(succ(zero)))).
false.                    false.
```

Pero debido al uso del predicado $\backslash+/1$, no es posible utilizar el predicado **even/1** para listar todos los números de Peano que son pares:

```
?- even(X).
X = zero;
false.
```

Considérese ahora el siguiente programa $\mathcal{P}'$ formado por el predicado **odd/1**, que es cierto cuando un número de Peano N es impar, planteado sin el uso de la negación.

```

R1 : odd(succ(zero)).
R2 : odd(succ(succ(N))) :- odd(N).

```

Si ahora preguntamos por todos los números impares, se observa como se obtienen las infinitas respuestas:

```

?- odd(X).
X = succ(zero);
X = succ(succ(succ(zero)));
X = succ(succ(succ(succ(succ(zero)))));
...

```

Predicados dinámicos

Para que un programa Prolog pueda ejecutarse es preciso que los hechos y las reglas que lo constituyen se hayan cargado en el espacio de trabajo del sistema Prolog. Un sistema Prolog solo reconoce aquello que está en su espacio de trabajo. Por consiguiente, los efectos y acciones que se realicen durante la ejecución de un programa y un objetivo dependen de lo que allí se encuentre almacenado [8].

Los cambios dinámicos en la base de datos interna de Prolog no son revertidos en la vuelta atrás de un punto de elección. Los predicados dinámicos son muy discutidos por algunos autores que desean mantener la pureza y declaratividad del lenguaje [8].

Para declarar un predicado dinámico, hay que utilizar el predicado `dynamic/1`. Esta directiva indica al compilador que un predicado puede no tener cláusulas, y que otras cláusulas pueden ser añadidas en tiempo de ejecución [32]. Para realizar las distintas acciones de inserción y eliminación de cláusulas dinámicamente, se proporcionan los siguientes predicados:

- `asserta/1`. Inserta en la base de datos un hecho o una cláusula del predicado determinado como el primer hecho o cláusula del predicado.
- `assertz/1`. Equivalente al predicado `asserta/1`, pero se inserta como el último hecho o cláusula del predicado.
- `retract/1`. Cuando el término del argumento unifica con el primer hecho o cláusula, este se elimina de la base de datos.
- `retractall/1`. Todos los hechos o cláusulas que unifican con el término del argumento son eliminados de la base de datos.

2.4.4. Gramáticas de cláusulas definidas

Una *gramática de cláusulas definidas* (DCG) representa una gramática como un conjunto de cláusulas definidas en una lógica de primer orden [6]. La notación de DCG es simplemente azúcar sintáctico⁸ para las cláusulas definidas normalmente en Prolog.

Una DCG se define mediante reglas de la forma `Cabeza --> Cuerpo.`, que son internamente traducidas a cláusulas Prolog. El cuerpo de una regla puede contener terminales y no terminales. Un *terminal* es una lista, que representa los elementos que contiene. Un *no terminal* se refiere a otra construcción de la gramática, que representa los elementos que ella misma describe.

Ejemplo 2.9. Definamos una gramática simple para analizar oraciones en lenguaje natural. En primer lugar, declaramos las reglas para un conjunto de determinantes, sustantivos y verbos:

```
R1 : determinante --> [e,l].
R2 : determinante --> [l,a].
R3 : sustantivo    --> [g,a,t,o].
R4 : sustantivo    --> [p,e,s,c,a,d,o].
R5 : verbo        --> [c,o,r,r,e].
R6 : verbo        --> [c,o,m,e].
```

En segundo lugar, definimos la estructura de una oración, que está compuesta por un sujeto y un predicado. Diremos que un sujeto está formado por un determinante y un sustantivo, y un predicado está compuesto por un verbo y un sustantivo:

```
R7 : oracion      --> sujeto, predicado.
R8 : sujeto       --> determinante, sustantivo.
R9 : predicado    --> verbo, sustantivo.
```

Ahora podemos comprobar si una cadena pertenece a esta gramática, es decir, si una determinada cadena puede derivarse de la gramática. Por ejemplo:

```
?- phrase(oracion,[e,l, g,a,t,o, c,o,m,e, p,e,s,c,a,d,o]).
true;
?- phrase(oracion,[e,l, g,a,t,o, c,o,r,r,e]).
false;
```

Vemos que la segunda cadena no puede derivarse como una oración, debido a que el predicado debe ser un verbo seguido de un sustantivo. Podríamos introducir una nueva regla `predicado --> verbo.` para aceptar también este tipo de producciones.

⁸Una construcción en un lenguaje de programación es llamada *azúcar sintáctico* si esta puede ser eliminada del lenguaje sin ningún efecto en la potencia del mismo.

2.5. Lógica difusa

Repasamos en esta sección los fundamentos de la lógica difusa tal como se contemplan en [27].

Las nociones básicas de la lógica difusa han sido formuladas por Zadeh [34, 35], Goguen [4] y Pavelka [26] con la intención de incorporar a la lógica formal los predicados de carácter *vago* del lenguaje ordinario, que han permitido iniciar la construcción del razonamiento aproximado.

Para Zadeh [36], creador de esta disciplina, el término de lógica difusa tiene dos significados diferentes. En el sentido más estricto, la lógica difusa constituye un sistema lógico que se ocupa de la formalización de modos de razonamiento aproximados. En este aspecto, resulta una extensión de los sistemas lógicos polivalentes⁹ y como la lógica simbólica, busca establecer la corrección y completitud de los sistemas que estudia. En el sentido más amplio, la lógica difusa coexiste con la teoría de conjuntos difusos, que es una teoría de clases con fronteras no nítidas.

2.5.1. Conjuntos difusos

La noción de conjunto difuso es introducida por Zadeh [35] y sobre ella puede formalizarse el concepto de interpretación difusa, las operaciones lógicas, los modificadores lingüísticos, etcétera. Mientras que para conjuntos ordinarios la relación de pertenencia tiene un carácter discreto, es decir, un elemento x (del correspondiente universo $\mathcal{U}$) pertenece o no al conjunto A :

$$\forall x \in \mathcal{U}, \quad x \in A \vee x \notin A; \quad A \subset \mathcal{U}$$

en los conjuntos difusos la pertenencia tiene un grado. Para formalizar estos conjuntos es esencial dar sentido a la nueva pertenencia, lo que se hace del siguiente modo. Un conjunto difuso A , en un universo $\mathcal{U}$, se expresa como:

$$A = \{x | \mu_A(x) : \mu_A(x) \neq 0, x \in \mathcal{U}\}$$

donde la aplicación $\mu_A : \mathcal{U} \rightarrow [0, 1]$ es la función grado de pertenencia. Es decir, un conjunto difuso está determinado por la función μ_A . Para cada $x \in \mathcal{U}$, $\mu_A(x) \in [0, 1]$ es un número real que estima la compatibilidad de x con la característica (predicado) que define el conjunto A .

⁹Una lógica polivalente es un sistema lógico que admite más valores de verdad que los tradicionales verdadero y falso.

Si observamos que la pertenencia ordinaria está determinada por la función característica

$$\chi_A : \mathcal{U} \rightarrow \{0, 1\}, \quad \chi_A(x) = \begin{cases} 1, & \text{si } x \in A \\ 0, & \text{si } x \notin A \end{cases}$$

vemos que la pertenencia difusa, graduada por μ_A , es una generalización de la clásica ya que χ_A es una función μ_A particular. De esta observación se deriva el hecho de que todo conjunto ordinario es un conjunto difuso; es decir, la noción de conjunto difuso extiende la de conjunto ordinario.

La definición esencial de la lógica difusa, desde el punto de vista semántico, es la de interpretación, en la que se asocia a cada fórmula atómica un número real en el intervalo $[0, 1]$ ¹⁰. En efecto, se asigna un grado de verdad a una proposición difusa mediante el concepto de conjunto difuso del siguiente modo.

Dado un predicado $A(x)$ en un universo $\mathcal{U}$ y un elemento $x_0 \in \mathcal{U}$, la fórmula $A(x_0)$ se interpreta como verdadera con grado de verdad $\mu_A(x_0)$. En tal caso escribimos:

$$\mathcal{I}(A(x_0)) = \mu_A(x_0)$$

Por tanto, venimos a decir que la proposición $A(x_0)$ se satisface con grado $\mu_A(x_0)$, el grado de pertenencia de x_0 al conjunto difuso A . Y este conjunto es, necesariamente,

$$A = \{x \in \mathcal{U} : A(x)\},$$

es decir, el conjunto cuyo predicado asociado es $A(x)$.

Debemos advertir que, posteriormente, a través de los modificadores lingüísticos, será posible contemplar las proposiciones difusas como falsas, muy verdaderas, muy falsas, algo verdaderas, etc. De este modo, podemos incorporar también el carácter difuso al concepto de interpretación. Más concretamente, podrá asociarse una nueva interpretación a cada modificador de predicado que se formalice.

2.5.2. t-normas, t-conormas y agregadores

La sintaxis de la lógica difusa no presenta muchas novedades. Una vez que se ha interpretado una expresión elemental, fórmulas al efecto otorgan verdad a las expresiones compuestas [18]. Por ejemplo, la conjunción se define habitualmente por la expresión

$$\mathcal{I}(A(x_0) \wedge B(y_0)) = \min\{\mathcal{I}(A(x_0)), \mathcal{I}(B(y_0))\},$$

donde $A(x)$ y $B(y)$ son predicados cualesquiera en universos $\mathcal{U}$, $\mathcal{V}$, respectivamente, y $x_0 \in \mathcal{U}$, $y_0 \in \mathcal{V}$. Si tomamos predicados $A(x)$ y $B(x)$ sobre el mismo universo $\mathcal{U}$ y

¹⁰De manera más general se puede tomar un retículo completo (ver Definición 2.26).

definen, respectivamente, los conjuntos difusos $A, B \subset \mathcal{U}$, se tiene además

$$\mathcal{I}(A(x_0) \wedge B(x_0)) = \mu_{A \cap B}(x_0),$$

donde $\mu_{A \cap B}(x_0)$ es el grado de pertenencia de x_0 al conjunto intersección $A \cap B$.

Pero de manera más general, la función de verdad de la conjunción difusa se puede definir también por todo un conjunto de funciones: las normas triangulares, introducidas por Schweizer y Sklar [29]. La definición de estas funciones, en el intervalo $[0, 1]$, es la que sigue.

Definición 2.15. (Norma triangular, [25]). Una operación $T : [0, 1] \times [0, 1] \rightarrow [0, 1]$ es una norma triangular o t -norma si, y solo si, verifica

- i) Es conmutativa, es decir, $T(x, y) = T(y, x)$, $\forall x, y \in [0, 1]$.
- ii) Es asociativa, es decir, $T(x, T(y, z)) = T(T(x, y), z)$, $\forall x, y, z \in [0, 1]$.
- iii) $T(x, 1) = x$, $\forall x \in [0, 1]$.
- iv) Es monótona en cada componente, es decir¹¹, si $x_1 \leq x_2$, entonces $T(x_1, y) \leq T(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.

De manera análoga, la disyunción se caracteriza habitualmente por la expresión

$$\mathcal{I}(A(x_0) \vee B(x_0)) = \max\{\mathcal{I}(A(x_0)), \mathcal{I}(B(x_0))\},$$

y si consideramos $A(x)$ y $B(x)$ sobre el mismo universo $\mathcal{U}$ definiendo, respectivamente, los conjuntos difusos $A, B \subset \mathcal{U}$, se tiene además

$$\mathcal{I}(A(x_0) \vee B(x_0)) = \mu_{A \cup B}(x_0),$$

donde $\mu_{A \cup B}(x_0)$ es el grado de pertenencia de x_0 al conjunto unión $A \cup B$.

Además, como para la conjunción, la función de verdad de la disyunción se puede definir también por todo un conjunto de funciones: las co-normas triangulares, caracterizadas del siguiente modo en el intervalo cerrado $[0, 1]$.

Definición 2.16. (Co-norma triangular, [25]). Una operación $S : [0, 1] \times [0, 1] \rightarrow [0, 1]$ es una co-norma triangular o t -conorma si, y solo si, verifica

- i) Es conmutativa, es decir, $S(x, y) = S(y, x)$, $\forall x, y \in [0, 1]$.

¹¹De la caracterización dada (solo para la primera componente) se sigue también la monotonía en la segunda componente usando las condiciones i) y iv).

- ii) Es asociativa, es decir, $S(x, S(y, z)) = S(S(x, y), z)$, $\forall x, y, z \in [0, 1]$.
- iii) $S(x, 0) = x$, $\forall x \in [0, 1]$.
- iv) Es monótona en cada componente, es decir¹², si $x_1 \leq x_2$, entonces $S(x_1, y) \leq S(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.

Mediante los operadores de agregación se gestiona la agregación de información de manera eficiente y flexible [5], aspecto que se ha convertido en tarea principal de los problemas de decisión multicriterio, en los que es necesario procesar mucha información de modo que su cantidad y precisión es muy variada.

La definición más general de operador de agregación, en el intervalo $[0, 1]$, es la considerada en [14], que damos en los siguientes términos.

Definición 2.17. (Operador de agregación, [27]). Un operador de agregación $@$ es una aplicación $@ : [0, 1]^n \rightarrow [0, 1]$ que satisface:

- i) Condiciones de frontera: $@(0, \dots, 0) = 0$, $@(1, \dots, 1) = 1$.
- ii) Monotonía: $\forall (x_1, \dots, x_n), (y_1, \dots, y_n) \in [0, 1]^n$,
 $(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \Rightarrow @(x_1, \dots, x_n) \leq @(y_1, \dots, y_n)$

A las condiciones de la definición anterior se añaden, en ocasiones, otras como la continuidad, simetría e idempotencia. En particular, $@$ es simétrico si, y solo si, para toda permutación σ de $\{1, \dots, n\}$ y toda n -tupla $(x_1, \dots, x_n) \in [0, 1]^n$ se cumple $@(x_1, \dots, x_n) = @(x_{\sigma(1)}, \dots, x_{\sigma(n)})$; además, $@$ es idempotente (es decir, $@(x, \dots, x) = x$) si, y solo si, para toda n -tupla $(x_1, \dots, x_n) \in [0, 1]^n$ se cumple $\min\{x_1, \dots, x_n\} \leq @(x_1, \dots, x_n) \leq \max\{x_1, \dots, x_n\}$.

2.5.3. Programación lógica difusa

Desde un punto de vista formal, la programación lógica difusa es un área de la lógica difusa dedicada al estudio de teorías difusas o programas difusos, que son un conjunto de expresiones lógicas difusas en un lenguaje de primer orden [27].

La gestión de la incertidumbre y de la imprecisión en los procesos de deducción es una cuestión relevante dado que en el mundo real la información a procesar es de naturaleza imperfecta. En programación lógica, este asunto ha atraído la atención de muchos investigadores y han sido propuestos distintos marcos para su formalización.

¹²La monotonía en la segunda componente resulta también de i) y iv).

Los marcos actuales para gestionar la imprecisión en programación lógica difusa pueden clasificarse en basados en anotaciones y basados en implicaciones [27]:

- Los basados en anotaciones admiten reglas de la forma $A : f(\beta_1, \dots, \beta_n) \leftarrow B_1 : \beta_1, \dots, B_n : \beta_n$ cuyo significado puede entenderse del modo “el grado de verdad de A es al menos $f(\beta_1, \dots, \beta_n)$ siempre que el grado de verdad de cada átomo B_i sea al menos β_i , $1 \leq i \leq n$ ”, siendo f una función computable y β_i una constante o una variable sobre un dominio apropiado.
- Los basados en implicaciones tienen reglas que podemos dar del modo $\langle A \leftarrow \varsigma(B_1, \dots, B_n); v \rangle$ donde v es el grado de verdad asociado a la fórmula $A \leftarrow \varsigma(B_1, \dots, B_n)$, en la que ς es una conectiva que combina las expresiones atómicas B_i . Computacionalmente, dada una interpretación $\mathcal{I}$, los valores de verdad $\mathcal{I}(B_i)$ se calculan conforme determina la función de verdad de la conectiva ς y posteriormente se propagan convenientemente al átomo A de la cabeza de la regla. Además, los grados de verdad pueden tomarse en un retículo, es decir, la aplicación $\mathcal{I}$ puede tomar valores sobre un cierto retículo.

Es de destacar que la mayoría de las aproximaciones no contemplan ningún tipo de razonamiento no-monótono ni admiten negación, el concepto lógico más relevante que no ha sido contemplado originariamente por la programación lógica difusa, debido a que su incorporación conlleva una complejidad significativa [27].

En cuanto al mecanismo operacional contemplado en las distintas aproximaciones, en muchos de ellos se reemplaza el mecanismo clásico de inferencia, la resolución-SLD, por una variante difusa que permita razonar con incertidumbre y evaluar grados de verdad. La mayoría de estos implementan el principio de resolución difuso introducido por Lee en [18].

Las propiedades de corrección y completitud para los diferentes tipos de semántica operacional han sido propuestas en relación con una semántica declarativa apropiada, que en muchos casos ha sido concebida como una extensión difusa del clásico modelo mínimo de Herbrand [19].

2.6. Lenguajes lógicos difusos

2.6.1. MALP

La programación lógica multi-adjunta, MALP (*Multi-Adjoint Logic Programming*), utiliza un lenguaje de primer orden $\mathcal{L}$ construido sobre un alfabeto Σ , que contiene los elementos de un conjunto infinito numerable de variables $\mathcal{V}$, símbolos de función y

símbolos de predicado con su aridad asociada –normalmente expresados como f^n o P^n donde n representa su aridad–, el símbolo de implicación $\leftarrow$ y un amplio conjunto de otros conectivos. El lenguaje combina elementos de Σ como términos, átomos, reglas y fórmulas. Una *constante* c es un símbolo de función con aridad cero. Un *término* es una variable, una constante o un símbolo de función f aplicado a n términos $t_1, t_2, \dots, t_n$, y se denota como $f(t_1, t_2, \dots, t_n)$ [12].

Asumimos también que en el lenguaje $\mathcal{L}$ se admiten elementos $r \in L$ y conjunciones $\&_i$ de un retículo multi-adjunto, $(L, \leq, \leftarrow_1, \&_1, \dots, \leftarrow_n, \&_n)$, equipado con una colección de pares adjuntos $(\leftarrow_i, \&_i)$, donde cada $\&_i$ es una conjunción ligada a la evaluación del *modus ponens* [27]. Por lo tanto, una fórmula bien formada es:

- r , si $r \in L$.
- $P(t_1, t_2, \dots, t_n)$, si $t_1, t_2, \dots, t_n$ son términos y P^n es un predicado de aridad n . Esta fórmula es un *átomo*.
- $\varsigma(\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_n)$, si $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_n$ son fórmulas bien formadas y ς es una conectiva de aridad n con una función de verdad $\varsigma : L^n \rightarrow L$.

Definición 2.18. (Retículo multi-adjunto, [27]). Sea $(L, \leq)$ un retículo. Un retículo multi-adjunto es una tupla $(L, \leq, \leftarrow_1, \&_1, \dots, \leftarrow_n, \&_n)$ que satisface las siguientes condiciones:

1. $(L, \leq)$ es acotado (es decir, existe el ínfimo, $\perp$, y el supremo, $\top$, de $(L, \leq)$).
2. $(L, \leq)$ es completo¹³ (es decir, para todo $S \subseteq L$ existen $\inf(S)$ y $\sup(S)$).
3. $\&_i(\top, v) = \&_i(v, \top) = v$, $\forall v \in L, i = 1, \dots, n$.
4. Cada par $(\leftarrow_i, \&_i)$, para $i = 1, \dots, n$, es un par adjunto, es decir,
 - a) la operación $\&_i$ es creciente en ambos argumentos.
 - b) la operación $\leftarrow_i$ es creciente en el primer argumento y decreciente en el segundo argumento.
 - c) propiedad adjunta: $\forall x, y, z \in L, x \leq (y \leftarrow_i z) \Leftrightarrow (x \&_i y) \leq z$.

El lenguaje está equipado con un conjunto de *conectivas* interpretadas en el retículo multi-adjunto, incluyendo:

- *Agregadores*, denotados por $\@$, cuyas funciones de verdad $\dot{\@}$ satisfacen la condición de frontera: $\dot{\@}(\top, \top) = \top$, $\dot{\@}(\perp, \perp) = \perp$, y la condición de monotonía: $(x_1, y_1) \leq (x_2, y_2) \Rightarrow \dot{\@}(x_1, y_1) \leq \dot{\@}(x_2, y_2)$.

¹³Si el retículo es completo, entonces es acotado, y podríamos prescindir de la condición 1.

- *t-normas* y *t-conormas* (también llamadas *conjunciones* y *disyunciones*), denotadas por $\&$ y $|$, respectivamente, cuyas funciones de verdad $\dot{\&}$ y $\dot{|}$ satisfacen las siguientes propiedades (vistas en las Definiciones 2.15 y 2.16):

- Conmutativa: $\dot{\&}(x, y) = \dot{\&}(y, x)$ $\dot{|}(x, y) = \dot{|}(y, x)$
- Asociativa: $\dot{\&}(x, \dot{\&}(y, z)) = \dot{\&}(\dot{\&}(x, y), z)$ $\dot{|}(x, \dot{|}(y, z)) = \dot{|}(\dot{|}(x, y), z)$
- Elemento identidad: $\dot{\&}(x, \top) = x$ $\dot{|}(x, \perp) = x$
- Monotonía en cada componente:

$$z \leq t \Rightarrow \begin{cases} \dot{\&}(z, y) \leq \dot{\&}(t, y) & \dot{\&}(x, z) \leq \dot{\&}(x, t) \\ \dot{|}(z, y) \leq \dot{|}(t, y) & \dot{|}(x, z) \leq \dot{|}(x, t) \end{cases}$$

Aunque las conectivas $\&_i$, $|_i$ y $@_i$ son operadores binarios, habitualmente los generalizamos para contemplarlos como funciones con un número arbitrario de argumentos. De este modo, escribiremos¹⁴ $@_i(x_1, x_2, \dots, x_n)$ en lugar de $@_i(x_1, @_i(x_2, \dots, @_i(x_{n-1}, x_n)))$.

$\dot{\&}_{\text{prod}}(x, y) = x \cdot y$	$\dot{\leftarrow}_{\text{prod}}(x, y) = \begin{cases} 1 & \text{si } y \leq x \\ x/y & \text{si } 0 < x < y \end{cases}$
$\dot{\&}_{\text{godel}}(x, y) = \min\{x, y\}$	$\dot{\leftarrow}_{\text{godel}}(x, y) = \begin{cases} 1 & \text{si } y \leq x \\ x & \text{en cualquier otro caso} \end{cases}$
$\dot{\&}_{\text{luka}}(x, y) = \max\{0, x + y - 1\}$	$\dot{\leftarrow}_{\text{luka}}(x, y) = \min\{x - y + 1, 1\}$

Figura 2.1: Pares adjuntos de las lógicas de Gödel, Łukasiewicz y producto sobre $([0, 1], \leq)$

Ejemplo 2.10. En esta memoria utilizaremos el retículo $([0, 1], \leq)$, donde $\leq$ es el orden usual; y tres conjuntos de conectivos correspondientes a la lógica difusa de Gödel, Łukasiewicz y Producto, definidos en la Figura 2.1. Las etiquetas `godel`, `luka` y `prod` significan, respectivamente, *lógica de Gödel*, *lógica de Łukasiewicz* y *lógica del producto* (con diferentes capacidades para modelar escenarios optimistas, pesimistas y realistas).

Es posible incluir otros conectivos, por ejemplo la media aritmética, definida por el conectivo $@_{\text{aver}}$ (con función de verdad $\dot{@}_{\text{aver}}(x, y) = \frac{x+y}{2}$). También es posible incluir conectivos con aridades distintas a dos, como el agregador $@_{\text{very}}$, definido por $\dot{@}_{\text{very}}(x) = x^2$, que es un conectivo unario o modificador lingüístico [12].

¹⁴Puesto que para algunas conectivas no puede generalizarse la asociatividad (ni la conmutatividad) los paréntesis son necesarios y convenimos en estos términos el uso de las conectivas de cualquier número de argumentos.

Definición 2.19. (Programa MALP, [27]). Un programa lógico multi-adjunto es un conjunto $\mathcal{P}$ de reglas de la forma $A \leftarrow_i \mathcal{B}$ with v (también denotadas como un par $\langle A \leftarrow_i \mathcal{B}; v \rangle$) verificando:

1. A es una fórmula atómica, llamada cabeza.
2. $\mathcal{B}$ es una fórmula arbitraria, llamada cuerpo, construida con fórmulas atómicas $B_1, \dots, B_n, n \geq 0$ y cualesquiera conjunciones, disyunciones, agregadores y grados de verdad $r \in L$.
3. $v \in L$ es el grado de verdad de la fórmula $A \leftarrow_i \mathcal{B}$, donde $(L, \leq)$ es el retículo multi-adjunto asociado al programa $\mathcal{P}$.

Como cabe esperar, llamamos hechos a las reglas con cuerpo $\top$ (a menudo, consideraremos también que son reglas con cuerpo vacío).

Ejemplo 2.11. El conjunto de reglas Π mostrado a continuación y el retículo L del Ejemplo 2.10 forman un programa MALP $\mathcal{P}$.

$$\Pi = \begin{cases} R_1 : p(X) & \leftarrow_{\text{prod}} q(X, Y) \ \&_{\text{godel}} \ r(Y) & \text{with } 0.8. \\ R_2 : q(a, Y) & \leftarrow_{\text{prod}} s(Y) & \text{with } 0.7. \\ R_3 : q(Y, a) & \leftarrow_{\text{luka}} r(Y) & \text{with } 0.8. \\ R_4 : r(Y) & \leftarrow & \text{with } 0.7. \\ R_5 : s(b) & \leftarrow & \text{with } 0.9. \end{cases}$$

Semántica procedural de MALP

La semántica procedural de MALP puede ser concebida como una fase operacional seguida de una interpretativa. Denotaremos a las fórmulas del lenguaje $\mathcal{L}$ por $\mathcal{L}$ -fórmulas. Además, un $\mathcal{L}$ -programa se entenderá por un conjunto de pares $\langle \mathcal{L}$ -regla, grado de verdad $\rangle$.

El mecanismo operacional que vamos a definir es un proceso de razonamiento que, usando una generalización del *modus ponens*, proporciona una cota inferior del grado de verdad de un objetivo bajo cualquier modelo de un programa. En esencia, en este contexto, una computación puede verse como un proceso con dos fases procedurales: una primera de tipo operacional y la segunda de carácter interpretativo [27].

De manera más precisa, partiendo de un objetivo extendido y aplicando el *modus ponens* sobre un átomo seleccionado A de dicho objetivo y una regla $\langle A' \leftarrow_i B_i; v \rangle$ del

programa, si existe una sustitución $\theta = mgu(\{A = A'\})$, reemplazamos el átomo A por la expresión extendida $(v \&_i B)\theta$. El proceso se repite hasta obtener una fórmula extendida que no contiene átomos. Entonces, el grado de verdad de esta se obtiene sin más que interpretarla en el retículo asociado. La propiedad adjunta del par $(\leftarrow_i, \&_i)$ garantiza que el grado de verdad resultante es, en efecto, una cota inferior del grado de verdad del objetivo.

Definición 2.20. (Computación admisible difusa, [27]). Sea $\mathcal{Q}$ un objetivo y σ una sustitución, un estado es un par $\langle \mathcal{Q}; \sigma \rangle$. Sea $\mathcal{E}$ el conjunto de estados. Dado ahora un programa MALP $\mathcal{P}$, definimos una computación admisible difusa como un sistema de transición de estados, cuya relación de transición $\rightarrow_{AS} \in (\mathcal{E} \times \mathcal{E})$ es la menor relación que satisface las siguientes reglas admisibles:

1. $\langle \mathcal{Q}[A]; v \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$,
si A es el átomo seleccionado en $\mathcal{Q}$, $\theta = mgu(\{A' = A\})$, $\langle A' \leftarrow_i \mathcal{B}; v \rangle \in \mathcal{P}$ y $\mathcal{B}$ no es vacío.
2. $\langle \mathcal{Q}[A]; v \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$,
si A es el átomo seleccionado en $\mathcal{Q}$, $\theta = mgu(\{A' = A\})$ y $\langle A' \leftarrow; v \rangle \in \mathcal{P}$.
3. $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\perp]); \sigma \rangle$,
si A es el átomo seleccionado en $\mathcal{Q}$ y no existe una regla en $\mathcal{P}$ cuya cabeza unifique con A .

Todos los conceptos usuales de la programación lógica (derivación, paso de derivación, etc.) pueden extenderse a este contexto, al igual que se supone que las fórmulas involucradas en pasos de computación admisibles difusos se renombran antes de ser utilizadas.

Obsérvese que la tercera regla se introduce para contemplar las (posibles) derivaciones admisibles de fallo, lo que supone una novedad importante con respecto a la programación lógica pura en la que no cabe contrapartida al cese de una derivación. En lo que sigue, usaremos los símbolos $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ y $\rightarrow_{FS}$ para referirnos explícitamente a la aplicación de cada una de las reglas admisibles. Cuando sea necesario, anotaremos la regla que intervenga en el paso correspondiente como un superíndice del símbolo $\rightarrow_{AS}$. Usaremos también $\rightarrow_{AS}^n$ para denotar una secuencia de n pasos de una computación admisible y $\rightarrow_{AS}^*$ para una secuencia arbitraria de cero o más pasos.

Definición 2.21. (Derivación admisible completa, [27]). Sea $\mathcal{P}$ un programa MALP y $\mathcal{Q}$ un objetivo. Una secuencia $\mathcal{E}_0 \rightarrow_{AS} \mathcal{E}_1 \rightarrow_{AS}^* \mathcal{E}_n$ es una derivación admisible completa o de éxito si:

1. $\mathcal{E}_0 = \langle \mathcal{Q}; id \rangle$, donde id es la sustitución vacía.
2. Para cada $i, 0 \leq i < n$, $\mathcal{E}_i \rightarrow_{AS} \mathcal{E}_{i+1}$ es un paso de derivación admisible.
3. $\mathcal{E}_n = \langle \mathcal{Q}'; \theta' \rangle$ y $\mathcal{Q}'$ es una $\mathcal{L}$ -fórmula que no contiene átomos.

Tengamos en cuenta que, si después de una secuencia de pasos admisibles, un objetivo se transforma en una $\mathcal{L}$ -fórmula que no contiene átomos, puede ser interpretado directamente en el retículo multi-adjunto L . Esto justifica la siguiente extensión de la noción de respuesta computada a este contexto. Usaremos $\mathcal{V}ar(s)$ para referirnos al conjunto de variables distintas que aparecen en el objeto sintáctico s , y $\theta[\mathcal{V}ar(s)]$ para denotar la sustitución obtenida a partir de θ al restringir su dominio, $Dom(\theta)$, a $\mathcal{V}ar(s)$.

Definición 2.22. (Respuesta computada admisible, [27]). Sea $\mathcal{P}$ un programa MALP y $\mathcal{Q}$ un objetivo. Una derivación admisible es una secuencia $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^* \langle \mathcal{Q}'; \theta \rangle$. Cuando $\mathcal{Q}'$ es una fórmula que no contiene átomos, el par $\langle \mathcal{Q}'; \sigma \rangle$, donde $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, se denomina una respuesta computada admisible (a.c.a.) para esta derivación.

Definición 2.23. (Respuesta computada difusa, [27]). Sea $\mathcal{P}$ un programa multi-adjunto y $\mathcal{Q}$ un objetivo. Dada una derivación admisible $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^* \langle \varsigma(r_1, \dots, r_n); \theta \rangle$, con $r_i \in L$ para todo $i \in \{1, \dots, n\}$, el par $\langle \varsigma(r_1, \dots, r_n); \sigma \rangle$, donde $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, es una respuesta computada difusa (f.c.a.) para esta derivación.

Ejemplo 2.12. Sea $\mathcal{P}$ el programa del Ejemplo 2.11. En la siguiente derivación admisible para el programa $\mathcal{P}$ y el objetivo $\mathbf{p}(X) \&_{\text{godel}} \mathbf{r}(\mathbf{a})$, subrayamos la expresión seleccionada en cada paso admisible:

$$\begin{array}{ll}
\langle \underline{\mathbf{p}(X)} \&_{\text{godel}} \mathbf{r}(\mathbf{a}); id \rangle & \rightarrow_{AS1R1} \\
\langle (0.8 \&_{\text{prod}} (\underline{\mathbf{q}(X1, Y1)} \&_{\text{godel}} \mathbf{r}(Y1))) \&_{\text{godel}} \mathbf{r}(\mathbf{a}); \{X/X1\} \rangle & \rightarrow_{AS1R2} \\
\langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} \underline{\mathbf{s}(Y2)}) \&_{\text{godel}} \mathbf{r}(Y2))) \&_{\text{godel}} \mathbf{r}(\mathbf{a}); \{X/a, X1/a, Y1/Y2\} \rangle & \rightarrow_{AS2R5} \\
\langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} 0.9) \&_{\text{godel}} \underline{\mathbf{r}(b)})) \&_{\text{godel}} \mathbf{r}(\mathbf{a}); \{X/a, X1/a, Y1/b, Y2/b\} \rangle & \rightarrow_{AS2R4} \\
\langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} 0.9) \&_{\text{godel}} 0.7)) \&_{\text{godel}} \underline{\mathbf{r}(a)}; \{X/a, X1/a, Y1/b, Y2/b\} \rangle & \rightarrow_{AS2R4} \\
\langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} 0.9) \&_{\text{godel}} 0.7)) \&_{\text{godel}} 0.7; \{X/a, X1/a, Y1/b, Y2/b\} \rangle &
\end{array}$$

El par $\langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} 0.9) \&_{\text{godel}} 0.7)) \&_{\text{godel}} 0.7; \{X/a, X1/a, Y1/b, Y2/b\} \rangle$ es entonces la respuesta computada admisible para esta derivación admisible. Además, la respuesta computada difusa es el par $\langle 0.504; \{X/a\} \rangle$ (ver Ejemplo 2.13).

Si explotamos todos los átomos de un objetivo, aplicando tantos pasos admisibles como sea necesario en la fase operacional, llegamos a una fórmula que no contiene átomos y que puede ser directamente interpretada en el correspondiente retículo multi-adjunto L .

En lo que sigue, diseñamos la fase interpretativa en términos de un sistema de transición de estados. Veamos la noción de paso de computación interpretativo, derivación interpretativa y (recordemos la de) respuesta computada difusa.

Definición 2.24. (Paso interpretativo, [27]). Sea $\mathcal{P}$ un programa, $\mathcal{Q}$ un objetivo y σ una sustitución. Formalizamos la noción de computación interpretativa como un sistema de transición de estados, cuya relación de transición $\rightarrow_{IS} \subset (\mathcal{E} \times \mathcal{E})$ se define como

$$\langle \mathcal{Q}[\varsigma(r_1, r_2)]; \sigma \rangle \rightarrow_{IS} \langle \mathcal{Q}[\varsigma(r_1, r_2)/\dot{\varsigma}(r_1, r_2)]; \sigma \rangle$$

donde $\dot{\varsigma}$ es la función de verdad de la conectiva ς en el retículo $(L, \leq)$ asociado al programa $\mathcal{P}$.

Definición 2.25. (Derivación interpretativa, [27]). Sea $\mathcal{P}$ un programa y $\langle \mathcal{Q}; \sigma \rangle$ una respuesta computada admisible (a.c.a.), es decir, $\mathcal{Q}$ es un objetivo que no contiene átomos. Una derivación interpretativa es una secuencia $\langle \mathcal{Q}; \sigma \rangle \rightarrow_{IS}^* \langle \mathcal{Q}'; \sigma \rangle$.

Recordemos que si $\mathcal{Q}' = r \in L$, siendo $(L, \leq)$ el retículo asociado a $\mathcal{P}$, el estado $\langle r; \sigma \rangle$ se dice una respuesta computada difusa o f.c.a. (ver Definición 2.23).

Habitualmente, nos referiremos a una *derivación completa* como la secuencia de pasos admisibles/interpretativos de la forma $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^* \langle \mathcal{Q}'; \sigma \rangle \rightarrow_{IS}^* \langle r; \sigma \rangle$, donde $\langle \mathcal{Q}'; \sigma[\text{Var}(\mathcal{Q})] \rangle$ y $\langle r; \sigma[\text{Var}(\mathcal{Q})] \rangle$ son, respectivamente, la a.c.a. y la f.c.a. para la derivación. En ocasiones, la denotaremos por $\langle \mathcal{Q}; id \rangle \rightarrow_{AS/IS}^* \langle r; \sigma \rangle$ (y diremos que $\langle r; \sigma \rangle$ es la respuesta computada difusa de la correspondiente derivación en la que se agrupan los pasos admisibles e interpretativos).

Ejemplo 2.13. Completamos ahora la derivación previa del Ejemplo 2.12, ejecutando los pasos interpretativos necesarios para obtener la respuesta computada difusa (f.c.a.) con respecto al retículo $([0, 1], \leq)$.

$$\begin{aligned} \langle (0.8 \&_{\text{prod}} ((0.7 \&_{\text{prod}} 0.9) \&_{\text{godel}} 0.7)) \&_{\text{godel}} 0.7; \{X/a\} \rangle &\rightarrow_{IS} \\ \langle (0.8 \&_{\text{prod}} (0.63 \&_{\text{godel}} 0.7)) \&_{\text{godel}} 0.7; \{X/a\} \rangle &\rightarrow_{IS} \\ \langle (0.8 \&_{\text{prod}} 0.63) \&_{\text{godel}} 0.7; \{X/a\} \rangle &\rightarrow_{IS} \\ \langle 0.504 \&_{\text{godel}} 0.7; \{X/a\} \rangle &\rightarrow_{IS} \\ \langle 0.504; \{X/a\} \rangle &\end{aligned}$$

Entonces el par $\langle 0.504; \{X/a\} \rangle$ es la f.c.a. para esta derivación completa.

2.6.2. FASILL

FASILL (*Fuzzy Aggregators and Similarity Into a Logic Language*) es un lenguaje de primer orden que integra y extiende las capacidades de MALP al incorporar, entre otras características, relaciones de similaridad. Al igual que otros lenguajes difusos que preceden a FASILL, acepta el formato de reglas con pesos explícitos de la forma $A \leftarrow_{\text{label}} \mathcal{B}$ *with* r , que son internamente traducidas a la sintaxis estándar de FASILL (ver Definición 2.28) como $A \leftarrow r \ \&_{\text{label}} \ \mathcal{B}$. Es importante destacar que en FASILL ya no es necesario exigir la propiedad adjunta para los pares adjuntos $(\leftarrow_{\text{label}}, \&_{\text{label}})$, permitiéndose en este nuevo marco mucho más amplio y flexible el uso de retículos completos, en vez de retículos multi-adjuntos, para la interpretación de programas.

Definición 2.26. (Retículo completo, [12]). *Un retículo completo es un conjunto ordenado $(L, \leq)$ tal que todo subconjunto S de L tiene ínfimo y supremo.*¹⁵

En lo que sigue formalizamos el concepto de relación de similaridad tal como lo usamos en el lenguaje FASILL. El concepto es debido a [33].

Definición 2.27. (Relación de similaridad, [12]). *Dado un dominio $\mathcal{U}$ y un retículo L con una t -norma $\wedge$ fija, una relación de similaridad $\mathcal{R}$ es una relación binaria difusa sobre $\mathcal{U}$, es decir, un subconjunto difuso sobre $\mathcal{U} \times \mathcal{U}$ (caracterizada por una función grado de pertenencia $\mathcal{R} : \mathcal{U} \times \mathcal{U} \rightarrow L$), que satisface las siguientes propiedades:*

- *Reflexiva:* $\mathcal{R}(x, x) = \top, \forall x \in \mathcal{U}$.
- *Simétrica:* $\mathcal{R}(x, y) = \mathcal{R}(y, x), \forall x, y \in \mathcal{U}$.
- *Transitiva:* $\mathcal{R}(x, z) \geq \mathcal{R}(x, y) \wedge \mathcal{R}(y, z), \forall x, y, z \in \mathcal{U}$.

Como cabe esperar, nos interesan las relaciones binarias difusas sobre el dominio semántico. En primer lugar, definimos similaridades sobre símbolos del alfabeto Σ de un lenguaje de primer orden $\mathcal{L}$. Esto hace posible tratar como indistinguibles dos símbolos que están relacionados por $\mathcal{R}$. Además, una relación de similaridad $\mathcal{R}$ sobre el alfabeto de $\mathcal{L}$ se puede extender a los términos mediante inducción estructural de la forma usual [30]. La extensión, $\hat{\mathcal{R}}$, de una relación de similaridad $\mathcal{R}$ se define como sigue [12]:

¹⁵En particular, existe el ínfimo de L y el supremo de L , denotados por $\perp$ y $\top$, respectivamente.

1. Sea x una variable, $\hat{\mathcal{R}}(x, x) = \mathcal{R}(x, x) = \top$.
2. Sean f y g dos símbolos de función de aridad n y sean $t_1, t_2, \dots, t_n, s_1, s_2, \dots, s_n$ términos, $\hat{\mathcal{R}}(f(t_1, t_2, \dots, t_n), g(s_1, s_2, \dots, s_n)) = \mathcal{R}(f, g) \wedge (\bigwedge_{i=1}^n \hat{\mathcal{R}}(t_i, s_i))$.
3. En cualquier otro caso, el grado de similaridad de dos términos es el ínfimo $\perp$.

Análogamente para fórmulas atómicas. Las fórmulas condicionales de la forma $\mathcal{C} \equiv A \leftarrow \mathcal{B}$, donde A es un átomo, tienen una especial relevancia. Para este tipo de fórmulas usaremos una noción diferente de similaridad más restrictiva que la definida en [30]. La idea es que una fórmula condicional $\mathcal{C}$ es similar a otra fórmula condicional $\mathcal{C}'$ si sus cabezas son similares pero mantienen el mismo cuerpo. Así, dados $\mathcal{C} \equiv A \leftarrow \mathcal{B}$ y $\mathcal{C}' \equiv A' \leftarrow \mathcal{B}'$, $\hat{\mathcal{R}}(\mathcal{C}, \mathcal{C}') = \hat{\mathcal{R}}(A, A')$ si $\mathcal{B} \equiv \mathcal{B}'$; y en cualquier otro caso, $\hat{\mathcal{R}}(\mathcal{C}, \mathcal{C}') = \perp$.

Nótese que en esta ampliación no hacemos una distinción de notación entre la relación $\mathcal{R}$ y su extensión $\hat{\mathcal{R}}$.

Ejemplo 2.14. La relación de similaridad $\mathcal{R}$ sobre $\mathcal{U} = \{\text{vanguardist}, \text{elegant}, \text{metro}, \text{taxi}, \text{bus}\}$ está definida mediante la matriz mostrada en la Tabla 2.2, donde se

$\mathcal{R}$	vanguardist	elegant	metro	taxi	bus
vanguardist	1	0.6	0	0	0
elegant	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

Tabla 2.2: Grados de similaridad de la relación del Ejemplo 2.14

puede comprobar que $\mathcal{R}$ satisface las propiedades reflexiva, simétrica y transitiva. Particularmente, utilizando como t-norma $\wedge$ la conjunción de Gödel, tenemos que: $\mathcal{R}(\text{metro}, \text{taxi}) \geq \mathcal{R}(\text{metro}, \text{bus}) \wedge \mathcal{R}(\text{bus}, \text{taxi}) = 0.5 \wedge 0.4$.

Además, la extensión $\hat{\mathcal{R}}$ de la relación $\mathcal{R}$ determina que los términos $\text{elegant}(\text{taxi})$ y $\text{vanguardist}(\text{metro})$ son similares, pues: $\hat{\mathcal{R}}(\text{elegant}(\text{taxi}), \text{vanguardist}(\text{metro})) = \mathcal{R}(\text{elegant}, \text{vanguardist}) \wedge \hat{\mathcal{R}}(\text{taxi}, \text{metro}) = 0.6 \wedge \mathcal{R}(\text{taxi}, \text{metro}) = 0.6 \wedge 0.4 = 0.4$.

Definición 2.28. (Regla FASILL, [12]). Una regla FASILL es de la forma $A \leftarrow \mathcal{B}$, donde A es una fórmula atómica llamada cabeza y $\mathcal{B}$ es una fórmula bien formada (construida a partir de fórmulas atómicas $B_1, B_2, \dots, B_n$, grados de verdad y conectivos de L), que llamamos cuerpo. En particular, cuando el cuerpo de una regla es $r \in L$ (un elemento del retículo L), esta regla se llama hecho y puede ser escrita como $A \leftarrow r$ (o

simplemente como A si $r = \top$).

Definición 2.29. (Programa FASILL, [12]). Un programa FASILL es una 3-tupla $\langle \Pi, \mathcal{R}, L \rangle$ donde Π es un conjunto de reglas, $\mathcal{R}$ es una relación de similaridad entre los elementos de Σ , y L es un retículo completo.

Ejemplo 2.15. El conjunto de reglas Π mostrado a continuación, la relación de similaridad $\mathcal{R}$ del Ejemplo 2.14, y el retículo L del Ejemplo 2.10 forman un programa FASILL $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$.

$$\Pi = \begin{cases} R_1: \text{vanguardist}(\text{hydropolis}) & \leftarrow 0.9. \\ R_2: \text{elegant}(\text{ritz}) & \leftarrow 0.8. \\ R_3: \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7. \\ R_4: \text{good_hotel}(X) & \leftarrow @_{\text{aver}}(\text{elegant}(X), \\ & @_{\text{very}}(\text{close}(X, \text{metro}))). \end{cases}$$

Semántica operacional de FASILL

Las reglas en un programa FASILL tienen el mismo papel que las cláusulas en los programas Prolog o MALP [11, 20], indicando que un cierto predicado relaciona algunos términos (la cabeza) si algunas condiciones (el cuerpo) se mantienen.

Como lenguaje lógico, FASILL hereda los conceptos de sustitución, unificador y unificador más general ($m.g.u.$) descritos en la Sección 2.3. Algunos de estos se han extendido para trabajar con relaciones de similaridad. Concretamente, siguiendo la línea adoptada en Bousi~Prolog [13], el unificador más general se reemplaza por el concepto de *unificador más general débil* ($w.m.g.u.$) y se introduce el *algoritmo de unificación débil* para calcularlo. Este algoritmo especifica que dos expresiones (términos o fórmulas atómicas) $f(t_1, t_2, \dots, t_n)$ y $g(s_1, s_2, \dots, s_n)$ unifican débilmente si los símbolos de la raíz, f y g , están relacionados con un determinado valor de verdad ($\mathcal{R}(f, g) = r > \perp$) y cada uno de los argumentos t_i y s_i unifican débilmente. Entonces, existe un unificador débil para dos expresiones incluso si los símbolos de sus raíces no son sintácticamente iguales ($f \neq g$).

De manera más técnica, el algoritmo de unificación débil que usaremos es una extensión del primero que aparece en [30] para retículos completos arbitrarios. Será formalizado como un sistema de transición de estados soportado por una relación de unificación basada en similaridad “ $\Rightarrow$ ”. La unificación de las expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ se

obtiene mediante una secuencia de transformaciones de estados, empezando desde el estado inicial $\langle G \equiv \{\mathcal{E}_1 \approx \mathcal{E}_2\}, id, \alpha_0 \rangle$, donde id es la sustitución identidad y $\alpha_0 = \top$ es el supremo de $(L, \leq)$: $\langle G, id, \alpha_0 \rangle \Rightarrow \langle G_1, \theta_1, \alpha_1 \rangle \Rightarrow \dots \Rightarrow \langle G_n, \theta_n, \alpha_n \rangle$. Cuando el estado final $\langle G_n, \theta_n, \alpha_n \rangle$, con $G_n = \emptyset$, es alcanzado (es decir, las ecuaciones del estado inicial están resueltas), las expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ son unificables por similitud con w.m.g.u. θ_n y *grado de unificación* α_n . Por lo tanto, el estado final $\langle G_n, \theta_n, \alpha_n \rangle$ devuelve el resultado de la unificación. Por otra parte, cuando las expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ no son unificables, la secuencia de transformaciones de estados termina en fallo ($G_n = \text{Fallo}$).

La *relación de unificación basada en similitud*, “ $\Rightarrow$ ”, se define como la menor relación binaria determinada por el siguiente conjunto de reglas de transición (donde $\mathcal{V}ar(t)$ denota el conjunto de variables de un término t dado) [12]:

$$\frac{\langle \{f(t_1, \dots, t_n) \approx g(s_1, \dots, s_n)\} \cup E, \theta, r_1 \rangle \quad \mathcal{R}(f, g) = r_2 > \perp}{\langle \{t_1 \approx s_1, \dots, t_n \approx s_n\} \cup E, \theta, r_1 \wedge r_2 \rangle} \quad (1)$$

$$\frac{\langle \{X \approx X\} \cup E, \theta, r_1 \rangle}{\langle E, \theta, r_1 \rangle} \quad (2) \qquad \frac{\langle \{X \approx t\} \cup E, \theta, r_1 \rangle \quad X \notin \mathcal{V}ar(t)}{\langle (E)\{X/t\}, \theta\{X/t\}, r_1 \rangle} \quad (3)$$

$$\frac{\langle \{t \approx X\} \cup E, \theta, r_1 \rangle}{\langle \{X \approx t\} \cup E, \theta, r_1 \rangle} \quad (4) \qquad \frac{\langle \{X \approx t\} \cup E, \theta, r_1 \rangle \quad X \in \mathcal{V}ar(t)}{\langle \text{Fallo}, \theta, r_1 \rangle} \quad (5)$$

$$\frac{\langle \{f(t_1, \dots, t_n) \approx g(s_1, \dots, s_n)\} \cup E, \theta, r_1 \rangle \quad \mathcal{R}(f, g) = \perp}{\langle \text{Fallo}, \theta, r_1 \rangle} \quad (6)$$

La primera regla descompone dos expresiones y anota la relación entre el símbolo de función (o de predicado) en su raíz. La segunda regla elimina falsa información y la cuarta regla intercambia la posición de los símbolos que manejan otras reglas. La tercera y quinta reglas realizan una comprobación de la variable X en un término t . En caso de éxito, genera una sustitución $\{X/t\}$; en cualquier otro caso el algoritmo termina con un fallo. También puede terminar en fallo si la relación entre el símbolo de función (o de predicado) en $\mathcal{R}$ es $\perp$, como se especifica en la sexta regla.

Usualmente, dadas dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$, si hay una secuencia de transiciones satisfactoria, $\langle \{\mathcal{E}_1 \approx \mathcal{E}_2\}, id, \top \rangle \Rightarrow^* \langle \emptyset, \theta, r \rangle$, entonces podemos decir que $wmgu(\mathcal{E}_1, \mathcal{E}_2) = \langle \theta, r \rangle$, siendo θ el *unificador débil más general* de $\mathcal{E}_1$ y $\mathcal{E}_2$, y r es su *grado de unificación*.

Finalmente, nótese que, en general, un w.m.g.u. de dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ no es único [30]. Ciertamente, el algoritmo de unificación débil únicamente calcula una clase representativa de un w.g.m.u., en el sentido de que, si $\theta = \{x_1/t_1, x_2/t_2, \dots, x_n/t_n\}$ es un w.m.g.u. con grado β , entonces, por definición, cualquier sustitución $\theta' = \{x_1/s_1, x_2/s_2,$

$\dots, x_n/s_n\}$ que satisfaga $\mathcal{R}(s_i, t_i) > \perp$ para cualquier $1 \leq i \leq n$, es también un w.m.g.u. con un grado de aproximación $\beta' = \beta \wedge (\bigwedge_1^n \mathcal{R}(s_i, t_i))$, donde “ $\wedge$ ” es la t-norma seleccionada. En cualquier caso, se puede observar que el w.m.g.u. representativo calculado por el algoritmo de unificación débil tiene un grado de aproximación igual o mayor que cualquier otro w.m.g.u. Como en el caso del algoritmo de unificación clásico, este algoritmo siempre termina devolviendo el resultado o fallo.

Ejemplo 2.16. Considérese el retículo $L = ([0, 1], \leq)$ del Ejemplo 2.10 y la relación de similaridad $\mathcal{R}$ del Ejemplo 2.14. Es posible el siguiente proceso de unificación débil, dados los términos `elegant(taxi)` y `vanguardist(metro)`.

$$\begin{aligned} \langle \{\text{elegant}(\text{taxi}) \approx \text{vanguardist}(\text{metro})\}, id, 1 \rangle &\xRightarrow{R1} \langle \{\text{taxi} \approx \text{metro}\}, id, 0.6 \rangle \xRightarrow{R1} \\ \langle \{\}, id, 0.6 \wedge 0.4 \rangle &= \langle \{\}, id, 0.4 \rangle \end{aligned}$$

También es posible unificar los términos `elegant(taxi)` y `vanguardist(X)`, dado que:

$$\begin{aligned} \langle \{\text{elegant}(\text{taxi}) \approx \text{vanguardist}(X)\}, id, 1 \rangle &\xRightarrow{R1} \langle \{\text{taxi} \approx X\}, id, 0.6 \rangle \xRightarrow{R4} \\ \langle \{X \approx \text{taxi}\}, id, 0.6 \rangle &\xRightarrow{R3} \langle \{\}, \{X/\text{taxi}\}, 0.6 \rangle \end{aligned}$$

y la sustitución $\{X/\text{taxi}\}$ es su w.m.g.u. con un grado de unificación 0.6.

Con el objetivo de describir la semántica operacional del lenguaje FASILL, en adelante denotaremos por $\mathcal{C}[A]$ a una fórmula donde A es una sub-expresión (normalmente un átomo) que ocurre en el –posiblemente vacío– contexto $\mathcal{C}[]$, mientras que $\mathcal{C}[A/A']$ significa el reemplazo de A por A' en el contexto $\mathcal{C}[]$. Además, $\mathcal{V}ar(s)$ denota el conjunto de las distintas variables que se encuentran en el objeto sintáctico s y $\theta[\mathcal{V}ar(s)]$ se refiere a la sustitución obtenida de θ restringiendo su dominio a $\mathcal{V}ar(s)$.

Definición 2.30. (Paso de computación, [12]). Sea $\mathcal{Q}$ un objetivo y σ una sustitución, y sea A el átomo seleccionado del objetivo $\mathcal{Q}$. El par $\langle \mathcal{Q}; \sigma \rangle$ es un estado. Dado un programa $\langle \Pi, \mathcal{R}, L \rangle$ y una t-norma $\wedge$ en L , una computación se formaliza como un sistema de transición de estados, cuya relación de transición de estados $\rightsquigarrow$ es la menor relación binaria que satisface las siguientes reglas:

1) Paso de éxito (denotado como $\overset{SS}{\rightsquigarrow}$):

$$\frac{\langle \mathcal{Q}[A]; \sigma \rangle \quad A' \leftarrow \mathcal{B} \in \Pi \quad \text{wmgu}(A, A') = \langle \theta; r \rangle}{\langle \mathcal{Q}[A/\mathcal{B} \wedge r]; \theta; \sigma\theta \rangle} \text{SS}$$

2) Paso de fallo (denotado como $\overset{FS}{\rightsquigarrow}$):

$$\frac{\langle \mathcal{Q}[A]; \sigma \rangle \quad \nexists A' \leftarrow \mathcal{B} \in \Pi : \text{wmgu}(A, A') = \langle \theta; r \rangle, r > \perp}{\langle \mathcal{Q}[A/\perp]; \sigma \rangle} \text{FS}$$

3) Paso interpretativo (denotado como $\xrightarrow{IS}$):

$$\frac{\langle \mathcal{Q}[\varsigma(r_1, \dots, r_n)]; \sigma \rangle \quad \varsigma(r_1, \dots, r_n) = r_{n+1}}{\langle \mathcal{Q}[\varsigma(r_1, \dots, r_n)/r_{n+1}]; \sigma \rangle} \text{IS}$$

Una *derivación* se define como una secuencia de longitud arbitraria $\langle \mathcal{Q}; id \rangle \rightsquigarrow^* \langle \mathcal{Q}'; \sigma \rangle$. Como es usual, las reglas se renombran antes de ser usadas. Cuando $\mathcal{Q}' = r \in L$, el estado $\langle r; \sigma \rangle$ se llama *respuesta computada difusa* (f.c.a.) para esa derivación.

Ejemplo 2.17. Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ el programa del Ejemplo 2.15. Es posible realizar una derivación con la respuesta difusa computada $\langle 0.4; \{X/\text{ritz}\} \rangle$ para $\mathcal{P}$ y el objetivo $\mathcal{Q} = \text{good_hotel}(X)$:

$$\begin{array}{ll} \langle \text{good_hotel}(X); id \rangle & \xrightarrow{SS^{R4}} \\ \langle @_{\text{aver}}(\text{elegant}(X), @_{\text{very}}(\text{close}(X, \text{metro}))); \{X_1/X\} \rangle & \xrightarrow{SS^{R2}} \\ \langle @_{\text{aver}}(0.8, @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))); \{X_1/\text{ritz}, X/\text{ritz}\} \rangle & \xrightarrow{\text{FS}} \\ \langle @_{\text{aver}}(0.8, @_{\text{very}}(0)); \{X_1/\text{ritz}, X/\text{ritz}\} \rangle & \xrightarrow{\text{IS}} \\ \langle @_{\text{aver}}(0.8, 0); \{X_1/\text{ritz}, X/\text{ritz}\} \rangle & \xrightarrow{\text{IS}} \\ \langle 0.4; \{X_1/\text{ritz}, X/\text{ritz}\} \rangle & \end{array}$$

Aparte de esta, existe otra f.c.a. alternativa $\langle 0.38; \{X/\text{hydropolis}\} \rangle$ asociada al mismo objetivo.

2.6.3. El entorno FLOPER

FLOPER (*Fuzzy LOGic Programming Environment for Research*) es una herramienta desarrollada en la Universidad de Castilla-La Mancha por el grupo de investigación DEC-TAU. En [10] se describe un método potente para traducir programas difusos a código Prolog estándar directamente ejecutable, mostrando las capacidades básicas de esta herramienta que aspira a ser una plataforma experimental determinante en la tarea de optimización, transformación, especialización y depuración de programas difusos, que el grupo está abordando en los últimos años.

En su estado actual, FLOPER es capaz de analizar y depurar programas lógicos difusos con sintaxis MALP y FASILL sobre cualquier plataforma Prolog, así como de ejecutar objetivos sobre los mismos y generar árboles de derivación para obtener las respuestas computadas difusas.

Herramienta online

Además de las técnicas de transformación y calibrado de programas implementadas sobre el entorno FLOPER a lo largo de este trabajo, se ha desarrollado una nueva herramienta web para el análisis, depuración y ejecución de programas lógicos difusos, accesible desde <http://dectau.uclm.es/unfold/try>, que servirá de base como herramienta para el despliegado y el calibrado de programas MALP y FASILL, como veremos en las Secciones 3.3.4 y 5.5.4.

La página web se divide principalmente en dos zonas. En la zona superior se encuentran los campos de texto para introducir las reglas del programa MALP o FASILL, el retículo asociado al programa y la relación de similaridad, en caso de haberla. En la zona inferior están los campos de texto para introducir un objetivo, la profundidad y el modo de los pasos interpretativos:

- El entorno FLOPER, a diferencia de Prolog, realiza una *búsqueda en anchura* en lugar de una búsqueda en profundidad, y devuelve todas las respuestas computadas difusas encontradas hasta la profundidad especificada.
- El entorno FLOPER cuenta con tres modos para el cómputo de los pasos interpretativos: largo, medio y corto. El modo largo calcula en una sola derivación la respuesta computada difusa; el modo medio calcula en cada paso un conectivo de la expresión; y el modo corto calcula en cada paso una de las operaciones internas de las que se compone un conectivo.

Los retículos están formados por una serie de predicados Prolog, que especifican los miembros que pertenecen al conjunto subyacente del mismo, su relación de orden, el ínfimo y el supremo, y un conjunto de conectivos y agregadores.

- El predicado `member/1` se satisface si el elemento pasado como argumento pertenece al retículo. Nótese que no es necesario que este predicado sea reversible, es decir, no es necesario que por reevaluación pueda listar todos los elementos que pertenecen al retículo.
- El predicado `members/1` devuelve una lista con todos los elementos de un retículo finito numerable. En caso de ser un retículo con infinitos elementos, este predicado puede devolver una muestra representativa¹⁶ de los elementos del mismo.
- Los predicados `bot/1` y `top/1` devuelven, respectivamente, los elementos ínfimo y supremo del retículo.

¹⁶Como veremos en el Capítulo 5, la elección de los miembros que forman parte de este predicado resulta esencial para el rendimiento y la optimalidad del proceso de calibrado de programas simbólicos.

- El predicado `leq/2` se satisface si el primer miembro del retículo pasado como argumento es menor o igual que el segundo miembro del retículo pasado como argumento.
- Los predicados de aridad 3 denotados por un identificador comenzando por `and_` hacen referencia a t-normas o conjunciones. Análogamente, los predicados de aridad 3 denotados por un identificador comenzando por `or_` hacen referencia a t-conormas o disyunciones. Estos predicados toman dos elementos del retículo y devuelven otro.
- Los predicados de cualquier aridad n , $n > 1$, denotados por un identificador comenzando por `agr_` hacen referencia a agregadores, que toman $n - 1$ elementos del retículo y devuelven otro.

Ejemplo 2.18. El siguiente conjunto de reglas Prolog caracteriza el retículo $([0, 1], \leq)$ que utilizaremos en esta memoria.

```

R1 : member(X)                :- number(X), 0=<X, X=<1.
R2 : members([0.3,0.5,0.7]).
R3 : leq(X,Y)                  :- X=<Y.
R4 : bot(0).
R5 : top(1).
R6 : and_prod(X,Y,Z)           :- Z is X*Y.
R7 : and_godel(X,Y,Z)          :- (X=<Y,Z=X;X>Y,Z=Y).
R8 : and_luka(X,Y,Z)           :- U is X+Y-1, (U=<0,Z=U;U>0,Z=0).
R9 : or_prod(X,Y,Z)            :- Z is X+Y-X*Y.
R10 : or_godel(X,Y,Z)          :- (X=<Y,Z=Y;X>Y,Z=X).
R11 : or_luka(X,Y,Z)           :- U is X+Y, (U=<1,Z=U;U>1,Z=1).
R12 : agr_aver(X,Y,Z)          :- Z is (X+Y)/2.
R13 : agr_geom(X,Y,Z)          :- Z is sqrt(X*Y).
R14 : agr_very(X,Y)            :- Y is X*X.

```

donde vemos que las reglas R_6 , R_7 y R_8 se corresponden con conjunciones, las reglas R_9 , R_{10} y R_{11} se corresponden con disyunciones, y las reglas R_{12} , R_{13} y R_{14} se corresponden con agregadores.

En la Figura 2.2 se muestra el programa FASILL del Ejemplo 2.15 introducido en el entorno FLOPER online, y en la Figura 2.3 se muestran las respuestas computadas difusas al ejecutar el objetivo `good_hotel(X)` sobre dicho programa.

Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz) <- 0.8.
close(hydropolis, taxi) <- 0.7.
good_hotel(X) <- @aver(elegant(X), @very(close(X, metro))).
```

[symbolic.fpl](#) [lists.fpl](#) [good_hotel.fpl](#)

Lattice

```
% Elements
member(X) :- number(X), 0 <= X, X <= 1.
members([0.3, 0.5, 0.7]).

% Distance
distance(X,Y,Z) :- Z is abs(Y-X).
```

[bool.lat.pl](#) [num.lat.pl](#)

Similarity equations

```
elegant/1 ~ vanguardist/1 = 0.6.
metro ~ bus = 0.5.
bus ~ taxi = 0.4.
~tnorm = godel.
```

[good_hotel.sim](#)

Figura 2.2: Introducción de programas en el entorno FLOPER online

Goal	Tree depth	Mode	Answers & Tree
good_hotel(X)	12	S M L	Generate

Answers

Computed answers

< 0.4, {X/ritz} >

< 0.38, {X/hydropolis} >

Derivation tree

```
R0 < good_hotel(X), {} >
  R4 < @aver(elegant(X),@very(close(X,metro))), {X1/X} >
    R2 < @aver(0.8,@very(close(ritz,metro))), {X/ritz,X1/ritz} >
      FS < @aver(0.8,@very(0)), {X/ritz,X1/ritz} >
        is < @aver(0.8 0) {X/ritz X1/ritz} >
```

Figura 2.3: Respuestas computadas difusas en el entorno FLOPER online

Entorno de desarrollo en SWI-Prolog

SWI-Prolog [32] es una implementación versátil y libre del lenguaje Prolog, ampliamente utilizada tanto en investigación y educación como en aplicaciones comerciales. SWI-Prolog cuenta con un paquete gráfico, *XPCE*, un conjunto de herramientas para desarrollar aplicaciones gráficas en Prolog y otros lenguajes interactivos y de tipo dinámico.

El motor de inferencia de FLOPER solo podía ejecutarse sobre el entorno *SICStus Prolog*, que únicamente se distribuye bajo licencia, al tiempo que la antigua interfaz gráfica estaba codificada en Java. Con el objetivo de facilitar su portabilidad, nos planteamos migrar esta herramienta al estándar de libre difusión SWI-Prolog, que permitió mejorar la interfaz de usuario de FLOPER e integrarla de forma compacta con su núcleo.

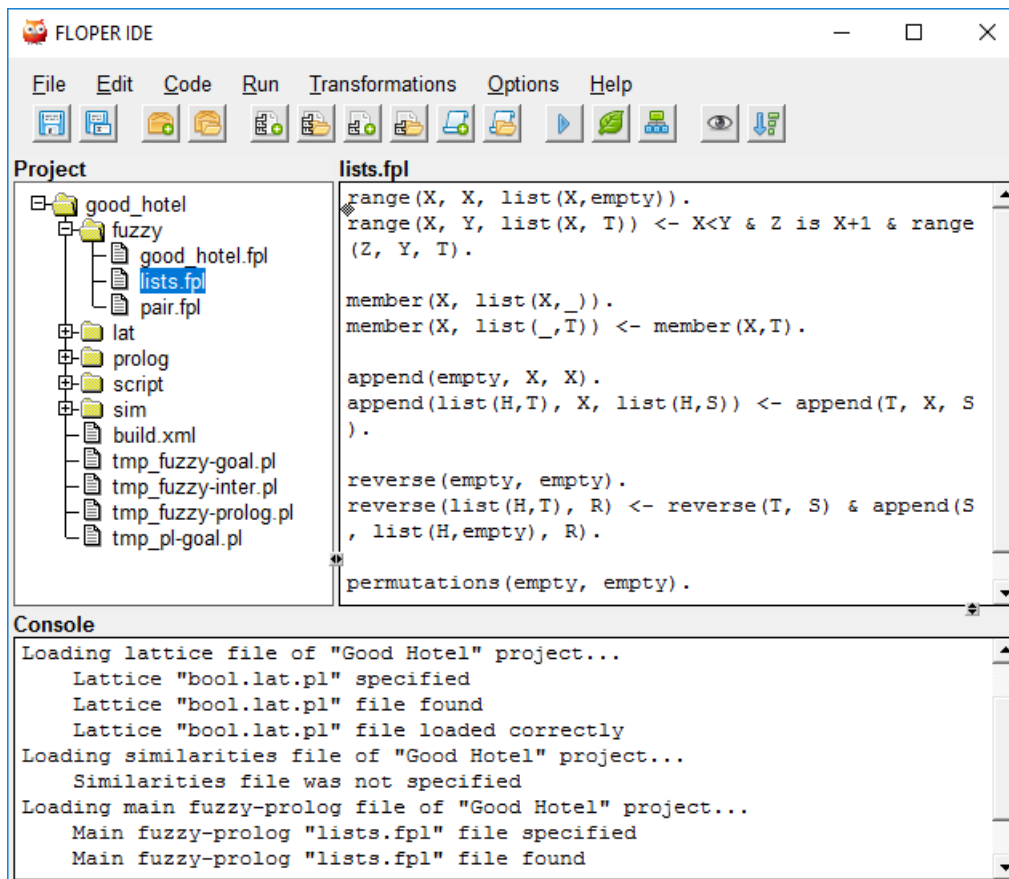


Figura 2.4: Nuevo entorno gráfico de FLOPER en SWI-Prolog

En la Figura 2.4 se muestra el nuevo entorno gráfico desarrollado con SWI-Prolog y XPCE que, actualmente, es capaz de cargar programas con sintaxis MALP y FASILL, ejecutar objetivos sobre los mismos y generar árboles de derivación.

Capítulo 3

Desplegado de programas lógicos difusos

3.1. Desplegado de programas FASILL

La *transformación de programas* es una técnica de optimización para programas declarativos, que empezando con un programa inicial $\mathcal{P}_0$, deriva una secuencia de programas transformados $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n$ por medio de la aplicación de *transformaciones elementales de reglas* (de plegado/desplegado) que mejoran la eficiencia del programa original. Este enfoque de transformaciones de plegado/desplegado se introdujo por primera vez en [2] para optimizar programas funcionales, y más adelante fue utilizado para programas lógicos [31] y programas lógicos funcionales (perezosos) [1].

La transformación de desplegado considerada tradicionalmente en programación lógica pura consiste en el reemplazo de una cláusula C del programa por el conjunto de cláusulas obtenidas tras la aplicación de pasos de computación, en todas sus posibles formas sobre el cuerpo de C [28, 31]. En el Ejemplo 3.1, se muestra la aplicación de un paso de desplegado sobre un programa Prolog.

Ejemplo 3.1. Sea $\mathcal{P}$ el programa Prolog que sigue, formado por el predicado `member/2`, que es cierto cuando el elemento X se encuentra en la lista X_s .

$R_1 : \text{member}(X, [X|_]).$

$R_2 : \text{member}(X, [_|Xs]) \text{ :- member}(X, Xs).$

El programa $\mathcal{P}'$ se obtiene después de desplegar la regla R_2 , seleccionando el átomo `member(X, Xs)`.

$R_1 : \text{member}(X, [X|_]).$

$R_{2-1} : \text{member}(X, [_|[X|_]).$

$R_{2-2} : \text{member}(X, [_|[_|Xs]]) \text{ :- member}(X, Xs).$

Donde se observa que la regla de $\mathcal{P}$ desplegada ha sido eliminada en $\mathcal{P}'$, y en su lugar se han generado dos nuevas reglas, R_{2-1} y R_{2-2} , al unificar el átomo seleccionado con las cabezas de las reglas R_1 y R_2 , respectivamente.

Como se detalla en [9, 10], estas transformaciones han sido adaptadas para trabajar con programas lógicos multi-adjuntos, definiéndose el desplegado en términos de pasos operacionales e interpretativos; en estas aportaciones, se demuestra que la aplicación de pasos de desplegado sobre programas MALP acelera la evaluación de objetivos reduciendo la longitud de las derivaciones. El objetivo de esta sección es adaptar, en los términos que recoge nuestra aportación [21], la definición de la transformación de desplegado a programas FASILL, que mejoran el marco precedente incorporando, entre otros aspectos, relaciones de similaridad.

Definición 3.1. (Desplegado basado en similaridad, [21]). Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa FASILL y sea $R : A \leftarrow B \in \Pi$ una regla del programa no unitaria. Entonces, el desplegado basado en similaridad de la regla R en el programa $\mathcal{P}$ es un nuevo programa $\mathcal{P}' = (\mathcal{P} - \{R\}) \cup \{A\sigma \leftarrow B' \mid \langle B; id \rangle \rightsquigarrow \langle B'; \sigma \rangle\}$.

En adelante, consideraremos que la función de selección utilizada para aplicar pasos de computación sobre un objetivo dado, únicamente aplicará pasos $\overset{\text{IS}}{\rightsquigarrow}$ (siempre de izquierda a derecha) cuando no queden átomos que explotar mediante pasos $\overset{\text{SS}}{\rightsquigarrow}$ o $\overset{\text{FS}}{\rightsquigarrow}$ (nuevamente de izquierda a derecha).

Ejemplo 3.2. Construyamos una secuencia de transformación de programas FASILL donde cada programa de la secuencia se obtiene aplicando una transformación de desplegado sobre el programa inmediatamente anterior, empezando con el programa inicial $\mathcal{P}_0$ del Ejemplo 2.15,

$$\begin{aligned} R_1 : \text{vanguardist}(\text{hydropolis}) &\leftarrow 0.9. \\ R_2 : \text{elegant}(\text{ritz}) &\leftarrow 0.8. \\ R_3 : \text{close}(\text{hydropolis}, \text{taxi}) &\leftarrow 0.7. \\ R_4 : \text{good_hotel}(X) &\leftarrow @_{\text{aver}}(\text{elegant}(X), @_{\text{very}}(\text{close}(X, \text{metro}))). \end{aligned}$$

El programa $\mathcal{P}_1$ con $\Pi_1 = \{R_1, R_2, R_3, R_{4-1}, R_{4-2}\}$ se obtiene después de desplegar la regla R_4 , seleccionando el átomo $\text{elegant}(X)$, al aplicar un paso $\overset{\text{SS}}{\rightsquigarrow}$ con las reglas R_1 y R_2 :

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.9, 0.6), \\
& \quad \quad \quad @_{\text{very}}(\text{close}(\text{hydropolis}, \text{metro}))). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_2$ con $\Pi_2 = \{R_1, R_2, R_3, R_{4-1-3}, R_{4-2}\}$ se obtiene tras desplegar la regla R_{4-1} , seleccionando el átomo $\text{close}(\text{hydropolis}, \text{metro})$, al aplicar un paso $\xrightarrow{\text{SS}}$ con la regla R_3 :

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.9, 0.6), \\
& \quad \quad \quad @_{\text{very}}(\&_{\text{godel}}(0.7, 0.4))). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_3$ con $\Pi_3 = \{R_1, R_2, R_3, R_{4-1-3I}, R_{4-2}\}$ se obtiene desplegando la regla R_{4-1-3} , seleccionando la expresión $\&_{\text{godel}}(0.9, 0.6)$, al aplicar un paso $\xrightarrow{\text{IS}}$:

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3I} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @_{\text{aver}}(0.6, @_{\text{very}}(\&_{\text{godel}}(0.7, 0.4))). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_4$ con $\Pi_4 = \{R_1, R_2, R_3, R_{4-1-3II}, R_{4-2}\}$ se obtiene tras desplegar la regla R_{4-1-3I} , seleccionando la expresión $\&_{\text{godel}}(0.7, 0.4)$, al aplicar un paso $\xrightarrow{\text{IS}}$:

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3II} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @_{\text{aver}}(0.6, @_{\text{very}}(0.4)). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_5$ con $\Pi_5 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2}\}$ se obtiene después de desplegar la regla $R_{4-1-3II}$, seleccionando la expresión $@_{\text{very}}(0.4)$, al aplicar un paso $\xrightarrow{\text{IS}}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow @_{\text{aver}}(0.6, 0.16). \\
R_{4-2} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_6$ con $\Pi_6 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2}\}$ se obtiene desplegando la regla $R_{4-1-3III}$, seleccionando la expresión $@_{\text{aver}}(0.6, 0.16)$, al aplicar un paso $\overset{\text{IS}}{\rightsquigarrow}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow 0.38. \\
R_{4-2} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), \\
& \quad \quad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

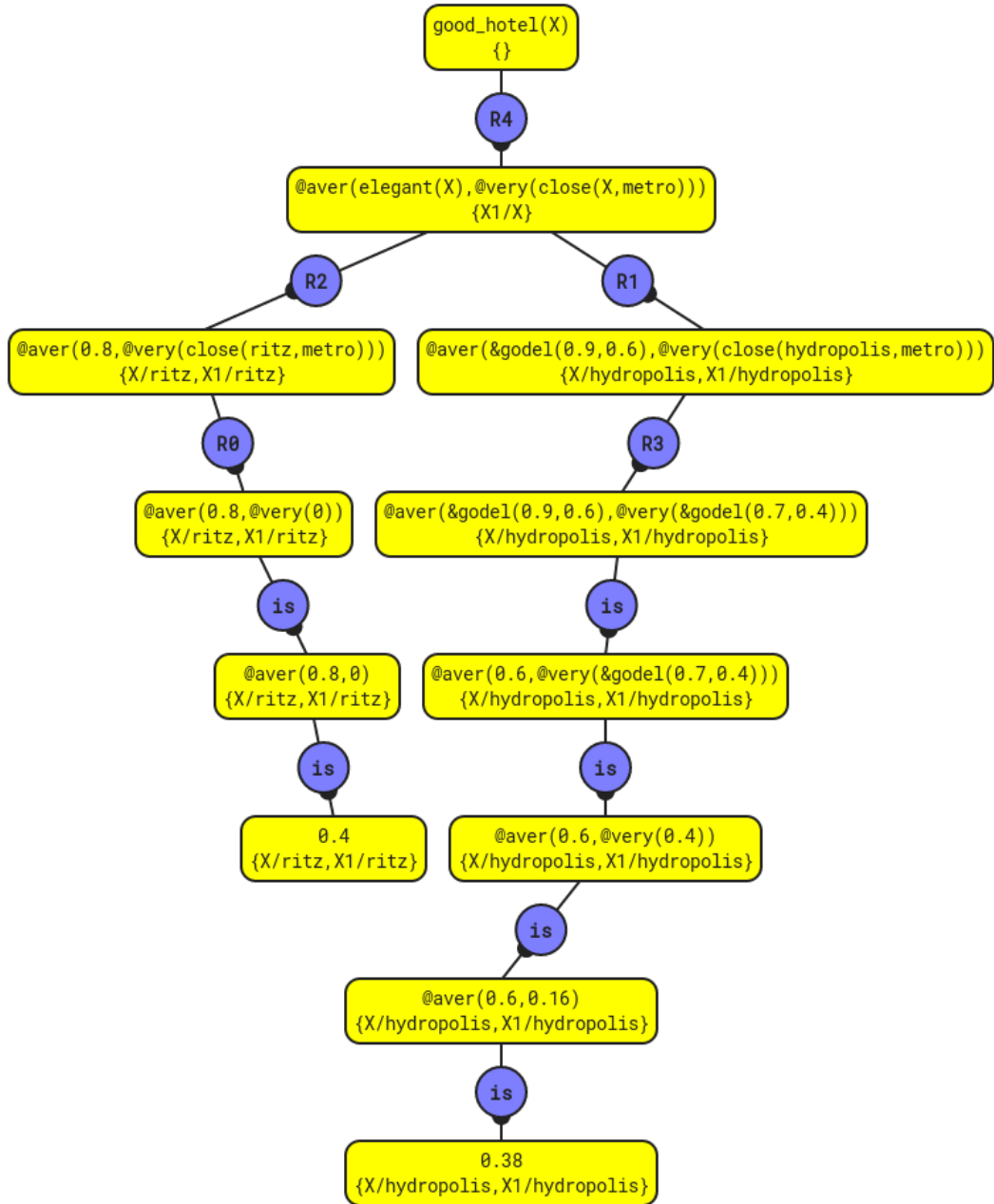
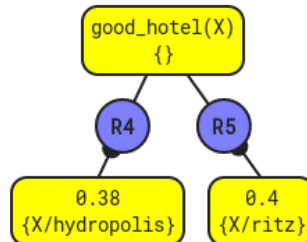
El programa $\mathcal{P}_7$ con $\Pi_7 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2F}\}$ se obtiene después de desplegar la regla R_{4-2} , seleccionando el átomo $\text{close}(\text{ritz}, \text{metro})$, al aplicar un paso $\overset{\text{FS}}{\rightsquigarrow}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow 0.38. \\
R_{4-2F} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 1), @_{\text{very}}(0)).
\end{aligned}$$

Finalmente, tras 3 desplegados más sobre la regla R_{4-2F} (basados en pasos $\overset{\text{IS}}{\rightsquigarrow}$) se obtiene el programa final $\mathcal{P}_{10}$ con $\Pi_{10} = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2FIII}\}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow 0.8. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow 0.38. \\
R_{4-2FIII} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow 0.4.
\end{aligned}$$

En el ejemplo anterior es fácil ver que cada programa de la secuencia generada proporciona las mismas respuestas computadas difusas para un objetivo dado, reduciendo cada vez más la longitud de las derivaciones. Por ejemplo, la derivación realizada con el programa $\mathcal{P}_0$ en el Ejemplo 2.17, puede ejecutarse con el programa $\mathcal{P}_{10}$ con tan solo un paso $\overset{\text{SS}}{\rightsquigarrow}$. En la Figura 3.1 y en la Figura 3.2 se muestran, respectivamente, los árboles de derivación de los programas $\mathcal{P}_0$ y $\mathcal{P}_{10}$ para el objetivo $\text{good_hotel}(X)$ (véase de nuevo [21]).

Figura 3.1: Árbol de derivación del programa $\mathcal{P}_0$ del Ejemplo 3.2Figura 3.2: Árbol de derivación del programa $\mathcal{P}_{10}$ del Ejemplo 3.2

3.2. Corrección del desplegado

Hemos encontrado algunas restricciones que deben ser impuestas cuando desplegamos programas FASILL, debido a que en general, la transformación no es segura, como se ilustra en los siguientes ejemplos.

Ejemplo 3.3. Considérese que el grado de similaridad entre los símbolos de predicado p y q es $\mathcal{R}(p, q) = 0.8$ en el siguiente programa $\mathcal{P}$ y su versión desplegada $\mathcal{P}'$.

$$\begin{aligned}\mathcal{P} &= \begin{cases} R_1 : p(X) \leftarrow @_{\text{aver}}(q(X), 1). \\ R_2 : r(a) \leftarrow 0.6. \end{cases} \\ \mathcal{P}' &= \begin{cases} R_{1-2} : p(a) \leftarrow @_{\text{aver}}(\&_{\text{godel}}(0.8, 0.6), 1). \\ R_2 : r(a) \leftarrow 0.6. \end{cases}\end{aligned}$$

Si observamos las siguientes derivaciones producidas con cada uno de los programas, vemos que no se obtienen las mismas respuestas computadas difusas para el mismo objetivo, para los que, en general, el desplegado no preserva las respuestas generadas en derivaciones que utilizan pasos $\overset{\text{FS}}{\rightsquigarrow}$.

$$\begin{aligned}D_1 : \langle \underline{p(b)}; id \rangle &\overset{\text{SS}^{R_1}}{\rightsquigarrow} \langle @_{\text{aver}}(\underline{q(b)}, 1); \{X/b\} \rangle \overset{\text{FS}}{\rightsquigarrow} \langle @_{\text{aver}}(0, 1); \{X/b\} \rangle \overset{\text{IS}}{\rightsquigarrow} \langle 0.5; \{X/b\} \rangle \\ D'_1 : \langle \underline{p(b)}; id \rangle &\overset{\text{FS}}{\rightsquigarrow} \langle 0; id \rangle\end{aligned}$$

Ejemplo 3.4. Considérese que el grado de similaridad entre las constantes a y b es $\mathcal{R}(a, b) = 0.4$ y entre los símbolos de predicado q y r es $\mathcal{R}(q, r) = 0.5$; considérese, además, el programa $\mathcal{P}$ y su versión desplegada $\mathcal{P}'$ que siguen.

$$\begin{aligned}\mathcal{P} &= \begin{cases} R_1 : p(X) \leftarrow @_{\text{very}}(q(X)). \\ R_2 : r(b) \leftarrow 1. \end{cases} \\ \mathcal{P}' &= \begin{cases} R_{1-2} : p(b) \leftarrow @_{\text{very}}(\&_{\text{godel}}(0.5, 1)). \\ R_2 : r(b) \leftarrow 1. \end{cases}\end{aligned}$$

Nuevamente, observamos en las siguientes derivaciones del programa $\mathcal{P}$ y $\mathcal{P}'$ para un mismo objetivo, que el desplegado no preserva las respuestas originales. En este caso, tenemos que $@_{\text{very}}(\&_{\text{godel}}(0.4, 0.5)) \neq \&_{\text{godel}}(0.4, @_{\text{very}}(0.5))$. Todavía estamos tratando de formalizar condiciones de las conectivas $@$ del retículo (por ejemplo, que se

conservar la conjunción $\wedge$ en cada componente, es decir, $@(t_1, t_2, \dots, (t_i \wedge t_{i+1}), \dots, t_n) = t_i \wedge @(t_1, t_2, \dots, t_{i+1}, \dots, t_n)$, $\forall i, i \leq n$, siendo $\wedge$ la t-norma establecida en el retículo para la propagación de similaridades) que permitan conservar las respuestas computadas difusas del programa original.

$$\begin{array}{ll}
 D_2 : & \langle \underline{p(a)}; id \rangle \quad \begin{array}{l} \text{SS}^{R_1} \\ \rightsquigarrow \end{array} \\
 & \langle @_{\text{very}}(\underline{q(a)}); \{X/a\} \rangle \quad \begin{array}{l} \text{SS}^{R_2} \\ \rightsquigarrow \end{array} \\
 & \langle @_{\text{very}}(\underline{\&_{\text{godel}}(0.4, 0.5)}); \{X/a\} \rangle \quad \begin{array}{l} \text{IS} \\ \rightsquigarrow \end{array} \\
 & \langle @_{\text{very}}(0.4); \{X/a\} \rangle \quad \begin{array}{l} \text{IS} \\ \rightsquigarrow \end{array} \\
 & \langle 0.16; \{X/a\} \rangle \\
 \\
 D'_2 : & \langle \underline{p(a)}; id \rangle \quad \begin{array}{l} \text{SS}^{R_{1-2}} \\ \rightsquigarrow \end{array} \\
 & \langle \underline{\&_{\text{godel}}(0.4, @_{\text{very}}(\underline{\&_{\text{godel}}(0.5, 1)}))}; id \rangle \quad \begin{array}{l} \text{IS} \\ \rightsquigarrow \end{array} \\
 & \langle \underline{\&_{\text{godel}}(0.4, @_{\text{very}}(0.5))}; id \rangle \quad \begin{array}{l} \text{IS} \\ \rightsquigarrow \end{array} \\
 & \langle \underline{\&_{\text{godel}}(0.4, 0.25)}; id \rangle \quad \begin{array}{l} \text{IS} \\ \rightsquigarrow \end{array} \\
 & \langle 0.25; id \rangle
 \end{array}$$

3.3. Implementación del desplegado

En el estado actual, FLOPER permite cargar programas MALP y FASILL y ejecutar objetivos sobre ellos. No obstante, no permite almacenar ni gestionar diferentes versiones de un programa al mismo tiempo. En esta sección se pretende implementar el concepto de transformación, así como el desplegado para programas MALP y FASILL.

3.3.1. Reglas

Como se explica en la Sección 2.4.3 de esta memoria, Prolog permite la modificación de su base de datos durante la ejecución de un programa, añadiendo o eliminando dinámicamente cláusulas de un predicado determinado. Cuando se carga un programa en el entorno FLOPER mediante el predicado `parse/1`, las reglas se analizan sintácticamente y se almacenan como cláusulas Prolog del predicado `rule/5`.

El predicado `rule(?X, ?H, ?I, ?B, ?T)` contiene un identificador numérico `X` (asignado automáticamente de manera incremental), la cabeza de la regla `H`, la implicación `I` utilizada para conectar la cabeza con el cuerpo, el cuerpo de la regla `B` y el grado de verdad `T` de la misma. Esta estructura se utiliza indistintamente para reglas MALP y FASILL, aunque el grado de verdad no tienen utilidad en reglas FASILL, debido a que no utiliza anotaciones explícitas de grados de verdad. En consecuencia, el término `T` en reglas cuyo grado de verdad no se indica, es rellenado con el supremo $\top$ del retículo.

La cabeza y el cuerpo de la regla se expresan como átomos `head/1` y `body/1`, respectivamente, que contienen una representación estructurada en Prolog de la regla analizada, formada como sigue:

- Las conjunciones (`&`), disyunciones (`|`) y agregadores (`@`) se representan mediante los átomos `and/3`, `or/3` y `agr/3`, respectivamente, y contienen el nombre del conector (`godel`, `very`, ...), la aridad n y una lista con los n argumentos.
- Los predicados se representan mediante el átomo `atom/2`, que contiene un término `pred/2` con el nombre y la aridad n del predicado, y una lista con los n argumentos.
- Los términos de función se representan mediante el átomo `term/2`, que contiene un término `fun/2` con el nombre y la aridad n del término, y una lista con los n argumentos.
- Los números, variables y grados de verdad se representan mediante los átomos `num/1`, `var/1` y `td/1`, que contienen el número, el nombre de la variable y el elemento del retículo, respectivamente.

Ejemplo 3.5. Considérese el programa FASILL mostrado en el Ejemplo 2.15, cuyas reglas son las siguientes.

```

R1: vanguardist(hydropolis) ← 0.9.
R2: elegant(ritz)           ← 0.8.
R3: close(hydropolis, taxi) ← 0.7.
R4: good_hotel(X)           ← @aver(elegant(X), @very(close(X, metro))).

```

La representación interna de las reglas de este programa mediante el predicado `rule/5` del entorno FLOPER es la siguiente.

```

R1: rule(1, head(atom(pred(vanguardist,1),[term(fun(hydropolis,0),[])])),
        impl(luka), body(td(0.9)), td(1)).
R2: rule(2, head(atom(pred(elegant,1),[term(fun(ritz,0),[])])),
        impl(luka), body(td(0.8)), td(1)).
R3: rule(3, head(atom(pred(close,2),[term(fun(hydropolis,0),[]),
        term(fun(taxi,0),[])])), impl(luka), body(td(0.7)), td(1)).
R4: rule(4, head(atom(pred(good_hotel,1),[var('X')])), impl(luka), body(
        agr(aver,2,[atom(pred(elegant,1),[var('X')]),agr(very,1,[atom(
        pred(close,2),[var('X'),term(fun(metro,0),[])]))])), td(1)).

```

Podemos observar que todas las reglas FASILL tienen grado de verdad 1, que es el supremo del retículo $([0, 1], \leq)$ asociado al programa. De ser declarado, el grado de verdad de los hechos en FASILL es almacenado en el cuerpo de la regla.

Cada vez que se carga un nuevo programa en el entorno, todas las cláusulas referentes al predicado `rule/5` son eliminadas antes de añadir las nuevas reglas, por lo que es imposible almacenar al mismo tiempo varios programas. Cuando es necesario obtener las reglas del programa actual, basta consultar el predicado `rule/5`.

Con el fin de poder trabajar con las diversas transformaciones de los programas, se hace necesario modificar los mecanismos de almacenamiento y consulta de las reglas, ya que nos interesa guardar las reglas de todas las transformaciones de un programa de la forma más eficiente posible. Para ello, se introducen dos nuevos predicados dinámicos en el entorno FLOPER:

- `rule_insert_id/1`. Contiene el identificador de la última regla insertada hasta el momento. Al iniciar el entorno y antes de cargar un nuevo programa, se inserta el hecho `rule_insert_id(0)`. Cuando se carga un programa, por cada regla analizada correctamente se elimina el último valor y se inserta dicho valor incrementado en uno.
- `rule_current_id/1`. Contiene el identificador de las reglas activas en la versión actual del programa. Al iniciar el entorno no hay ningún hecho de este predicado. Antes de cargar un nuevo programa se eliminan todas las cláusulas de este predicado, y por cada regla R analizada correctamente se inserta un hecho `rule_current_id(IDR)`, donde ID_R es el identificador de la regla R .

En adelante, diremos que una regla R está activa si la cláusula `rule_current_id(IDR)` forma parte de la base de datos del entorno actual.

Si bien antes era posible obtener, por reevaluación, la información de todas las reglas activas simplemente consultando el predicado `rule(?X,?H,?I,?B,?T)`, ahora es necesario consultar primero el identificador de las reglas activas con el predicado `rule_current_id(?X)`, y a partir del identificador X obtener la información de la regla mediante `rule(+X,?H,?I,?B,?T)`, ya que ahora `rule/5` almacena reglas que pueden no estar activas.

3.3.2. Transformaciones

Ahora que es posible almacenar reglas sin necesidad de que estas formen parte del programa actual, se deben crear los predicados necesarios para gestionar las diferentes transformaciones de un programa. Nuevamente, introducimos tres predicados dinámicos en el entorno FLOPER:

- `transformation/6`. Contiene la información de las distintas transformaciones: un identificador numérico (asignado automáticamente de manera incremental), un

término especificando el tipo de transformación (por el momento siempre serán transformaciones del tipo **unfold**), un átomo **from/1** que contiene la versión del programa desde la que se realizó la transformación, un átomo **add/1** que contiene una lista de las reglas añadidas con respecto a la versión anterior, un átomo **remove/1** que contiene una lista de las reglas eliminadas con respecto a la versión anterior, y un átomo **rules/1** que contiene una lista de las reglas que pertenecen a la versión actual.

- **transformation_insert_id/1** y **transformation_current_id/1**. De manera análoga a los predicados **rule_insert_id/1** y **rule_current_id/1**, estos predicados contienen el identificador de la última transformación insertada y el identificador de la transformación activa en ese instante. Solo puede haber un hecho de cada uno de estos predicados en cada momento, debido a que no puede haber más de una versión activa simultáneamente.

Además, se necesita un predicado para cambiar entre las distintas versiones del programa, es decir, para cambiar el conjunto de reglas activas por las reglas de otra versión. El predicado **transformation_goto/1** (ver Anexo B) activa las reglas de una versión concreta del programa: si recibe un número como parámetro, activa las reglas de la versión del programa que tenga dicho identificador; si recibe el término **prev**, **next**, **first** o **last** como parámetro, activa las reglas de la versión previa, siguiente, original o última, respectivamente.

Para facilitar la introducción de nuevas reglas y transformaciones, se han creado los predicados **insert_rules/2** e **insert_transformation/2** (ver Anexo B), que se encargan de asignar los identificadores y añadir las cláusulas correspondientes, devolviendo los identificadores que se han asignado a las nuevas reglas y transformaciones, respectivamente.

3.3.3. Desplegado

El predicado encargado de realizar el despliegado de una regla es **unfold/1**, y para ello hace uso de los siguientes predicados disponibles en el entorno FLOPER:

- **rename/2**. Dada una expresión, devuelve la misma expresión con nuevas variables.
- **subs/3**. Dado un estado inicial $s = \langle Q; \sigma \rangle$ y una sustitución θ , devuelve el nuevo estado $s' = \langle Q\theta; \sigma\theta \rangle$ obtenido tras aplicar la sustitución θ al estado s .
- **step/4**. Dado un estado inicial s , devuelve el estado s' obtenido tras dar un paso de computación sobre el estado s .

Algoritmo 3.1: Desplegado de programas lógicos difusos

Entrada: un programa $\mathcal{P}$ y una regla $R : A \leftarrow \mathcal{B} \in \mathcal{P}$, donde R es la regla a desplegar

Salida: un programa $\mathcal{P}'$ si se ha podido desplegar la regla R , o **falso** en cualquier otro caso

```

1 renombrar las variables de la cabeza  $A$  y del cuerpo  $\mathcal{B}$  de la regla  $R$ ;
2 dar un paso de computación en todas sus posibles formas sobre  $\mathcal{P}$ , tomando el
  cuerpo  $\mathcal{B}$  como objetivo,  $\langle \mathcal{B}; id \rangle \rightarrow \langle \mathcal{B}'_i; \theta_i \rangle$ ;
3 establecer  $\Pi' = \emptyset$ ;
4 para cada  $\langle \mathcal{B}'_i; \theta_i \rangle$  obtenido hacer
5    $A'_i = A\theta_i$ ;  $\triangleright$  aplicar la sustitución  $\theta_i$  sobre la cabeza  $A$  de la regla  $R$ 
6   almacenar en  $\Pi'$  la regla  $R'_i : A'_i \leftarrow \mathcal{B}'_i$ ;
7 fin
8 si  $\Pi' \neq \emptyset$  entonces
9   devolver nueva versión del programa  $\mathcal{P}'$ , tal que  $\mathcal{P}' = (\mathcal{P} - \{R\}) \cup \Pi'$ ;
10 fin
11 devolver falso

```

El predicado `unfold/1` mostrado a continuación, utiliza un predicado dinámico `unfolding/2` para almacenar las reglas que se van generando. El predicado `fail/0` fuerza la reevaluación para dar los pasos de computación de todas las posibles formas mediante el predicado `step/4`.

Cuando se han generado todas las reglas, se recogen en una lista mediante el predicado `findall/3` y se genera una nueva transformación del programa, tal y como especifica el Algoritmo 3.1 de desplegado de programas lógicos difusos.

```

1 unfold(Ri) :-
2     rule_current_id(Ri),
3     retractall(unfolding(Ri, _)),
4     retractall(varcont(_)),
5     assert(varcont(0)),
6     fasiller_ismode(I),
7     rule(Ri, head(Hi), Ir, body(Bi), Tr),
8     rename(Hi, Hv),
9     rename(Bi, Bv),
10    step(I, state(Bv, []), state(Br, S), _),
11    subs(Hv, S, Hr),

```

```

12     assertz(unfolding(Ri, rule(head(Hr), Ir, body(Br), Tr))),
13     fail.
14
15 unfold(Ri) :-
16     retractall(varcont(_)),
17     retractall(state(_, _)),
18     findall(X, unfolding(Ri, X), Rules),
19     Rules \= [],
20     retractall(unfolding(Ri, _)),
21     insert_rules(Rules, Add),
22     insert_transformation(
23         transformation(unfold, add(Add), remove([Ri])), _).

```

3.3.4. Herramienta online para el despliegado

Ampliando la funcionalidad del entorno FLOPER online mostrado en la Sección 2.6.3, bajo la sección de la página donde se introduce el programa se ha añadido una nueva opción, **[+] Unfold program**, para habilitar el despliegado de programas MALP y FASILL, como se muestra en la Figura 3.3.

Program

```

vanguardist(hydropolis) <- 0.9.
elegant(ritz)           <- 0.8.
close(hydropolis, taxi) <- 0.7.
good_hotel(X)           <- @aver(elegant(X), @very(close(X, metro))).

```

symbolic.fpl lists.fpl good_hotel.fpl

[+] Unfold program (0) | « Previous | Next »

Figura 3.3: Menú de despliegado del entorno FLOPER online

El menú de despliegado incluye información sobre la versión del programa actual (mostrada entre paréntesis) y la opción de cargar las versiones previas y posteriores del programa actual, si las hubiera.

Al pulsar la acción de despliegado, se listan las diferentes reglas del programa cargado en el campo de texto y se solicita al usuario que seleccione la regla que desea desplegar. En la Figura 3.4 se muestra la selección de la regla a desplegar para el programa del Ejemplo 2.15.

Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz) <- 0.8.
close(hydropolis, taxi) <- 0.7.
good_hotel(X) <- @aver(elegant(X), @very(close(X, metro))).
```

symbolic.fpl lists.fpl good_hotel.fpl

[−] Unfold program (0) | « Previous | Next »

Select the rule to unfold:

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

Figura 3.4: Selección de la regla a desplegar en el entorno FLOPER online

Si se ha podido desplegar la regla seleccionada, se carga la nueva versión del programa en el campo de texto y se actualiza la información sobre la versión en el menú de desplegado.

En la Figura 3.5 se muestra el resultado de desplegar el programa del Ejemplo 2.15 seleccionando la regla R_4 . Se observa que, como resultado del desplegado, el programa ha pasado de su versión 0 a su versión 1, y la acción de cargar el programa previo se ha activado.

Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz) <- 0.8.
close(hydropolis, taxi) <- 0.7.
good_hotel(ritz) <- @aver(0.8, @very(close(ritz, metro))).
good_hotel(hydropolis) <- @aver((0.9 &godel 0.6), @very(close(hydropolis, metro))).
```

symbolic.fpl lists.fpl good_hotel.fpl

[+] Unfold program (1) | « Previous | Next »

Figura 3.5: Programa desplegado en el entorno FLOPER online

Si el usuario modifica el programa actual (escribiendo directamente sobre el campo de texto del programa) o carga un nuevo programa, el entorno borra toda la información almacenada sobre las versiones y considera que se está trabajando con un nuevo programa, restaurando el contador de versiones a 0.

Capítulo 4

Análisis y ejecución de programas simbólicos

4.1. Programación lógica difusa simbólica

En esta sección se introduce, en los términos que recoge nuestra propuesta [24], una extensión *simbólica* de la programación lógica difusa. En esencia, permitiremos que las reglas de los programas contengan algunos valores (grados de verdad) y conectivos indefinidos, para que esos elementos puedan ser calculados automáticamente más tarde.

En adelante, utilizaremos las abreviaturas *sMALP* (*symbolic Multi-Adjoint Logic Programming*) y *sFASILL* (*symbolic Fuzzy Aggregators and Similarity Into a Logic Language*) para referirnos a estos programas. Los objetos simbólicos se representarán genéricamente como o^s .

4.1.1. sMALP

Dado un retículo multi-adjunto L , consideraremos un lenguaje $\mathcal{L}_L^s \subseteq \mathcal{L}_L$ que puede incluir un conjunto de valores simbólicos, pares adjuntos simbólicos y conectivos simbólicos que no pertenecen a L .

Definición 4.1. (Programa sMALP, [24]). Sea L un retículo multi-adjunto. Un programa sMALP $\mathcal{P}^s$ sobre L es un conjunto de reglas simbólicas, donde cada regla simbólica es una fórmula $(H \leftarrow_i \mathcal{B} \text{ with } v)$ que satisface las siguientes condiciones:

- H es una fórmula atómica de $\mathcal{L}_L$ (la cabeza de la regla).
- $\leftarrow_i$ es una implicación (posiblemente simbólica) procedente de un par adjunto simbólico $(\&^s, \leftarrow^s)$ o de un par adjunto de L .

- $\mathcal{B}$ (el cuerpo de la regla) es un objetivo simbólico, es decir, una fórmula bien formada de $\mathcal{L}_L^s$.
- v es un grado de verdad (un elemento de L) o un valor simbólico.

Ejemplo 4.1. El conjunto de reglas mostrado a continuación y el retículo L del Ejemplo 2.10 forman un programa sMALP.

$$\mathcal{P}^s = \begin{cases} R_1 : p(X) \leftarrow^{s_1} \&^{s_2}(q(X), @_{\text{aver}}(r(X), s(X))) & \text{with } 0.9. \\ R_2 : q(a) & \text{with } v^s. \\ R_3 : r(X) & \text{with } 0.7. \\ R_4 : s(X) & \text{with } 0.5. \end{cases}$$

Donde $(\&^{s_1}, \leftarrow^{s_1})$ es un par adjunto simbólico (es decir, no es un par definido en L), $\&^{s_2}$ es una conjunción simbólica, y v^s es un valor simbólico.

Semántica operacional de sMALP

La semántica operacional de sMALP se diseña de manera análoga a como recoge la Sección 2.6.1 para MALP. En primer lugar, se introduce un paso *operativo* que también recuerda a la resolución SLD de la programación lógica pura. A diferencia de la programación lógica estándar, nuestra etapa operativa devuelve una expresión que puede contener un conjunto de valores y conectivos (posiblemente simbólicos). Entonces, un paso *interpretativo* evalúa los conectivos y produce una respuesta final (posiblemente con algún valor o conectivo simbólico).

La semántica operacional de los programas MALP y sMALP está basada en un esquema similar. La principal diferencia es que, para programas MALP, el paso interpretativo siempre devuelve un valor, mientras que para los programas sMALP es posible obtener una expresión que contenga valores y conectivos simbólicos que deben ser reemplazados primero para poder computarlos.

En adelante, denotaremos por $\mathcal{C}[A]$ a una fórmula donde A es una sub-expresión (normalmente un átomo) que ocurre en el –posiblemente vacío– contexto $\mathcal{C}[]$, mientras que $\mathcal{C}[A/A']$ significa el reemplazo de A por A' en el contexto $\mathcal{C}[]$. Además, $\text{Var}(s)$ denota el conjunto de las distintas variables que se encuentran en el objeto sintáctico s y $\theta[\text{Var}(s)]$ se refiere a la sustitución obtenida de θ restringiendo su dominio a $\text{Var}(s)$. Un *estado de sMALP* tiene la forma $\langle \mathcal{Q}; \sigma \rangle$ donde $\mathcal{Q}$ es un objetivo simbólico y σ es una sustitución. Diremos que $\mathcal{E}^s$ denota el conjunto de todos los posibles estados de sMALP.

Definición 4.2. (Paso admisible, [24]). Sea L un retículo multi-adjunto y $\mathcal{P}^s$ un programa sMALP sobre L . Un paso admisible se define como un sistema de transición de estados, cuya relación de transición $\rightarrow_{AS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$ es la menor relación binaria que satisface las siguientes reglas de transición:

1. $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$,
si $\theta = \text{mgu}(\{H = A\}) \neq \text{Fallo}$, $(H \leftarrow_i \mathcal{B} \text{ with } v) \ll \mathcal{P}^s$ y $\mathcal{B}$ no es vacío¹;
2. $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\perp]); \sigma \rangle$,
si no hay ninguna regla $(H \leftarrow_i \mathcal{B} \text{ with } v) \ll \mathcal{P}^s$ tal que $\text{mgu}(\{H = A\}) \neq \text{Fallo}$;

donde $(H \leftarrow_i \mathcal{B} \text{ with } v) \ll \mathcal{P}^s$ denota que $(H \leftarrow_i \mathcal{B} \text{ with } v)$ es una variante de la regla renombrada en $\mathcal{P}^s$ (es decir, todas sus variables son frescas). Nótese que los valores y conectivos simbólicos no se renombran.

Obsérvese que la segunda regla de la Definición 4.2 es necesaria para trabajar con expresiones como $\mathcal{Q}_{\text{aver}}(\mathbf{p}(\mathbf{a}), 0.8)$, que puede ser evaluada satisfactoriamente incluso cuando no hay una regla $\mathbf{p}(\mathbf{a})$ que coincida, dado que $\mathcal{Q}_{\text{aver}}(0, 0.8) = 0.4$.

En adelante, dada una relación $\rightarrow$, denotaremos por $\rightarrow^*$ a su clausura reflexiva y transitiva. Además, una L^s -expresión ahora es una fórmula bien formada de $\mathcal{L}_L^s$ que está compuesta por valores y conectivos de L así como de grados de verdad y conectivos simbólicos [24].

Definición 4.3. (Derivación admisible, [24]). Sea L un retículo multi-adjunto y $\mathcal{P}^s$ un programa sMALP sobre L . Dado un objetivo $\mathcal{Q}$, una derivación admisible es una secuencia $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^* \langle \mathcal{Q}'; \theta \rangle$. Cuando $\mathcal{Q}'$ es una L^s -expresión, la derivación se dice final y el par $\langle \mathcal{Q}'; \sigma \rangle$, donde $\sigma = \theta[\text{Var}(\mathcal{Q})]$, es llamado una respuesta computada admisible simbólica (s.a.c.a.) para el objetivo $\mathcal{Q}$ en $\mathcal{P}^s$.

Ejemplo 4.2. Considérese de nuevo el retículo multi-adjunto L y el programa sMALP $\mathcal{P}^s$ del Ejemplo 4.1. Entonces, tenemos la siguiente derivación admisible final para $\mathbf{p}(\mathbf{X})$ en $\mathcal{P}^s$ (el átomo seleccionado en cada paso de derivación figura subrayado).

$$\begin{array}{ll}
\langle \underline{\mathbf{p}(\mathbf{X})}; id \rangle & \rightarrow_{AS} \\
\langle \&^{s1}(0.9, \&^{s2}(\underline{\mathbf{q}(\mathbf{X}_1)}, \mathcal{Q}_{\text{aver}}(\mathbf{r}(\mathbf{X}_1), \mathbf{s}(\mathbf{X}_1)))); \{X/X_1\} \rangle & \rightarrow_{AS} \\
\langle \&^{s1}(0.9, \&^{s2}(\mathbf{v}^s, \mathcal{Q}_{\text{aver}}(\underline{\mathbf{r}(\mathbf{a})}, \mathbf{s}(\mathbf{a})))); \{X/a, X_1/a\} \rangle & \rightarrow_{AS} \\
\langle \&^{s1}(0.9, \&^{s2}(\mathbf{v}^s, \mathcal{Q}_{\text{aver}}(0.7, \underline{\mathbf{s}(\mathbf{a})}))); \{X/a, X_1/a, X_2/a\} \rangle & \rightarrow_{AS} \\
\langle \&^{s1}(0.9, \&^{s2}(\mathbf{v}^s, \mathcal{Q}_{\text{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\} \rangle &
\end{array}$$

¹Por simplicidad, consideraremos que los hechos $(H \text{ with } v)$ son vistos como reglas de la forma $(H \leftarrow_i \top \text{ with } v)$ para una implicación $\leftarrow_i$. Además, en este caso, derivaremos directamente el estado $\langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$ ya que $\&_i(v, \top) = v$ para todo $\&_i$.

Por lo tanto, la s.a.c.a. asociada es $\langle \&^{s1}(0.9, \&^{s2}(v^s, @_{\text{aver}}(0.7, 0.5))) ; \{X/a\} \rangle$.

Dado un objetivo $\mathcal{Q}$ y una derivación admisible final $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^* \langle \mathcal{Q}'; \sigma \rangle$, tenemos que $\mathcal{Q}'$ no contiene fórmulas atómicas. Ahora, $\mathcal{Q}'$ puede ser resuelto utilizando los pasos interpretativos descritos en la siguiente definición.

Definición 4.4. (Paso interpretativo, [24]). Sea L un retículo multi-adjunto y $\mathcal{P}^s$ un programa sMALP sobre L . Dada una s.a.c.a. $\langle \mathcal{Q}; \sigma \rangle$, el estado interpretativo se formaliza mediante la siguiente relación de transición $\rightarrow_{IS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$, que se define como la menor relación de transición que satisface:

$$\langle \mathcal{Q}[\varsigma(r_1, r_2, \dots, r_n)]; \sigma \rangle \rightarrow_{IS} \langle \mathcal{Q}[\varsigma(r_1, r_2, \dots, r_n)/r_{n+1}]; \sigma \rangle$$

donde ς denota una conectiva definida sobre L y $\varsigma(r_1, r_2, \dots, r_n) = r_{n+1}$.

Definición 4.5. (Derivación interpretativa, [24]). Sea L un retículo multi-adjunto y $\mathcal{P}^s$ un programa sMALP sobre L . Dado un objetivo $\mathcal{Q}$, una derivación interpretativa es una secuencia $\langle \mathcal{Q}; \sigma \rangle \rightarrow_{IS}^* \langle \mathcal{Q}'; \sigma \rangle$. Cuando no puede reducirse más la expresión $\mathcal{Q}'$, la derivación se dice final. En este caso, $\langle \mathcal{Q}'; \sigma \rangle$ es una respuesta computada difusa simbólica (s.f.c.a.). Además, si $\mathcal{Q}' \in L$, diremos que $\langle \mathcal{Q}'; \sigma \rangle$ es una respuesta computada difusa (f.c.a.).

Ejemplo 4.3. Dada la respuesta computada admisible simbólica del Ejemplo 4.2, $\langle \&^{s1}(0.9, \&^{s2}(v^s, @_{\text{aver}}(0.7, 0.5))) ; \{X/a\} \rangle$, tenemos la siguiente derivación interpretativa final (el conectivo reducido en cada momento está subrayado).

$$\begin{aligned} \langle \&^{s1}(0.9, \&^{s2}(v^s, @_{\text{aver}}(0.7, 0.5))) ; \{X/a\} \rangle &\rightarrow_{IS} \\ \langle \&^{s1}(0.9, \&^{s2}(v^s, 0.6)) ; \{X/a\} \rangle \end{aligned}$$

con $@(0.7, 0.5) = 0.6$. Entonces, $\langle \&^{s1}(0.9, \&^{s2}(v^s, 0.6)) ; \{X/a\} \rangle$ es una s.f.c.a. de $p(X)$ en $\mathcal{P}$ dado que la expresión no se puede reducir más.

4.1.2. sFASILL

Dado un retículo completo L , de manera análoga a como hemos procedido para sMALP, consideraremos un lenguaje $\mathcal{L}_L^s \subseteq \mathcal{L}_L$ que puede incluir un número de valores simbólicos y conectivos simbólicos que no pertenecen a L . Nótese que la principal diferencia con sMALP es el uso de un retículo completo en lugar de uno multi-adjunto,

lo que permite prescindir de los pares adjuntos simbólicos en el lenguaje.

La Definición 2.29 dada para *programas FASILL* sigue siendo válida para *programas sFASILL*, teniendo en cuenta que ahora un programa (simbólico) $\mathcal{P}^s = \langle \Pi^s, \mathcal{R}, L \rangle$ contiene un conjunto de reglas FASILL simbólicas Π^s (o reglas sFASILL), que definimos a continuación.

Definición 4.6. (Regla sFASILL). *Una regla sFASILL es de la forma $A \leftarrow \mathcal{B}$, donde A es una fórmula atómica llamada cabeza y $\mathcal{B}$ es una fórmula bien formada construida a partir de fórmulas atómicas $B_1, B_2, \dots, B_n$, grados de verdad de L , valores simbólicos, y conectivos –posiblemente simbólicos–, que llamamos cuerpo. En particular, cuando el cuerpo de una regla es $r \in L$ (un elemento del retículo L) o un valor simbólico v^s , la regla se llama hecho.*

Ejemplo 4.4. El conjunto de reglas Π^s mostrado a continuación, la relación de similitud $\mathcal{R}$ del Ejemplo 2.14, y el retículo L del Ejemplo 2.10 forman un programa sFASILL $\mathcal{P}^s = \langle \Pi^s, \mathcal{R}, L \rangle$.

$$\Pi^s = \begin{cases} R_1 : \text{vanguardist}(\text{hydropolis}) & \leftarrow 0.9. \\ R_2 : \text{elegant}(\text{ritz}) & \leftarrow v^{s1}. \\ R_3 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7. \\ R_4 : \text{good_hotel}(X) & \leftarrow @^{s2}(\text{elegant}(X), \\ & \quad @_{\text{very}}(\text{close}(X, \text{metro}))). \end{cases}$$

Respecto a la semántica operacional de los programas sFASILL, es idéntica a la semántica operacional de los programas FASILL, teniendo en cuenta que ahora los pasos interpretativos $\xrightarrow{\text{IS}}$ pueden devolver valores o expresiones con valores y conectivos simbólicos, que al igual que en los programas sMALP, deben ser reemplazados antes de poder computarse su valor.

4.2. Desplegado de programas simbólicos

Tal y como recoge nuestra aportación [22], haciendo uso de la nueva semántica operacional de los programas sMALP y sFASILL es posible utilizar la técnica de despliegado de programas difusos mostrada en la Sección 3, para desplegar programas simbólicos. Veámoslo mediante el siguiente ejemplo.

Ejemplo 4.5. Construyamos una secuencia de transformación de programas sFASILL donde cada programa de la secuencia se obtiene aplicando una transformación de desplegado sobre el programa inmediatamente anterior, empezando con el programa inicial $\mathcal{P}_0$ del Ejemplo 4.4.

El programa $\mathcal{P}_1$ con $\Pi_1 = \{R_1, R_2, R_3, R_{4-1}, R_{4-2}\}$ se obtiene después de desplegar la regla R_4 , seleccionando el átomo `elegant(X)`, al aplicar un paso $\overset{\text{SS}}{\rightsquigarrow}$ con las reglas R_1 y R_2 :

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow v^{s^1}. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @^{s^2}(\&_{\text{godel}}(0.9, 0.6), \\
& \qquad \qquad \qquad @_{\text{very}}(\text{close}(\text{hydropolis}, \text{metro}))). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @^{s^2}(\&_{\text{godel}}(v^{s^1}, 1), \\
& \qquad \qquad \qquad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_2$ con $\Pi_2 = \{R_1, R_2, R_3, R_{4-1-3}, R_{4-2}\}$ se obtiene tras desplegar la regla R_{4-1} , seleccionando el átomo `close(hydropolis,metro)`, al aplicar un paso $\overset{\text{SS}}{\rightsquigarrow}$ con la regla R_3 :

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow v^{s^1}. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @^{s^2}(\&_{\text{godel}}(0.9, 0.6), \\
& \qquad \qquad \qquad @_{\text{very}}(\&_{\text{godel}}(0.9, 0.6))). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @^{s^2}(\&_{\text{godel}}(v^{s^1}, 1), \\
& \qquad \qquad \qquad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_5$ con $\Pi_5 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2}\}$ se obtiene tras 3 desplegados basados en pasos $\overset{\text{IS}}{\rightsquigarrow}$:

$$\begin{aligned}
R_1 : \quad & \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : \quad & \text{elegant}(\text{ritz}) \leftarrow v^{s^1}. \\
R_3 : \quad & \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : \quad & \text{good_hotel}(\text{hydropolis}) \leftarrow @^{s^2}(0.6, 0.16). \\
R_{4-2} : \quad & \text{good_hotel}(\text{ritz}) \leftarrow @^{s^2}(\&_{\text{godel}}(v^{s^1}, 1), \\
& \qquad \qquad \qquad @_{\text{very}}(\text{close}(\text{ritz}, \text{metro}))).
\end{aligned}$$

El programa $\mathcal{P}_6$ con $\Pi_6 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2F}\}$ se obtiene desplegando la regla R_{4-2} , seleccionando la expresión `close(ritz,metro)`, al aplicar un paso $\overset{\text{FS}}{\rightsquigarrow}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow v^{s1}. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow @^{s2}(0.6, 0.16). \\
R_{4-2F} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow @^{s2}(\&_{\text{godel}}(v^{s1}, 1), @_{\text{very}}(0)).
\end{aligned}$$

Finalmente, el programa $\mathcal{P}_7$ con $\Pi_7 = \{R_1, R_2, R_3, R_{4-1-3III}, R_{4-2FI}\}$ se obtiene tras desplegar la regla R_{4-2F} , seleccionando la expresión $@_{\text{very}}(0)$, al aplicar un paso $\xrightarrow{\text{IS}}$:

$$\begin{aligned}
R_1 : & \quad \text{vanguardist}(\text{hydropolis}) \leftarrow 0.9. \\
R_2 : & \quad \text{elegant}(\text{ritz}) \leftarrow v^{s1}. \\
R_3 : & \quad \text{close}(\text{hydropolis}, \text{taxi}) \leftarrow 0.7. \\
R_{4-1-3III} : & \quad \text{good_hotel}(\text{hydropolis}) \leftarrow @^{s2}(0.6, 0.16). \\
R_{4-2F} : & \quad \text{good_hotel}(\text{ritz}) \leftarrow @^{s2}(\&_{\text{godel}}(v^{s1}, 1), 0).
\end{aligned}$$

En el ejemplo anterior podemos observar que al desplegar la regla 4 y tras dar todos los pasos interpretativos posibles, no llegamos a obtener ninguna respuesta computada difusa, debido a que el agregador $@^{s2}$ no puede ser interpretado en el retículo. En la Figura 4.1 y en la Figura 4.2 se muestran, respectivamente, los árboles de derivación de los programas $\mathcal{P}_0$ y $\mathcal{P}_7$ para el objetivo $\text{good_hotel}(X)$.

4.3. Implementación de sMALP y sFASILL

En el estado actual, el entorno FLOPER es capaz de analizar léxica y sintácticamente programas MALP y FASILL, y permite ejecutar objetivos y obtener respuestas computadas difusas para estos. En esta sección nos proponemos incorporar al entorno FLOPER los recursos necesarios para poder trabajar con programas simbólicos.

4.3.1. Análisis léxico y sintáctico

El primer paso para introducir los programas simbólicos en el entorno FLOPER es ampliar la sintaxis de los programas MALP y FASILL, con el fin de reconocer los grados de verdad y conectivos simbólicos.

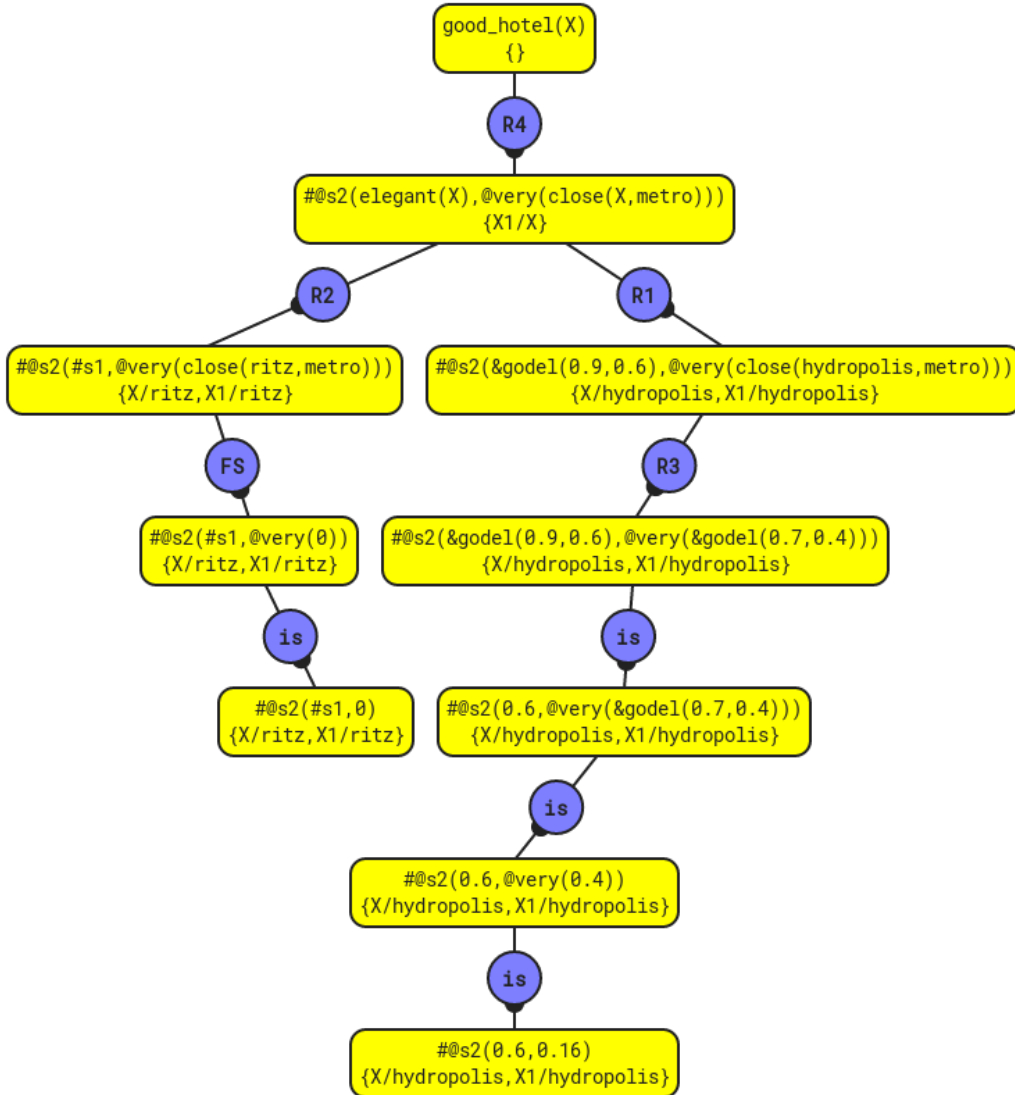
Para ello, se introducen en ambos lenguajes los símbolos mostrados en la Tabla 4.1, donde un conectivo simbólico ($\#?_{\text{label}}$) puede ser una conjunción, una disyunción o un agregador simbólico; y una constante simbólica ($\#_{\text{label}}$) hace referencia a un grado de verdad desconocido del retículo.

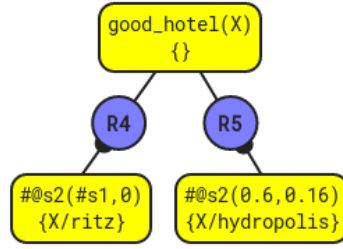
La etiqueta (label) es un identificador del elemento simbólico en cuestión, y está formado por uno o varios caracteres alfanuméricos, donde el primer carácter es una letra minúscula. Aunque dos elementos simbólicos distintos utilicen una misma etiqueta, por

$\# \&_{\text{label}}$	Conjunción simbólica
$\# _{\text{label}}$	Disyunción simbólica
$\# <_{\text{label}}$	Implicación simbólica
$\# @_{\text{label}}$	Agregador simbólico
$\# ?_{\text{label}}$	Conectivo simbólico
$\#_{\text{label}}$	Constante simbólica

Tabla 4.1: Elementos simbólicos de sMALP

ejemplo $\# \&_v$ y $\# |_v$, se consideran etiquetas distintas, en el sentido en que cada uno de estos conectivos puede ser sustituido por uno de cualquiera de las lógicas habituales, por ejemplo $\{\# \&_v / \&_{\text{godel}}, \# |_v / |_{\text{luka}}\}$; a excepción de las implicaciones ($\# <_v$), que siempre van emparejadas con su correspondiente conjunción ($\# \&_v$).

Figura 4.1: Árbol de derivación del programa $\mathcal{P}_0$ del Ejemplo 4.4

Figura 4.2: Árbol de derivación del programa $\mathcal{P}_7$ del Ejemplo 4.4

Ejemplo 4.6. El conjunto de reglas Π mostrado a continuación y el retículo L del Ejemplo 2.10 forman un programa sMALP.

$$\Pi = \left\{ \begin{array}{ll} R_1 : & \text{popularity}(X) \quad \#<_{s1} \text{facilities}(X) \ \# \&_{s1} \\ & \quad \quad \quad \# @_{s2}(\text{location}(X), \text{rates}(X)) \text{ with } 0.9. \\ R_2 : & \text{facilities}(\text{sun}) \quad \text{with } \#_{s3}. \\ R_3 : & \text{location}(\text{sun}) \quad \text{with } \#_{s3}. \\ R_4 : & \text{rates}(\text{sun}) \quad \text{with } 0.7. \\ R_5 : & \text{facilities}(\text{sweet}) \quad \text{with } 0.5. \\ R_6 : & \text{location}(\text{sweet}) \quad \text{with } 0.3. \\ R_7 : & \text{rates}(\text{sweet}). \\ R_8 : & \text{facilities}(\text{lux}) \quad \text{with } 0.9. \\ R_9 : & \text{location}(\text{lux}) \quad \text{with } 0.8. \\ R_{10} : & \text{rates}(\text{lux}) \quad \text{with } 0.2. \end{array} \right.$$

Donde $\#<_{s1}$ y $\# \&_{s1}$ en R_1 representan un par adjunto simbólico, $\# @_{s2}$ en R_1 representa un agregador simbólico, y $\#_{s3}$ en R_2 y R_3 representan un mismo grado de verdad simbólico.

El proceso de análisis léxico y sintáctico se implementa en el entorno FLOPER como una gramática de cláusulas definidas (ver Sección 2.4.4). Debido a que la aplicación contiene alrededor de 300 cláusulas, nos limitaremos a señalar qué cláusulas de la gramática han sido añadidas o modificadas para introducir los conectivos y grados de verdad simbólicos.

La gramática hace uso de una regla **need/2** para controlar los errores sintácticos. Esta regla recibe una expresión y un mensaje de error, y si la expresión no se puede derivar se lanza el mensaje de error mediante el predicado **malp_newError_foundExpected/1**. Después, se sigue analizando el programa desde la siguiente regla.

Las conjunciones, disyunciones y agregadores son tratadas por las siguientes reglas de la gramática, donde el predicado `con2/1` devuelve un identificador válido para un conectivo:

```

1 con('c#'(and,I)) --> ['&'], !, con2(I), blanks, !.
2 con('c#'(or,I)) --> ['|'], !, con2(I), blanks, !.
3 con('c#'(agr,I)) --> ['@'], !, con2(I), blanks, !.
4 con2(I) --> [X], {minus(X)}, !, id_(D), {list2id([X|D],I)}, !.
5 con2(_) --> [], !.

```

De forma similar, añadiremos las siguientes reglas para analizar las conectivas simbólicas. Las implicaciones simbólicas se analizan análogamente por la regla **implication/1**):

[illegible]

Los grados de verdad pueden formar parte del cuerpo de las reglas, o pueden ser anotaciones explícitas mediante la palabra reservada **with** (ver Definición 2.19). El grado de verdad de una regla se analiza mediante las siguientes reglas de la gramática:

```

1 truth_degree(PT) --> [w,i,t,h], !, blanks, need(term(T), 'Truth
2                        degree after with'), {fpl2pl_term(T,PT),
3                        (lat:member(PT); loaded_file_lat(L),
4                        malp_newWarning([PT, ' is not a member of
5                        lattice ', L]))}, !.
6 truth_degree(empty) --> [].

```

Vemos que cuando se especifica el grado de verdad, el entorno comprueba si dicho elemento es un miembro del retículo. De no pertenecer al retículo (o si no hay un retículo cargado en el entorno), se lanza un mensaje de error y se sigue analizando el programa. Añadimos la siguiente regla para permitir grados de verdad simbólicos como anotación:

[illegible]

Finalmente, añadimos otra regla a la gramática para permitir constantes simbólicas en el cuerpo de las reglas. La regla `termX/1` devuelve la etiqueta de la constante simbólica, y la regla `bodyX_exp/2` introduce las constantes simbólicas en el cuerpo de las reglas:

```
1 bodyX_exp(B,td('#sym'(T))) --> [], {fpl2pl_term(B,'sc#'(T))}.
2 termX('sc#'(T)) --> ['#'], need(id(T),'Symbolic constant'), !.
```

Con estas modificaciones a la gramática, ya es posible analizar programas simbólicos con sintaxis sMALP y sFASILL. Recordamos que las reglas de un programa son almacenadas internamente en el entorno FLOPER como cláusulas del predicado `rule/5`, como se explica en la Sección 3.3.1. Los elementos simbólicos serán representados internamente por el átomo `sym/1`, como se ilustra en el siguiente ejemplo.

Ejemplo 4.7. Considérese el programa sFASILL mostrado en el Ejemplo 4.4, cuyas reglas son las siguientes.

$$\begin{aligned} R_1 : \text{vanguardist}(\text{hydropolis}) &\leftarrow 0.9. \\ R_2 : \text{elegant}(\text{ritz}) &\leftarrow v^{s1}. \\ R_3 : \text{close}(\text{hydropolis}, \text{taxi}) &\leftarrow 0.7. \\ R_4 : \text{good_hotel}(X) &\leftarrow @^{s2}(\text{elegant}(X), @_{\text{very}}(\text{close}(X, \text{metro}))). \end{aligned}$$

La representación interna de las reglas de este programa mediante el predicado `rule/5` del entorno FLOPER es la siguiente.

$$\begin{aligned} R_1 : \text{rule}(1, \text{head}(\text{atom}(\text{pred}(\text{vanguardist}, 1), [\text{term}(\text{fun}(\text{hydropolis}, 0), [])])), \\ \text{impl}(\text{luka}), \text{body}(\text{td}(0.9)), \text{td}(1)). \\ R_2 : \text{rule}(2, \text{head}(\text{atom}(\text{pred}(\text{elegant}, 1), [\text{term}(\text{fun}(\text{ritz}, 0), [])])), \\ \text{impl}(\text{luka}), \text{body}(\text{sym}(\text{td}(s1))), \text{td}(1)). \\ R_3 : \text{rule}(3, \text{head}(\text{atom}(\text{pred}(\text{close}, 2), [\text{term}(\text{fun}(\text{hydropolis}, 0), []), \\ \text{term}(\text{fun}(\text{taxi}, 0), [])])), \text{impl}(\text{luka}), \text{body}(\text{td}(0.7)), \text{td}(1)). \\ R_4 : \text{rule}(4, \text{head}(\text{atom}(\text{pred}(\text{good_hotel}, 1), [\text{var}('X')])), \text{impl}(\text{luka}), \text{body}(\\ \text{sym}(\text{agr}(s2, 2, [\text{atom}(\text{pred}(\text{elegant}, 1), [\text{var}('X')]), \text{agr}(\text{very}, 1, [\text{atom}(\text{pred}(\text{close}, 2), [\text{var}('X'), \text{term}(\text{fun}(\text{metro}, 0), [])])]))), \text{td}(1)). \end{aligned}$$

Vemos en la regla R_2 que en el cuerpo se almacena una constante simbólica v^{s1} como `sym(td(s1))`, y en la regla R_4 se almacena el agregador simbólico $@^{s2}$ como `sym(agr(s2, 2, [...]))`, donde el nombre de la etiqueta simbólica ocupa el lugar del valor del grado de verdad o del nombre de la conectiva.

4.3.2. Ejecución parcial

Explicamos en esta sección las modificaciones necesarias en los predicados del entorno FLOPER para poder realizar derivaciones sobre programas simbólicos.

El predicado `select_atom/4` recibe un objetivo Q y devuelve: un átomo A de Q , el objetivo Q con una variable V en lugar del átomo seleccionado, y la variable V . Este predicado se utiliza para seleccionar el siguiente átomo a explotar. Análogamente, el predicado `select_expression/4` recibe un objetivo y devuelve una expresión para ser interpretada.

Dado que internamente un elemento simbólico simplemente está envuelto en el átomo `sym/1`, podemos buscar recursivamente un átomo dentro de este. En el caso de las expresiones, debemos ignorar los elementos simbólicos (dado que no se pueden interpretar en el retículo) y buscar una expresión en sus argumentos:

```

1 select_atom(sym(S),X,sym(S_),V) :- select_atom(S,X,S_,V).
2 select_expression(sym(S),X,sym(S_),V) :-
3     S =.. [O,C,A,L],
4     select_expression_2(L,X,L_,V),
5     S_ =.. [O,C,A,L_].

```

Como ya hemos visto, el predicado `subs/3` toma un estado inicial y una sustitución, y devuelve el estado resultante de aplicar la sustitución al estado inicial; y el predicado `rename/2` toma una expresión y renombra sus variables.

Para aplicar una sustitución a un elemento simbólico, añadimos la siguiente cláusula, que aplica la sustitución recursivamente al contenido del átomo `sym/1`, y posteriormente envuelve el resultado en dicho átomo. Procedemos del mismo modo para renombrar sus variables.

```

1 subs(sym(X),S,sym(X_)) :- subs(X,S,X_), !.
2 rename(sym(X),sym(X_)) :- rename(X,X_).

```

Finalmente, para proporcionar las respuestas e imprimir los estados, añadimos las siguientes cláusulas del predicado `prints/1`, que permiten visualizar los elementos simbólicos precedidos del símbolo “#”.

```

1 prints(sym(X)) :- write('#'), prints(X).
2 prints(con(Label,A,List)) :-
3     write('?'),
4     write(Label),
5     prints2(A,List).

```

4.3.3. Herramienta online para programas simbólicos

El entorno FLOPER ahora está capacitado para trabajar con programas simbólicos, permitiendo ejecutar objetivos y desplegados sobre los mismos. Introducimos estas capacidades en el entorno online. En la Figura 4.3 se muestra la introducción del programa simbólico del Ejemplo 4.4 en el entorno.

Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz) <- #s1.
close(hydropolis, taxi) <- 0.7.
good_hotel(X) <- #s2(elegant(X), @very(close(X, metro))).
```

[+] Unfold program (0) | « Previous | Next »

Figura 4.3: Programa simbólico en el entorno FLOPER online

Una vez introducido el programa, el retículo, y las ecuaciones de similaridad, es posible desplegar el programa. En la Figura 4.4 se muestran las reglas del programa simbólico tras ser analizadas, donde se selecciona la regla R_4 para desplegar.

Select the rule to unfold:

```
1 vanguardist(hydropolis) <- 0.9.
2 elegant(ritz) <- #s1.
3 close(hydropolis, taxi) <- 0.7.
4 good_hotel(X) <- #s2(elegant(X), @very(close(X, metro))).
```

Figura 4.4: Desplegado simbólico en el entorno FLOPER online

Tras realizar siete pasos de desplegado sobre el programa sFASILL original, obtenemos el programa mostrado en la Figura 4.5.

Program

```
vanguardist(hydropolis) <- 0.9.
elegant(ritz) <- #s1.
close(hydropolis, taxi) <- 0.7.
good_hotel(ritz) <- #s2(#s1, 0).
good_hotel(hydropolis) <- #s2(0.6, 0.16).
```

[+] Unfold program (7) | « Previous | Next »

Figura 4.5: Programa simbólico desplegado en el entorno FLOPER online

Tras ejecutar el objetivo `good_hotel(X)` sobre el programa previamente desplegado, obtenemos las respuestas computadas difusas simbólicas y el árbol de derivación mostrados en la Figura 4.6.

Answers

Computed answers

```
< #@s2(#s1,0), {X/ritz} >
```

```
< #@s2(0.6,0.16), {X/hydropolis} >
```

Derivation tree

```
R0 < good_hotel(X), {} >
  R4 < #@s2(#s1,0), {X/ritz} >
  R5 < #@s2(0.6,0.16), {X/hydropolis} >
```

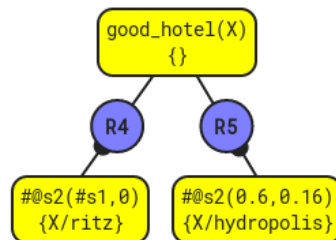


Figura 4.6: Ejecución simbólica en el entorno FLOPER online

Capítulo 5

Calibrado de programas simbólicos

Habitualmente, los programadores tienen en mente un modelo donde los parámetros tienen un valor claro. Por ejemplo, el grado de verdad de una regla puede ser determinado estáticamente y su valor puede ser fácil de obtener. En otros casos, los grados y conectivos más apropiados dependen de nociones subjetivas y los programadores no saben cómo obtenerlos. En un escenario típico, disponemos de un conjunto extenso de respuestas computadas *esperadas* (esto es, *casos de prueba*), así que el programador puede optar por la estrategia de “probar y evaluar”. Desafortunadamente, esta es una tarea tediosa y que consume tiempo. Actualmente, puede ser incluso una tarea impracticable cuando el programa debe modelar correctamente un gran número de casos de prueba.

En esta sección introduciremos un conjunto de técnicas automáticas para el calibrado de programas lógicos difusos utilizando programas sMALP y sFASILL, reproduciendo la propuesta que se formula en [23].

En adelante, denotaremos indistintamente a los programas referidos como *programas simbólicos*, dado que los algoritmos considerados son válidos para ambos lenguajes.

5.1. Sustituciones simbólicas

Dado un retículo L y un lenguaje simbólico $\mathcal{L}_L^s$, en adelante consideraremos *sustituciones simbólicas*, que son aplicaciones que asocian a valores y conectivos simbólicos expresiones sobre $\Sigma_L^T \cup \Sigma_L^C$. Las sustituciones simbólicas son denotadas por Θ , Γ , ... Además, para toda sustitución simbólica Θ , requeriremos la siguiente condición: $\leftarrow^s / \leftarrow_i \in \Theta$ si y solo si $\&^s / \&_i \in \Theta$, donde $(\&^s, \leftarrow^s)$ es un par adjunto simbólico y $(\&_i, \leftarrow_i)$ es un par adjunto¹ en L . Intuitivamente, esta condición es necesaria para que

¹Nótese que esta condición solo es requerida para retículos multi-adjuntos en programas sMALP, no así para retículos completos en programas sFASILL.

la sustitución tenga el mismo efecto sobre el programa y sobre la L^s -expresión.

Dado un programa simbólico $\mathcal{P}^s$ sobre L , denotaremos los valores y conectivos simbólicos en $\mathcal{P}^s$ como $\mathcal{S}ym(\mathcal{P}^s)$. Dada una sustitución simbólica Θ para $\mathcal{S}ym(\mathcal{P}^s)$, denotaremos por $\mathcal{P}^s\Theta$ el programa resultante al reemplazar cada elemento simbólico e^s por $e^s\Theta$ en $\mathcal{P}^s$. Trivialmente, $\mathcal{P}^s\Theta$ es ahora un programa MALP o FASILL², esto es, un programa sin elementos simbólicos, como se muestra en el siguiente ejemplo.

Ejemplo 5.1. Sea $\mathcal{P}^s = \langle \Pi^s, L, R \rangle$ el programa simbólico del Ejemplo 4.4, cuyas reglas Π^s se muestran a continuación, y sea $\Theta = \{v^{s1}/0.5, @^{s2}/@_{very}\}$ una sustitución simbólica para $\mathcal{P}^s$.

$$\Pi^s = \begin{cases} R_1 : \text{vanguardist}(\text{hydropolis}) & \leftarrow v^{s1}. \\ R_2 : \text{elegant}(\text{ritz}) & \leftarrow 0.8. \\ R_3 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7. \\ R_4 : \text{good_hotel}(X) & \leftarrow @_{\text{aver}}(\text{elegant}(X), \\ & @^{s2}(\text{close}(X, \text{metro})). \end{cases}$$

Al aplicar la sustitución Θ sobre el programa $\mathcal{P}^s$, obtenemos un nuevo programa $\mathcal{P}' = \langle \Pi', L, R \rangle$ cuyas reglas Π' son las siguientes.

$$\Pi' = \begin{cases} R_1 : \text{vanguardist}(\text{hydropolis}) & \leftarrow 0.5. \\ R_2 : \text{elegant}(\text{ritz}) & \leftarrow 0.8. \\ R_3 : \text{close}(\text{hydropolis}, \text{taxi}) & \leftarrow 0.7. \\ R_4 : \text{good_hotel}(X) & \leftarrow @_{\text{aver}}(\text{elegant}(X), \\ & @_{\text{very}}(\text{close}(X, \text{metro})). \end{cases}$$

5.2. Distancia entre elementos del retículo

Con el fin de obtener la sustitución simbólica Θ que minimiza la suma de los errores de las respuestas computadas difusas frente a un conjunto de casos de prueba, debemos introducir el concepto de *distancia* entre dos elementos cualesquiera del retículo.

Es decir, para calcular la diferencia entre el grado de verdad esperado y el grado de verdad de una respuesta computada difusa, se hace necesario incluir en el retículo asociado al programa simbólico una nueva operación que devuelva la distancia entre ambos elementos. Formalizaremos este concepto mediante la siguiente definición.

²Consideraremos que las sustituciones simbólicas contienen un valor para todos los elementos simbólicos del programa sobre el que se aplican.

Definición 5.1. (Distancia). Sea X un conjunto. Se dice que la aplicación $d : X \times X \rightarrow \mathbb{R}$ define una distancia en X si, para todo $x, y \in X$, se cumple:

- a) $d(x, y) \geq 0$, y además $d(x, y) = 0$, si y solo si, $x = y$,
- b) $d(x, y) = d(y, x)$,
- c) $d(x, z) \leq d(x, y) + d(y, z)$ (desigualdad triangular).

Ejemplo 5.2. La distancia más común en $\mathbb{R}$ es dada por el valor absoluto, 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

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

En adelante, denotaremos por $d(x, y)$ a la distancia entre dos elementos del retículo, cuyo valor en el caso del retículo $([0, 1], \leq)$ será $d(x, y) = |x - y|$.

5.3. Algoritmos de calibrado

Por simplicidad, consideraremos únicamente la primera respuesta computada de un objetivo. Nótese que esta no es una restricción significativa, dado que uno puede 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 los siguientes algoritmos para múltiples soluciones no es difícil, pero hace la formalización más incómoda. Por consiguiente, diremos que un *caso de prueba* es un par (Q, f) donde Q es un objetivo y f es una respuesta computada difusa.

Como se verá a continuación, la precisión de estos algoritmos puede ser parametrizada dependiendo del conjunto de sustituciones simbólicas consideradas. Por ejemplo, para el retículo del Ejemplo 2.10 se puede considerar únicamente el conjunto de valores $\{0.3, 0.5, 0.8\}$ o el conjunto con más elementos $\{0.0, 0.1, 0.2, \dots, 0.9, 1.0\}$; otro tanto ocurre con los conectivos. Obviamente, cuanto mayor es el dominio de valores y conectivos, más preciso será el resultado, pero más extensiva será la búsqueda.

Ejemplo 5.3. Para ilustrar el funcionamiento de los algoritmos de calibrado, utilizaremos el programa sMALP $\mathcal{P}$ del Ejemplo 4.6, tomando $\{0.3, 0.5, 0.8\}$ como posibles valores del retículo, $\{\&_{prod}, \&_{godel}, \&_{luka}\}$ como posibles conjunciones, $\{|_{prod}, |_{godel}, |_{luka}\}$ como posibles disyunciones y $\{@_{aver}, @_{very}, @_{geom}\}$ ³ como posibles agregadores. El programa contiene tres elementos simbólicos distintos (teniendo en cuenta que la implicación y la conjunción simbólicas de la regla R_1 forman un par adjunto): $\{\leftarrow^{s1}, \&^{s1}, @^{s2}, v^{s3}\}$. Por lo tanto, debemos considerar las siguientes posibles sustituciones simbólicas a la hora de calibrar $\mathcal{P}$:

$$\begin{aligned}
\Theta_1 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{aver}, v^{s3} / 0.3\} \\
\Theta_2 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{aver}, v^{s3} / 0.5\} \\
\Theta_3 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{aver}, v^{s3} / 0.7\} \\
\Theta_4 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{geom}, v^{s3} / 0.3\} \\
\Theta_5 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{geom}, v^{s3} / 0.5\} \\
\Theta_6 &= \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{geom}, v^{s3} / 0.7\} \\
\Theta_7 &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{aver}, v^{s3} / 0.3\} \\
\Theta_8 &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{aver}, v^{s3} / 0.5\} \\
\Theta_9 &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{aver}, v^{s3} / 0.7\} \\
\Theta_{10} &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{geom}, v^{s3} / 0.3\} \\
\Theta_{11} &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{geom}, v^{s3} / 0.5\} \\
\Theta_{12} &= \{\leftarrow^{s1} / \leftarrow_{luka}, \&^{s1} / \&_{luka}, @^{s2} / @_{geom}, v^{s3} / 0.7\} \\
\Theta_{13} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{aver}, v^{s3} / 0.3\} \\
\Theta_{14} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{aver}, v^{s3} / 0.5\} \\
\Theta_{15} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{aver}, v^{s3} / 0.7\} \\
\Theta_{16} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{geom}, v^{s3} / 0.3\} \\
\Theta_{17} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{geom}, v^{s3} / 0.5\} \\
\Theta_{18} &= \{\leftarrow^{s1} / \leftarrow_{prod}, \&^{s1} / \&_{prod}, @^{s2} / @_{geom}, v^{s3} / 0.7\}
\end{aligned}$$

Nótese que aunque el conjunto de agregadores es $\{@_{aver}, @_{very}, @_{geom}\}$, el agregador $@^{s2}$ recibe dos argumentos en el programa $\mathcal{P}$. Por lo tanto, teniendo en cuenta la aridad, el agregador 1-ario $@_{very}$ no puede ser aplicado en lugar de $@^{s2}$.

Además de las sustituciones simbólicas, necesitamos definir una serie de casos de prueba. En este caso, para este programa $\mathcal{P}$ consideraremos los siguientes casos de prueba.

$$\begin{aligned}
t_1 &= (\text{popularity}(\text{sun}), \langle 0.8; id \rangle) \\
t_2 &= (\text{popularity}(\text{sweet}), \langle 0.5; id \rangle) \\
t_3 &= (\text{popularity}(\text{lux}), \langle 0.35; id \rangle)
\end{aligned}$$

³Definimos el agregador $@_{geom}$ como la media geométrica de dos elementos del retículo, cuya función de verdad es $\hat{@}_{geom}(x, y) = \sqrt{x \cdot y}$.

Es decir, queremos conocer la mejor sustitución simbólica para minimizar el error del objetivo $\text{popularity}(\mathbf{X})$, esperando obtener el grado de verdad 0.8 para **sun**, el grado de verdad 0.5 para **sweet** y el grado de verdad 0.35 para **lux**.

5.3.1. Calibrado básico

El método *básico* de calibrado es el más simple, dado que no hace uso del mecanismo operacional de los programas simbólicos, y a su vez es el más lento, debido a que por cada posible sustitución simbólica debe computar completamente los objetivos de todos los casos de prueba. Es decir, primero se aplica cada sustitución simbólica sobre el programa original, y luego se computan los objetivos de los casos de prueba, devolviendo la sustitución simbólica que minimiza el error respecto a los valores esperados en los casos de prueba.

Algoritmo 5.1: Calibrado básico de programas simbólicos

Entrada: un programa simbólico $\mathcal{P}^s$ y un número de casos de prueba esperados $(Q_i, \langle v_i; \theta_i \rangle)$, con $i \in \{1, \dots, k\}$

Salida: una sustitución simbólica Θ

```

1 considerar un número finito de posibles sustituciones simbólicas para  $\text{Sym}(\mathcal{P}^s)$ ,
   denotadas por  $\Theta_1, \Theta_2, \dots, \Theta_n$ , con  $n > 0$ ;
2 para cada sustitución simbólica  $\Theta_j$ , con  $j \in \{1, \dots, n\}$  hacer
3   | calcular  $\mathcal{P}'_j = \mathcal{P}^s \Theta_j$ ;
4   | para cada caso de prueba  $t_i$ , con  $i \in \{1, \dots, k\}$  hacer
5   |   | computar  $\langle Q_i; id \rangle \rightarrow^* \langle v_{i,j}; \theta_{i,j} \rangle$  en  $\mathcal{P}'_j$ ;
6   |   | calcular  $d_{i,j} = d(v_{i,j}, v_i)$ ;
7   | fin
8 fin
9 devolver  $\Theta_j$  que minimiza  $\sum_{i=1}^k d_{i,j}$ 

```

Ejemplo 5.4. Calibremos el programa del Ejemplo 5.3 por el método básico. Para ello, primero se aplica cada sustitución Θ_i , con $i \in \{1, \dots, 18\}$, sobre el programa $\mathcal{P}^s$, y luego se computan los objetivos de los tres casos de prueba, obteniendo los resultados mostrados en la Tabla 5.1 (en las columnas **sun**, **sweet** y **lux** se muestran las f.c.a. obtenidas junto al error cometido respecto a los valores esperados).

$\leftarrow^{s1}, \&^{s1}$	@ ^{s2}	v^{s3}	Θ	sun		sweet		lux		z
$\leftarrow_{godel}, \&_{godel}$	@ _{aver}	0.3	Θ_1	0.3	0.5	0.5	0	0.5	0.15	0.65
		0.5	Θ_2	0.5	0.3	0.5	0	0.5	0.15	0.45
		0.7	Θ_3	0.7	0.1	0.5	0	0.5	0.15	0.25
	@ _{geom}	0.3	Θ_4	0.3	0.5	0.5	0	0.4	0.05	0.55
		0.5	Θ_5	0.5	0.3	0.5	0	0.4	0.05	0.35
		0.7	Θ_6	0.7	0.1	0.5	0	0.4	0.05	0.15
$\leftarrow_{luka}, \&_{luka}$	@ _{aver}	0.3	Θ_7	0	0.8	0.05	0.45	0.3	0.05	1.3
		0.5	Θ_8	0	0.8	0.05	0.45	0.3	0.05	1.3
		0.7	Θ_9	0.3	0.5	0.05	0.45	0.3	0.05	1
	@ _{geom}	0.3	Θ_{10}	0	0.8	0	0.5	0.2	0.15	1.45
		0.5	Θ_{11}	0	0.8	0	0.5	0.2	0.15	1.45
		0.7	Θ_{12}	0.3	0.5	0	0.5	0.2	0.15	1.15
$\leftarrow_{prod}, \&_{prod}$	@ _{aver}	0.3	Θ_{13}	0.13	0.67	0.29	0.21	0.4	0.05	0.93
		0.5	Θ_{14}	0.27	0.53	0.29	0.21	0.4	0.05	0.79
		0.7	Θ_{15}	0.44	0.36	0.29	0.21	0.4	0.05	0.62
	@ _{geom}	0.3	Θ_{16}	0.12	0.68	0.24	0.26	0.32	0.03	0.97
		0.5	Θ_{17}	0.26	0.54	0.24	0.26	0.32	0.03	0.83
		0.7	Θ_{18}	0.44	0.36	0.24	0.26	0.32	0.03	0.65

Tabla 5.1: Resumen de pesos al calibrar el programa del Ejemplo 5.3

Como se puede observar, la sustitución simbólica que minimiza el error respecto a los casos de prueba es $\Theta_6 = \{\leftarrow^{s1} / \leftarrow_{godel}, \&^{s1} / \&_{godel}, @^{s2} / @_{geom}, v^{s3} / 0.7\}$ con un error de 0.15. Por lo tanto, el algoritmo de calibrado devuelve la sustitución simbólica Θ_6 .

5.3.2. Calibrado simbólico

El método *simbólico* de calibrado computa primero las s.f.c.a. de los objetivos de los casos de prueba desde el programa simbólico $\mathcal{P}^s$. Luego, por cada sustitución simbólica, se aplica dicha sustitución a las s.f.c.a. computadas anteriormente (obteniendo expresiones sin elementos simbólicos) y tras una serie de derivaciones interpretativas se obtienen las respuestas computadas difusas de cada objetivo.

La mejora de este algoritmo frente al anterior es que no hay que computar varias veces los objetivos de los casos de prueba desde el programa original; en su lugar, se realizan las derivaciones admisibles (y de fallo) una sola vez, y luego, por cada sustitución, solo hay que calcular las derivaciones interpretativas en cada s.f.c.a. Es decir, las derivaciones son más cortas.

Algoritmo 5.2: Calibrado simbólico de programas simbólicos

Entrada: un programa simbólico $\mathcal{P}^s$ y un número de casos de prueba esperados $(Q_i, \langle v_i; \theta_i \rangle)$, con $i \in \{1, \dots, k\}$

Salida: una sustitución simbólica Θ

```

1 considerar un número finito de posibles sustituciones simbólicas para  $\mathcal{Sym}(\mathcal{P}^s)$ ,
  denotadas por  $\Theta_1, \Theta_2, \dots, \Theta_n$ , con  $n > 0$ ;
2 para cada caso de prueba  $t_i$ , con  $i \in \{1, \dots, k\}$  hacer
3   | calcular la s.f.c.a.  $\langle Q'_i; \theta_i \rangle$  de  $\langle Q_i; id \rangle$  en  $\mathcal{P}^s$ ;
4 fin
5 para cada sustitución simbólica  $\Theta_j$ , con  $j \in \{1, \dots, n\}$  hacer
6   | para cada caso de prueba  $t_i$ , con  $i \in \{1, \dots, k\}$  hacer
7     | computar  $\langle Q'_i \Theta_j; \theta_i \rangle \rightarrow_{IS}^* \langle v_{i,j}; \theta_i \rangle$ ;
8     | calcular  $d_{i,j} = d(v_{i,j}, v_i)$ ;
9   | fin
10 fin
11 devolver  $\Theta_j$  que minimiza  $\sum_{i=1}^k d_{i,j}$ 

```

Ejemplo 5.5. Calibremos el programa del Ejemplo 5.3 por el método simbólico. Para ello, primero debemos ejecutar el objetivo de los casos de prueba sobre el programa simbólico original, obteniendo las siguientes s.f.c.a.

$$\begin{aligned}
Q'_1 &: \langle \&^{s1}(0.9, \&^{s1}(v^{s3}, @^{s2}(v^{s3}, 0.7))) \rangle; \{\} \rangle \\
Q'_2 &: \langle \&^{s1}(0.9, \&^{s1}(0.5, @^{s2}(0.3, 1))) \rangle; \{\} \rangle \\
Q'_3 &: \langle \&^{s1}(0.9, \&^{s1}(0.9, @^{s2}(0.8, 0.2))) \rangle; \{\} \rangle
\end{aligned}$$

A partir de estas s.f.c.a. aplicamos las sustituciones simbólicas Θ_i a Q'_1 , Q'_2 y Q'_3 , con $i \in \{1, \dots, 18\}$, y computamos las derivaciones interpretativas necesarias obteniendo los mismos resultados de la tabla 5.1. Nótese que hemos tenido que calcular el mismo número de respuestas computadas difusas que en el método básico, pero las derivaciones son más cortas.

5.3.3. Calibrado umbralizado

El método *umbralizado* de calibrado es una modificación del método anterior, que descarta una sustitución simbólica cuando la suma de los errores de las respuestas computadas difusas calculadas hasta el momento supera el error de la mejor sustitución simbólica encontrada.

Por lo tanto, este algoritmo suele calcular menos respuestas computadas difusas que los algoritmos anteriores, aunque la poda depende en gran medida del orden en que se comprueban las sustituciones simbólicas.

Algoritmo 5.3: Calibrado umbralizado de programas simbólicos

Entrada: un programa simbólico $\mathcal{P}^s$ y un número de casos de prueba esperados $(Q_i, \langle v_i; \theta_i \rangle)$, con $i \in \{1, \dots, k\}$

Salida: una sustitución simbólica Θ

```

1  considerar un número finito de posibles sustituciones simbólicas para  $\mathcal{Sym}(\mathcal{P}^s)$ ,
   denotadas por  $\Theta_1, \Theta_2, \dots, \Theta_n$ , con  $n > 0$ ;
2  establecer  $\tau = +\infty$ ;
3  para cada caso de prueba  $t_i$ , con  $i \in \{1, \dots, k\}$  hacer
4  |   calcular la s.f.c.a.  $\langle Q'_i; \theta_i \rangle$  de  $\langle Q_i; id \rangle$  en  $\mathcal{P}^s$ ;
5  fin
6  para cada sustitución simbólica  $\Theta_j$ , con  $j \in \{1, \dots, n\}$  hacer
7  |   establecer  $z = 0$ ;
8  |   para cada caso de prueba  $t_i$ , con  $i \in \{1, \dots, k\}$  hacer
9  |   |   si  $z \geq \tau$  entonces
10 |   |   |   parar;
11 |   |   fin
12 |   |   computar  $\langle Q'_i \Theta_j; \theta_i \rangle \rightarrow_{IS}^* \langle v_{i,j}; \theta_i \rangle$ ;
13 |   |   actualizar  $z = z + d(v_{i,j}, v_i)$ ;
14 |   fin
15 |   si  $z < \tau$  entonces
16 |   |   actualizar  $\tau = z$ ;
17 |   |   actualizar  $\Theta_\tau = \Theta_j$ ;
18 |   fin
19 fin
20 devolver  $\Theta_\tau$ 

```

Ejemplo 5.6. Calibremos el programa del Ejemplo 5.3 por el método umbralizado. El procedimiento es similar al del ejemplo anterior, pero ahora descartaremos algunas soluciones antes de computar las f.c.a. para todas las sustituciones simbólicas. En la Tabla 5.2 se muestran remarcadas con un asterisco (*) las f.c.a. podadas.

$\leftarrow^{s1}, \&^{s1}$	@ ^{s2}	v^{s3}	Θ	sun		sweet		lux		z
$\leftarrow_{godel},$ $\&_{godel}$	@ _{aver}	0.3	Θ_1	0.3	0.5	0.5	0	0.5	0.15	0.65
		0.5	Θ_2	0.5	0.3	0.5	0	0.5	0.15	0.45
		0.7	Θ_3	0.7	0.1	0.5	0	0.5	0.15	0.25
	@ _{geom}	0.3	Θ_4	0.3	0.5	0.5*	0*	0.4*	0.05*	0.55*
		0.5	Θ_5	0.5	0.3	0.5*	0*	0.4*	0.05*	0.35*
		0.7	Θ_6	0.7	0.1	0.5	0	0.4	0.05	0.15
$\leftarrow_{luka},$ $\&_{luka}$	@ _{aver}	0.3	Θ_7	0	0.8	0.05*	0.45*	0.3*	0.05*	1.3*
		0.5	Θ_8	0	0.8	0.05*	0.45*	0.3*	0.05*	1.3*
		0.7	Θ_9	0.3	0.5	0.05*	0.45*	0.3*	0.05*	1*
	@ _{geom}	0.3	Θ_{10}	0	0.8	0*	0.5*	0.2*	0.15*	1.45*
		0.5	Θ_{11}	0	0.8	0*	0.5*	0.2*	0.15*	1.45*
		0.7	Θ_{12}	0.3	0.5	0*	0.5*	0.2*	0.15*	1.15*
$\leftarrow_{prod},$ $\&_{prod}$	@ _{aver}	0.3	Θ_{13}	0.13	0.67	0.29*	0.21*	0.4*	0.05*	0.93*
		0.5	Θ_{14}	0.27	0.53	0.29*	0.21*	0.4*	0.05*	0.79*
		0.7	Θ_{15}	0.44	0.36	0.29*	0.21*	0.4*	0.05*	0.62*
	@ _{geom}	0.3	Θ_{16}	0.12	0.68	0.24*	0.26*	0.32*	0.03*	0.97*
		0.5	Θ_{17}	0.26	0.54	0.24*	0.26*	0.32*	0.03*	0.83*
		0.7	Θ_{18}	0.44	0.36	0.24*	0.26*	0.32*	0.03*	0.65*

Tabla 5.2: Resumen de pesos al calibrar el programa del Ejemplo 5.3 con poda

Como se observa en la tabla, en este caso se han podado más de la mitad de las respuestas computadas difusas de las que habría que calcular en el método simbólico. En comparación con el método básico, este método encuentra la mejor sustitución calculando menos f.c.a. y realizando derivaciones más cortas.

5.4. Resultados experimentales

Como veremos en la siguiente sección, el entorno FLOPER ha sido adaptado para calibrar programas simbólicos mediante los tres algoritmos vistos anteriormente, devolviendo además el tiempo de ejecución de cada método.

5.4.1. Sustituciones simbólicas

En la Tabla 5.3 se resumen los resultados experimentales⁴ del tiempo de ejecución necesario para calibrar el programa del Ejemplo 4.6 mediante los tres algoritmos, variando el número de elementos simbólicos del programa, así como el número de grados de verdad considerados en el retículo ($[0, 1], \leq$). El número de casos de prueba ha sido fijado a 3 en todas las ejecuciones.

	Grados de verdad			Elementos simbólicos						
	10	100	1000	5	6	7	8	9	10	11
Básico	120	1130	11360	320	990	3030	9180	28170	86760	264850
Simbólico	30	290	2860	100	290	980	2970	9930	30570	93360
Umbralizado	15	130	1300	50	140	420	1580	4390	13460	38310

Tabla 5.3: Tiempo de ejecución del calibrado (en milisegundos)

En la primera columna, consideramos tres elementos simbólicos variando el número de grados de verdad del retículo. Nótese que en el caso más complejo, considerando 9000 sustituciones simbólicas⁵ diferentes, el método umbralizado es hasta 3 veces más eficiente que el método simbólico, y hasta 8 veces más eficiente que el método básico, mostrando las ventajas de nuestro mecanismo de calibrado mejorado.

En la última columna, consideramos variaciones del número de elementos simbólicos (entre conectivos y grados de verdad) de 5 a 11 fijando a 3 el número de grados de verdad en el retículo, mostrando así que el método umbralizado se ajusta bien y resuelve el problema en tan solo unos segundos.

5.4.2. Casos de prueba

Para medir la escalabilidad de los algoritmos de calibrado variando el número de casos de prueba, utilizaremos el siguiente ejemplo, que muestra además el uso de las técnicas de calibrado como herramienta para la clasificación automática en el campo de la minería de datos.

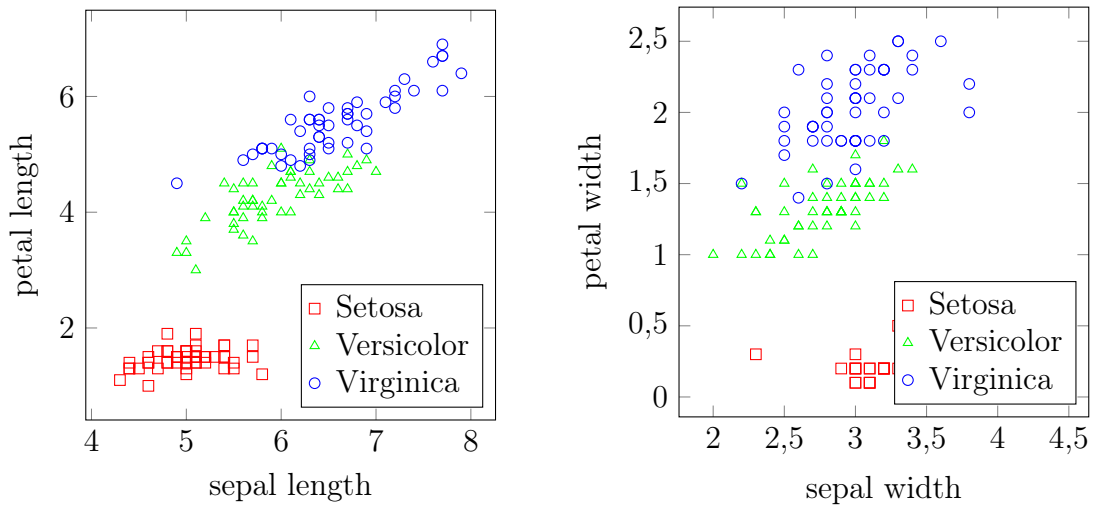
⁴Cada celda contiene la media tras 100 ejecuciones utilizando un ordenador de sobremesa equipado con un i3-2310M CPU @ 2.10 GHz y 4,00 GB RAM.

⁵El programa contiene tres elementos simbólicos: una conjunción, una disyunción y un grado de verdad. La conjunción y la disyunción simbólicas pueden tomar 3 valores distintos cada una, y el grado de verdad simbólico puede tomar 1000 valores del retículo, por lo tanto tenemos que comprobar $3 \cdot 3 \cdot 1000 = 9000$ sustituciones simbólicas.

Ejemplo 5.7. El conjunto de datos *Iris* [3] contiene cuatro medidas para 150 flores que representan 3 especies de flores iris: *iris setosa*, *iris versicolor* e *iris virginica*; contando con 50 registros de cada una de las especies. Los atributos son los siguientes (todos ellos medidos en centímetros).

- `sepal_length`: longitud del sépalo.
- `sepal_width`: ancho del sépalo.
- `petal_length`: longitud del pétalo.
- `petal_width`: ancho del pétalo.

El objetivo es crear un programa que nos ayude a clasificar la especie de nuevas flores iris en base a estos cuatro atributos, devolviendo el grado de verdad con el que la flor pertenece a cada especie. Mediante los siguientes diagramas de dispersión⁶ es fácil ver que hay una clara separación entre las especies:



Aunque podemos clasificar perfectamente las flores de la especie setosa, el límite no es tan claro para las flores de las especies versicolor y virginica. Por lo tanto, crearemos el programa simbólico $\mathcal{P}^s$ dando pesos simbólicos a las reglas, y dejaremos que el algoritmo de calibrado elija los mejores umbrales para minimizar el error en el conjunto de datos del que disponemos.⁷

$$\Pi^s = \begin{cases} R_1 : \text{iris}(_, _, \text{PL}, _, \text{setosa}) & \leftarrow \text{PL} \leq 2.5. \\ R_2 : \text{iris}(_, _, \text{PL}, _, \text{versicolor}) & \leftarrow \text{PL} > 2.5 \ \& \ ((\text{PL} < 4.8 \ \& \ v^{s1}) \\ & \quad |_{\text{godel}} (\text{PL} < 5.2 \ \& \ v^{s2})). \\ R_3 : \text{iris}(_, _, \text{PL}, _, \text{virginica}) & \leftarrow \text{PL} > 2.5 \ \& \ ((\text{PL} \geq 4.8 \ \& \ v^{s1}) \\ & \quad |_{\text{godel}} (\text{PL} \geq 5.2 \ \& \ v^{s2})). \end{cases}$$

⁶Un diagrama de dispersión utiliza coordenadas cartesianas para mostrar los valores de dos variables para un conjunto de datos.

⁷Por simplicidad, consideramos únicamente la variable `petal_length` para la clasificación.

Utilizaremos el conjunto de datos para especificar los casos de prueba. Por ejemplo, si tenemos un registro clasificado como **versicolor** cuya longitud del sépalo, ancho del sépalo, longitud del pétalo y ancho del pétalo es, respectivamente, 7, 3.2, 4.7 y 1.4, insertaremos los siguientes casos de prueba.

```
1 → iris(7, 3.2, 4.7, 1.4, versicolor).
0 → iris(7, 3.2, 4.7, 1.4, virginica).
```

En el Anexo A se muestran los 100 registros de flores versicolor y virginica del conjunto de datos *Iris* componiendo 200 casos de prueba. Para calibrar este programa consideraremos únicamente dos valores del retículo: 0 y 1.

Tras calibrar este programa con 2 constantes simbólicas y 200 casos de prueba, el método básico tarda 400 milisegundos, el método simbólico 250 milisegundos, y el método umbralizado 200 milisegundos, corroborando la eficacia de las técnicas de calibrado incluso con un gran número de casos de prueba.

En este caso, el calibrado nos devuelve la sustitución simbólica $\Theta = \{v^{s1}/1, v^{s2}/0\}$ con un error de 14. Es decir, 4.8 es mejor umbral que 5.2 para la variable **petal_length**, clasificando mal únicamente 7 de las 100 muestras del conjunto de datos.

5.5. Implementación del calibrado

5.5.1. Análisis de los casos de prueba

Para calibrar un programa, es necesario introducir en el entorno un conjunto de casos de prueba. Para almacenar estos casos de prueba, introducimos el predicado dinámico **testcase/2**, que por reevaluación devolverá todos los casos de prueba cargados en el entorno, siendo el primer argumento el grado de verdad esperado y el segundo argumento el objetivo.

Nuevamente, haremos uso de las gramáticas de cláusulas definidas para analizar léxica y sintácticamente el conjunto de casos de prueba. Un caso de prueba está formado por un grado de verdad, seguido del símbolo “ $\rightarrow$ ”, seguido del objetivo, y terminando en un punto “.”. Por lo tanto las reglas de la gramática son las siguientes:

```
1 dcg_testcases(_, [], []).
2 dcg_testcases([testcase(T,B)|Ts]) --> dcg_testcase(T,B),
3                                     dcg_testcases(Ts).
4 dcg_testcase(T,B) --> body(T), ['-', '>'], body(B), ['.', ''].
```

Las dos primeras reglas componen y devuelven la lista de casos de prueba, y la tercera regla es la que analiza y devuelve cada uno de los casos de prueba.

Para cargar un nuevo fichero de casos de prueba en el entorno, se ha creado el predicado `parse_dcg_testcases/1`, que recibe la ruta del fichero, elimina todas las cláusulas anteriores de `testcase/2`, llama al proceso de análisis, escribe en un fichero temporal los casos de prueba en formato Prolog, y finalmente consulta dicho fichero. Para escribir el fichero hace uso del predicado `write_dcg_testcases/1` que recibe una lista de casos de prueba y los escribe en la salida estándar.

Antes de poder utilizar los casos de prueba para calibrar un programa, debemos comprobar que están bien formados, esto es, que en la parte izquierda tenemos un grado de verdad del retículo, y en la parte derecha un objetivo válido sin elementos simbólicos, ya que para calibrar un programa simbólico necesitamos casos de prueba cuyos objetivos nos devuelvan respuestas computadas difusas.

Dado que en el proceso de análisis léxico no nos hemos preocupado de comprobar estas restricciones, se introduce el predicado `check_testcases/0`, que comprueba todos los casos de prueba del predicado `testcase/2`.

```

1 check_testcases :-
2     testcase(T,B),
3     ( T = td(_),
4         all_instantiated(B)
5     ; !, fail).

```

El predicado `all_instantiated/1` (ver Anexo C) recibe el objetivo de cada uno de los casos de prueba, y comprueba que no haya ningún elemento simbólico, es decir, será verdad cuando el objetivo no contenga ningún átomo `sym/1`. Si encuentra algún grado de verdad o conector simbólico, muestra un mensaje de error y el proceso de calibrado falla.

5.5.2. Generación de sustituciones simbólicas

Para generar todas las posibles sustituciones simbólicas necesarias para calibrar un programa, es necesario obtener primero todos los símbolos contenidos en las reglas del mismo. Incorporamos otro predicado dinámico, `tuning_sym/3`, que almacena para cada símbolo: el tipo de elemento simbólico (`td`, `and`, `con`, etc), su aridad y su etiqueta o identificador simbólico.

El predicado `update_rule_symbols/0` (ver Anexo C) busca todos los elementos simbólicos del programa activo, y los almacena como cláusulas de `tuning_sym/3`.

El predicado `tuning_lat/4` (ver Anexo C) proporciona, por reevaluación, todos los elementos del retículo, es decir, todos los grados de verdad considerados para el calibrado, y todos los conectivos. El primer argumento es el tipo de elemento simbólico al que responde, el segundo argumento es su aridad, el tercer argumento es el tipo de conectivo, y el cuarto argumento es el nombre del conectivo (`godel`, `very`, etc).

Por lo general, el primer y tercer argumento serán idénticos, excepto cuando el primer argumento es un conectivo genérico (`con`), en cuyo caso el tercer argumento puede ser una conjunción (`and`), una disyunción (`or`) o un agregador (`agr`).

El predicado `tuning_com/2` recibe la lista de elementos simbólicos obtenida mediante `update_rule_symbols/0`, y genera todas las posibles sustituciones: el producto cartesiano de los conjuntos de elementos obtenidos por `tuning_lat/4` para cada elemento simbólico de la lista.

```

1 tuning_combinations([], []).
2 tuning_combinations([sym(C,L,S)|Xs],
3   [pair(sym(C,L,S),val(C2,L,P))|Ys]) :-
4   tuning_lat(C,L,C2,P),
5   tuning_combinations(Xs,Ys).
```

Como se observa en la definición, cada sustitución se representa mediante un átomo de la forma `pair(sym(C,L,S),val(C2,L,P))`, donde el primer elemento es el símbolo y el segundo elemento es el valor por el que será sustituido, donde `C` y `C2` representan el tipo de elemento, `L` es la aridad, y `S` y `P` son la etiqueta y el nombre del conectivo (o el grado de verdad), respectivamente.

5.5.3. Algoritmos de calibrado

Para la implementación de los distintos métodos de calibrado, se introducen dos nuevos predicados dinámicos en el entorno FLOPER. El predicado `tuning_tau/1` almacenará el error de la mejor sustitución encontrada durante el proceso de calibrado. Asimismo, el predicado `tuning_sub/1` contendrá la mejor sustitución.

Al iniciarse un proceso de calibrado, todas las cláusulas de estos predicados son eliminadas de la base de datos, y se inserta el hecho `tuning_tau(inf)`, expresando que la mejor sustitución actual tiene un error de $+\infty$. Además, también hay que comprobar los casos de prueba mediante el predicado `check_testcases/0` y actualizar los elementos simbólicos mediante el predicado `update_rule_symbols/0`.

Al terminar el proceso de calibrado, el predicado `write_symbolic_sub/0` (ver Anexo C) se encarga de imprimir por la salida estándar la mejor sustitución encontrada.

Calibrado básico

El método básico de calibrado se implementa en el entorno FLOPER mediante el predicado `tuning_basic/0`. Para cada sustitución simbólica, se eliminan todas las reglas del programa activo, se aplica la sustitución simbólica a las reglas originales, se insertan dichas reglas, y se ejecutan los objetivos de cada caso de prueba (obteniendo la f.c.a. mediante el predicado `tuning_leaves/1`).

```

1 tuning_basic :-
2     check_testcases ,
3     retractall(tuning_tau(_)),
4     retractall(tuning_sub(_)),
5     assertz(tuning_tau(inf)),
6     once(update_rule_symbols),
7     findall(X,rule(_,X),Rules),
8     findall(testcase(A,B),testcase(A,B),TestCases),
9     findall(sym(C,L,S),tuning_sym(C,L,S),Sym),
10    tuning_basic_do(Rules,TestCases,Sym),
11    retractall(rule(_,_,_,_,_)),
12    tuning_basic_reassert(Rules),
13    write_symbolic_sub.

```

El predicado `tuning_basic_do/3` recibe una lista con las reglas originales, una lista con los casos de prueba y una lista con los símbolos del programa, y en cada iteración elimina las reglas e inserta las nuevas reglas con cada sustitución simbólica.

```

1 tuning_basic_do(Rules,TestCases,Sym) :-
2     ( tuning_combinations(Sym,C),
3       retractall(rule(_,_,_,_,_)),
4       tuning_basic_assert(Rules,C),
5       once(tuning_basic_tests(TestCases,C,0)),
6       fail
7     ; true).

```

El predicado `tuning_basic_assert/2` (ver Anexo C) recibe la lista de reglas y una sustitución simbólica, y se encarga de aplicar la sustitución a las reglas e insertarlas.

El predicado `tuning_basic_tests/3` recibe la lista con los casos de prueba, una sustitución simbólica, y el error cometido hasta el momento por dicha sustitución (inicialmente 0); e itera los casos de prueba para ir calculando el error de cada objetivo.

Cuando la lista de casos de prueba queda vacía, se comprueba si el error cometido es menor que la mejor sustitución encontrada hasta el momento, y se actualiza la sustitución si corresponde.

```

1 tuning_basic_tests([],Sub,N) :-
2     tuning_tau(Tau),
3     (tuning_tau(inf) ; N < Tau),
4     retractall(tuning_tau(_)),
5     retractall(tuning_sub(_)),
6     assertz(tuning_tau(N)),
7     assertz(tuning_sub(Sub)), !.
8
9 tuning_basic_tests([],_,_).
10
11 tuning_basic_tests([testcase(td(T),B)|Xs],Sub,N) :-
12     retractall(leaf(_)),
13     once(tuning_leaves(B)),
14     leaf(td(TD)),
15     lat:distance(TD,T,M),
16     N2 is N+M,
17     tuning_basic_tests(Xs,Sub,N2).

```

Calibrado simbólico

El método de calibrado simbólico se implementa en el entorno FLOPER mediante el predicado `tuning_symbolic/0`. Este predicado ejecuta el objetivo de cada caso de prueba obteniendo su respuesta computada difusa simbólica, y, mediante el predicado `tuning_symbolic_try/3`, se aplica cada sustitución simbólica a las s.f.c.a y devuelve la mejor sustitución simbólica.

```

1 tuning_symbolic :-
2     check_testcases,
3     retractall(leaf(_)),
4     retractall(tuning_tau(_)),
5     retractall(tuning_sub(_)),
6     assertz(tuning_tau(inf)),
7     ( testcase(T,B),
8         once(tuning_leaves(B)),
9         fail
10    ; !, true),
11     once(update_rule_symbols),
12     findall(sym(C,L,S),tuning_sym(C,L,S),Sym),

```

```

13     findall(Leaf, leaf(Leaf), Tests),
14     findall(TD, testcase(TD, _), Expected),
15     tuning_symbolic_try(Tests, Expected, Sym),
16     write_symbolic_sub.

```

El predicado `tuning_symbolic_try/3` recibe una lista con las s.f.c.a., una lista con los valores esperados (extraídos de los casos de prueba) y una lista con los símbolos del programa; y para cada sustitución calcula el error respecto a los casos de prueba mediante `tuning_symbolic_try/4`.

```

1 tuning_symbolic_try(Ts, Es, Sym) :-
2     ( tuning_combinations(Sym, C),
3       once(tuning_symbolic_try(Ts, Es, C, 0)),
4       fail
5       ; true).

```

El predicado `tuning_symbolic_try/4` recibe una lista con las s.f.c.a., una lista con los valores esperados para cada objetivo, una sustitución simbólica y el error cometido hasta el momento por dicha sustitución (inicialmente 0). Por cada s.f.c.a., aplica la sustitución y calcula la f.c.a., actualizando el error. Cuando la lista de casos de prueba queda vacía, se comprueba si el error cometido es menor que la mejor sustitución encontrada hasta el momento, y se actualiza la sustitución si corresponde.

```

1 tuning_symbolic_try([], _, Sub, N) :-
2     tuning_tau(Tau),
3     (Tau = inf ; N < Tau),
4     retractall(tuning_tau(_)),
5     retractall(tuning_sub(_)),
6     assertz(tuning_tau(N)),
7     assertz(tuning_sub(Sub)).
8
9 tuning_symbolic_try([T|Ts], [td(E)|Es], Sub, N) :-
10    tuning_apply(T, Sub, T2),
11    retractall(leaf(_)),
12    once(tuning_leaves(T2)),
13    leaf(td(TD)),
14    lat:distance(TD, E, M),
15    N2 is N+M, !,
16    tuning_symbolic_try(Ts, Es, Sub, N2).
17
18 tuning_symbolic_try(_, _, _, _) :- !.

```

Calibrado umbralizado

El método de calibrado umbralizado se implementa en el entorno FLOPER mediante el predicado `tuning_thresholded/0`. Este predicado es idéntico al predicado `tuning_symbolic/0`, añadiendo una poda en el método `tuning_thresholded_try/4`, esto es, cada vez que se calcula una nueva f.c.a., se comprueba si el error sigue siendo menor al error de la mejor sustitución.

```

1 tuning_thresholded_try([],[],Sub,N) :- !,
2     tuning_tau(Tau),
3     (Tau = inf ; N < Tau),
4     retractall(tuning_tau(_)),
5     retractall(tuning_sub(_)),
6     assertz(tuning_tau(N)),
7     assertz(tuning_sub(Sub)).
8
9 tuning_thresholded_try([T|Ts],[td(E)|Es],Sub,N) :- !,
10    tuning_tau(Tau),
11    (Tau = inf ; N < Tau),
12    tuning_apply(T,Sub,T2),
13    retractall(leaf(_)),
14    once(tuning_leaves(T2)),
15    leaf(td(TD)),
16    lat:distance(TD,E,M),
17    N2 is N+M,
18    tuning_thresholded_try(Ts,Es,Sub,N2).
19
20 tuning_thresholded_try(_,_,_,-) :- !.

```

5.5.4. Herramienta online para el calibrado

Para equipar el entorno FLOPER online con las entradas necesarias para calibrar programas simbólicos, se ha modificado la interfaz utilizando un patrón de diseño de pestañas, separando la entrada y ejecución de objetivos, de la entrada de casos de prueba y calibrado. En la Figura 5.1 y en la Figura 5.2 se muestran, respectivamente, el contenido de la pestaña de ejecución y el contenido de la pestaña de calibrado.

Para calibrar un programa simbólico es necesario ejecutar objetivos. Por lo tanto, también hay que establecer la profundidad hasta la que se buscarán las respuestas, así como el modo interpretativo (largo, medio o corto). Para evitar sobrecargar la interfaz, los valores para estos parámetros se tomarán de la pestaña de ejecución.

Run Tune

Goal Tree depth Mode Answers & Tree

popularity(X) 12 S M L Generate

Figura 5.1: Pestaña de ejecución en el entorno FLOPER online

Run Tune

Test cases

0.6 -> popularity(X). symbolic.test

Apply substitution Generate substitution

Figura 5.2: Pestaña de calibrado en el entorno FLOPER online

Cuando se han introducido los casos de prueba en el campo de texto y se pulsa el botón **Generate substitution** para buscar la mejor sustitución simbólica, la primera acción del entorno es tratar de unificar las variables libres de todas las formas posibles, transformando los casos de prueba, tal y como se muestra en el siguiente ejemplo.

Ejemplo 5.8. Sea $\mathcal{P}$ el programa sMALP del Ejemplo 4.6. Al introducir el caso de prueba $0.6 \rightarrow \text{popularity}(X).$ en el entorno FLOPER, este es reemplazado por los siguientes casos de prueba

$$\begin{aligned} 0.6 &\rightarrow \text{popularity}(\text{sun}). \\ 0.6 &\rightarrow \text{popularity}(\text{sweet}). \\ 0.6 &\rightarrow \text{popularity}(\text{lux}). \end{aligned}$$

dado que el objetivo $\text{popularity}(X)$ unifica con la regla R_1 para las sustituciones $\{X/\text{sun}\}$, $\{X/\text{sweet}\}$ y $\{X/\text{lux}\}$.

Una vez sustituidos los casos de prueba, se puede reajustar el grado de verdad esperado de cada uno ellos, y ejecutar nuevamente el proceso de calibrado. En la Figura 5.3 se muestran los casos de prueba tras ser reemplazados.

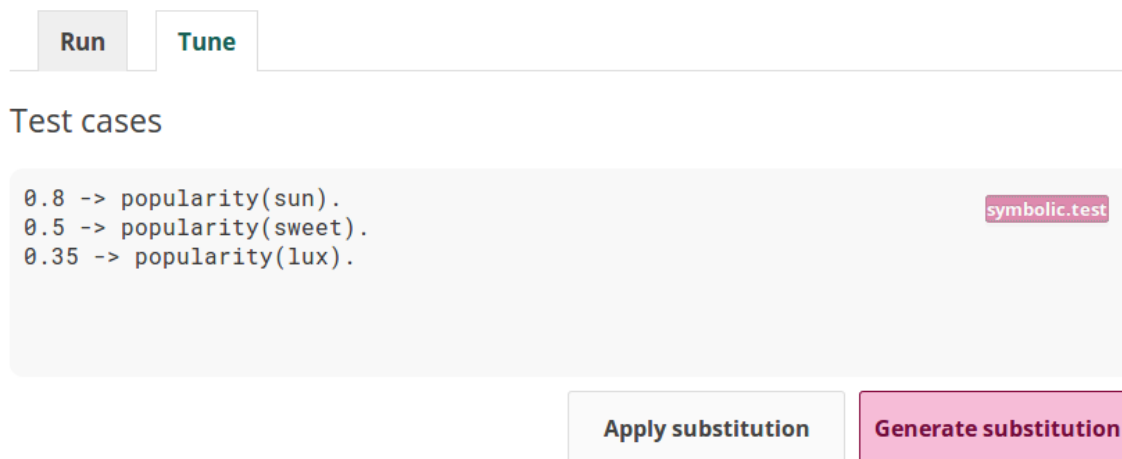


Figura 5.3: Casos de prueba en el entorno FLOPER online

Obsérvese que los algoritmos de calibrado pueden trabajar perfectamente con objetivos incluyendo variables, pero este mecanismo ofrece mayor expresividad al usuario.

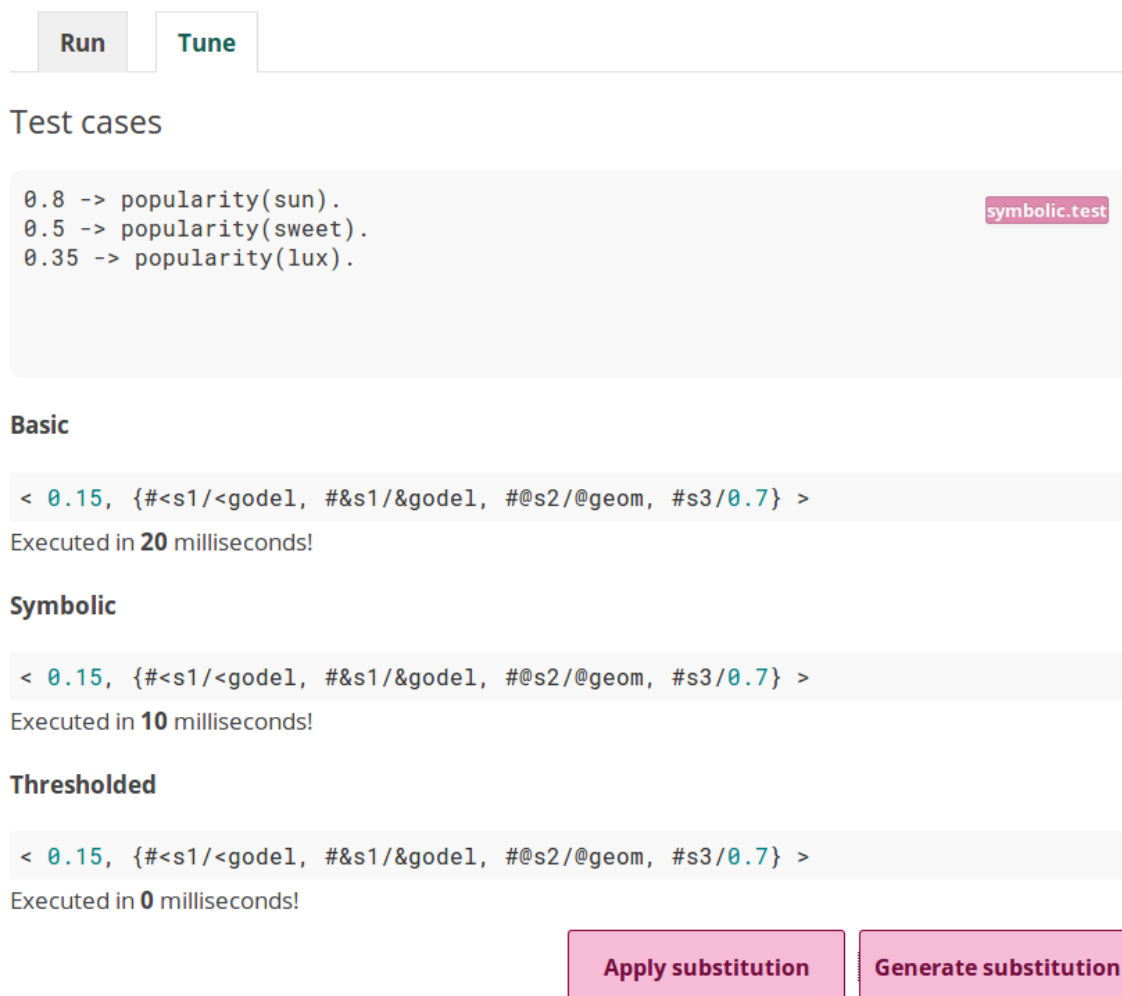


Figura 5.4: Proceso de calibrado en el entorno FLOPER online

Tras comprobar los casos de prueba, el entorno ejecuta los tres algoritmos de calibrado vistos en la Sección 5.3, y muestra la mejor sustitución simbólica y el error cometido frente al conjunto de casos de prueba, así como el tiempo de ejecución de cada uno de los métodos de búsqueda. En la Figura 5.4 se muestra la mejor sustitución simbólica para el calibrado del Ejemplo 5.3.

Program

```
popularity(X) <- (0.9 &godel (facilities(X) &godel @geom(location(X),
rates(X)))).
facilities(sun) <- 0.7.
location(sun) <- 0.7.
rates(sun) <- 0.7.
facilities(sweet) <- 0.5.
location(sweet) <- 0.3.
rates(sweet).
facilities(lux) <- 0.9.
location(lux) <- 0.8.
rates(lux) <- 0.2.
```

[+] Unfold program (0) | « Previous | Next »

Figura 5.5: Aplicación de una sustitución simbólica en el entorno FLOPER online

Una vez generada una sustitución simbólica, el entorno activa el botón **Apply substitution**, que aplica la sustitución simbólica encontrada al programa introducido en el campo de texto del programa. En la Figura 5.5 se muestra el programa del Ejemplo 4.6 tras aplicar la sustitución simbólica mostrada en la Figura 5.4. Nótese que, aunque el programa original estuviera escrito con sintaxis sMALP, el programa siempre es sobrescrito con sintaxis FASILL.

Run

Tune

Test cases

```
0.8 -> popularity(sun).
0.5 -> popularity(sweet).
0.35 -> popularity(lux)
```

symbolic.test

There was an error.

Apply substitution

Generate substitution

Figura 5.6: Casos de prueba erróneos en el entorno FLOPER online

Si se produce algún error a la hora de analizar el conjunto de casos de prueba, como se ilustra en la Figura 5.6 (donde el último caso de prueba no termina en punto “.”), el entorno muestra un mensaje de error y se detiene el proceso de calibrado.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Este Trabajo Fin de Grado se ha centrado en la ampliación de las capacidades de FLOPER, un entorno de programación lógica difusa desarrollado por el grupo de investigación DEC-TAU de la Universidad de Castilla-La Mancha. En su estado inmediatamente anterior, el entorno era capaz de analizar y evaluar objetivos difusos sobre programas MALP y FASILL. Este trabajo ha ampliado las herramientas disponibles en el entorno referido para transformar programas lógicos difusos, incorporando mecanismos de desplegado y calibrado automático, e incluyendo extensiones simbólicas de los lenguajes difusos MALP y FASILL.

Se detallan los formalismos subyacentes de la lógica difusa y la programación lógica, así como de las técnicas de desplegado y calibrado simbólico implementadas, indicando las capacidades y limitaciones del entorno FLOPER al inicio del trabajo. Al tratarse de un trabajo realizado sobre una implementación previa, ha sido necesario un esfuerzo inicial muy notorio para comprender el código del sistema en el estado en que se encontraba.

Como el área en la que se ubica este trabajo, la transformación automática de programas lógicos difusos, es de reciente consolidación, los fundamentos teóricos necesarios para las tareas de implementación abordadas, han sido proporcionados por los directores del trabajo. Con este Trabajo Fin de Grado hemos contribuido al desarrollo de los conceptos teóricos referidos, a la vez que a su implementación, en los términos que detallamos posteriormente; todo ello, ha dado lugar a las aportaciones [21, 22, 23, 24].

Las contribuciones esenciales aportadas al entorno FLOPER se concretan en los siguientes resultados:

1. Implementación del desplegado difuso para programas MALP y FASILL, y su integración en la herramienta online.
2. Implementación de los mecanismos necesarios para gestionar diversas versiones de los programas simultáneamente.
3. Definición del formato de los casos de prueba y de la sintaxis para los grados de verdad y conectivos simbólicos.
4. Actualización de la implementación del proceso de análisis léxico y sintáctico para la interpretación de las extensiones simbólicas sMALP y sFASILL.
5. Actualización de la implementación de la semántica operacional de MALP y FASILL para trabajar con sus extensiones simbólicas.
6. Implementación del proceso de análisis léxico y sintáctico para la interpretación de los casos de prueba.
7. Implementación de los tres métodos de calibrado para programas sMALP y sFASILL, y su integración en la herramienta online.
8. Medición y comparación del tiempo de ejecución de los diversos métodos de calibrado, variando el número de elementos simbólicos considerados en el retículo, y el número de casos de prueba.

6.2. Trabajo futuro

En primer lugar, como hemos visto en la Sección 3.2, la transformación de desplegado difuso para programas FASILL no siempre preserva la semántica de los programas originales. Todavía no hemos establecido condiciones generales para las cuales podamos garantizar la corrección y la completitud de dichas transformaciones, siendo este el siguiente objetivo en la investigación sobre el desplegado.

Las técnicas de desplegado son la base para otros sistemas de transformaciones más poderosos, basados en las sucesivas aplicaciones de pasos de plegado/desplegado sobre las reglas de un programa. Las transformaciones de plegado se basan en la idea contraria a las de desplegado. Mediante el uso de heurísticas, podemos combinar estas transformaciones para optimizar programas automáticamente.

Por otra parte, la extensión simbólica del lenguaje FASILL actualmente solo permite introducir grados de verdad y conectivos simbólicos en las reglas del programa. En el futuro, nos proponemos permitir que las relaciones de similaridad acepten también

elementos simbólicos, esto es, valores simbólicos en las relaciones de los términos y predicados, así como una t-norma simbólica.

Para llevar a cabo esta ampliación de las capacidades del calibrado simbólico, además de tener que modificar el proceso de análisis de las ecuaciones de similaridad, también es necesario modificar el proceso de unificación débil de FASILL. Esta mejora podría suponer un gran impacto en el calibrado de programas sFASILL, debido a que un elemento simbólico en una relación de similaridad (por ejemplo en la t-norma asociada) puede afectar a muchas más reglas de un programa que un elemento simbólico en una sola regla.

En este trabajo nos hemos centrado, asimismo, en la integración de estas técnicas sobre el entorno FLOPER online, pero, debido al volumen de tareas abordadas, no ha sido posible integrarlas también en el nuevo entorno gráfico mostrado en la Sección [2.6.3](#). Por tanto, en el futuro nos proponemos integrar estos mecanismos en el nuevo editor implementado en SWI-Prolog, ampliando su interfaz con el paquete gráfico XPCE.

Bibliografía

- [1] M. Alpuente, M. Falaschi, G. Moreno, y G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Elsevier, TCS*, 311(1-3):479–525, 2004. [43](#)
- [2] R. M. Burstall y J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977. [43](#)
- [3] R. A. Fisher. The species problem in Iris. *Annals of the Missouri Botanical Garden*, 23(3):457–509, 1936. [81](#)
- [4] J. A. Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1969. [22](#)
- [5] F. Herrera, E. Herrera-Viedma, y J. L. Verdegay. Direct approach processes in group decision making using linguistic OWA operators. *Fuzzy Sets and Systems*, 79(2):175–190, 1996. [25](#)
- [6] M. Johnson. Two ways of formalizing grammars. *Linguistics and Philosophy*, 17(3):221–240, 1994. [21](#)
- [7] P. Julián. *Lógica Simbólica para Informáticos*. Ra-Ma, 2004. [5](#), [6](#), [8](#), [9](#)
- [8] P. Julián y M. Alpuente. *Programación lógica. Teoría y práctica*. Pearson, 2007. [5](#), [10](#), [11](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [19](#), [20](#)
- [9] P. Julián, G. Moreno, y J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems*, 154:16–33, 2005. [2](#), [44](#)
- [10] P. Julián, G. Moreno, y J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science*, 12(11):1679–1699, 2006. [2](#), [38](#), [44](#)
- [11] P. Julián, G. Moreno, y J. Penabad. On the declarative semantics of multi-adjoint logic programs. En *Proc. of 10th International Work-Conference on Artificial Neural Networks, IWANN’09*, páginas 253–260. Springer-Verlag, LNCS 5517, 2009. [35](#)

- [12] P. Julián, G. Moreno, J. Penabad, y C. Vázquez. A Fuzzy Logic Programming Environment for Managing Similarity and Truth Degrees. En *Post-Proc. of XIV Jornadas sobre Programación y Lenguajes, PROLE'14*, volumen 173, páginas 71–86, Cádiz, España, 2015. EPTCS. [27](#), [28](#), [33](#), [34](#), [35](#), [36](#), [37](#)
- [13] P. Julián y C. Rubio. A declarative semantics for bousi~prolog. En *Proc. of 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'09*, páginas 149–160, Coimbra, Portugal, 2009. [35](#)
- [14] A. Kolesárová y M. Komorníková. Triangular norm-based iterative compensatory operators. *Fuzzy Sets and Systems*, 104(1):109–120, 1999. [25](#)
- [15] R. A. Kowalski. Predicate Logic as a Programming Language. *Information Processings 74*, páginas 569–574, 1974. [9](#)
- [16] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979. [9](#)
- [17] R. A. Kowalski. *Logic for Problem Solving*. North-Holland, 1979. [9](#)
- [18] R. C. T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972. [23](#), [26](#)
- [19] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin. Segunda edición, 1987. [26](#)
- [20] J. Medina, M. Ojeda-Aciego, y P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004. [35](#)
- [21] G. Moreno, J. Penabad, y J. A. Ríaza. On Similarity-based Unfolding. En *Proc. of International Conference on Scalable Uncertainty Management, SUM'17*, páginas 1–7. (aceptado), 2017. [44](#), [46](#), [93](#)
- [22] G. Moreno, J. Penabad, y J. A. Ríaza. Symbolic Unfolding of Multi-adjoint Logic Programs. En *Proc. of 9th European Symposium on Computational Intelligence and Mathematics, ESCIM'17*, páginas 1–6. (sometido), 2017. [61](#), [93](#)
- [23] G. Moreno, J. Penabad, J. A. Ríaza, y G. Vidal. Symbolic Execution and Thresholding for Efficiently Tuning Fuzzy Logic Programs. En *Post-Proc. of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'16*, páginas 1–16. Springer, LNCS 10184, 2017. [71](#), [93](#)
- [24] G. Moreno y J. A. Ríaza. An Online Tool for Tuning Fuzzy Logic Programs. En *Proc. of International Joint Conference on Rules and Reasoning, RuleML+RR'17*, páginas 1–15. Springer, LNCS 10364 (aceptado), 2017. [57](#), [59](#), [60](#), [93](#)

- [25] H. T. Nguyen y E. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida. Tercera edición, 2006. [24](#)
- [26] J. Pavelka. On fuzzy logic I, II, III. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:45–52, 119–134, 447–464, 1979. [22](#)
- [27] J. Penabad. *Desplegado de Programas Lógicos Difusos*. Tesis doctoral, Universidad de Castilla-La Mancha, 2010. [2](#), [22](#), [25](#), [26](#), [27](#), [29](#), [30](#), [31](#), [32](#)
- [28] A. Pettorossi y M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996. [43](#)
- [29] B. Schweizer y A. Sklar. *Probabilistic Metric Spaces*. North-Holland, New York, 1983. [24](#)
- [30] M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Elsevier, TCS*, 275(1-2):389–426, 2002. [33](#), [34](#), [35](#), [36](#)
- [31] H. Tamaki y T. Sato. Unfold/Fold Transformations of Logic Programs. En *Proc. of Second International Conference on Logic Programming*, páginas 127–139, 1984. [43](#)
- [32] J. Wielemaker. Swi-Prolog. Disponible en: <http://www.swi-prolog.org/>, 2017. [20](#), [42](#)
- [33] M. S. Ying. A logic for approximated reasoning. *Journal of Symbolic Logic*, 59(3):830–837, 1994. [33](#)
- [34] L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428, 1965. [22](#)
- [35] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965. [22](#)
- [36] L. A. Zadeh. Nacimiento y evolución de la lógica borrosa, el soft computing y la computación con palabras: un punto de vista personal. *Psicothema*, 8(2):421–429, 1996. [22](#)

Apéndice A

Casos de prueba del conjunto de datos Iris

$1 \rightarrow \text{iris}(7,3.2,4.7,1.4,\text{versicolor}).$	$0 \rightarrow \text{iris}(7,3.2,4.7,1.4,\text{virginica}).$
$1 \rightarrow \text{iris}(6.4,3.2,4.5,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.4,3.2,4.5,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(6.9,3.1,4.9,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.9,3.1,4.9,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(5.5,2.3,4,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.5,2.3,4,1.3,\text{virginica}).$
$1 \rightarrow \text{iris}(6.5,2.8,4.6,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.5,2.8,4.6,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(5.7,2.8,4.5,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.7,2.8,4.5,1.3,\text{virginica}).$
$1 \rightarrow \text{iris}(6.3,3.3,4.7,1.6,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.3,3.3,4.7,1.6,\text{virginica}).$
$1 \rightarrow \text{iris}(4.9,2.4,3.3,1,\text{versicolor}).$	$0 \rightarrow \text{iris}(4.9,2.4,3.3,1,\text{virginica}).$
$1 \rightarrow \text{iris}(6.6,2.9,4.6,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.6,2.9,4.6,1.3,\text{virginica}).$
$1 \rightarrow \text{iris}(5.2,2.7,3.9,1.4,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.2,2.7,3.9,1.4,\text{virginica}).$
$1 \rightarrow \text{iris}(5,2,3.5,1,\text{versicolor}).$	$0 \rightarrow \text{iris}(5,2,3.5,1,\text{virginica}).$
$1 \rightarrow \text{iris}(5.9,3,4.2,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.9,3,4.2,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(6,2.2,4,1,\text{versicolor}).$	$0 \rightarrow \text{iris}(6,2.2,4,1,\text{virginica}).$
$1 \rightarrow \text{iris}(6.1,2.9,4.7,1.4,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.1,2.9,4.7,1.4,\text{virginica}).$
$1 \rightarrow \text{iris}(5.6,2.9,3.6,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.6,2.9,3.6,1.3,\text{virginica}).$
$1 \rightarrow \text{iris}(6.7,3.1,4.4,1.4,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.7,3.1,4.4,1.4,\text{virginica}).$
$1 \rightarrow \text{iris}(5.6,3,4.5,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.6,3,4.5,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(5.8,2.7,4.1,1,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.8,2.7,4.1,1,\text{virginica}).$
$1 \rightarrow \text{iris}(6.2,2.2,4.5,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.2,2.2,4.5,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(5.6,2.5,3.9,1.1,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.6,2.5,3.9,1.1,\text{virginica}).$
$1 \rightarrow \text{iris}(5.9,3.2,4.8,1.8,\text{versicolor}).$	$0 \rightarrow \text{iris}(5.9,3.2,4.8,1.8,\text{virginica}).$
$1 \rightarrow \text{iris}(6.1,2.8,4,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.1,2.8,4,1.3,\text{virginica}).$
$1 \rightarrow \text{iris}(6.3,2.5,4.9,1.5,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.3,2.5,4.9,1.5,\text{virginica}).$
$1 \rightarrow \text{iris}(6.1,2.8,4.7,1.2,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.1,2.8,4.7,1.2,\text{virginica}).$
$1 \rightarrow \text{iris}(6.4,2.9,4.3,1.3,\text{versicolor}).$	$0 \rightarrow \text{iris}(6.4,2.9,4.3,1.3,\text{virginica}).$

1 → iris(6.6,3,4.4,1.4,versicolor).	0 → iris(6.6,3,4.4,1.4,virginica).
1 → iris(6.8,2.8,4.8,1.4,versicolor).	0 → iris(6.8,2.8,4.8,1.4,virginica).
1 → iris(6.7,3,5,1.7,versicolor).	0 → iris(6.7,3,5,1.7,virginica).
1 → iris(6,2.9,4.5,1.5,versicolor).	0 → iris(6,2.9,4.5,1.5,virginica).
1 → iris(5.7,2.6,3.5,1,versicolor).	0 → iris(5.7,2.6,3.5,1,virginica).
1 → iris(5.5,2.4,3.8,1.1,versicolor).	0 → iris(5.5,2.4,3.8,1.1,virginica).
1 → iris(5.5,2.4,3.7,1,versicolor).	0 → iris(5.5,2.4,3.7,1,virginica).
1 → iris(5.8,2.7,3.9,1.2,versicolor).	0 → iris(5.8,2.7,3.9,1.2,virginica).
1 → iris(6,2.7,5.1,1.6,versicolor).	0 → iris(6,2.7,5.1,1.6,virginica).
1 → iris(5.4,3,4.5,1.5,versicolor).	0 → iris(5.4,3,4.5,1.5,virginica).
1 → iris(6,3.4,4.5,1.6,versicolor).	0 → iris(6,3.4,4.5,1.6,virginica).
1 → iris(6.7,3.1,4.7,1.5,versicolor).	0 → iris(6.7,3.1,4.7,1.5,virginica).
1 → iris(6.3,2.3,4.4,1.3,versicolor).	0 → iris(6.3,2.3,4.4,1.3,virginica).
1 → iris(5.6,3,4.1,1.3,versicolor).	0 → iris(5.6,3,4.1,1.3,virginica).
1 → iris(5.5,2.5,4,1.3,versicolor).	0 → iris(5.5,2.5,4,1.3,virginica).
1 → iris(5.5,2.6,4.4,1.2,versicolor).	0 → iris(5.5,2.6,4.4,1.2,virginica).
1 → iris(6.1,3,4.6,1.4,versicolor).	0 → iris(6.1,3,4.6,1.4,virginica).
1 → iris(5.8,2.6,4,1.2,versicolor).	0 → iris(5.8,2.6,4,1.2,virginica).
1 → iris(5,2.3,3.3,1,versicolor).	0 → iris(5,2.3,3.3,1,virginica).
1 → iris(5.6,2.7,4.2,1.3,versicolor).	0 → iris(5.6,2.7,4.2,1.3,virginica).
1 → iris(5.7,3,4.2,1.2,versicolor).	0 → iris(5.7,3,4.2,1.2,virginica).
1 → iris(5.7,2.9,4.2,1.3,versicolor).	0 → iris(5.7,2.9,4.2,1.3,virginica).
1 → iris(6.2,2.9,4.3,1.3,versicolor).	0 → iris(6.2,2.9,4.3,1.3,virginica).
1 → iris(5.1,2.5,3,1.1,versicolor).	0 → iris(5.1,2.5,3,1.1,virginica).
1 → iris(5.7,2.8,4.1,1.3,versicolor).	0 → iris(5.7,2.8,4.1,1.3,virginica).
1 → iris(6.3,3.3,6,2.5,virginica).	0 → iris(6.3,3.3,6,2.5,versicolor).
1 → iris(5.8,2.7,5.1,1.9,virginica).	0 → iris(5.8,2.7,5.1,1.9,versicolor).
1 → iris(7.1,3,5.9,2.1,virginica).	0 → iris(7.1,3,5.9,2.1,versicolor).
1 → iris(6.3,2.9,5.6,1.8,virginica).	0 → iris(6.3,2.9,5.6,1.8,versicolor).
1 → iris(6.5,3,5.8,2.2,virginica).	0 → iris(6.5,3,5.8,2.2,versicolor).
1 → iris(7.6,3,6.6,2.1,virginica).	0 → iris(7.6,3,6.6,2.1,versicolor).
1 → iris(4.9,2.5,4.5,1.7,virginica).	0 → iris(4.9,2.5,4.5,1.7,versicolor).
1 → iris(7.3,2.9,6.3,1.8,virginica).	0 → iris(7.3,2.9,6.3,1.8,versicolor).
1 → iris(6.7,2.5,5.8,1.8,virginica).	0 → iris(6.7,2.5,5.8,1.8,versicolor).
1 → iris(7.2,3.6,6.1,2.5,virginica).	0 → iris(7.2,3.6,6.1,2.5,versicolor).
1 → iris(6.5,3.2,5.1,2,virginica).	0 → iris(6.5,3.2,5.1,2,versicolor).
1 → iris(6.4,2.7,5.3,1.9,virginica).	0 → iris(6.4,2.7,5.3,1.9,versicolor).
1 → iris(6.8,3,5.5,2.1,virginica).	0 → iris(6.8,3,5.5,2.1,versicolor).

1 → iris(5.7,2.5,5,2, virginica).	0 → iris(5.7,2.5,5,2, versicolor).
1 → iris(5.8,2.8,5.1,2.4, virginica).	0 → iris(5.8,2.8,5.1,2.4, versicolor).
1 → iris(6.4,3.2,5.3,2.3, virginica).	0 → iris(6.4,3.2,5.3,2.3, versicolor).
1 → iris(6.5,3,5.5,1.8, virginica).	0 → iris(6.5,3,5.5,1.8, versicolor).
1 → iris(7.7,3.8,6.7,2.2, virginica).	0 → iris(7.7,3.8,6.7,2.2, versicolor).
1 → iris(7.7,2.6,6.9,2.3, virginica).	0 → iris(7.7,2.6,6.9,2.3, versicolor).
1 → iris(6,2.2,5,1.5, virginica).	0 → iris(6,2.2,5,1.5, versicolor).
1 → iris(6.9,3.2,5.7,2.3, virginica).	0 → iris(6.9,3.2,5.7,2.3, versicolor).
1 → iris(5.6,2.8,4.9,2, virginica).	0 → iris(5.6,2.8,4.9,2, versicolor).
1 → iris(7.7,2.8,6.7,2, virginica).	0 → iris(7.7,2.8,6.7,2, versicolor).
1 → iris(6.3,2.7,4.9,1.8, virginica).	0 → iris(6.3,2.7,4.9,1.8, versicolor).
1 → iris(6.7,3.3,5.7,2.1, virginica).	0 → iris(6.7,3.3,5.7,2.1, versicolor).
1 → iris(7.2,3.2,6,1.8, virginica).	0 → iris(7.2,3.2,6,1.8, versicolor).
1 → iris(6.2,2.8,4.8,1.8, virginica).	0 → iris(6.2,2.8,4.8,1.8, versicolor).
1 → iris(6.1,3,4.9,1.8, virginica).	0 → iris(6.1,3,4.9,1.8, versicolor).
1 → iris(6.4,2.8,5.6,2.1, virginica).	0 → iris(6.4,2.8,5.6,2.1, versicolor).
1 → iris(7.2,3,5.8,1.6, virginica).	0 → iris(7.2,3,5.8,1.6, versicolor).
1 → iris(7.4,2.8,6.1,1.9, virginica).	0 → iris(7.4,2.8,6.1,1.9, versicolor).
1 → iris(7.9,3.8,6.4,2, virginica).	0 → iris(7.9,3.8,6.4,2, versicolor).
1 → iris(6.4,2.8,5.6,2.2, virginica).	0 → iris(6.4,2.8,5.6,2.2, versicolor).
1 → iris(6.3,2.8,5.1,1.5, virginica).	0 → iris(6.3,2.8,5.1,1.5, versicolor).
1 → iris(6.1,2.6,5.6,1.4, virginica).	0 → iris(6.1,2.6,5.6,1.4, versicolor).
1 → iris(7.7,3,6.1,2.3, virginica).	0 → iris(7.7,3,6.1,2.3, versicolor).
1 → iris(6.3,3.4,5.6,2.4, virginica).	0 → iris(6.3,3.4,5.6,2.4, versicolor).
1 → iris(6.4,3.1,5.5,1.8, virginica).	0 → iris(6.4,3.1,5.5,1.8, versicolor).
1 → iris(6,3,4.8,1.8, virginica).	0 → iris(6,3,4.8,1.8, versicolor).
1 → iris(6.9,3.1,5.4,2.1, virginica).	0 → iris(6.9,3.1,5.4,2.1, versicolor).
1 → iris(6.7,3.1,5.6,2.4, virginica).	0 → iris(6.7,3.1,5.6,2.4, versicolor).
1 → iris(6.9,3.1,5.1,2.3, virginica).	0 → iris(6.9,3.1,5.1,2.3, versicolor).
1 → iris(5.8,2.7,5.1,1.9, virginica).	0 → iris(5.8,2.7,5.1,1.9, versicolor).
1 → iris(6.8,3.2,5.9,2.3, virginica).	0 → iris(6.8,3.2,5.9,2.3, versicolor).
1 → iris(6.7,3.3,5.7,2.5, virginica).	0 → iris(6.7,3.3,5.7,2.5, versicolor).
1 → iris(6.7,3,5.2,2.3, virginica).	0 → iris(6.7,3,5.2,2.3, versicolor).
1 → iris(6.3,2.5,5,1.9, virginica).	0 → iris(6.3,2.5,5,1.9, versicolor).
1 → iris(6.5,3,5.2,2, virginica).	0 → iris(6.5,3,5.2,2, versicolor).
1 → iris(6.2,3.4,5.4,2.3, virginica).	0 → iris(6.2,3.4,5.4,2.3, versicolor).
1 → iris(5.9,3,5.1,1.8, virginica).	0 → iris(5.9,3,5.1,1.8, versicolor).

Apéndice B

Predicados de desplegado en FLOPER

transformation_goto/1

```
1 transformation_goto(prev) :- !,
2     transformation_current_id(C),
3     P is C - 1,
4     transformation_goto(P).
5
6 transformation_goto(next) :- !,
7     transformation_current_id(C),
8     N is C - 1,
9     transformation_goto(N).
10
11 transformation_goto(first) :- !,
12     transformation_unsafe_goto(1).
13
14 transformation_goto(last) :- !,
15     transformation_insert_id(I),
16     transformation_unsafe_goto(I).
17
18 transformation_goto(T) :-
19     transformation_insert_id(I),
20     number(T),
21     T >= 1,
22     T =< I,
23     transformation_unsafe_goto(T).
```

transformation_unsafe_goto/1

```

1 transformation_unsafe_goto(T) :-
2     transformation(T, _, _, _, rules(R)),
3     retract(transformation_current_id(_)),
4     assertz(transformation_current_id(T)),
5     retractall(rule_current_id(_)),
6     assert_rules(R).

```

assert_rules/1

```

1 assert_rules([]).
2 assert_rules([X|Xs]) :-
3     assertz(rule_current_id(X)),
4     assert_rules(Xs).

```

retract_rules/1

```

1 retract_rules([]).
2 retract_rules([X|Xs]) :-
3     retract(rule_current_id(X)),
4     retract_rules(Xs).

```

insert_transformation/2

```

1 insert_transformation(transformation(0,add(A),remove(R)),N) :-
2     transformation_insert_id(C),
3     N is C + 1,
4     retract(transformation_insert_id(C)),
5     assertz(transformation_insert_id(N)),
6     retract(transformation_current_id(F)),
7     assertz(transformation_current_id(N)),
8     assert_rules(A),
9     retract_rules(R),
10    findall(X, rule_current_id(X), Rules),
11    assertz(
12        transformation(N,0,from(F),add(A),remove(R),rules(Rules))
13    ).

```

insert_rules/2

```
1 insert_rules([], []).
2 insert_rules([rule(H, I, B, T)|Xs], [N|Ns]) :-
3     rule_insert_id(C),
4     N is C + 1,
5     retract(rule_insert_id(C)),
6     assertz(rule_insert_id(N)),
7     assertz(rule(N, H, I, B, T)),
8     insert_rules(Xs, Ns).
```


Apéndice C

Predicados de calibrado en FLOPER

update_rule_symbols/0

```
1 update_rule_symbols :-
2     retractall(tuning_sym(_,_,_)),
3     findall(B,(rule_current_id(X), rule(X,_,_,body(B),_)),Bs),
4     update_rule_symbols(Bs).
```

update_rule_symbols/1

```
1 update_rule_symbols([X|Xs]) :-
2     update_rule_symbols(X),
3     update_rule_symbols(Xs).
4 update_rule_symbols(atom(_,Xs)) :- update_rule_symbols(Xs).
5 update_rule_symbols(term(_,Xs)) :- update_rule_symbols(Xs).
6 update_rule_symbols(and(_,_,Xs)) :- update_rule_symbols(Xs).
7 update_rule_symbols(or(_,_,Xs)) :- update_rule_symbols(Xs).
8 update_rule_symbols(agr(_,_,Xs)) :- update_rule_symbols(Xs).
9 update_rule_symbols(pri(_,_,Xs)) :- update_rule_symbols(Xs).
10 update_rule_symbols(sym(td(S))) :-
11     tuning_sym(td,1,S);
12     assertz(tuning_sym(td,1,S)).
13 update_rule_symbols(sym(P)) :-
14     P =.. [C,S,L,A],
15     ( tuning_sym(C,L,S);
16       assertz(tuning_sym(C,L,S))
17     ), update_rule_symbols(A).
18 update_rule_symbols(_).
```

all_instantiated/1

```

1 all_instantiated([]).
2 all_instantiated([X|Xs]) :-
3     all_instantiated(X),
4     all_instantiated(Xs).
5 all_instantiated(num(_)).
6 all_instantiated(td(_)).
7 all_instantiated(var(_)).
8 all_instantiated(atom(_,Xs)) :- all_instantiated(Xs).
9 all_instantiated(term(_,Xs)) :- all_instantiated(Xs).
10 all_instantiated(and(_,_,Xs)) :- all_instantiated(Xs).
11 all_instantiated(or(_,_,Xs)) :- all_instantiated(Xs).
12 all_instantiated(agr(_,_,Xs)) :- all_instantiated(Xs).
13 all_instantiated(pri(_,_,Xs)) :- all_instantiated(Xs).
14 all_instantiated(sym(td(_))) :-
15     write('Symbolic constants are not allowed.'), nl, !, fail.
16 all_instantiated(sym(_)) :-
17     write('Symbolic connectives are not allowed.'), nl, !,
18     fail.

```

tuning_lat/4

```

1 tuning_lat(td, _, td, M) :- lat:members(X), member(M,X).
2 tuning_lat(and, L, and, Ps) :-
3     N is L+1,
4     current_predicate(lat:P/N),
5     sub_atom(P, _, 4, _, and_),
6     sub_atom(P, 4, _, 0, Ps).
7 tuning_lat(or, L, or, Ps) :-
8     N is L+1, current_predicate(lat:P/N),
9     sub_atom(P, _, 3, _, or_),
10    sub_atom(P, 3, _, 0, Ps).
11 tuning_lat(agr, L, agr, Ps) :-
12    N is L+1, current_predicate(lat:P/N),
13    sub_atom(P, _, 4, _, agr_),
14    sub_atom(P, 4, _, 0, Ps).
15 tuning_lat(con, L, C, P) :-
16    tuning_lat(and, L, C, P);
17    tuning_lat(or, L, C, P);
18    tuning_lat(agr, L, C, P).

```

write_symbolic_sub/0

```

1 write_symbolic_sub :-
2     tuning_sub(Best),
3     tuning_tau(Tau),
4     write('< '),
5     write(Tau),
6     write(', {'),
7     write_symbolic_sub(Best),
8     write('} >').

```

write_symbolic_sub/1

```

1 write_symbolic_sub([]).
2 write_symbolic_sub(pair(sym(and,_,N),val(and,_,M))) :- !,
3     write_symbolic_sub(sym(imp,_,N)),
4     write('/'),
5     write_symbolic_sub(val(imp,_,M)),
6     write(', '),
7     write_symbolic_sub(sym(and,_,N)),
8     write('/'),
9     write_symbolic_sub(val(and,_,M)).
10 write_symbolic_sub(pair(S,V)) :- !,
11     write_symbolic_sub(S),
12     write('/'),
13     write_symbolic_sub(V).
14 write_symbolic_sub([X]) :- !,
15     write_symbolic_sub(X).
16 write_symbolic_sub([X|Xs]) :- !,
17     write_symbolic_sub(X),
18     write(', '),
19     write_symbolic_sub(Xs).

```

tuning_basic_assert/2

```

1 tuning_basic_assert([],_).
2 tuning_basic_assert([rule(R,H,I,body(B),TD)|Xs],C) :-
3     tuning_apply(B,C,B2),
4     assertz(rule(R,H,I,body(B2),TD)),
5     tuning_basic_assert(Xs,C).

```