UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Calibrado de programas lógicos difusos mediante satisfacibilidad módulo teorías

José Antonio Riaza Valverde

Julio de 2019

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

Departamento de Sistemas Informáticos

TRABAJO FIN DE MÁSTER

Calibrado de programas lógicos difusos mediante satisfacibilidad módulo teorías

Autor: José Antonio Riaza Valverde

Director: Ginés Damián Moreno Valverde

Julio de 2019

Resumen

El objetivo fundamental de este Trabajo Fin de Máster es la introducción, en el entorno FLOPER, de una nueva técnica de calibrado de programas lógicos difusos basada en el uso de solucionadores de problemas de satisfacibilidad módulo teorías. Estas técnicas de calibrado automático de programas difusos simbólicos se utilizan para elegir los pesos y conectivos adecuados en las reglas de un programa simbólico, partiendo de un conjunto de casos de prueba.

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 y del lenguaje FASILL, centrándonos en la implementación de la técnica de calibrado mencionada. Asimismo, se ha desarrollado una nueva plataforma online donde poder desplegar y calibrar programas lógicos difusos.

Por otra parte, mostramos en esta memoria distintas aplicaciones reales del calibrado de programas difusos a problemas no triviales como, por ejemplo, regresión lineal, validación de circuitos e incluso la web semántica.

Queremos destacar que la descripción de estas técnicas de transformación y calibrado automático de programas lógicos difusos, incluyendo los detalles de su implementación, han originado recientemente seis publicaciones científicas presentadas en congresos internacionales (LOPSTR'16 [41], RuleML+RR'17 [42], SUM'17 [39], ESCIM'17 [40], IEEE SSCI'17 [18] y IEEE SSCI'18 [19]), además de una publicación en un congreso nacional (PROLE'18 [43]). Recientemente hemos sometido otro artículo en un congreso internacional en el ámbito de la lógica difusa (FUZZ-IEEE'19 [2]).

Agradecimientos

Deseo agradecer la colaboración y el apoyo de las personas que han hecho posible este Trabajo Fin de Máster. 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 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 a la hora de supervisar mi Trabajo Fin de Grado, *\Implementacion de tecnicas de desplegado difuso sobre el entorno FLOPER"* [54], que precede a este. 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

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

La lógica difusa supone un marco teórico, formal y matemáticamente muy desarrollado, que ya se ha aplicado con notable éxito en distintas facetas que abarcan nuestra vida cotidiana. En los últimos años, hemos sido testigos del importante papel que ha jugado en el desarrollo de sofisticadas aplicaciones software en campos tan diversos como los sistemas expertos, medicina, control industrial, etcétera. Con el objetivo de facilitar el desarrollo de tales aplicaciones, ha surgido el interés por diseñar lenguajes declarativos difusos que incorporen entre sus recursos expresivos el tratamiento de información imprecisa de forma natural.

Con todos estos antecedentes, el presente trabajo está en concordancia con buena parte de los objetivos perseguidos en el plan de estudios del máster profesional en el ámbito general de las tecnologías de la información y las comunicaciones. Nuestros desarrollos se orientan especialmente a los sistemas inteligentes y a su posible impacto final en el mundo empresarial en relación a la inteligencia aplicada a negocio.

El entorno de programación lógica difusa FLOPER ha sido desarrollado en la Universidad de Castilla-La Mancha a partir de otros proyectos fin de grado/máster que preceden al que proponemos ahora. En su estado actual, la herramienta es capaz de ejecutar y depurar programas difusos sobre cualquier plataforma Prolog, e incorpora diversas técnicas para la transformación, optimización, especialización, entre otros, diseñadas por el grupo de investigación DEC-TAU en los últimos años. Es posible ejecutar su última versión online a través de http://dectau.uclm.es/fasill.

FASILL (*Fuzzy Aggregators and Similarity Into a Logic Language"*) es un lenguaje de programación lógico difuso con anotaciones de grados de verdad implícitas/explícitas, una gran variedad de conectivos y unificación por similitud. FASILL integra y extiende las características de MALP (*Multi-Adjoint Logic Programming"*, un lenguaje de programación lógico difuso con reglas explícitamente anotadas) y Bousi∼Prolog

(que utiliza un algoritmo de unificación débil y está bien adaptado para la respuesta de consultas flexibles).

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 introduce una extensión simbólica de los programas FASILL, con el objetivo de poder escribir reglas con grados de verdad y conectivos simbólicos (que no están definidos en el retículo asociado al programa). El entorno FLOPER ya cuenta con tres técnicas diferentes de calibrado, que encuentran los mejores valores para esos grados de verdad y conectivos simbólicos de un programa, con respecto a un conjunto de casos de prueba proporcionados por el usuario.

El *problema de satisfacibilidad booleana* (*SAT*) consiste en determinar si una fórmula proposicional puede evaluarse como cierta, y se manifiesta en varios dominios de aplicación importantes, tales como el diseño y la verificación de sistemas software y hardware, así como en aplicaciones de inteligencia artificial.

No obstante, a menudo, las aplicaciones en estos campos requieren determinar la satisfacibilidad de fórmulas en lógicas más expresivas, tales como la lógica de primer orden, aunque muchas de ellas no requieren satisfacibilidad de primer orden en general, sino satisfacibilidad respecto a alguna teoría de fondo que fije la interpretación de ciertos símbolos de predicado y de función. El campo de investigación centrado en la satisfacibilidad de estas fórmulas con respecto a alguna teoría de fondo se denomina *satisfacibilidad modulo teor as* (*SMT*). En analogía con los procedimientos de SAT, los procedimientos de SMT (ya sean de decisión o no) generalmente se conocen como *solucionadores* de SMT.

En este Trabajo Fin de Máster se pretende incorporar al entorno FLOPER una nueva técnica de calibrado simbólico de programas sFASILL basada en el uso de solucionadores SMT, dado que una vez obtenidas las respuestas computadas difusas simbólicas a partir de los casos de prueba, el calibrado puede ser expresado como un problema de optimización en el que se debe minimizar la desviación obtenida por las respuestas simbólicas con respecto a las respuestas esperadas en los casos de prueba.

1.2. Objetivos

En este Trabajo Fin de Máster nos centraremos en extender las funcionalidades del entorno de programación lógica difusa FLOPER, equipándolo con poderosos mecanismos de calibrado de programas lógicos difusos. Este trabajo se centra en tres objetivos fundamentales.

- En primer lugar, investigaremos las posibles conexiones entre las técnicas de ca-

librado de programas lógicos difusos y los problemas de satisfacibilidad booleana y satisfacibiliad módulo teorías, estudiando algunas teorías de interés que más tarde adaptaremos a nuestro problema.

- En segundo lugar, implementaremos en el entorno FLOPER un nuevo método de calibrado automático de programas basado en el uso de solucionadores SMT. También incorporaremos esta nueva funcionalidad al entorno FLOPER online, donde es posible ejecutar, desplegar y calibrar programas lógicos difusos sin necesidad de descargar e instalar la herramienta.

- En tercer lugar, expondremos en esta memoria una serie de casos de uso y aplicaciones donde el calibrado de programas lógicos difusos puede ser de utilidad. Además, utilizaremos estos ejemplos para medir y comparar el rendimiento de las técnicas de calibrado ya implementadas en el entorno FLOPER con la nueva técnica propuesta en este trabajo.

1.3. Estructura de la memoria

Tras un primer capítulo donde introducimos la motivación y los objetivos fundamentales del presente Trabajo Fin de Máster, la estructura de esta memoria es la que se enumera a continuación.

- **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, programación lógica, satisfacibilidad booleana y satisfacibilidad módulo teorías, así como la sintaxis y la semántica operacional de los lenguajes y herramientas involucrados: FASILL, SMT-LIB y Z3. Además, se muestran las últimas herramientas implementadas para el entorno FLOPER: una nueva consola interactiva implementada en SWI-Prolog, y una nueva versión online en la que se incorporará la nueva técnica de calibrado expuesta en este trabajo.

- **Capítulo 3. Calibrado de programas lógicos difusos en el entorno FLOPER.**
  En el Capítulo 3 se presenta la extensión simbólica del lenguaje FASILL y se introduce la idea de calibrado de programas lógicos difusos. Además, se formaliza su definición para el lenguaje FASILL, presentando un método para el calibrado automático de programas simbólicos. Por último, se muestra su implementación en el entorno FLOPER y su integración en la versión online.

- **Capítulo 4. Calibrado de programas lógicos difusos en solucionadores SMT.**
  En el Capítulo 4 se recoge la principal aportación de este trabajo, introduciendo un nuevo método de calibrado automático de programas lógicos difusos basado en el uso de solucionadores SMT. Se detalla la implementación de esta técnica en el entorno FLOPER, y su adopción en el entorno online.

- **Capítulo 5. Casos de uso del calibrado.**
  En el Capítulo 5 se recogen una serie de casos de uso donde las técnicas de calibrado pueden resultar útiles, y se establecen relaciones con otros campos de aplicación como la verificación automática, el aprendizaje automático, o la web semántica. Estos casos de uso se utilizan además para comparar el rendimiento del nuevo método de calibrado con los métodos ya implementados en el entorno FLOPER.

- **Capítulo 6. Conclusiones y trabajo futuro.**
  Finalmente, en el Capítulo 6 se recogen las conclusiones del trabajo, destacando la técnica de calibrado de programas lógicos difusos implementada 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

2.1. Lógica

La lógica es una disciplina que tiene sus orígenes en los trabajos de Aristóteles 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. Los diferentes sistemas lógicos elementales tienen en común, en su presentación, una etapa previa de formalización simbólica que suele hacerse a dos niveles [22]:

- *Logica de proposiciones*: Se ocupa de enunciados declarativos simples o proposiciones que se contemplan como un todo indivisible y que pueden ser combinados mediante *conectivas* (como 'no', 'y', 'o', 'si ... entonces ...').
- *Logica 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 *logica de proposiciones* o *logica 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 *formula bien formada*, que se construye haciendo uso de:

- Una cantidad finita de variables proposicionales: $p, q, r, \ldots$
- Un conjunto de símbolos para las conectivas: $\neg$, $\wedge$, $\vee$, $\oplus$, $\Rightarrow$ 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, [22]). *Una formula bien formada es una expresion, en la que intervienen variables proposicionales y s mbolos de conectiva, conforme a las siguientes reglas:*

1. *Toda variable proposicional $p, q, r, \ldots$ es una formula bien formada.*
2. *Si $\mathcal{A}$ y $\mathcal{B}$ son formulas bien formadas, tambien lo son: $\neg\mathcal{A}$, $\neg\mathcal{B}$, $\mathcal{A} \wedge \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \oplus \mathcal{B}$, $\mathcal{A} \Rightarrow \mathcal{B}$ y $\mathcal{A} \Leftrightarrow \mathcal{B}$.*

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

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

reproduce un ejemplo clásico de *silogismo* 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 *conclusion*. Ahora bien, en la lógica referida no puede justificarse la tesis r: no existen propiedades de las conectivas $\wedge$, $\Rightarrow$ que garanticen dicha tesis [54].

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 *Socrates 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 *termino*, 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 *logica de predicados* (que para nosotros será sinónimo de *logica 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 que se listan a continuación.

- Constantes: $a, b, c, \ldots$ El conjunto infinito numerable de símbolos de constante lo denotaremos por $\mathcal{C}$.
- Símbolos de variable: $x, y, z, \ldots$ El conjunto infinito numerable de símbolos de variable lo denotaremos por $\mathcal{V}$.
- Símbolos de función: $f, g, h, \ldots$ El conjunto infinito numerable de símbolos de función lo denotaremos por $\mathcal{F}$.
- Símbolos de predicado: $P, Q, R, \ldots$
- 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 $\mathcal{L}$, [22]). *Un termino de $\mathcal{L}$ es una expresion, en la que intervienen s mbolos de constante y s mbolos de funcion, que se de ne de acuerdo a las siguientes reglas:*

1. *Si $t \in \mathcal{V} \cup \mathcal{C}$ entonces t es un termino. Esto es, toda variable o constante de $\mathcal{L}$ es un termino de $\mathcal{L}$.*
2. *Si $t_1, t_2, \ldots, t_n$ son terminos de $\mathcal{L}$ y f es un s mbolo de funcion n-ario de $\mathcal{L}$, entonces $f(t_1, t_2, \ldots, t_n)$ es un termino de $\mathcal{L}$.*

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

Definición 2.3. (Fórmula atómica). *Una formula atomica de $\mathcal{L}$ es una expresion $P(t_1, t_2, \ldots, t_n)$, en la que $t_1, t_2, \ldots, t_n$ son terminos de $\mathcal{L}$, y P es un s mbolo de predicado n-ario de $\mathcal{L}$.*

Además, por *`atomica'* 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, [22]). *Una formula bien formada (fbf) de $\mathcal{L}$ es una expresion, en la que intervienen formulas atomicas, conectivas o cuanti cadores, que cumplen las siguientes reglas:*

1. *Toda formula atomica de $\mathcal{L}$ es una formula bien formada.*
2. *Si $\mathcal{A}$ y $\mathcal{B}$ son formulas bien formadas de $\mathcal{L}$, tambien 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 formula bien formada de $\mathcal{L}$ que incorpora la variable $x \in \mathcal{V}$ entonces $\forall x, \mathcal{A}$ y $\exists x : \mathcal{A}$ son formulas bien formadas.*

Concluimos esta subsección con una serie de definiciones de conceptos que utilizaremos más adelante:

1. *Ocurrencia* de una variable es cualquier aparición de una variable en una fórmula.
2. *Radio de accion*, ámbito o alcance de un cuantificador:
   - En la fbf $(\forall x, \mathcal{A})$, el radio de acción de $(\forall x)$ es $\mathcal{A}$.
   - En la fbf $(\exists x : \mathcal{A})$, el radio de acción de $(\exists x)$ es $\mathcal{A}$.
3. Una ocurrencia de la variable x en una fbf se dice que es *ligada* si aparece dentro del radio de acción de un cuantificador universal $(\forall x)$ o uno existencial $(\exists x)$.
4. Una ocurrencia de la variable x en una fbf se dice que es *libre*, si su aparición no es ligada.
5. Una fórmula *cerrada* es la que no tiene ocurrencias de variables libres.

2.1.2.1. Semántica

La *semantica* estudia la adscripción de significado a los lenguajes de los sistemas formales. La semántica de un sistema lógico, suele abordarse desde la llamada *teor a 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 [22]:

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, [22]). *Una interpretacion $\mathcal{I}$ de $\mathcal{L}$ es un par $(\mathcal{D}, \mathcal{J})$ que consiste en:*

1. *Un conjunto no vac o $\mathcal{D}$, el dominio de $\mathcal{I}$.*

2. *Una aplicacion $\mathcal{J}$ que asigna:*

   a) *A cada s mbolo de constante de $\mathcal{L}$, a, un elemento distinguido de $\mathcal{D}$; $\mathcal{J}(a) = \bar{a}$.*

   b) *A cada s mbolo de funcion f de $\mathcal{L}$ una funcion $\mathcal{J}(f) = \bar{f}$, tal que $\bar{f} : \mathcal{D}^n \to \mathcal{D}$.*

   c) *A cada s mbolo de predicado P de $\mathcal{L}$ una relacion $\mathcal{J}(P) = R$, tal que $R \subset \mathcal{D}^n$.*

Solo podremos hablar de verdad y falsedad en el contexto de una interpretación, pero para ello todavía falta un paso previo (si la fbf no es cerrada): asignar valores a las ocurrencias de las variables (libres) en la fórmula.

Definición 2.6. (Valoración, [23]). *Una valoracion $\vartheta : \mathcal{T} \to \mathcal{D}$ en $\mathcal{I}$ es una aplicacion que asigna a cada termino de $\mathcal{L}$ un elemento del dominio de la interpretacion $\mathcal{D}$ y que de nimos inductivamente mediante las siguientes reglas:*

$$\vartheta(t) = \begin{cases} v(x) & si \quad t \in \mathcal{V} \wedge t \equiv x \\ \mathcal{J}(a) & si \quad t \in \mathcal{C} \wedge t \equiv a \\ \mathcal{J}(f)(\vartheta(t_1), \ldots, \vartheta(t_n)) & si \quad f \in \mathcal{F} \wedge t_1, \ldots, t_n \in \mathcal{T} \wedge t \equiv f(t_1, \ldots, t_n) \end{cases}$$

donde v es una aplicacion que asigna a cada variable de $\mathcal{L}$ un elemento del dominio de interpretacion $\mathcal{D}$.

Una valoración ϑ_x que coincide exactamente con la valoración ϑ, salvo quizá en el valor asignado a la variable $x \in \mathcal{V}$, se denomina valoración x-equivalente de ϑ.

Definición 2.7. (Satisfacibilidad, [23]). *Sea una interpretacion $\mathcal{I} = (\mathcal{D}, \mathcal{J})$. Sea $\mathcal{A}$ una fbf de $\mathcal{L}$. Decimos que la valoracion ϑ en $\mathcal{I}$ satisface la fbf $\mathcal{A}$ si y solo si se cumple que:*

1. *Si $\mathcal{A} \equiv r(t_1, \ldots, t_n)$ entonces $\bar{r}(\vartheta(t_1), \ldots, \vartheta(t_n))$ es verdadero, donde $\bar{r} = \mathcal{J}(r)$ es una relacion en $\mathcal{D}$.*

2. *Si $\mathcal{A}$ es de la forma:*

   a) $\neg\mathcal{B}$ *entonces* ϑ *no satisface* $\mathcal{B}$*;*

   b) $(\mathcal{B} \wedge \mathcal{C})$ *entonces* ϑ *satisface* $\mathcal{B}$ *y* ϑ *satisface* $\mathcal{C}$*;*

   c) $(\mathcal{B} \vee \mathcal{C})$ *entonces* ϑ *satisface* $\mathcal{B}$ *o* ϑ *satisface* $\mathcal{C}$*;*

   d) $(\mathcal{B} \Rightarrow \mathcal{C})$ *entonces* ϑ *satisface* $\neg\mathcal{B}$ *o* ϑ *satisface* $\mathcal{C}$*;*

   e) $(\mathcal{B} \Leftrightarrow \mathcal{C})$ *entonces* ϑ *satisface* $\mathcal{B}$ *y* $\mathcal{C}$*, o* ϑ *no satisface ni* $\mathcal{B}$ *ni* $\mathcal{C}$*.*

3. *Si* $\mathcal{A} \equiv (\forall x, \mathcal{B})$*, para* ***toda*** *valoracion* ϑ_x x*-equivalente de* ϑ*,* ϑ_x *satisface* $\mathcal{B}$*.*

4. *Si* $\mathcal{A} \equiv (\exists x : \mathcal{B})$*, para* ***alguna*** *valoracion* ϑ_x x*-equivalente de* ϑ*,* ϑ_x *satisface* $\mathcal{B}$*.*

Dada una fórmula bien formada de $\mathcal{L}$, es conveniente introducir la siguiente nomenclatura:

1. Una fbf $\mathcal{A}$ es *verdadera* en la interpretación $\mathcal{I}$ si y solo si toda valoración ϑ en $\mathcal{I}$ satisface $\mathcal{A}$.

2. Una fbf $\mathcal{A}$ es *falsa* en $\mathcal{I}$ si y solo si no existe valoración ϑ en $\mathcal{I}$ que satisfaga $\mathcal{A}$. Escribiremos $\mathcal{I} \models \mathcal{A}$ para denotar que $\mathcal{A}$ es verdadera en $\mathcal{I}$.

3. $\mathcal{A}$ es *logicamente valida* (denotado $\models \mathcal{A}$) si y solo si para toda interpretación $\mathcal{I}$, $\mathcal{A}$ es verdadera en $\mathcal{I}$.

4. $\mathcal{A}$ es *insatisfacible* si y solo si para toda interpretación $\mathcal{I}$, $\mathcal{A}$ es falsa en $\mathcal{I}$.

5. $\mathcal{A}$ es satisfacible si y solo si existe una interpretación $\mathcal{I}$ y una valoración en ella de las variables libres tal que ϑ satisface $\mathcal{A}$ en $\mathcal{I}$.

Un resultado interesante, que tiene repercusiones cuando se estudia la forma clausal, es que al añadir un cuantificador universal para una variable x que aparece libre en $\mathcal{A}$, la nueva fórmula sigue siendo verdadera en $\mathcal{I}$. De este modo, cuando consideramos la verdad o falsedad de fbf con variables libres, los cuantificadores universales se sobreentienden en cierto sentido.

El valor de verdad de una fórmula cerrada no depende de la valoración concreta en $\mathcal{I}$. Si encontramos una valoración ϑ que satisface una fórmula en $\mathcal{I}$ entonces cualquier otra valoración también la satisfará. Si una valoración no la satisface sucede lo contrario, es decir, ninguna valoración la satisfará y la fórmula será falsa en $\mathcal{I}$.

Dado que para fbf cerradas los conceptos de satisfacibilidad por una valoración en $\mathcal{I}$ y verdad en $\mathcal{I}$ son equivalentes, el concepto de "satisfacible" puede expresarse en términos de verdad en una interpretación $\mathcal{I}$: *una fbf cerrada $\mathcal{A}$ es satisfacible si y solo si existe una interpretacion $\mathcal{I}$ en la cual $\mathcal{A}$ sea verdadera*. También decimos que $\mathcal{I}$ satisface $\mathcal{A}$ o que $\mathcal{I}$ es modelo de $\mathcal{A}$ [23].

Tabla 2.1: Definición de las conectivas lógicas proposicionales

x	y	$\neg x$	$\neg y$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \Rightarrow y$	$x \Leftrightarrow y$
0	0	1	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0
1	0	0	1	0	1	1	0	0
1	1	0	0	1	1	0	1	1

Definición 2.8. (Modelo, [23]). *Dada una fbf cerrada $\mathcal{A}$ de $\mathcal{L}$, decimos que una interpretacion $\mathcal{I}$ es modelo de $\mathcal{A}$ si y solo si la fbf $\mathcal{A}$ es verdadera en la interpretacion $\mathcal{I}$. Sea Γ un conjunto de fbf cerradas de $\mathcal{L}$, entonces $\mathcal{I}$ es modelo de Γ si y solo si $\mathcal{I}$ es modelo para cada una de las formulas de Γ.*

2.2. Satisfacibilidad booleana

El *problema de satisfacibilidad booleana* (*SAT*) consiste en determinar si una fórmula proposicional puede evaluarse como cierta. SAT fue el primer problema en ser clasificado como NP-completo, y se manifiesta en varios dominios de aplicación importantes, tales como el diseño y la verificación de sistemas software y hardware, así como en aplicaciones de inteligencia artificial [33].

2.2.1. Notación

Una *formula proposicional* es una fórmula bien formada (véase la Definición 2.1) que tiene un valor de verdad. Sea $\mathcal{V}$ el conjunto de variables proposicionales y sean 0 y 1 los elementos del dominio lógico B representando los valores *falso* y *cierto*, respectivamente. Toda función lógica $f : \mathsf{B}^n \to \mathsf{B}$ puede expresarse como una fórmula proposicional $\mathcal{F}$ con n variables $x_1, x_2, \ldots, x_n \in \mathcal{V}$.

La interpretación del conjunto de conectivas lógicas $\{\neg, \wedge, \vee, \oplus, \Rightarrow, \Leftrightarrow\}$ está descrita en la Tabla 2.1[1]. Utilizaremos el símbolo $\equiv$ para denotar equivalencia lógica. Una *asignacion de verdad* $\mathcal{A}$ es una correspondencia de $\mathcal{V}$ a B, donde $\mathcal{A}(x)$ denota el valor que $\mathcal{A}$ asigna a x. Llamaremos *asignacion total* a $\mathcal{A}$ si es una función total[2]. En otro caso, $\mathcal{A}$ es una asignación parcial. $\mathcal{A}$ satisface una fórmula $\mathcal{F}(x_1, \ldots, x_n)$ si y solo si $\mathcal{F}(\mathcal{A}(x_1), \ldots, \mathcal{A}(x_n))$ está definido y se evalúa a 1 (denotado por $\mathcal{A} \models \mathcal{F}$). Una fórmula $\mathcal{F}$ es satisfacible si y solo si $\exists \mathcal{A} : \mathcal{A} \models \mathcal{F}$, y es insatisfacible (es decir, lógicamente inconsistente) en cualquier otro caso [33].

[1] En ocasiones, representaremos la negación $\neg x$ como $\bar{x}$.

[2] Una función es total si está definida para todo el conjunto de partida.

2.2.2. Forma normal conjuntiva

La *forma normal conjuntiva* (*CNF*) de una fórmula es una forma restringida de una fórmula proposicional. Una fórmula en CNF es un producto de sumas (una conjunción de cláusulas). Nótese que la cláusula vacía corresponde con el valor lógico 0.

Ejemplo 2.2. La fórmula $(\bar{x}_3) \wedge (\bar{x}_1) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee x_2 \vee x_3)$ está en forma normal conjuntiva. Una notación alternativa (y más compacta) utilizada comúnmente para representar esta fórmula es $(\bar{x}_3)(\bar{x}_1)(x_1\bar{x}_2)(\bar{x}_1x_2)(x_1x_2x_3)$.

Toda fórmula $\mathcal{F}$ de la lógica proposicional puede ser transformada a CNF. Desafortunadamente, la fórmula resultante puede ser exponencialmente más larga que $\mathcal{F}$. No obstante, es posible construir una fórmula $\mathcal{G}$ en CNF tal que $\mathcal{F}$ y $\mathcal{G}$ sean equisatisfacibles (es decir, $(\exists \mathcal{A} : \mathcal{A} \models \mathcal{F}) \Leftrightarrow (\exists \mathcal{A} : \mathcal{A} \models \mathcal{G})$) y el tamaño de $\mathcal{G}$ sea polinomial respecto al tamaño de la fórmula $\mathcal{F}$ [33]. Tal fórmula equisatisfacible se puede obtener aplicando la transformación de Tseitin, presentada originalmente en [62].

2.2.3. Comprobación de la satisfacibilidad booleana

En esta sección se introduce el problema de satisfacibilidad booleana y se presenta una técnica para abordarlo.

Definición 2.9. (Problema de satisfacibilidad booleana, [33]). *Dada una formula proposicional $\mathcal{F}$, determinar cuando $\mathcal{F}$ es satisfacible.*

Aunque la Definición 2.9 se refiere a fórmulas proposicionales en general, el problema puede ser fácilmente reducido a fórmulas en CNF, dado que toda fórmula puede ser transformada en una fórmula equisatisfacible en forma clausal.

El *principio de resolucion* establece que una asignación de verdad que satisface las cláusulas $\mathcal{C} \vee x$ y $\mathcal{D} \vee \bar{x}$ también satisface $\mathcal{C} \vee \mathcal{D}$. Las cláusulas $\mathcal{C} \vee x$ y $\mathcal{D} \vee \bar{x}$ son los *antecedentes*, x es el *pivote* y $\mathcal{C} \vee \mathcal{D}$ es el *resolvente*. Denotaremos por $Res(C, D, x)$ al resolvente de las cláusulas $\mathcal{C}$ y $\mathcal{D}$ con el pivote x. La regla de resolución correspondiente se describe formalmente a continuación:

$$\frac{\mathcal{C} \vee x \qquad \mathcal{D} \vee \bar{x}}{C \vee \mathcal{D}} Res$$

La resolución corresponde a la cuantificación existencial del pivote y posterior eliminación del cuantificador, como se demuestra mediante la siguiente secuencia de pasos de transformación lógica (donde $\mathcal{F}_{(x \quad e)}$ denota la sustitución de todas las ocurrencias

libres de x en $\mathcal{F}$ por la expresión e) [33]:

$$\begin{aligned} & \exists x : (\mathcal{C} \vee x) \wedge (\mathcal{D} \vee \bar{x}) \\ \equiv\ & ((\mathcal{C} \vee x) \wedge (\mathcal{D} \vee \bar{x}))_{(x \quad 1)} \vee ((\mathcal{C} \vee x) \wedge (\mathcal{D} \vee \bar{x}))_{(x \quad 0)} \\ \equiv\ & ((\mathcal{C} \vee 1) \wedge (\mathcal{D} \vee \bar{1})) \vee ((\mathcal{C} \vee 0) \wedge (\mathcal{D} \vee \bar{0})) \\ \equiv\ & (1 \wedge \mathcal{D}) \vee (\mathcal{C} \wedge 1) \\ \equiv\ & \mathcal{D} \vee \mathcal{C} \equiv \mathcal{C} \vee \mathcal{D} \end{aligned}$$

La aplicación repetida de la regla de resolución resulta en una *prueba de resolucion*.

Definición 2.10. (Prueba de resolución, [33]). *Una prueba de resolucion R es un grafo dirigido ac clico $(V_R, E_R, piv_R, \lambda_R, s_R)$, donde V_R es un conjunto de nodos, E_R es un conjunto de aristas, piv_R es la funcion de pivote, λ_R es la funcion de clausula, y $s_R \in V_R$ es un nodo sumidero. Un nodo inicial tiene grado de entrada 0. El resto de nodos son internos y tienen grado de entrada 2. La funcion de pivote asigna los nodos internos a las variables pivote de los respectivos pasos de resolucion. Para cada nodo interno v y $(v_1, v), (v_2, v) \in E_R, \lambda_R = Res(\lambda_R(v_1), \lambda_R(v_2), piv(R(v)))$.*

Una prueba de resolución es una *refutacion* si $\lambda_R(s_R) =$. Una refutación R es una refutación para una fórmula $\mathcal{F}$ (en CNF) si la etiqueta de cada nodo inicial de R es una cláusula de $\mathcal{F}$. En la Figura 2.1 se muestra una prueba de resolución para la fórmula proposicional $(\bar{x}_3) \wedge (\bar{x}_1) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee x_2 \vee x_3)$.

La regla de resolución es suficiente para diseñar un algoritmo completo para decidir si una fórmula en CNF es satisfacible o no [55]. Nos referiremos al algoritmo presentado en [14] como el procedimiento *Davis-Putnam* o *DP*, que comprende tres reglas:

1. *Regla de 1 literal.* Cuando una de las cláusulas de $\mathcal{F}$ es una clausula unitaria, es decir, contiene un solo literal ℓ, obtenemos una nueva fórmula $\mathcal{F}_1$ al eliminar cualquier instancia de $\bar{\ell}$ del resto de cláusulas, y eliminar cualquier cláusula que contenga ℓ, incluyendo la propia cláusula unitaria.

2. *Regla a rmativa-negativa.* Si cualquier literal ℓ ocurre solo positiva o negativamente en $\mathcal{F}$, eliminamos todas las cláusulas que contengan ℓ.

3. *Regla para eliminar formulas atomicas*. Para todas las cláusulas $(C+x)$ y $(D+\bar{x})$ en $\mathcal{F}$, donde ni C ni D contienen x o $\bar{x}$, el resolvente $Res((C+x), (D+\bar{x}), x)$ se encuentra en $\mathcal{F}_1$. Además, toda cláusula C en $\mathcal{F}$ que no contenga x o $\bar{x}$ también está en $\mathcal{F}_1$.

Esta última regla puede incrementar el tamaño de la fórmula significativamente. No obstante, elimina completamente todas las ocurrencias del átomo x. La corrección de la transformación está justificada por el principio de resolución. En la práctica, la regla

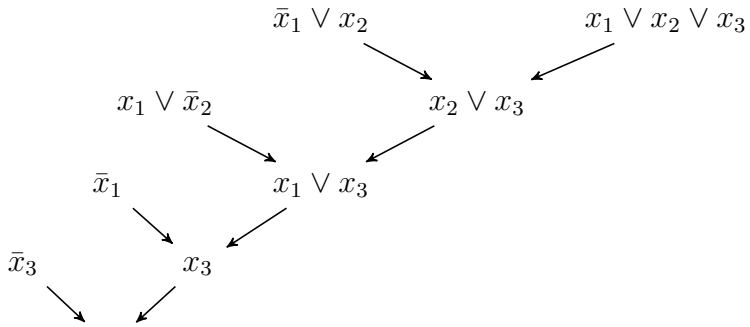


Figura 2.1: Prueba de resolución

de resolución solo debería aplicarse tras las dos primeras reglas. La regla de 1 literal es también conocida como *propagacion de unidades*.

Para problemas realistas, el número de cláusulas generadas por el procedimiento DP crece rápidamente. Para evitar esta explosión, Davis, Logemann y Loveland [13] sugirieron reemplazar la regla de resolución por una división de casos. Este algoritmo modificado es comúnmente llamado ***DPLL***. Está basado en la identidad conocida como expansión de Shannon [59]:

$$\mathcal{F} \equiv x \cdot \mathcal{F}_{(x \quad 1)} + \bar{x} \cdot \mathcal{F}_{(x \quad 0)}$$

En consecuencia, comprobar la satisfacibilidad de una fórmula $\mathcal{F}$ puede ser reducido a comprobar $\mathcal{F} \cdot x$ y $\mathcal{F} \cdot \bar{x}$ por separado. La aplicación posterior de la propagación de unidades puede reducir el tamaño de estas fórmulas significativamente. Esta transformación, aplicada recursivamente, lleva a un procedimiento completo de decisión.

2.3. Satisfacibilidad módulo teorías

Tal y como mencionamos en la sección anterior, las aplicaciones en inteligencia artificial y los métodos formales para el desarrollo de hardware y software se han beneficiado enormemente de los grandes avances en SAT. No obstante, a menudo, las aplicaciones en estos campos requieren determinar la satisfacibilidad de fórmulas en lógicas más expresivas, tales como la lógica de primer orden. A pesar de los avances realizados, los "demostradores" de teoremas de primer orden de propósito general no son capaces de resolver estas fórmulas directamente. La principal razón es que muchas aplicaciones no requieren satisfacibilidad de primer orden en general, sino satisfacibilidad respecto a alguna teoría de fondo que fije la interpretación de ciertos símbolos de predicado y de función. Por ejemplo, una aplicación que utilice aritmética entera podría estar interesada en conocer si la fórmula

$$x < y \wedge \neg(x < y + 0)$$

es satisfacible en una interpretación donde $<$ es la relación de orden usual sobre los enteros, $+$ es la adición de enteros, y 0 es la identidad de la función suma. Los métodos de razonamiento de propósito general pueden ser forzados a considerar solo interpretaciones consistentes con una teoría de fondo $\mathcal{T}$, pero solo incorporando explícitamente los axiomas para $\mathcal{T}$ en sus fórmulas de entrada. Incluso cuando esto es posible[3], el rendimiento de tales demostradores es a menudo inaceptable. Para algunas teorías de fondo, una alternativa más viable es utilizar métodos de razonamiento adaptados a la teoría en cuestión.

El campo de investigación centrado en la satisfacibilidad de estas fórmulas con respecto a alguna teoría de fondo se denomina *satisfacibilidad modulo teor as* (*SMT*). En analogía con los procedimientos de SAT, los procedimientos de SMT (ya sean de decisión o no) generalmente se conocen como *solucionadores* de SMT [6].

2.3.1. Sintaxis

En esta sección trabajaremos en el contexto de la lógica (clásica) de primer orden. Como vimos en la Sección 2.1.2, el alfabeto Σ del lenguaje formal de la lógica de predicados $\mathcal{L}$ es un conjunto de símbolos de predicado $(P, Q, \dots)$ y símbolos de función $(f, g, \dots)$ con su aridad asociada, donde los símbolos de predicado con aridad 0 son *s mbolos constantes* $(a, b, \dots)$ y los símbolos de predicado con aridad 0 son *s mbolos proposicionales* $(A, B, \dots)$. Aquí, estamos interesados principalmente en términos y fórmulas sin cuantificadores, construidos con los símbolos de un determinado alfabeto Σ. Por conveniencia, trataremos las variables (libres) de una fórmula cuantificada como constantes en una extensión adecuada de Σ. Por ejemplo, si Σ es el alfabeto de la aritmética entera, consideraremos que la fórmula $x < y + 1$ es una fórmula básica (es decir, sin variables libres) en la que x e y son símbolos constantes adicionales [6].

2.3.2. Semántica

En la Sección 2.1.2.1 se describe la noción de interpretación utilizada para adscribir significado a los lenguajes de los sistemas formales mediante la selección de un modelo, permitiendo así discutir sobre la verdad o falsedad de una fórmula. En SMT, no estamos interesados en modelos arbitrarios, sino en modelos que pertenecen a una teoría $\mathcal{T}$ dada que restringe la interpretación de los símbolos de Σ. Por lo tanto, definimos teorías en Σ (en adelante Σ-teorías) generalmente como uno o más (posiblemente infinitos)

[3] Algunas teorías de fondo, como la teoría de los números reales o la teoría de los árboles finitos, no pueden ser capturadas por un conjunto finito de fórmulas de primer orden, o, como en el caso de la teoría de la aritmética entera (con multiplicación), por ningún conjunto decidible de fórmulas de primer orden.

modelos[4], y decimos que una fórmula básica φ es satisfacible en una Σ-teoría $\mathcal{T}$ (esto es, $\mathcal{T}$-satisfacible) si y solo si hay un elemento del conjunto $\mathcal{T}$ que satisface φ.

Análogamente, un conjunto Γ de fórmulas básicas $\mathcal{T}$-implica una fórmula básica φ, denotado como $\Gamma \models_{\mathcal{T}} \varphi$, si y solo si todo modelo de $\mathcal{T}$ que satisface todas las fórmulas en Γ satisface φ también. Diremos que φ es $\mathcal{T}$-válido si y solo si $\models_{\mathcal{T}} \varphi$. Llamaremos *lema de la teoría* a una cláusula $\mathcal{C}$ si es $\mathcal{T}$-válida. Todas estas nociones se reducen exactamente a las correspondientes nociones en la lógica de primer orden estándar, escogiendo como $\mathcal{T}$ al conjunto de todos los modelos en Σ.

Normalmente, dada una Σ-teoría $\mathcal{T}$, nos interesa conocer la $\mathcal{T}$-satisfacibilidad de una fórmula básica que contiene símbolos no interpretados, es decir, símbolos de predicado o de función que no pertenecen a Σ. Este es el caso particular de los símbolos constantes no interpretados—que, como hemos visto, juegan el papel de variables libres—y de los símbolos proposicionales no interpretados—que pueden ser usados como abstracciones de otras fórmulas—. Formalmente, los símbolos no interpretados son incluidos en las definiciones anteriores considerando en lugar de $\mathcal{T}$, la teoría $\mathcal{T}^0$ definida como sigue. Sea Σ^0 un alfabeto que contiene a Σ. Una *extensión* $\mathcal{A}^0$ de un modelo $\mathcal{A}$ en Σ a Σ^0 es un modelo en Σ^0 que tiene el mismo universo que $\mathcal{A}$ y que coincide con $\mathcal{A}$ en las interpretaciones de los símbolos de Σ. La teoría $\mathcal{T}^0$ es un conjunto de todas las posibles extensiones de los modelos de $\mathcal{T}$ a Σ^0 [6].

2.3.3. Algunas teorías de interés

Muchas aplicaciones de SMT tratan con fórmulas que involucran dos o más teorías al mismo tiempo. En tal caso, la satisfacibilidad se entiende como *módulo* alguna combinación de varias teorías. En esta sección, introduciremos algunas de las teorías de interés.

2.3.3.1. Igualdad

Generalmente, una teoría impone algunas restricciones sobre cómo se pueden interpretar los símbolos de función o de predicado. Sin embargo, el caso más general es una teoría que no impone tales restricciones, en otras palabras, una teoría que incluye todos los modelos posibles para un alfabeto determinado.

Dado un alfabeto, denotaremos la teoría que incluye todos los posible modelos de dicha teoría como $\mathcal{T}_E$. También suele llamarse la teoría *vacía* dado que su axiomatización finita es el conjunto vacío $\emptyset$. El problema de satisfacibilidad para conjunciones de fórmulas básicas módulo $\mathcal{T}_E$ es decidible en tiempo polinomial utilizando un procedimiento denominado *cierre de congruencia* [4].

[4]La forma más tradicional de definir una teoría como un conjunto de axiomas se puede simular como un caso especial tomando todos los modelos en Σ de los axiomas de la teoría.

Las funciones no interpretadas son utilizadas a menudo como una técnica de abstracción para eliminar complejidad innecesaria o detalles irrelevantes a la hora de modelar un sistema. Por ejemplo, supongamos que queremos demostrar que el siguiente conjunto de literales es insatisfacible: $\{a * (f(b) + f(c)) = d, b * (f(a) + f(c)) \neq d, a = b\}$. A primera vista, podría parecer que es necesario un razonamiento en la teoría de aritmética. No obstante, si abstraemos los símbolos + y $*$ reemplazándolos por las funciones no interpretadas g y h respectivamente, obtenemos un nuevo conjunto de literales: $\{h(a, g(f(b), f(c))) = d, h(b, g(f(a), f(c))) \neq d, a = b\}$, que puede ser demostrado insatisfacible utilizando únicamente el cierre de congruencia.

2.3.3.2. Aritmética

Sea Σ_Z el alfabeto $(0, 1, +, -, \leq)$. Sea la teoría $\mathcal{T}_Z$ consistente en el modelo que interpreta estos símbolos de la manera usual sobre los enteros[5]. Podemos definir la teoría $\mathcal{T}_R$ consistente en el modelo que interpreta estos mismos símbolos de la manera usual sobre los reales.

Sea $\mathcal{T}_Z^0$ la extensión de $\mathcal{T}_Z$ con un número arbitrario de constantes no interpretadas (y similarmente para $\mathcal{T}_R^0$). La pregunta de satisfacibilidad para conjunciones de fórmulas básicas en ambas teorías es decidible. La satisfacibilidad básica en $\mathcal{T}_R^0$ es actualmente decidible en tiempo polinomial [29], aunque métodos exponenciales como los basados en el *algoritmo Simplex* consiguen un mayor rendimiento en la práctica. Por otra parte, el problema de satisfacibilidad básica en $\mathcal{T}_Z^0$ es NP-completo [50].

Una extensión obvia de la teoría de aritmética introducida hasta el momento es la inclusión de la multiplicación. Desafortunadamente, esto incrementa drásticamente la complejidad del problema, y es evitado en la práctica. De hecho, en el caso de los enteros el problema se vuelve indecidible incluso para conjunciones de fórmulas básicas [36]. En el caso de los reales, el problema es decidible pero exponencial [12].

Hay muchos usos prácticos de los procedimientos de decisión para la aritmética. En particular, al modelar y razonar sobre sistemas, la aritmética es útil para el modelado de conjuntos finitos, manipulación de punteros y memoria, restricciones en tiempo real, propiedades físicas del entorno, etcétera.

2.3.3.3. Vectores

Sea σ_A el alfabeto $(leer, escribir)$, Dado un vector a, el término $leer(a, i)$ denota el valor de a en el índice i, y el término $escribir(a, i, v)$ denota un vector que es idéntico a a excepto porque el valor en el índice i es v. Más formalmente, sea Λ_A los siguientes

[5]Por supuesto, se pueden añadir símbolos adicionales para incluir números distintos a 0 y 1 o $<$, $>$, $\geq$, pero estos no añaden expresividad alguna a la teoría, por lo que los omitiremos por simplicidad.

axiomas:

$$\forall a \forall i \forall v(leer(escribir(a, i, v), i) = v)$$
$$\forall a \forall i \forall j \forall v(i \neq j \rightarrow leer(escribir(a, i, v), j) = leer(a, j))$$

Entonces, la teoría $\mathcal{T}_A$ de *vectores* es el conjunto de todos los modelos de estos axiomas. Es común incluir además el siguiente *axioma de extensionalidad*:

$$\forall a \forall b((\forall i(leer(a, i) = leer(b, i))) \rightarrow a = b)$$

Denotaremos la teoría resultante como $\mathcal{T}_{Aex}$ La satisfacibilidad de fórmulas básicas sobre $\mathcal{T}_A^0$ o $\mathcal{T}_{Aex}^0$ es un problema NP-completo, pero se han desarrollado varios algoritmos que funcionan bien en la práctica [61]. La teoría de vectores es comúnmente utilizada para modelar las estructuras de datos de vectores en programas. También son utilizados frecuentemente como una abstracción de la memoria. La ventaja de modelar memoria utilizando vectores es que el tamaño del modelo depende del número de accesos a memoria en lugar del tamaño de la memoria modelada.

2.3.3.4. Vectores de bits de longitud fija

Una teoría natural para el razonamiento a alto nivel de circuitos y programas es la teoría de vectores de bits. Varias teorías de vectores de bits han sido propuestas y estudiadas [5, 11]. Generalmente, se utilizan símbolos de constante para representar vectores de bits, y cada símbolo de constante tiene asociado una longitud de bits que es fija para cada símbolo. Los símbolos de predicado y de función en estas teorías pueden incluir extracción, concatenación, operaciones lógicas bit a bit, y operaciones aritméticas. Para teorías no triviales de vectores de bits, es fácil ver que el problema de satisfacibilidad es NP-completo, mediante una simple reducción a SAT. Los vectores de bits proporcionan una representación más compacta y normalmente permiten resolver problemas de manera más eficiente que si fueran representados a nivel de bits [10].

2.3.3.5. Tipos de datos inductivos

Un *tipo de dato inductivo* (*IDT*) define uno o más *constructores*, y posiblemente también *selectores* y *probadores*. Un ejemplo simple es el IDT *lista*, con los constructores *cons* y *null*, los selectores *car* y *cdr*, y los probadores *is_cons* e *is_null*. El *alfabeto de primer orden* de un IDT asocia un símbolo de función a cada constructor y selector, y un símbolo de predicado a cada probador. El modelo estándar para dicho alfabeto es un modelo de términos construido utilizando únicamente los constructores. Para un IDT con un único constructor, una conjunción de literales es decidible en tiempo polinomial utilizando el algoritmo introducido por Oppen en [49]. Para un IDT más general, el problema es NP-completo, pero existen algoritmos razonablemente eficientes en la práctica [7]. Los tipos de datos inductivos son muy generales y pueden ser utilizados

para modelar una gran variedad de cosas, como por ejemplo enumeraciones, registros, tuplas, tipos de datos de programas, y sistemas de tipos.

2.4. Programación lógica

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

Definición 2.11. (Cláusula de Horn con cabeza). *Una clausula de Horn con cabeza (o clausula de nida) es una disyuncion de literales todos negados, salvo uno que se denomina cabeza. As pues, una clausula de Horn con cabeza es una clausula de la forma* $A \leftarrow B_1 \wedge \cdots \wedge B_n$, $n \geq 0$, *donde* A, B_i, $i = 1, \ldots, n$, *representan atomos.*

Definición 2.12. (Programa lógico definido, [23]). *Un programa de nido es un conjunto de clausulas de nidas* Π. *La de nicion de un s mbolo de predicado*[6] p *que aparece en* Π *es el subconjunto de clausulas de* Π *cuyas cabezas tienen como s mbolo de predicado el predicado* p.

Como semántica declarativa se utiliza un método de prueba por refutación que emplea el algoritmo de *uni cacion* 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.

2.4.1. Sustitución

Para formalizar adecuadamente los conceptos de *uni cacion*, *resolucion* y *respuesta computada*, es imprescindible introducir el concepto de sustitución.

Definición 2.13. (Sustitución, [23]). *Una sustitucion* σ *es una aplicacion que asigna, a cada variable* X *del conjunto de variables* $\mathcal{V}$ *de un lenguaje de primer orden* $\mathcal{L}$, *un termino* $\sigma(X)$ *del conjunto de los terminos* $\mathcal{T}$ *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

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

[6] A diferencia de la lógica de predicados, en la programación lógica los predicados se designan por letras minúsculas $p, q, r, \ldots$

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 $\{X_1, X_2, \ldots, X_n\}$ se denomina *dominio* de la sustitución y el conjunto $\{t_1, t_2, \ldots, t_n\}$ 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 *sustitucion identidad*, *id*, se representa mediante el conjunto vacío de elementos: {}.

Definición 2.14. (Instancia, [23]). *La aplicacion de una sustitucion $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ a una expresion E, denotado $\sigma(E)$ o $E\sigma$, se obtiene reemplazando simultaneamente cada ocurrencia de X_i en la expresion E por el correspondiente termino 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 $\sigma \circ \theta$[7] tal que $(\sigma \circ \theta)(E) = \sigma(\theta(E))$.

Ejemplo 2.3. 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)))`.

2.4.2. Unificación

La unificación de expresiones es un concepto fundamental para la programación lógica. Informalmente, *uni car* 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.

Definición 2.15. (Unificador, [23]). *Una sustitucion θ es un uni cador del conjunto de expresiones $\{E_1, E_2, \ldots, E_k\}$ si, y solo si, $\theta(E_1) = \theta(E_2) = \cdots = \theta(E_k)$. Se dice que el conjunto $E_1, E_2, \ldots, E_k$ es uni cable si existe un uni cador para el.*

[7] En ocasiones escribiremos $\sigma\theta$.

Definición 2.16. (Unificador más general, [23]). *Un uni cador σ de un conjunto de expresiones $\mathcal{S}$ es un uni cador mas general (m.g.u.[8]) para $\mathcal{S}$ si, y solo si, cualquier uni cador θ de $\mathcal{S}$ cumple que $\sigma \leq \theta$.*

Ejemplo 2.4. 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 $\mathcal{S}$.

2.4.3. 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 clausula*, que nos permite obtener una versión simplificada de la misma.

Si dos o más literales[9] (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 2.17. (Resolvente binario, [23]). *Sean $\mathcal{C}_1$ y $\mathcal{C}_2$ dos clausulas sin variables en comun. 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 clausula $(\sigma(\mathcal{C}_1)\backslash\sigma(L_1))\cup(\sigma(\mathcal{C}_2)\backslash\sigma(L_2))$ se conoce como resolvente binario de $\mathcal{C}_1$ y $\mathcal{C}_2$.*

Definición 2.18. (Resolvente, [23]). *Un resolvente de las clausulas $\mathcal{C}_1$ y $\mathcal{C}_2$ es uno de los siguientes resolventes binarios:*

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 C_2.*

Ejemplo 2.5. 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 $\mathcal{C}_1^{\emptyset} \equiv$ `p(f(Y))` $\vee$ `r(g(Y))`. Un resolvente binario de $\mathcal{C}_1^{\emptyset}$ 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 $\mathcal{C}_2$.

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

[9] Un literal es una fórmula atómica o su negación.

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.

2.4.4. Resolución SLD

La resolución SLD es una estrategia de resolución en la que, dado un programa Π y una cláusula A tal que A^0 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 2.19. (Regla de computación, [23]). *Llamamos regla de computacion (o funcion de seleccion) φ a una funcion que, cuando se aplica a un objetivo $\mathcal{G}$, selecciona uno, y solo uno, de los atomos 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:

$$\frac{(\mathcal{G} \equiv \leftarrow \mathcal{Q}_1 \wedge A^0 \wedge \mathcal{Q}_2), \varphi(\mathcal{G}) = A^0, \mathcal{C} \equiv (A \leftarrow \mathcal{Q}) \ll \Pi, \sigma = mgu(A, A^0)}{\langle \mathcal{G}; \theta \rangle \Rightarrow_{SLD} \langle \leftarrow \sigma(\mathcal{Q}_1 \wedge \mathcal{Q} \wedge \mathcal{Q}_2); \sigma\theta \rangle}$$

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 *refutacion SLD* es una derivación de éxito (es decir, que conduce a la cláusula vacía) $\langle \mathcal{G}_0; id \rangle \Rightarrow_{SLD} \langle \ \ ; \sigma \rangle$, donde σ es la *respuesta computada* en la derivación.

2.5. Lógica difusa

Repasamos en esta sección los fundamentos de la lógica difusa tal como se contemplan en [52].

Las nociones básicas de la lógica difusa han sido formuladas por Zadeh [64, 65], Goguen [17] y Pavelka [51] 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 [66], 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 polivalentes[10] 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 [65] 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 $\mu_A : \mathcal{U} \to [0,1]$ 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.

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

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

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

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

el intervalo [0,1][11]. 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 [31]. Por ejemplo, la conjunción se define habitualmente por la expresión

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

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

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

donde $\mu_{A \setminus B}(x_0)$ 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 [56]. La definición de estas funciones, en el intervalo $[0, 1]$, es la que sigue.

Definición 2.20. (Norma triangular, [48]). *Una operacion $T : [0, 1] \times [0, 1] \to [0, 1]$ es una norma triangular o t-norma si, y solo si, veri ca*

[11] De manera más general se puede tomar un retículo completo (ver Definición 2.23).

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 monotona en cada componente, es decir*[12], *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

$$\mathcal{I}(A(x_0) \vee B(x_0)) = \text{máx}\{\mathcal{I}(A(x_0)), \mathcal{I}(B(y_0))\},$$

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

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

donde $\mu_{A \text{ı} B}(x_0)$ 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 2.21. (Co-norma triangular, [48]). *Una operacion* $S : [0,1] \times [0,1] \to [0,1]$ *es una co-norma triangular o t-conorma si, y solo si, veri ca*

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

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 monotona en cada componente, es decir*[13], *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 [20], 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 [30], que damos en los siguientes términos.

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

[13] La monotonía en la segunda componente resulta también de i) y iv).

Definición 2.22. (Operador de agregación, [52]**).** *Un operador de agregacion @ es una aplicacion* $@ : [0,1]^n \to [0,1]$ *que satisface:*

i) Condiciones de frontera: $@(0,\ldots,0) = 0, @(1,\ldots,1) = 1.$

ii) Monoton a: $\forall (x_1,\ldots,x_n),(y_1,\ldots,y_n) \in [0,1]^n$,
$(x_1,\ldots,x_n) \leq (y_1,\ldots,y_n) \Rightarrow @(x_1,\ldots,x_n) \leq @(y_1,\ldots,y_n)$

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 $\{1,\ldots,n\}$ y toda n-tupla $(x_1,\ldots,x_n) \in [0,1]^n$ se cumple $@(x_1,\ldots,x_n) = @(x_{\sigma(1)},\ldots,x_{\sigma(n)})$; además, @ es idempotente (es decir, $@(x,\ldots,x) = x$) si, y solo si, para toda n-tupla $(x_1,\ldots,x_n) \in [0,1]^n$ se cumple $\text{mín}\{x_1,\ldots,x_n\} \leq @(x_1,\ldots,x_n) \leq \text{máx}\{x_1,\ldots,x_n\}$.

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 [52].

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.

Los marcos actuales para gestionar la imprecisión en programación lógica difusa pueden clasificarse en basados en anotaciones y basados en implicaciones [52]:

- Los basados en anotaciones admiten reglas de la forma $A : f(\beta_1,\ldots,\beta_n) \leftarrow B_1 : \beta_1,\ldots,B_n : \beta_n$ cuyo significado puede entenderse del modo "el grado de verdad de A es al menos $f(\beta_1,\ldots,\beta_n)$ 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 $\langle A \leftarrow \varsigma(B_1,\ldots,B_n); v\rangle$ donde v es el grado de verdad asociado a la fórmula $A \leftarrow \varsigma(B_1,\ldots,B_n)$, 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 [52].

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 [31].

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 [32].

2.6. El lenguaje FASILL

FASILL [25, 26] (*"Fuzzy Aggregators and Similarity Into a Logic Language"*) 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 *termino* es una variable, una constante o un símbolo de función f aplicado a n términos $t_1, t_2, \ldots, t_n$, y se denota como $f(t_1, t_2, \ldots, t_n)$ [27].

Asumimos también que en el lenguaje $\mathcal{L}$ se admiten elementos $r \in L$ de un retículo completo, $(L, \leq)$, equipado con una colección de conectivas (conjunciones, disyunciones y agregadores). Por lo tanto, una fórmula bien formada es:

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

Definición 2.23. (Retículo completo, [27]). *Un ret culo completo es un conjunto ordenado* $(L, \leq)$ *tal que todo subconjunto* S *de* L *tiene n mo y supremo.*[14]

[14] En particular, existe el ínfimo de L y el supremo de L, denotados por $\bot$ y $\top$, respectivamente.

$\dot{\&}_{prod}(x,y)$, xy	$\dot{\&}_{godel}(x,y)$, $\min\{x,y\}$	$\dot{\&}_{luka}(x,y)$, $\max\{0,x+y-1\}$
$\dot{\mid}_{prod}(x,y)$, $x+y-xy$	$\dot{\mid}_{godel}(x,y)$, $\max\{x,y\}$	$\dot{\mid}_{luka}(x,y)$, $\min\{x+y,1\}$
$\dot{@}_{aver}(x,y)$, $(x+y)/2$	$\dot{@}_{geom}(x,y)$, $\sqrt{xy}$	$\dot{@}_{very}(x)$, x^2

Figura 2.2: Conjunciones, disyunciones y agregadores en $([0,1],\leq)$

Ejemplo 2.6. En esta memoria utilizaremos el retículo $([0,1],\leq)$, donde $\leq$ es el orden usual; y tres conjuntos de conectivos correspondientes a las lógicas difusas de Gödel, Łukasiewicz y del producto, definidos en la Figura 2.2, con diferentes capacidades para modelar escenarios optimistas, pesimistas y realistas, respectivamente. Es posible incluir otros conectivos, por ejemplo la media aritmética, definida por el conectivo $@_{aver}$, o el agregador $@_{very}$ que es un conectivo unario o modificador lingüístico.

En lo que sigue formalizamos el concepto de relación de similitud tal como lo usamos en el lenguaje FASILL. El concepto es debido a [63].

Definición 2.24. (Relación de similitud, [27]). *Dado un dominio $\mathcal{U}$ y un ret culo L con una t-norma $\wedge$ ja, una relacion de similitud $\mathcal{R}$ es una relacion binaria difusa sobre $\mathcal{U}$, es decir, un subconjunto difuso sobre $\mathcal{U}\times\mathcal{U}$ (caracterizada por una funcion grado de pertenencia $\mathcal{R}:\mathcal{U}\times\mathcal{U}\to L$), que satisface las siguientes propiedades:*

- *Re exiva:* $\mathcal{R}(x,x)=\top, \forall x\in\mathcal{U}$.
- *Simetrica:* $\mathcal{R}(x,y)=\mathcal{R}(y,x), \forall x,y\in\mathcal{U}$.
- *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 similitudes 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 similitud $\mathcal{R}$ sobre el alfabeto de $\mathcal{L}$ se puede extender a los términos mediante inducción estructural de la forma usual [58]. La extensión, $\hat{\mathcal{R}}$, de una relación de similitud $\mathcal{R}$ se define como sigue [27]:

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 $t_1,\ldots,t_n,s_1,\ldots,s_n$ términos, $\hat{\mathcal{R}}(f(t_1,\ldots,t_n),g(s_1,\ldots,s_n))=\mathcal{R}(f,g)\wedge(\bigwedge_{i=1}^{n}\hat{\mathcal{R}}(t_i,s_i))$.
3. En cualquier otro caso, el grado de similitud de dos términos es el ínfimo $\bot$.

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 similitud más restrictiva que la definida en [58]. La idea es que una fórmula condicional $\mathcal{C}$ es similar a otra fórmula condicional $\mathcal{C}^0$ si sus cabezas son similares pero mantienen el mismo cuerpo. Así, dados $\mathcal{C} \equiv A \leftarrow \mathcal{B}$ y $\mathcal{C}^0 \equiv A^0 \leftarrow \mathcal{B}^0$, $\hat{\mathcal{R}}(\mathcal{C}, \mathcal{C}^0) = \hat{\mathcal{R}}(A, A^0)$ si $\mathcal{B} \equiv \mathcal{B}^0$; y en cualquier otro caso, $\hat{\mathcal{R}}(\mathcal{C}, \mathcal{C}^0) = \bot$.

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

Ejemplo 2.7. La relación de similitud $\mathcal{R}$ sobre $\mathcal{U} = \{$*vanguardist, elegant, metro, taxi, bus*$\}$ está definida mediante la matriz mostrada en la Tabla 2.2, donde se puede compro-

Tabla 2.2: Grados de similitud de la relación del Ejemplo 2.7

$\mathcal{R}$	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

bar que $\mathcal{R}$ satisface las propiedades reflexiva, simétrica y transitiva. Particularmente, utilizando como t-norma $\wedge$ la conjunción de Gödel, tenemos que: $\mathcal{R}(\texttt{metro}, \texttt{taxi}) \geq \mathcal{R}(\texttt{metro}, \texttt{bus}) \wedge \mathcal{R}(\texttt{bus}, \texttt{taxi}) = 0.5 \wedge 0.4$.
Además, la extensión $\hat{\mathcal{R}}$ de la relación $\mathcal{R}$ determina que los términos `elegant(taxi)` y `vanguardist(metro)` son similares, pues: $\hat{\mathcal{R}}$(`elegant(taxi)`,`vanguardist(metro)`) $= \mathcal{R}$(`elegant`,`vanguardist`) $\wedge \hat{\mathcal{R}}$(`taxi`,`metro`) $= 0.6 \wedge \mathcal{R}$(`taxi`,`metro`) $= 0.6 \wedge 0.4 = 0.4$.

Definición 2.25. (Regla FASILL, [27]). *Una regla FASILL es de la forma $A \leftarrow \mathcal{B}$, donde A es una formula atomica llamada cabeza y $\mathcal{B}$ es una formula bien formada (construida a partir de formulas atomicas $B_1, B_2, \ldots, B_n$, 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 ret culo L), esta regla se llama hecho y puede ser escrita como $A \leftarrow r$ (o simplemente como A si $r = \top$).*

Definición 2.26. (Programa FASILL, [27]). *Un programa FASILL es una 3-tupla $\langle \Pi, \mathcal{R}, L \rangle$ donde Π es un conjunto de reglas, $\mathcal{R}$ es una relacion de similitud entre los elementos de Σ, y L es un ret culo completo.*

Ejemplo 2.8. El conjunto de reglas Π mostrado a continuación, la relación de similitud $\mathcal{R}$ del Ejemplo 2.7, y el retículo L del Ejemplo 2.6 forman un programa FASILL $\mathcal{P} =$

$\langle \Pi, \mathcal{R}, L \rangle$.

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

2.6.1. Semántica operacional de FASILL

Las reglas en un programa FASILL tienen el mismo papel que las cláusulas en los programas Prolog o MALP [24, 37], 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 2.4. Algunos de estos se han extendido para trabajar con relaciones de similitud. Concretamente, siguiendo la línea adoptada en Bousi∼Prolog [28], el unificador más general se reemplaza por el concepto de *unificador mas general debil* (*w.m.g.u.*) y se introduce el *algoritmo de unificacion debil* para calcularlo. Este algoritmo especifica que dos expresiones (términos o fórmulas atómicas) $f(t_1, t_2, \ldots, t_n)$ y $g(s_1, s_2, \ldots, s_n)$ unifican débilmente si los símbolos de la raíz, f y g, están relacionados con un determinado valor de verdad ($\mathcal{R}(f, g) = r > \bot$) 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 ($f \not\equiv g$).

De manera más técnica, el algoritmo de unificación débil que usaremos es una extensión del primero que aparece en [58] para retículos completos arbitrarios. Será formalizado como un sistema de transición de estados soportado por una relación de unificación basada en similitud "⇒". 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 $(L, \leq)$: $\langle G, id, \alpha_0 \rangle \Rightarrow \langle G_1, \theta_1, \alpha_1 \rangle \Rightarrow \cdots \Rightarrow \langle G_n, \theta_n, \alpha_n \rangle$. 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 unificacion* 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 ($G_n = Fallo$).

La *relacion de unificacion basada en similitud*, "⇒", 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) [27]:

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

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

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

$$\frac{\langle\{f(t_1,\ldots,t_n)\approx g(s_1,\ldots,s_n)\}\cup E,\theta,r_1\rangle \qquad \mathcal{R}(f,g)=\bot}{\langle Fallo,\theta,r_1\rangle}\ (6)$$

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 $\bot$, 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, $\langle\{\mathcal{E}_1\approx\mathcal{E}_2\},id,\top\rangle\Rightarrow^\star\langle\emptyset,\theta,r\rangle$, entonces podemos decir que $wmgu(\mathcal{E}_1,\mathcal{E}_2)=\langle\theta,r\rangle$, siendo θ el *unificador debil mas general* de $\mathcal{E}_1$ y $\mathcal{E}_2$, y r es su *grado de unificacion*.

Finalmente, nótese que, en general, un w.m.g.u. de dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ no es único [58]. Ciertamente, el algoritmo de unificación débil únicamente calcula una clase representativa de un w.g.m.u., en el sentido de que, si $\theta=\{x_1/t_1,x_2/t_2,\ldots,x_n/t_n\}$ es un w.m.g.u. con grado β, entonces, por definición, cualquier sustitución $\theta^\emptyset=\{x_1/s_1,x_2/s_2,\ldots,x_n/s_n\}$ que satisfaga $\mathcal{R}(s_i,t_i)>\bot$ para cualquier $1\leq i\leq n$, es también un w.m.g.u. con un grado de aproximación $\beta^\emptyset=\beta\wedge(\bigwedge_1^n\mathcal{R}(s_i,t_i))$, 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 2.9. Considérese el retículo $L=([0,1],\leq)$ del Ejemplo 2.6 y la relación de similitud $\mathcal{R}$ del Ejemplo 2.7. Es posible el siguiente proceso de unificación débil, dados los términos `elegant(taxi)` y `vanguardist(metro)`.

$\langle\{\texttt{elegant(taxi)}\approx\texttt{vanguardist(metro)}\},id,\texttt{1}\rangle\overset{R1}{\Longrightarrow}\langle\{\texttt{taxi}\approx\texttt{metro}\},id,\texttt{0.6}\rangle\overset{R1}{\Longrightarrow}$
$\langle\{\},id,\texttt{0.6}\wedge\texttt{0.4}\rangle=\langle\{\},id,\texttt{0.4}\rangle$

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

$$\langle\{\texttt{elegant(taxi)} \approx \texttt{vanguardist(X)}\}, id, \texttt{1}\rangle \overset{R1}{\Longrightarrow} \langle\{\texttt{taxi} \approx X\}, id, \texttt{0.6}\rangle \overset{R4}{\Longrightarrow}$$
$$\langle\{X \approx \texttt{taxi}\}, id, \texttt{0.6}\rangle \overset{R3}{\Longrightarrow} \langle\{\}, \{X/taxi\}, \texttt{0.6}\rangle$$

y la sustitución $\{X/taxi\}$ 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 $\mathcal{C}[A/A']$

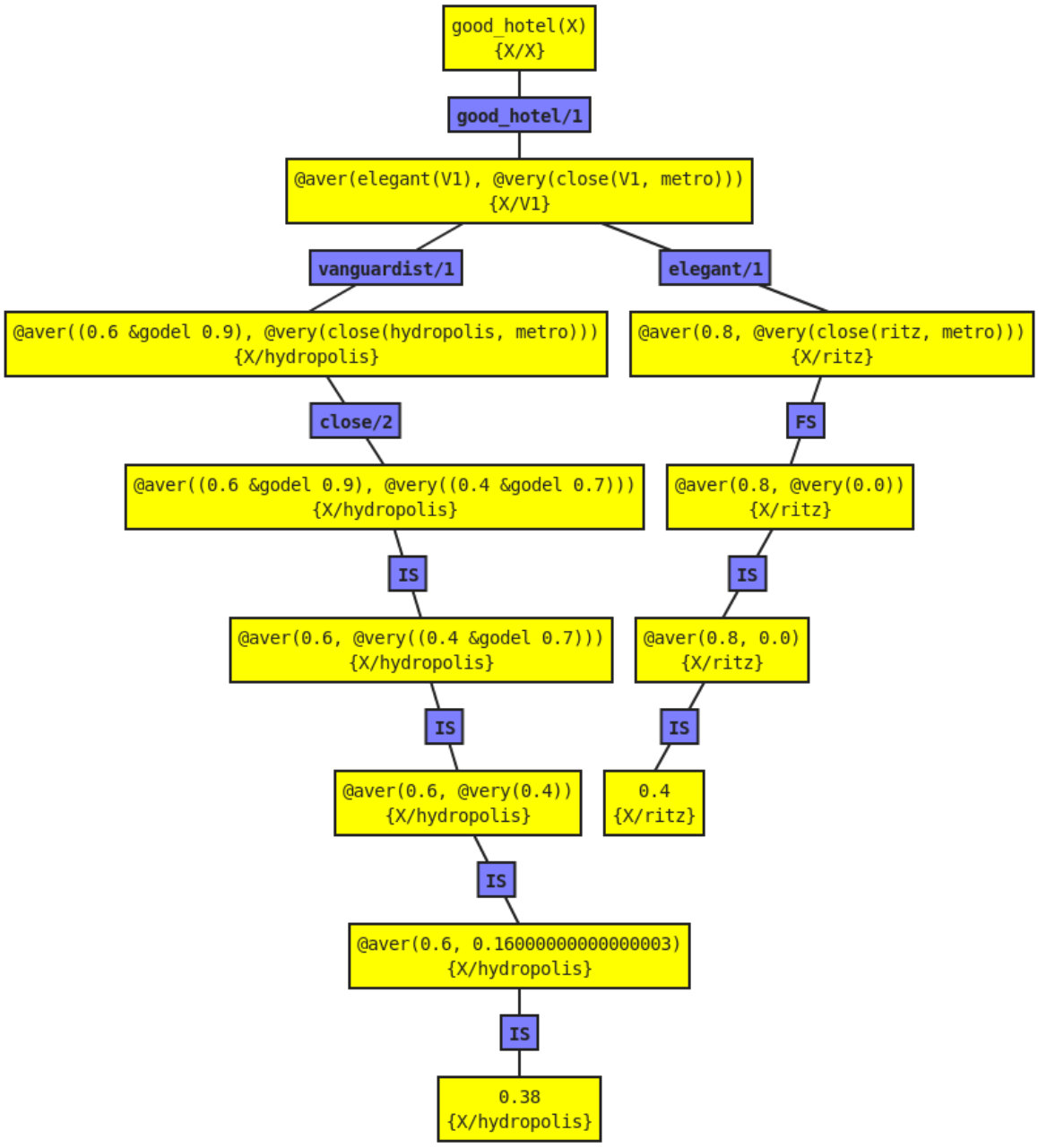


Figura 2.3: Árbol de derivación para el Ejemplo 2.10

significa el reemplazo de A por A^{θ} 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 2.27. (Paso de computación, [27]). *Sea $\mathcal{Q}$ un objetivo y σ una sustitucion, y sea A el atomo 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 computacion se formaliza como un sistema de transicion de estados, cuya relacion de transicion de estados es la menor relacion binaria que satisface las siguientes reglas:*

1) Paso de exito (denotado como ^{SS}):

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

2) Paso de fallo (denotado como ^{FS}):

$$\frac{\langle \mathcal{Q}[A]; \sigma \rangle \qquad @A^{\theta} \leftarrow \mathcal{B} \in \Pi : wmgu(A, A^{\theta}) = \langle \theta; r \rangle, r > \bot}{\langle \mathcal{Q}[A/\bot]; \sigma \rangle} \text{ FS}$$

3) Paso interpretativo (denotado como ^{IS}):

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

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

Ejemplo 2.10. Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ el programa del Ejemplo 2.8. Es posible realizar una derivación con la respuesta difusa computada $\langle \texttt{0.4}; \{X/ritz\} \rangle$ para $\mathcal{P}$ y el objetivo $\mathcal{Q} =$ `good_hotel(X)`:

$\langle$ `good_hotel(X)`$; id \rangle$ — SS^{R4}

$\langle$ `@aver(elegant(X),@very(close(X,metro)))`$; \{X_1/X\} \rangle$ — SS^{R2}

$\langle$ `@aver(0.8,@very(close(ritz,metro)))`$; \{X_1/ritz, X/ritz\} \rangle$ — FS

$\langle$ `@aver(0.8,@very(0))`$; \{X_1/ritz, X/ritz\} \rangle$ — IS

$\langle$ `@aver(0.8,0)`$; \{X_1/ritz, X/ritz\} \rangle$ — IS

$\langle$ `0.4`$; \{X_1/ritz, X/ritz\} \rangle$

Aparte de esta, existe otra fca alternativa $\langle \texttt{0.38}; \{X/hydropolis\} \rangle$ asociada al mismo objetivo. En la Figura 2.3 se muestra una representación gráfica del árbol de derivación para este objetivo.

```
fasill> consult('../sample/program/good_hotel.fpl').
< 1.0, {} >  ;

fasill> good_hotel(X).
< 0.4, {X/ritz} >  ;

fasill> consult_sim('../sample/sim/good_hotel.sim.pl').
< 1.0, {} >  ;

fasill> good_hotel(X).
< 0.38, {X/hydropolis} >  ;
< 0.4, {X/ritz} >  ;

fasill> findall(hotel(X,Y), truth_degree(good_hotel(X),Y), L).
< 1.0, {X/X, Y/Y, L/[hotel(hydropolis, 0.38), hotel(ritz, 0.4)]} >  ;

fasill> taxi = metro ; taxi ~ metro.
< 0.0, {} >  ;
< 0.4, {} >  ;
```

Figura 2.4: Consola interactiva del lenguaje FASILL

2.6.2. Implementación de FASILL

FASILL es un lenguaje interpretado que ha sido desarrollado sobre Prolog, y que actualmente es ejecutado sobre el intérprete SWI-Prolog. Su implementación está disponible a través de la URL http://dectau.uclm.es/fasill/ y su código puede encontrarse en el siguiente repositorio de GitHub: https://github.com/jariazavalverde/fasill. FASILL cuenta con una consola interactiva (véase la Figura 2.4) que provee a los programadores de una forma rápida de ejecutar objetivos lógicos difusos. Además, cuenta con un entorno de prueba online para ejecutar la herramienta sin necesidad de instalarla: http://dectau.uclm.es/fasill/sandbox.

En la Figura 2.5 se muestra la zona de entrada del entorno FASILL online, la cual está formada por tres campos de texto para introducir las reglas, el retículo y, opcionalmente, un conjunto de ecuaciones de similitud a partir del cual se genera la relación de similitud calculando su cierre reflexivo, simétrico y transitivo. Además, hay un cuarto campo de texto para fijar el límite de inferencias que el entorno puede realizar para encontrar respuestas computadas difusas, con el fin de evitar derivaciones infinitas.

Una vez introducido el programa, podemos consultar un objetivo en la zona de ejecución mostrada en la Figura 2.6. El entorno proporciona el conjunto de respuestas computadas difusas junto al tiempo de ejecución, así como el árbol de derivación en formato textual. Además, también genera un gráfico en forma de árbol que representa gráficamente la derivación (véase la Figura 2.3).

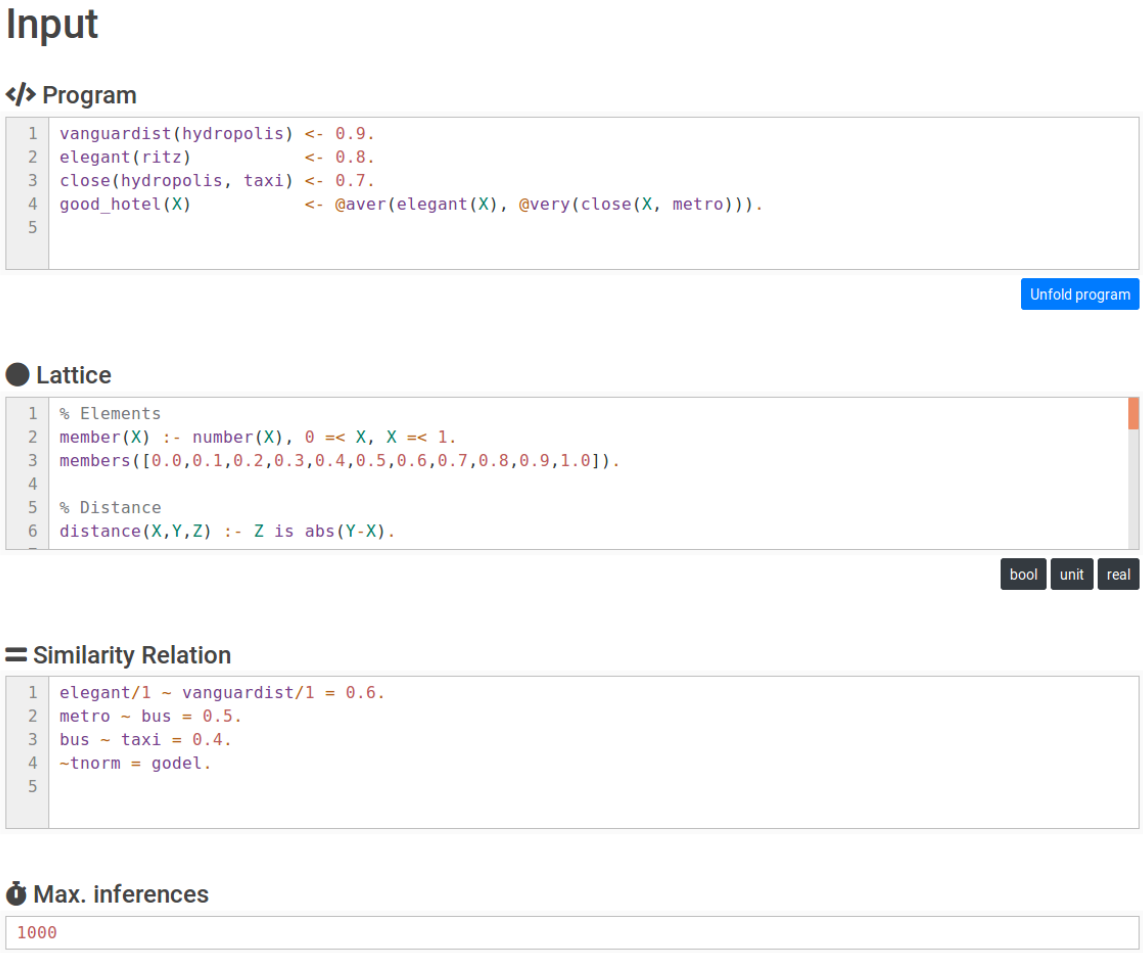


Figura 2.5: Captura de pantalla de la entrada de la herramienta FASILL online

2.7. El solucionador Z3

Z3 es un solucionador SMT de Microsoft Research, originalmente orientado a resolver problemas que surgen en la verificación y análisis de programas. Consecuentemente, integra una gran variedad de teorías utilizando algoritmos novedosos para la combinación de las mismas [45]. El primer lanzamiento externo de Z3 fue realizado en septiembre de 2007.

Z3 soporta varios formatos textuales de entrada, y dispone de una interfaz binaria. Los tres formatos de entrada textuales compatibles son: el formato SMT-LIB [8], el formato Simplify [15], y un formato nativo a bajo nivel al estilo del formato DIMACS[15] para fórmulas proposicionales SAT. También es posible llamar a Z3 proceduralmente utilizando el API para los lenguajes soportados por .NET, y para otros lenguajes como ANSI C, Haskell, OCaml o Python [46].

[15]DIMACS está ampliamente aceptado como el formato estándar para fórmulas lógicas en CNF.

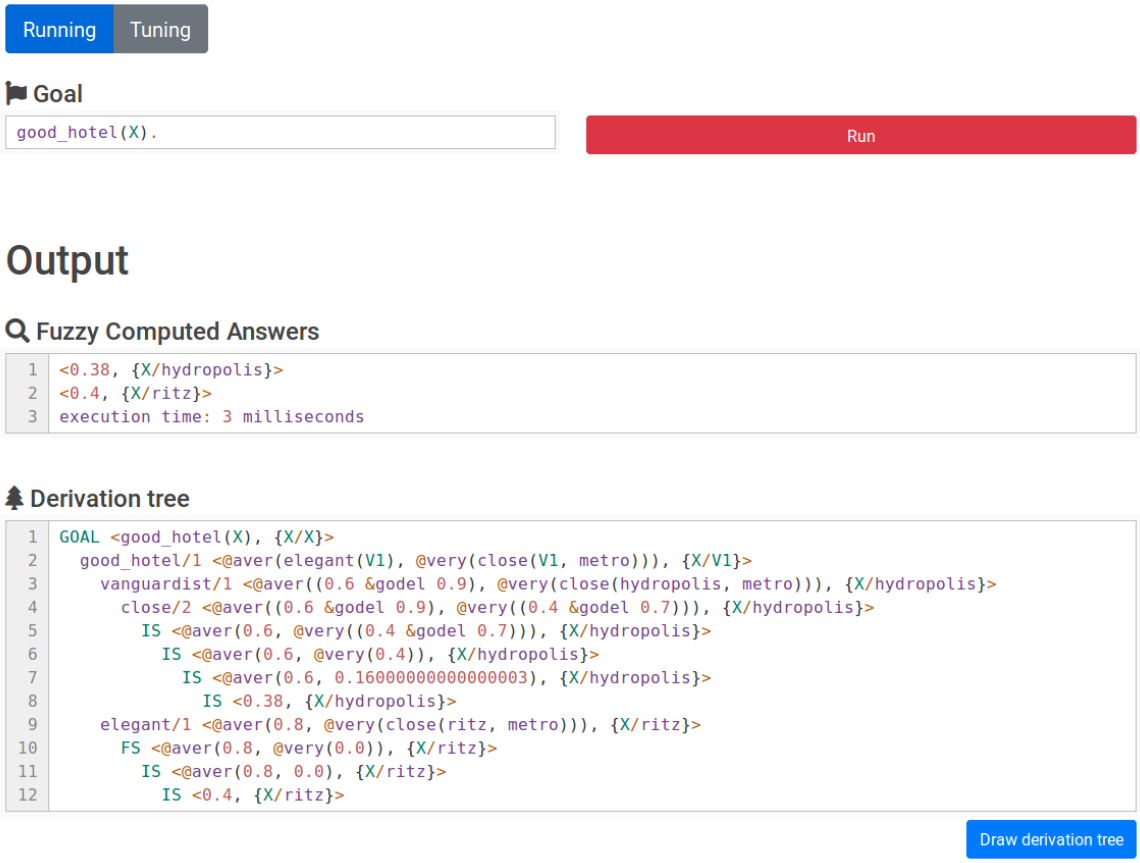


Figura 2.6: Captura de pantalla de la herramienta FASILL online tras ejecutar un objetivo

2.7.1. Arquitectura de Z3

Z3 integra un moderno solucionador SAT basado en DPLL, un *solucionador de teor a central* que maneja igualdades y funciones no interpretadas, *solucionadores satelite* (para aritmética, vectores, etcétera) y una *maquina abstracta de E-emparejamiento* (para cuantificadores). Z3 está implementado en C++. En la Figura 2.7 se muestra una descripción esquemática de Z3, cuyas principales componentes y características se describen a continuación tal y como se detallan en [46].

- **Simplificador.** Las fórmulas de entrada primero son procesadas utilizando una simplificación incompleta pero eficiente. El simplificador aplica reglas de reducción de expresiones algebraicas, como $p \wedge \texttt{true} \mapsto p$, pero también realiza simplificaciones contextuales limitadas, ya que identifica definiciones ecuacionales en un contexto y reduce la fórmula restante utilizando la definición, por ejemplo $x = 4 \wedge q(x) \mapsto x = 4 \wedge q(4)$. El conjuntor trivialmente satisfacible $x = 4$ no se compila en el núcleo, sino que se mantiene a un lado en el caso de que se requiera un modelo para evaluar x.

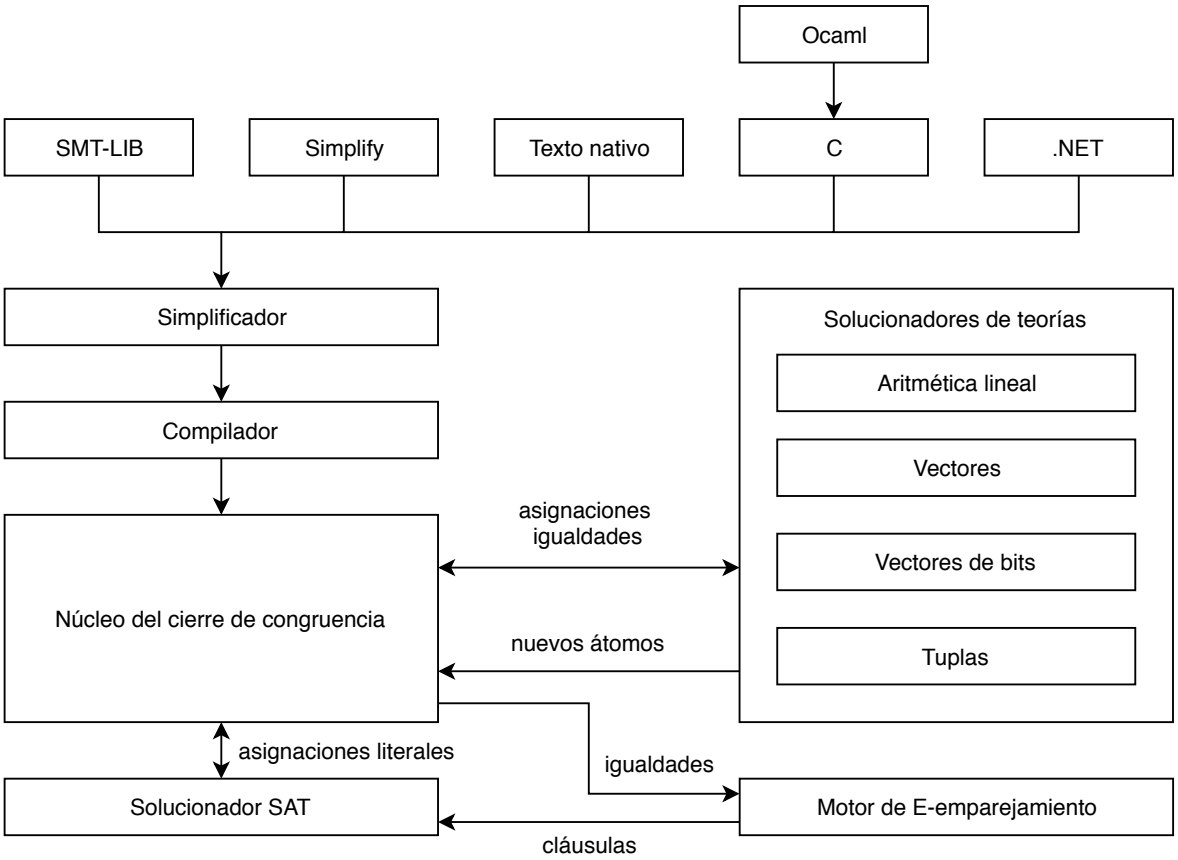


Figura 2.7: Arquitectura del solucionador Z3

- **Compilador.** La representación simplificada del árbol de sintaxis abstracta de la fórmula se convierte a una estructura de datos diferente que comprende un conjunto de cláusulas y nodos de cierre de congruencia.

- **Núcleo del cierre de congruencia.** El núcleo del cierre de congruencia recibe asignaciones de verdad para los átomos del solucionador SAT. Las igualdades asertadas por el solucionador SAT son propagadas por el núcleo del cierre de congruencia utilizando una estructura de datos llamada *E-grafo* [15]. Los nodos del E-grafo pueden apuntar a varios solucionadores de teorías. Cuando dos nodos se juntan, los conjuntos de referencias de los solucionadores de teorías se juntan, y la unión se propaga como una igualdad a los solucionadores de teorías en la intersección de los dos conjuntos de referencias de los solucionadores. El núcleo también propaga los efectos de los solucionadores de teorías, como por ejemplo las igualdades inferidas que se producen y los átomos asignados a verdadero o falso. Los solucionadores de teorías también pueden producir átomos frescos en el caso de las teorías no convexas, cuyos valores son posteriormente asignados por el solucionador SAT.

- **Combinación de teorías.** Los métodos tradicionales para combinar soluciona-

dores de teorías dependen de las capacidades de los solucionadores para producir todas las igualdades implicadas o un paso de preproceso que introduce literales adicionales en el espacio de búsqueda. Z3 utiliza un nuevo método de combinación de teorías que recoge incrementalmente los modelos mantenidos por cada teoría [45].

- **Solucionador SAT.** Las particiones lógicas son controladas por un solucionador SAT considerado estado del arte, que integra métodos de poda de búsqueda, aprendizaje utilizando cláusulas conflictivas, y vuelta atrás no cronológica.

- **Eliminación de cláusulas.** La instanciación de cuantificadores tiene el efecto secundario de producir nuevas cláusulas en el espacio de búsqueda, que contienen nuevos átomos. El recolector de basura de Z3 recoge cláusulas, junto con sus átomos y términos, que son inútiles en las ramas cercanas. No obstante, las cláusulas conflictivas y sus literales no son eliminados, por lo que las instanciaciones de cuantificadores que fueron útiles produciendo conflictos son retenidas como efecto secundario.

- **Propagación de relevancia.** Los solucionadores basados en DPLL asignan valores lógicos a potencialmente todos los átomos que aparecen en el objetivo. En la práctica, muchos de estos átomos no son relevantes. Z3 ignora estos átomos para teorías muy costosas, como la teoría de vectores de bits. El algoritmo para discriminar los átomos relevantes es descrito en [44].

- **Instanciación de cuantificadores mediante E-emparejamiento.** Z3 utiliza una aproximación bien conocida para el razonamiento de cuantificadores que funciona sobre un E-grafo para instanciar variables cuantificadas. Z3 utiliza nuevos algoritmos que identifican coincidencias en E-grafos incremental y eficientemente.

- **Solucionadores de teorías.** Z3 utiliza un solucionador de aritmética lineal basado en el algoritmo de Yices descrito en [16]. La teoría de vectores utiliza instanciación perezosa de los axiomas de vectores. La teoría de vectores de bits con longitud fija aplica técnicas de aplanamiento (*bit-blasting*) a todas las operaciones vectoriales, excepto a la de igualdad.

- **Generación de modelos.** Z3 puede producir modelos como parte de su salida. Los modelos asignan valores a las constantes de las entradas y generan funciones parciales para los símbolos de predicado y de función.

Tabla 2.3: Símbolos terminales del lenguaje SMT-LIB

Símbolo	**Expresión regular**
`lpar`	`/\(/`
`rpar`	`/\)/`
`whitespace`	`/[\n\t\r]+|;[^\n]*\n/`
`numeral`	`/0|[1-9][0-9]*/`
`decimal`	`/0|[1-9][0-9]*\.[0-9]*/`
`hexadecimal`	`/#x[a-fA-F0-9]+/`
`binary`	`/#b[01]+/`
`string`	`/"([^"]|)*"/`
`keyword`	`/:[a-zA-Z~!@$%^&*_-+=<>.?][a-zA-Z0-9~!@$%^&*_-+=<>.?]*/`
`symbol`	`/[a-zA-Z~!@$%^&*_-+=<>.?][a-zA-Z0-9~!@$%^&*_-+=<>.?]*/`
`symbol`	`/|[^|\\]+|/`

2.7.2. El lenguaje SMT-LIB

Tal y como mencionamos anteriormente, Z3 soporta varios formatos textuales de entrada, entre los que se encuentra el formato SMT-LIB. Este es el formato que utilizaremos en este TFM, por lo tanto en esta sección introducimos la sintaxis de SMT-LIB descrita en [8], centrándonos especialmente en la fase de análisis léxico.

Los archivos fuente de SMT-LIB consisten en caracteres Unicode en cualquier codificación de 8 bits, como UTF-8. La mayoría de los componentes léxicos (*tokens*) están limitados al conjunto de caracteres US-ASCII imprimibles. El resto de caracteres están permitidos en cadenas literales, símbolos entrecomillados y comentarios. Tanto los comentarios como los espacios en blanco son considerados elementos terminales del tipo `whitespace`. La única función léxica de los espacios en blanco es separar el texto de entrada en otros componentes léxicos.

Los elementos léxicos terminales del lenguaje son los paréntesis `(` y `)`, los componentes `whitespace`, `numeral`, `decimal`, `hexadecimal`, `binary`, `string`, `symbol` y `keyword`, así como un número de palabras reservadas, todos ellos definidos a continuación. En la Tabla 2.3 se muestran todos los símbolos terminales del lenguaje junto a su especificación como expresiones regulares con sintaxis PCRE[16].

- **Espacios en blanco.** Un token `whitespace` es una secuencia de caracteres cualesquiera comenzando por el carácter punto y coma `;` y terminando con un carácter de salto de línea; o una secuencia formada por cualquier combinación de los caracteres `\t` (tabulador), `\n` (salto de línea), `\r` (retorno de carro), y ␣ (espacio).

[16] Perl-compatible regular expressions.

- **Números enteros.** Un token `numeral` es el carácter `0` o una secuencia no vacía de dígitos (caracteres entre `0` y `9`) empezando por un carácter distinto de `0`.
- **Números reales.** Un token `decimal` es un token `numeral` seguido de un carácter punto `.`, cero o más caracteres `0`, y otro `numeral`.
- **Números hexadecimales.** Un token `hexadecimal` es una secuencia no vacía *insensible a mayusculas y minusculas* de dígitos (caracteres entre `0` y `9`) y letras desde la `A` hasta la `F`, precedida por los caracteres (sensible a mayúsculas y minúsculas) `#x`.
- **Números binarios.** Un token `binary` es una secuencia no vacía de caracteres `0` y `1` precedida por los caracteres `#b`.
- **Cadenas literales.** Un token `string` es cualquier secuencia de caracteres imprimibles delimitada por el carácter de doble comilla `"`. El carácter `"` puede aparecer dentro de una cadena literal únicamente si está duplicado. Es decir, el analizador léxico debería tratar la secuencia `""` como una secuencia de escape denotando una única ocurrencia de `"` dentro del literal.
- **Palabras reservadas.** El lenguaje utiliza una serie de palabras reservadas, secuencias de caracteres imprimibles que deben ser tratadas como tokens individuales. El conjunto básico de estas palabras es el siguiente: `BINARY`, `DECIMAL`, `HEXADECIMAL`, `NUMERAL`, `STRING`, `_`, `!`, `as`, `let`, `exists`, `forall`, `match` y `par`. Adicionalmente, cada nombre de comando (`set-logic`, `set-option`, etcétera) es también una palabra reservada.
- **Símbolos.** Un token `symbol` es un símbolo simple o un símbolo entrecomillado. Un *símbolo simple* es cualquier secuencia no vacía de letras (caracteres entre `a` y `z` y entre `A` y `Z`) dígitos (caracteres entre `0` y `9`), y los caracteres `~ ! @ $ % ^ & * _ - + = < > . ? /`, que no comienza por un dígito y no es una palabra reservada. Un *símbolo entrecomillado* es cualquier secuencia de caracteres imprimibles que comienza y termina con el carácter `|` y que no contiene `|` o `\`.
- **Palabras clave.** Un token `keyword` es el carácter dos puntos `:` seguido de un símbolo simple.

La sintaxis del lenguaje SMT-LIB es similar a la del lenguaje de programación Lisp. De hecho, cada expresión en SMT-LIB es una S-expresión[17] legal de Common Lisp [60]. La elección de la sintaxis de S-expresiones y el diseño concreto de la sintaxis de SMT-LIB fue principalmente dirigido por el objetivo de simplificar el análisis.

[17] Notación en forma de texto para representar una estructura de datos de árbol, basada en listas anidadas, donde cada sublista es un subárbol.

$$
\begin{array}{rcl}
\langle Program \rangle & \rightarrow & \langle S-expression \rangle \langle Program \rangle \mid \lambda \\
\langle S-expression \rangle & \rightarrow & \texttt{whitespace} \mid \texttt{numeral} \mid \texttt{decimal} \mid \texttt{hexadecimal} \mid \\
 & & \texttt{binary} \mid \texttt{string} \mid \texttt{symbol} \mid \texttt{keyword} \mid \\
 & & \texttt{lpar}\ (\langle S-expression \rangle^{*})\ \texttt{rpar}
\end{array}
$$

Figura 2.8: Reglas de producción del lenguaje SMT-LIB

Una S-expresión es un elemento terminal sin paréntesis o una secuencia (posiblemente vacía) de S-expresiones encerradas entre paréntesis. Un programa en Z3 es una secuencia de S-expresiones. En [8] se detallan las reglas de producción necesarias para cubrir la sintaxis completa del lenguaje. Desde el punto de vista de este trabajo, únicamente estamos interesados en analizar los modelos devueltos por el solucionador Z3, por lo que las reglas de producción mostradas en la Figura 2.8 son suficientes.

2.7.3. Introducción a Z3

Z3 es una herramienta de bajo nivel que se utiliza principalmente como un componente en el contexto de otras herramientas que requieren resolver fórmulas lógicas. En consecuencia, Z3 expone una serie de facilidades por medio de su API para que sea más conveniente para las herramientas comunicarse con él, pero no hay editores independientes ni instalaciones centradas en el usuario para interactuar con Z3.

En esta sección reproducimos las principales características de Z3 necesarias para la comprensión del Capítulo 4 tal y como se contemplan en [38]. Cabe mencionar que Z3 cuenta con muchas otras características que no son mencionadas en esta memoria.

2.7.3.1. Comandos básicos

Como hemos visto en la sección anterior, la entrada de Z3 es una extensión del estándar SMT-LIB 2.0. Un guión (*script*) de Z3 es una secuencia de comandos. El comando `help` muestra una lista de todos los comandos disponibles. El comando `echo` muestra un mensaje. Internamente, Z3 mantiene una pila de fórmulas y declaraciones introducidas por el usuario. Diremos que estas son las aserciones introducidas por el usuario. El comando `declare-cons` declara una constante de un determinado tipo (*sort*). Análogamente, el comando `declare-fun` declara una función. En el siguiente ejemplo, declaramos una función que recibe un dato entero y un dato lógico, y devuelve un dato entero.

```
1 (echo "empezando con Z3...")
2 (declare-const a Int)
3 (declare-fun f (Int Bool) Int)
```

El comando `assert` añade una fórmula a la pila interna de Z3. Diremos que las fórmulas en la pila de Z3 son satisfacibles si existe una interpretación (para las constantes y funciones declaradas por el usuario) que hace verdad todas las fórmulas asertadas.

```
1 (declare-const a Int)
2 (declare-fun f (Int Bool) Int)
3 (assert (> a 10))
4 (assert (< (f a true) 100))
5 (check-sat)
6 (get-model)
```

La primera aserción (línea 3) declara una fórmula especificando que la constante `a` debe ser mayor que `10`. La segunda fórmula (línea 4) declara que la función `f` aplicada a `a` y al valor `true` debe devolver un valor menor a `100`. El comando `check-sat` determina si las fórmulas actuales en la pila de Z3 son satisfacibles o no. Si las fórmulas son satisfacibles, Z3 devuelve `sat`. Si no son satisfacibles, entonces devuelve `unsat`. Z3 también puede devolver `unknown` cuando no puede determinar si una fórmula es satisfacible o no.

Cuando `check-sat` devuelve `sat`, es posible utilizar el comando `get-model` para recuperar la interpretación que hace verdaderas todas las fórmulas de la pila interna de Z3. Para el ejemplo anterior, obtendríamos la siguiente respuesta.

```
1 sat
2 (model
3     (define-fun a () Int 11)
4     (define-fun f ((x!1 Int) (x!2 Bool)) Int
5         (ite (and (= x!1 11) (= x!2 true)) 0 0)
6     )
7 )
```

La interpretación se proporciona por medio de definiciones. Por ejemplo, la definición `(define-fun a () Int [val])` declara que el valor de `a` en el modelo es `[val]`. La definición `(define-fun f ((x!1 Int) (x!2 Bool)) Int ...)` es muy similar a la definición de una función utilizada en un lenguaje de programación. En este ejemplo, `x!1` y `x!2` son los argumentos de la interpretación de la función creada por Z3. Para este ejemplo simple, la definición de la función `f` está basada en expresiones condicionales o `ite`'s (*if-then-elses*). Por ejemplo, la expresión `(ite (and (= x!1 11) (= x!2 false)) 21 0)` se evalúa a (devuelve) `21` si `x!1` es igual a `11` y `x!2` es igual a `false`. En cualquier otro caso, devuelve `0`.

2.7.3.2. Constantes y funciones no interpretadas

Los componentes básicos de las fórmulas SMT son las constantes y las funciones. Las constantes son tan solo funciones que no toman argumentos. Por lo tanto, todo es realmente una función. En el ejemplo anterior, podríamos haber declarado la constante `a` como `(declare-fun a () Int)`, y `Z3` habría proporcionado el mismo modelo. De hecho, en los modelos devueltos por `Z3` se definen directamente las constantes como funciones sin argumentos.

Las funciones y los símbolos constantes en la lógica pura de primer orden no se interpretan (son libres), lo que significa que no se adjunta ninguna interpretación a priori. Esto contrasta con las funciones que pertenecen al alfabeto de las teorías, como la aritmética, donde la función `+` tiene una interpretación estándar fija (suma dos números). Las funciones y constantes no interpretadas son muy flexibles, permitiendo cualquier interpretación que sea consistente con las restricciones sobre la función o constante.

2.7.3.3. Aritmética

`Z3` tiene soporte integrado para constantes enteras y reales. Estos dos tipos, `Int` y `Real`, representan los enteros y los reales desde el punto de vista matemático. Las fórmulas pueden incluir operadores aritméticos tales como `+`, `-` y `<`. El comando `check-sat` le indicará a `Z3` que intente encontrar una interpretación para las constantes declaradas que haga que todas las fórmulas sean verdaderas. La interpretación consiste básicamente en asignar un número a cada constante.

```
1  (declare-const a Int)
2  (declare-const b Int)
3  (declare-const c Int)
4  (declare-const d Real)
5  (declare-const e Real)
6  (assert (> a (+ b 2)))
7  (assert (= a (+ (* 2 c) 10)))
8  (assert (<= (+ c b) 1000))
9  (assert (>= d e))
10 (check-sat)
11 (get-model)
```

Para el ejemplo anterior, obtendríamos el siguiente modelo de `Z3`, donde todas las variables toman el valor `0` o `0.0`, excepto `a` que toma el valor `10`. Las constantes reales deben contener un punto decimal. A diferencia de la mayoría de los lenguajes de

programación, Z3 no convierte automáticamente enteros en reales (y viceversa).[18]

```
1 sat
2 (model
3     (define-fun b () Int 0)
4     (define-fun c () Int 0)
5     (define-fun e () Real 0.0)
6     (define-fun d () Real 0.0)
7     (define-fun a () Int 10)
8 )
```

2.7.3.4. Aritmética no lineal

Decimos que una fórmula no es lineal si contiene expresiones de la forma `(* t s)`, donde `t` y `s` no son números. La aritmética real no lineal es muy cara computacionalmente, y Z3 no es completo para este tipo de fórmulas. El comando `check-sat` puede devolver `unknown` o `loop`. La aritmética de enteros no lineal es indecidible: no hay ningún procedimiento que sea correcto y que finalice (para cada entrada) con una respuesta `sat` o `unsat`. Nótese que esto no impide que Z3 devuelva una respuesta para muchos problemas no lineales. El límite real es que siempre habrá una fórmula aritmética de enteros no lineal que fallará a la hora de producir una respuesta.

```
1 (declare-const a Int)
2 (declare-const b Real)
3 (declare-const c Real)
4 (assert (> (* a a) 3))
5 (assert (= (+ (* b b b) (* b c)) 3.0))
6 (check-sat)
```

En este ejemplo, Z3 devuelve la respuesta `unknown`. No obstante, cuando solo hay restricciones no lineales sobre reales, Z3 utiliza un solucionador especializado que sí es completo. Si eliminamos la primera restricción (línea 4) y solicitamos el modelo, obtenemos el siguiente resultado.

```
1 sat
2 (model
3     (define-fun b () Real (/ 1.0 8.0))
4     (define-fun c () Real (/ 1535.0 64.0))
5 )
```

[18] La función `to-real` puede utilizarse para convertir una expresión entera en una expresión real.

Para evitar problemas de precisión al redondear o trucar números, Z3 devuelve los valores reales como expresiones. Para obtener una aproximación de las expresiones como resultado, se debe establecer la opción `(set-option :pp.decimal true)`.

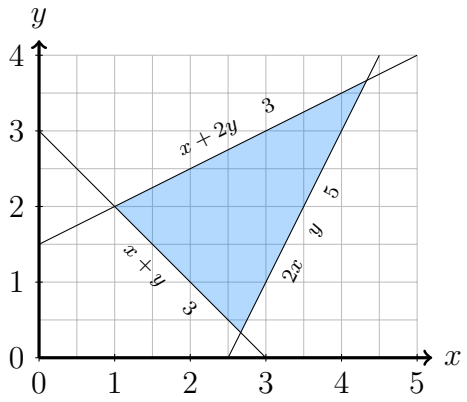
2.7.3.5. Optimización

La funcionalidad principal de Z3 es comprobar la satisfacibilidad de fórmulas lógicas sobre una o más teorías, y producir modelos para las fórmulas satisfacibles. No obstante, en muchos casos los modelos arbitrarios son insuficientes para aplicaciones que deben resolver problemas de optimización: uno o varios valores deben ser mínimos o máximos.

Z3 extiende el estándar SMT-LIB 2.0 para incorporar nuevos comandos que permiten expresar la optimización de objetivos. El comando `(maximize t)` obliga a `check-sat` a producir un modelo que maximice los valores del término `t`, cuyo tipo debe ser `Int`, `Real`, o `BitVec`. De forma análoga, `(minimize t)` se utiliza para minimizar los valores del término `t`.

Ejemplo 2.11. Supongamos que queremos resolver un problema típico de programación lineal[19], donde tenemos una serie de restricciones lineales y una función objetivo, también lineal, que debemos maximizar o minimizar:

$$
\begin{array}{ll}
\textit{minimizar} & x + y \\
\textit{sujeto a} & 2x - y \leq 5 \\
& -x + 2y \leq 3 \\
& x + y \geq 3
\end{array}
$$



Las variables del problema son declaradas en Z3 como constantes mediante el comando `declare-const` (líneas 1 y 2). Las restricciones del problema son introducidas como fórmulas lógicas mediante el comando `assert` (líneas 3, 4 y 5). La función objetivo se introduce mediante el comando `minimize` (línea 6). Y, como es usual, para comprobar la satisfacibilidad de las fórmulas y obtener la interpretación utilizamos los comandos `check-sat` y `get-model` (líneas 7 y 8).

```
1 (declare-const x Real)
2 (declare-const y Real)
3 (assert (<= (+ (- x) (* 2 y)) 3))
4 (assert (<= (- (* 2 x) y) 5))
```

[19] Campo de la optimización matemática dedicado a maximizar o minimizar una función lineal.

```
(assert (>= (+ x y) 3))
(minimize (+ x y))
(check-sat)
(get-model)
```

Como se puede observar en la gráfica, el problema es satisfacible y la solución óptima se encuentra en uno de los puntos extremos de la región factible (coloreada en azul), dado que es acotada, no vacía y conforma un polígono convexo. En concreto, el punto que minimiza la función objetivo es `(x,y) = (1,2)`, tal y como indica la interpretación proporcionada por Z3.

```
sat
(model
    (define-fun y () Int 2.0)
    (define-fun x () Int 1.0)
)
```

Capítulo 3

Calibrado de programas lógicos difusos en el entorno FLOPER

En esta sección se introduce, en los términos que recoge nuestra propuesta [41, 42, 54], una extensión *simbolica* 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 la abreviatura sFASILL (*\symbolic Fuzzy Aggregators and Similarity Into a Logic Language"*) para referirnos a estos programas. Los objetos simbólicos se representarán genéricamente como o^s.

3.1. El lenguaje sFASILL

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

Definición 3.1. (Regla sFASILL). *Una regla sFASILL es de la forma $A \leftarrow \mathcal{B}$, donde A es una formula atomica llamada cabeza y $\mathcal{B}$ es una formula bien formada (construida a partir de formulas atomicas $B_1, B_2, \ldots, B_n$, grados de verdad y conectivos | posiblemente simbolicos |).*

Ejemplo 3.1. El conjunto de reglas Π^s mostrado a continuación, la relación de similitud $\mathcal{R}$ del Ejemplo 2.7, y el retículo L del Ejemplo 2.6 forman un programa sFASILL $\mathcal{P}^s = \langle \Pi^s, \mathcal{R}, L \rangle$.

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

Donde $\mathtt{v}^{\mathtt{s1}}$ es un grado de verdad simbólico y $@^{\mathtt{s2}}$ es un agregador simbólico.

3.1.1. Semántica operacional de sFASILL

La semántica operacional de sFASILL se diseña de manera análoga a como recoge la Sección 2.6.1 para FASILL. En primer lugar, se introduce un paso *operativo* que, a diferencia de la programación lógica difusa estándar, 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 FASILL y sFASILL está basada en un esquema similar. La principal diferencia es que, para programas FASILL, el paso interpretativo siempre devuelve un valor, mientras que para programas sFASILL es posible obtener una expresión que contenga valores y conectivos simbólicos que deben ser reemplazados por elementos del retículo para poder computar su valor. Llamaremos a estas respuestas simbólicas evaluadas tanto como sea posible *respuestas computadas difusas simbolicas*, o sfca's para abreviar.

Ejemplo 3.2. Sea $\mathcal{P}^s = \langle \Pi^s, \mathcal{R}, L \rangle$ el programa del Ejemplo 3.1. Es posible realizar una derivación con la respuesta computada difusa simbólica $\langle @^{\mathtt{s2}}\mathtt{(0.8,0)}; \{X/ritz\}\rangle$ para $\mathcal{P}^s$ y el objetivo $\mathcal{Q} = \mathtt{good\_hotel(X)}$:

$$
\begin{array}{ll}
\langle \underline{\mathtt{good\_hotel(X)}}; id\rangle & \mathrm{SS}^{R4} \\
\langle @^{\mathtt{s2}}(\underline{\mathtt{elegant(X)}}, @_{\mathtt{very}}(\mathtt{close(X,metro)})); \{X_1/X\}\rangle & \mathrm{SS}^{R2} \\
\langle @^{\mathtt{s2}}(\mathtt{0.8}, @_{\mathtt{very}}(\underline{\mathtt{close(ritz,metro)}})); \{X_1/ritz, X/ritz\}\rangle & \mathrm{FS} \\
\langle @^{\mathtt{s2}}(\mathtt{0.8}, \underline{@_{\mathtt{very}}(\mathtt{0})}); \{X_1/ritz, X/ritz\}\rangle & \mathrm{IS} \\
\langle @^{\mathtt{s2}}(\mathtt{0.8,0}); \{X_1/ritz, X/ritz\}\rangle &
\end{array}
$$

Aparte de esta, existe otra sfca $\langle @^{\mathtt{s2}}(\mathtt{0.6}\ \&_{\mathtt{godel}}\ \mathtt{v}^{\mathtt{s1}}\mathtt{,0.16}); \{X/hydropolis\}\rangle$ alternativa asociada al mismo objetivo. En la Figura 3.1 se muestra una representación gráfica del árbol de derivación para este objetivo.

3.2. 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

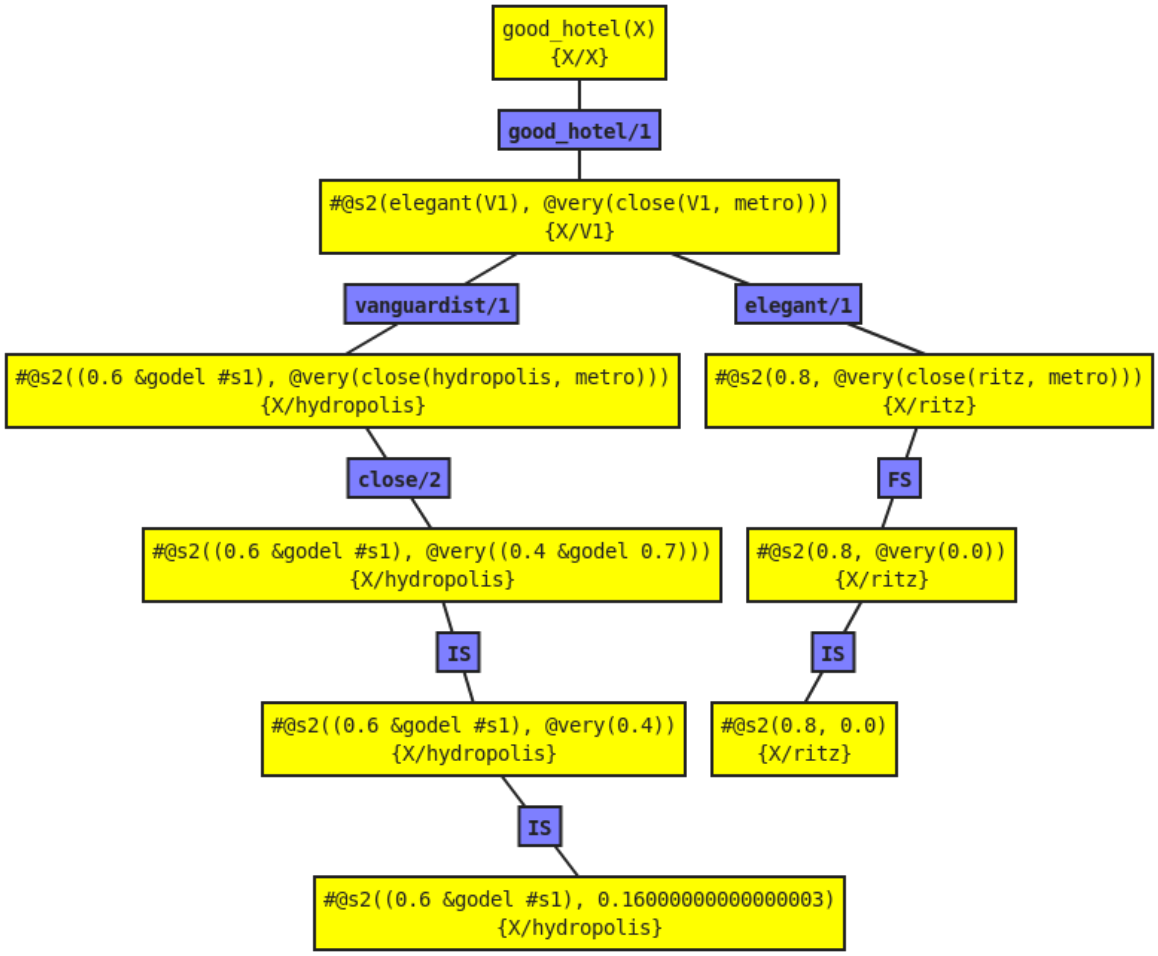


Figura 3.1: Árbol de derivación para el Ejemplo 3.2

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 una técnica automática para el calibrado de programas lógicos difusos utilizando programas sFASILL, reproduciendo la propuesta que se formula en [41].

3.2.1. Sustitución simbólica

Dado un retículo L y un lenguaje simbólico $\mathcal{L}_L^s$, en adelante consideraremos *sustituciones simbolicas*, 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 $\Theta, \Gamma, \ldots$

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 FASILL[1], esto es, un programa sin elementos simbólicos, como se muestra en el siguiente ejemplo.

Ejemplo 3.3. Sea $\mathcal{P}^s = \langle \Pi^s, L, R\rangle$ el programa simbólico del Ejemplo 2.8, cuyas reglas Π^s se muestran a continuación, y sea $\Theta = \{v^{s1}/0.5, @^{s2}/@_{aver}\}$ una sustitución simbólica para $\mathcal{P}^s$.

$$\Pi^s = \left\{ \begin{array}{lll} R_1: & \texttt{vanguardist(hydropolis)} & \leftarrow \texttt{v}^{\texttt{s1}}\texttt{.} \\ R_2: & \texttt{elegant(ritz)} & \leftarrow \texttt{0.8.} \\ R_3: & \texttt{close(hydropolis, taxi)} & \leftarrow \texttt{0.7.} \\ R_4: & \texttt{good\_hotel(X)} & \leftarrow @_{\texttt{aver}}\texttt{(elegant(X),} \\ & & \quad @^{\texttt{s2}}\texttt{(close(X,metro)).} \end{array} \right.$$

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

$$\Pi^\emptyset = \left\{ \begin{array}{lll} R_1: & \texttt{vanguardist(hydropolis)} & \leftarrow \texttt{0.5.} \\ R_2: & \texttt{elegant(ritz)} & \leftarrow \texttt{0.8.} \\ R_3: & \texttt{close(hydropolis, taxi)} & \leftarrow \texttt{0.7.} \\ R_4: & \texttt{good\_hotel(X)} & \leftarrow @_{\texttt{aver}}\texttt{(elegant(X),} \\ & & \quad @_{\texttt{very}}\texttt{(close(X,metro)).} \end{array} \right.$$

3.2.2. Distancia

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.

Definición 3.2. (Distancia). *Sea X un conjunto. Se dice que la aplicacion $d : X \times X \to \mathbb{R}$ de ne una distancia en X si, para todo $x, y \in X$, se cumple:*

a) $d(x, y) \geq 0$, *y ademas* $d(x, y) = 0$, *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).*

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

Ejemplo 3.4. La distancia más común en 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 $\dot{d}(x, y) = |x - y|$.

Algoritmo 3.1: 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 $i \in \{1, ..., k\}$

Salida: una sustitución simbólica Θ

considerar un número finito de posibles sustituciones simbólicas para $\mathcal{S}ym(\mathcal{P}^s)$, denotadas por Θ_1, Θ_2, ..., Θ_n, con $n > 0$;
establecer $\tau = +\infty$;
para cada *caso de prueba* t_i, *con* $i \in \{1, ..., k\}$ **hacer**
    calcular la sfca $\langle Q_i^{\emptyset}; \theta_i \rangle$ de $\langle Q_i; id \rangle$ en $\mathcal{P}^s$;
fin
para cada *sustitucion simbolica* Θ_j, *con* $j \in \{1, ..., n\}$ **hacer**
    establecer $z = 0$;
    para cada *caso de prueba* t_i, *con* $i \in \{1, ..., k\}$ **hacer**
        si $z \geq \tau$ **entonces**
            parar;
        fin
        computar $\langle Q_i^{\emptyset} \Theta_j; \theta_i \rangle \rightarrow_{IS} \langle v_{i,j}; \theta_i \rangle$;
        actualizar $z = z + d(v_{i,j}, v_i)$;
    fin
    si $z < \tau$ **entonces**
        actualizar $\tau = z$;
        actualizar $\Theta_\tau = \Theta_j$;
    fin
fin
devolver Θ_τ;

3.2.3. Algoritmo 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 el siguiente algoritmo 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 este algoritmo puede ser parametrizada dependiendo del conjunto de sustituciones simbólicas consideradas. Por ejemplo, para el retículo del Ejemplo 2.6 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, . . . , 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.

La técnica automática de calibrado detallada en el Algoritmo 3.1 computa primero las sfca's de los objetivos de los casos de prueba con el programa simbólico $\mathcal{P}^s$. Luego, por cada sustitución simbólica, se aplica dicha sustitución a las sfca's computadas anteriormente (obteniendo expresiones sin constantes simbólicas) y tras una serie de derivaciones interpretativas se obtienen las respuestas computadas difusas de cada objetivo. El algoritmo 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.

Figura 3.2: Captura de pantalla de la herramienta FASILL online tras calibrar un programa lógico difuso

Ejemplo 3.5. Para ilustrar el funcionamiento del algoritmo de calibrado, utilizaremos el programa sFASILL $\mathcal{P}^s$ del Ejemplo 3.1, 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 $\{@_{aver}, @_{very}, @_{geom}\}$ como posibles agregadores. El programa contiene dos elementos simbólicos distintos: $\{v^{s1}, @^{s2}\}$. Por lo tanto, debemos considerar las siguientes sustituciones simbólicas a la hora de calibrar $\mathcal{P}^s$:

$$\Theta_1 = \{v^{s1}/0.3, @^{s2}/@_{aver}\} \quad \Theta_3 = \{v^{s1}/0.5, @^{s2}/@_{aver}\} \quad \Theta_5 = \{v^{s1}/0.7, @^{s2}/@_{aver}\}$$
$$\Theta_2 = \{v^{s1}/0.3, @^{s2}/@_{geom}\} \quad \Theta_4 = \{v^{s1}/0.5, @^{s2}/@_{geom}\} \quad \Theta_6 = \{v^{s1}/0.7, @^{s2}/@_{geom}\}$$

Nótese que aunque el conjunto de agregadores es $\{@_{aver}, @_{very}, @_{geom}\}$, el agregador $@^{s2}$ recibe dos argumentos en el programa $\mathcal{P}^s$. 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 un conjunto de casos de prueba. En este ejemplo, consideraremos los siguientes casos de prueba:

$$t_1 = (\texttt{good\_hotel(hydropolis)}, \langle \texttt{0.3}; id \rangle)$$
$$t_2 = (\texttt{good\_hotel(ritz)}, \langle \texttt{0.4}; id \rangle)$$

Es decir, queremos conocer la mejor sustitución simbólica para minimizar el error del objetivo `good_hotel(X)`, esperando obtener el grado de verdad 0.3 para `hydropolis` y el grado de verdad 0.4 para `ritz`.

Para calibrar este programa, primero debemos ejecutar el objetivo de los casos de prueba, obteniendo las siguientes respuestas computadas difusas simbólicas:

$$Q_1^{\emptyset}: \quad \langle \texttt{@}^{\texttt{s2}}\texttt{((0.6 \&}_{\texttt{godel}}\ \texttt{v}^{\texttt{s1}}\texttt{), 0.16)}; \{\} \rangle$$
$$Q_2^{\emptyset}: \quad \langle \texttt{@}^{\texttt{s2}}\texttt{(0.8, 0.0)}; \{\} \rangle$$

A partir de estas sfca's aplicamos las sustituciones simbólicas Θ_i a $Q_1^{\emptyset}$ y $Q_2^{\emptyset}$, con $i \in \{1, \ldots, 6\}$, y computamos las derivaciones interpretativas necesarias. En la Tabla 3.1 se muestran remarcadas con un asterisco ( ) las fca's podadas, donde las columnas `hydropolis` y `ritz` contienes las fca's obtenidas junto al error cometido.

Tabla 3.1: Resumen de pesos al calibrar el programa del Ejemplo 3.5

v^{s1}	$@^{s2}$	Θ	*hydropolis*		*ritz*		z
0.3	$@_{aver}$	Θ_1	0.23	0.07	0.4	0.0	0.07
	$@_{geom}$	Θ_2	0.22	0.08	**0.0**	**0.4**	**0.48**
0.5	$@_{aver}$	Θ_3	0.33	0.03	0.4	0.0	0.03
	$@_{geom}$	Θ_4	0.28	0.02	0.0	0.4	**0.42**
0.7	$@_{aver}$	Θ_5	0.38	0.08	**0.4**	**0.0**	**0.48**
	$@_{geom}$	Θ_6	0.31	0.01	0.0	0.4	**0.41**

Tal y como se observa en el Tabla 3.1, la mejor sustitución simbólica encontrada es $\Theta_3 = \{v^{s1}/0.5, @^{s2}/@_{aver}\}$ con una desviación de 0.03 con respecto a los casos de prueba introducidos. Si aplicamos esta sustitución simbólica al programa inicial y ejecutamos el objetivo `good_hotel(X)`, obtenemos las fca's $\langle 0.33; \{X/hydropolis\}\rangle$ y $\langle 0.4; \{X/ritz\}\rangle$. Con esta precisión de los grados de verdad, no ha sido posible reducir la desviación al máximo. En la Figura 3.2 se muestra la herramienta FASILL online tras calibrar este programa.

Capítulo 4

Calibrado de programas lógicos difusos en solucionadores SMT

En esta sección se presenta la principal aportación del presente trabajo, introduciendo el nuevo algoritmo de calibrado de programas lógicos difusos basado en el uso de solucionadores SMT, y se detalla su implementación en el entorno FLOPER junto con la herramienta Z3.

Al igual que en el algoritmo original de calibrado introducido en la Sección 3.2, consideraremos únicamente la primera respuesta computada de un objetivo. No obstante, a diferencia del algoritmo anterior, donde hay que discretizar los elementos del retículo cuando este es infinito (como es el caso del retículo unitario), en Z3 esto no es necesario, ya que no buscará el modelo óptimo realizando una búsqueda exhaustiva entre todas las posibles combinaciones.

4.1. Traducción entre Prolog y SMT-LIB

El primer paso para calibrar programas lógicos difusos mediante Z3 es comunicar el entorno FLOPER, encargado de obtener las respuestas computadas difusas simbólicas a partir de un conjunto de casos de prueba, con el solucionador Z3, encargado de buscar un modelo que minimice la desviación de estas respuestas simbólicas respecto a los valores esperados de los casos de prueba.

Dado que Z3 no cuenta con una interfaz para Prolog, se hace necesario traducir las L^s-expresiones de sFASILL y los retículos expresados en Prolog al lenguaje SMT-LIB, con el fin de poder ejecutar el proceso de calibrado como un guión de Z3. Del mismo modo, la salida del solucionador Z3 en lenguaje SMT-LIB debe ser analizada y traducida nuevamente a Prolog para poder ser manipulada por el entorno FLOPER.

Como resultado de este trabajo se ha desarrollado y publicado un paquete para el intérprete SWI-Prolog, disponible en la URL http://www.swi-prolog.org/pack/

list?p=smtlib y cuyo código fuente se puede encontrar en el siguiente repositorio de GitHub: https://github.com/jariazavalverde/prolog-smtlib. Este paquete cuenta con un único módulo Prolog con predicados para leer y escribir expresiones, guiones, declaraciones de lógicas y declaraciones de teorías en SMT-LIB. Este módulo ha sido implementado utilizando gramáticas de cláusulas definidas.

Una *gramatica de clausulas de nidas* (*DCG*) representa una gramática como un conjunto de cláusulas definidas en una lógica de primer orden [21]. La notación de DCG es simplemente azúcar sintáctico[1] para las cláusulas definidas normalmente en Prolog. Una gramática en Prolog se define mediante reglas de la forma `Cabeza --> Cuerpo`, que son internamente traducidas a cláusulas Prolog estándar. 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. En el Ejemplo 4.1 se muestran las reglas implementadas para analizar uno de los componentes léxicos de SMT-LIB.

Ejemplo 4.1. Tal y como se describe en el la Sección 2.7.2, un componente léxico de la categoría `numeral` en SMT-LIB es el carácter `0` o una secuencia no vacía de dígitos empezando por un carácter distinto de `0`.

```
1  digits([X|Xs]) --> [X],
2      {char_code(X, C), C >= 48, C =< 57}, !,
3      digits(Xs).
4  digits([]) --> [].
5
6  numeral(numeral(Y)) -->
7      digits([X|Xs]),
8      {(Xs = [] ; X \= '0'),
9       number_chars(Y, [X|Xs])
10     }, whitespaces.
```

La primera regla de producción (línea 1) lee un carácter, comprueba que es un dígito entre 0 y 9 (línea 2) y lee los siguientes dígitos (línea 3). La segunda regla de producción (línea 4) permite leer una secuencia vacía de dígitos cuando la primera regla falla. La tercera regla de producción (línea 6) lee una secuencia de dígitos mediante las anteriores reglas (línea 7), y comprueba si la secuencia es de un único dígito o si comienza por un dígito distinto de 0, devolviendo un `numeral`.

[1] Una construcción en un lenguaje de programación es llamada *azucar sintactico* si esta puede ser eliminada del lenguaje sin ningún efecto en la potencia del mismo.

4.2. Traducción de retículos Prolog a SMT-LIB

En el entorno FLOPER, los retículos de grados de verdad se modelan como se describe a continuación. Todas las componentes relevantes de un retículo pueden ser codificadas como un programa Prolog que debe contenter la definición de un conjunto mínimo de predicados describiendo los elementos válidos del retículo (con una mención especial a los elementos supremo e ínfimo), el orden total o parcial entre los elementos, y el repertorio de conectivas difusas.

- `member/1` es cierto cuando es llamado con un término que representa un grado de verdad válido.
- `members/1` devuelve una lista con todos los elementos del retículo. Si el retículo es infinito, devuelve una lista de elementos representativos.
- `bot/1` y `top/1` devuelven el elemento ínfimo y supremo del retículo, respectivamente.
- `leq/2` modela la relación de orden entre todos los posibles pares de grados de verdad, y se satisface cuando el primer argumento es menor o igual al segundo.
- Dado un conjunto de conectivas de la forma $\&_{label1}$ (conjunción), $|_{label2}$ (disyunción) o $@_{label3}$ (agregador) con aridades n_1, n_2 y n_3 respectivamente, debemos proporcionar cláusulas que definan los predicados `and_label1/(`n_1`+1)`, `or_label2/(`n_2`+1)` y `agr_label3/(`n_3`+1)`, donde el parámetro adicional de cada predicado contiene el resultado de evaluar dicha conectiva con los parámetros anteriores.
- Opcionalmente, es posible establecer la t-norma y la t-conorma por defecto mediante los predicados `tnorm/1` y `tconorm/1`, respectivamente. El entorno utilizará las conectivas indicadas por este predicado cuando no se especifique una etiqueta en las conjunciones y disyunciones de las reglas.

En el Apéndice A.1 puede encontrarse una descripción de la implementación de cada uno de los retículos utilizados en esta memoria junto a su código Prolog.

Para poder evaluar las L^s-expresiones en el solucionador Z3, debemos traducir todas las conectivas al lenguaje SMT-LIB. Por ejemplo, la disyunción de la lógica del producto descrita en la Figura 2.2 se expresa en SMT-LIB de la siguiente forma:

```
1 (define-fun lat!or!prod!2 ((x Real) (y Real)) Real
2     (- (+ x y) (* x y)))
```

Los posibles valores de los grados de verdad simbólicos vendrán determinados por el sistema de tipos de Z3, por lo que no será obligatorio traducir el predicado `member/1` a menos que sea necesario restringir su dominio. Por ejemplo, los elementos del retículo lógico $\{false, true\}$ se corresponden exactamente con los datos del tipo `Bool` en SMT-LIB. No obstante, en el caso del retículo unitario $([0,1], \leq)$, el tipo de dato es `Real` pero los grados de verdad están acotados en el intervalo $[0,1]$, por lo que se hace necesario incluir la siguiente función del retículo en el guión de Z3:

```
1 (define-fun lat!member ((x Real)) Bool
2     (and (<= 0.0 x) (<= x 1.0)))
```

La función `lat!member` recibe un elemento con el tipo de dato de los grados de verdad del retículo en cuestión, y devuelve un valor lógico indicando si el elemento pertenece o no al retículo. Por otra parte, también es necesario traducir la función de distancia, que tomará dos grados de verdad y devolverá un dato del tipo `Real`. Por ejemplo, en el retículo unitario, la noción de distancia usual se expresa en SMT-LIB de la siguiente forma:

```
1 (define-fun lat!distance ((x Real) (y Real)) Real
2     (abs (- y x)))
```

Este proceso de traducción debe ser realizado manualmente para cada retículo. En el Apéndice A.2 se encuentra el código completo de los retículos en SMT-LIB utilizados en esta memoria.

4.3. Implementación del calibrado basado en SMT

Una vez traducidos los retículos a SMT-LIB, el resto del proceso de traducción y calibrado se realiza de forma automática en el entorno FLOPER. Nótese que la idea de este nuevo método de calibrado es la misma que la mostrada en el Algoritmo 3.1, con la diferencia de que será la herramienta Z3 quien realice la búsqueda, al expresar el calibrado como un problema de optimización. En adelante, describiremos los predicados que se han implementado en un nuevo módulo Prolog del lenguaje FASILL llamado `tuning_smt`, que se ha creado para tal fin.

El proceso de calibrado se ejecuta consultando el predicado `tuning_smt/4`, el único predicado que exporta el módulo `tuning_smt`. Este predicado recibe un átomo que representa el tipo de dato de los grados de verdad (`Bool`, `Real`, etcétera) y un átomo que representa la ruta al fichero que contiene el retículo en SMT-LIB; y devuelve la mejor sustitución y el error absoluto cometido con respecto a los casos de prueba. Más concretamente, este predicado se encarga de:

1. Recoger todas las constantes simbólicas contenidas en los casos de prueba que estén cargados en el entorno, mediante el predicado `findall_symbolic_cons/1` perteneciente al módulo `tuning`.

2. Declarar todas las constantes simbólicas en formato SMT-LIB mediante el predicado `tuning_smt_decl_const/3` y restringir su dominio si es necesario.

3. Ejecutar y traducir los casos de prueba al formato SMT-LIB mediante el predicado `tuning_smt_minimize/1`, y terminar de componer todos los comandos necesarios del guión de Z3.

4. Escribir el guión de Z3 mediante el predicado `smtlib_write_to_file/2` de la librería `smtlib`, invocando al proceso de Z3 por línea de comandos, para finalmente analizar la salida mediante el predicado `tuning_smt_read_expressions/2` de la librería `smtlib` y obtener así la sustitución y la desviación en formato Prolog.

```
1  tuning_smt(Domain, LatFile, Substitution, Deviation) :-
2      smtlib_read_script(LatFile, list(Lattice)),
3      findall_symbolic_cons(Cons),
4      tuning_smt_decl_const(Domain, Cons, Declarations),
5      (member([reserved('define-fun'), symbol('lat!member')|_],
       Lattice) ->
6          tuning_smt_members(Cons, Members);
7          Members = []),
8      tuning_smt_minimize(Minimize),
9      tuning_theory_options(Domain, TheoryOpts),
10     GetModel = [[reserved('check-sat')], [reserved('get-model
      ')]],
11     append([Lattice, Declarations, Members, Minimize,
      TheoryOpts, GetModel], Script),
12     smtlib_write_to_file('.tuning.smt2', list(Script)),
13     shell('z3 -smt2 .tuning.smt2 > .result.tuning.smt2', _),
14     smtlib_read_expressions('.result.tuning.smt2', Z3answer),
15     tuning_smt_answer(Z3answer, Substitution, Deviation).
```

El predicado `tuning_smt_decl_const/3` recibe el tipo de dato t de los grados de verdad y una lista de constantes simbólicas, y por cada constante simbólica declara una variable en formato SMT-LIB con el tipo `String` si se trata de una conectiva simbólica, o con el tipo t correspondiente si se trata de un grado de verdad simbólico; y devuelve una lista con todas las declaraciones.

```
1  tuning_smt_decl_const(_, [], [[reserved('declare-const'),
       symbol('deviation!'), symbol('Real')]]) :- !.
```

```
tuning_smt_decl_const(Domain, [X|Xs], [Y|Ys]) :- !,
    tuning_smt_decl_const(Domain, X, Y),
    tuning_smt_decl_const(Domain, Xs, Ys).
tuning_smt_decl_const(Domain, sym(td,Name,0), [reserved('
    declare-const'), symbol(Sym), symbol(Domain)]) :- !,
    atom_concat('sym!td!0!', Name, Sym).
tuning_smt_decl_const(_, sym(Type,Name,Arity), [reserved('
    declare-const'), symbol(Sym), symbol('String')]) :- !,
    atom_number(Arity_, Arity),
    atom_concat('sym!', Type, LatType),
    atom_concat(LatType, '!', LatType_),
    atom_concat(LatType_, Arity_, LatTypeArity),
    atom_concat(LatTypeArity, '!', LatTypeArity_),
    atom_concat(LatTypeArity_, Name, Sym).
```

Por cada grado de verdad simbólico v^s, además, se añade al guión un aserto de la forma `(assert (lat!member v))` si la función `member` está definida, para restringir su dominio. Del mismo modo, para cada conectiva simbólica ς^s se añade un aserto de la forma `(assert (dom!sym!type!n` ς`))`, donde `type` representa el tipo de conectiva (`and`, `or` o `agr`) y `n` representa su aridad. Estas funciones especifican las posibles conectivas por las que se pueden sustituir las conectivas simbólicas. Por ejemplo, en el caso de las conjunciones simbólicas (con aridad 2) en el retículo unitario:

```
(define-fun dom!sym!and!2 ((s String)) Bool
    (or
        (= s "and_godel")
        (= s "and_luka")
        (= s "and_prod")))
```

El predicado `tuning_smt_members/2` se encarga de generar estas declaraciones, recibiendo una lista de constantes simbólicas y devolviendo una lista que contiene el comando necesario en formato SMT-LIB para cada una de ellas:

```
tuning_smt_members([], []) :- !.
tuning_smt_members([sym(td,Name,0)|Cons], [[reserved('assert'
    ), [symbol('lat!member'), symbol(Sym)]]|Members]) :- !,
    atom_concat('sym!td!0!', Name, Sym),
    tuning_smt_members(Cons, Members).
tuning_smt_members([sym(Type,Name,Arity)|Cons], [[reserved('
    assert'), [symbol(Dom), symbol(Sym)]]|Members]) :- !,
    atom_concat('sym!', Type, SymType),
    atom_concat(SymType, '!', SymType_),
    atom_number(Arity_, Arity),
```

```
9       atom_concat(SymType_, Arity_, SymTypeArity),
10      atom_concat('dom!', SymTypeArity, Dom),
11      atom_concat(SymTypeArity, '!', SymTypeArity_),
12      atom_concat(SymTypeArity_, Name, Sym),
13      tuning_smt_members(Cons, Members).
```

Las L^s-expresiones son trivialmente traducidas, almacenando en una variable llamada `deviation!` la suma de las distancias entre las sfca's y los valores de verdad esperados, minimizando entonces dicha variable en Z3, de la siguiente forma:

```
1  (assert
2      (= deviation!
3          (+
4              (lat!distance td_1 expr_1)
5              (lat!distance td_2 expr_2)
6              ...
7              (lat!distance td_n expr_n))))
8
9  (minimize deviation!)
10 (check-sat)
11 (get-model)
```

El predicado `tuning_smt_minimize/1` genera los comandos SMT-LIB necesarios para calcular el valor de la variable `deviation!`—la variable objetivo a minimizar—teniendo en cuenta los casos de prueba que se hayan introducido anteriormente en el entorno. FASILL almacena los casos de prueba como cláusulas del predicado dinámico `fasill_testcase/2`. Por cada caso de prueba, el predicado `tuning_smt_minimize/1` obtiene la primera sfca correspondiente mediante el predicado `query/2` del módulo `semantics`, y la traduce a SMT-LIB mediante el predicado `sfca_to_smtlib/2`.

```
1  tuning_smt_minimize([Assert, Minimize]) :-
2      findall([symbol('lat!distance'), TD_, SMT], (
3          fasill_testcase(TD, Goal),
4          (query(Goal, state(SFCA, _)) -> true ;
       lattice_call_bot(SFCA)),
5          sfca_to_smtlib(TD, TD_),
6          sfca_to_smtlib(SFCA, SMT)
7      ), Distances),
8      Assert = [reserved(assert), [symbol(=), symbol('deviation
       !'), [symbol(+)|Distances]]],
9      Minimize = [reserved(minimize), symbol('deviation!')].
```

Ejemplo 4.2. Nuevamente, utilizaremos el programa $\mathcal{P}^s$ del Ejemplo 3.1 con dos constantes simbólicas $\{v^{s1}, @^{s2}\}$ para ilustrar el proceso de calibrado con Z3, con los mismos casos de prueba utilizados en el calibrado del Ejemplo 3.5:

$t_1 = (\texttt{good\_hotel(hydropolis)}, \langle \texttt{0.3}; id \rangle)$ $\quad Q_1^{\emptyset} : \langle \texttt{@}^{\texttt{s2}}\texttt{((0.6 \&}_{\texttt{godel}}\ \texttt{v}^{\texttt{s1}}\texttt{), 0.16)}; \{\} \rangle$

$t_2 = (\texttt{good\_hotel(ritz)}, \langle \texttt{0.4}; id \rangle)$ $\quad Q_2^{\emptyset} : \langle \texttt{@}^{\texttt{s2}}\texttt{(0.8, 0.0)}; \{\} \rangle$

Las constantes simbólicas contenidas en las sfca's $Q_1^{\emptyset}$ y $Q_2^{\emptyset}$ son declaradas en el guión de Z3, estableciendo sus posibles valores mediante las funciones del retículo `lat!member` y `dom!sym!agr!2`:

```
(declare-const deviation! Real)
(declare-const sym!td!0!s1 Real)
(declare-const sym!agr!2!s2 String)
(assert
    (lat!member sym!td!0!s1))
(assert
    (dom!sym!agr!2 sym!agr!2!s2))
```

La variable `deviation!` almacena el sumatorio de la distancia de cada respuesta computada difusa simbólica con su valor esperado, es decir, la distancia entre $Q_1^{\emptyset}$ y 0.3 más la distancia entre $Q_2^{\emptyset}$ y 0.4:

```
(assert (= deviation! (+
    (lat!distance 0.3
        (call!sym!agr!2 sym!agr!2!s2
            (lat!and!godel!2 0.6 sym!td!0!s1) 0.17))
    (lat!distance 0.4 (call!sym!agr!2 sym!agr!2!s2 0.8 0)))))

(minimize deviation!)
(set-option :pp.decimal true)
(check-sat)
(get-model)
```

Añadiendo todos estos comandos junto a la definición del retículo unitario en SMT-LIB, obtenemos la siguiente salida tras ejecutar el guión en Z3:

```
sat
(model
    (define-fun sym!td!0!s1 () Real
        0.43)
    (define-fun deviation! () Real
        0.0)
    (define-fun sym!agr!2!s2 () String
```

```
8           "agr_aver"))
```

Tal y como se observa en la salida de Z3, la mejor sustitución simbólica es $\Theta = \{v^{s1}/0.43, @^{s2}/@_{aver}\}$ con una desviación de 0.0 con respecto a los casos de prueba introducidos. Si aplicamos esta sustitución simbólica al programa inicial y ejecutamos el objetivo `good_hotel(X)`, obtenemos exactamente las respuestas computadas difusas esperadas $\langle 0.3; \{X/hydropolis\}\rangle$ y $\langle 0.4; \{X/ritz\}\rangle$. En la Figura 4.1 se muestra el entorno FASILL online tras calibrar el programa de este ejemplo mediante Z3.

4.4. Herramienta online para el calibrado con Z3

Tal y como se describe en la Sección 2.6.2, el entorno FASILL cuenta con una aplicación online que en la actualidad permite ejecutar, desplegar [39, 40, 43] y calibrar [41, 42] programas lógicos difusos simbólicos. Aquí, nos proponemos incorporar la nueva técnica de calibrado con Z3.

Tras la zona de entrada del programa (véase la Figura 2.5), el usuario puede escoger entre la pestaña de ejecución mostrada en la Figura 2.6 para lanzar un objetivo, o la pestaña de calibrado, que se muestra en la Figura 4.1. La zona de calibrado cuenta con un campo de texto para introducir los casos de prueba, y con un menú desplegable para seleccionar el algoritmo de calibrado, donde hasta el momento únicamente estaba disponible la opción *\FASILL | Thresholded method"*, que se corresponde con el algoritmo descrito en la Sección 3.2. La salida del calibrado está formada por un

Figura 4.1: Captura de pantalla de la herramienta FASILL online tras calibrar un programa lógico difuso con Z3

único campo de texto que muestra la mejor sustitución simbólica encontrada, junto a su desviación con respecto a los casos de prueba, y el tiempo de ejecución del proceso de calibrado (en milisegundos). Nótese que aunque en el Ejemplo 4.2 hemos mostrado la salida del solucionador Z3 (en formato SMT-LIB) como resultado del calibrado, en realidad FASILL analiza dicha salida y ofrece una respuesta con el mismo formato que el propio calibrado de FASILL, por lo que es imperceptible para el usuario final.

Dado que el proceso de calibrado con Z3 requiere que el retículo utilizado sea traducido manualmente a SMT-LIB, se ha introducido una nueva opción de calibrado con SMT en el menú desplegable por cada posible retículo. Por el momento, hemos adaptado la técnica a los siguientes retículos en el entorno online:

- *\SMT | Boolean lattice"*
- *\SMT | Real lattice"*
- *\SMT | Unit lattice"*

Para comunicar la interfaz web con FASILL, se realiza una petición asíncrona con JavaScript[2] a una página PHP[3] del servidor, enviando todos los datos necesarios: el programa, el retículo, las ecuaciones de similitud, los casos de prueba, etcétera. Esta página recoge estos datos y los almacena en ficheros temporales en el servidor, y se encarga de ejecutar el intérprete SWI-Prolog cargando los paquetes de FASILL e invocando al predicado que inicia el proceso de calibrado. FASILL proporciona una respuesta en formato de texto plano por la salida estándar, que es imprimida por el servidor, y que es recogida como respuesta de la llamada asíncrona con JavaScript. Este proceso es similar para cada una de las acciones realizadas en la herramienta online, ya sea ejecutar, desplegar o calibrar programas.

[2] JavaScript es un lenguaje de programación interpretado que permite interactuar con las páginas web en el lado del cliente.

[3] PHP es un lenguaje de programación de propósito general de código del lado del servidor originalmente diseñado para el desarrollo web.

Capítulo 5

Casos de uso del calibrado

En esta sección se presentan una serie de casos de uso o aplicaciones del calibrado de programas lógicos difusos, que relacionan estas técnicas de calibrado con otros campos de aplicación, como el aprendizaje automático o la web semántica. Utilizaremos estos ejemplos para mostrar las principales ventajas del nuevo algoritmo de calibrado en Z3 con respecto al algoritmo de calibrado del entorno FLOPER.

5.1. Equivalencia de circuitos combinacionales

5.1.1. Descripción

Un *circuito combinacional* es un circuito formado por funciones lógicas elementales, que tiene un determinado número de entradas y salidas, y donde las salidas dependen únicamente de la combinación de sus entradas. En la Figura 5.2 se muestran dos circuitos combinacionales con cuatro entradas (x_0, x_1, x_2, x_3) y dos salidas (y_0, y_1).

El problema de verificar la equivalencia de dos circuitos combinacionales es de vital importancia en el campo de la verificación de circuitos digitales. Como consecuencia, han surgido numerosos enfoques para solucionar este problema [35]. Aquí, proponemos un método basado en satisfacibilidad, expresando el problema como un programa lógico del cual se obtendrá la equivalencia mediante el calibrado de un objetivo.

Sean C_A y C_B dos circuitos combinacionales, ambos con n entradas $x_1, \ldots, x_n$ y ambos con m salidas, C_A con salidas $y_1, \ldots, y_m$ y C_B con salidas $w_1, \ldots, w_m$. La función implementada por cada uno de los circuitos se define como sigue: $f_A : \{0,1\}^n \to \{0,1\}^m$, y $f_B : \{0,1\}^n \to \{0,1\}^m$. Sea $x \in \{0,1\}^n$, y definamos $f_A(x) = (f_{A,1}(x), \ldots, f_{A,m}(x))$ y $f_B(x) = (f_{B,1}(x), \ldots, f_{B,m}(x))$. Ambos circuitos no son equivalentes si se satisface la siguiente condición:

$$\exists x \in \{0,1\}^n, \exists i \in [1,m], f_{A,i}(x) \neq f_{B,i}(x)$$

que puede ser expresada como el siguiente problema de satisfacibilidad, representado

en la Figura 5.1 como un circuito combinacional (conocido como *miter*) [34]:

$$\bigvee_{i=1}^{n} (f_{A,i}(x) \oplus f_{B,i}(x)) = 1 \tag{5.1}$$

A partir de estos resultados, es fácil codificar en forma normal clausal el problema de verificar la equivalencia de dos circuitos combinacionales y, por lo tanto, es susceptible de ser implementado y calibrado como un programa lógico.

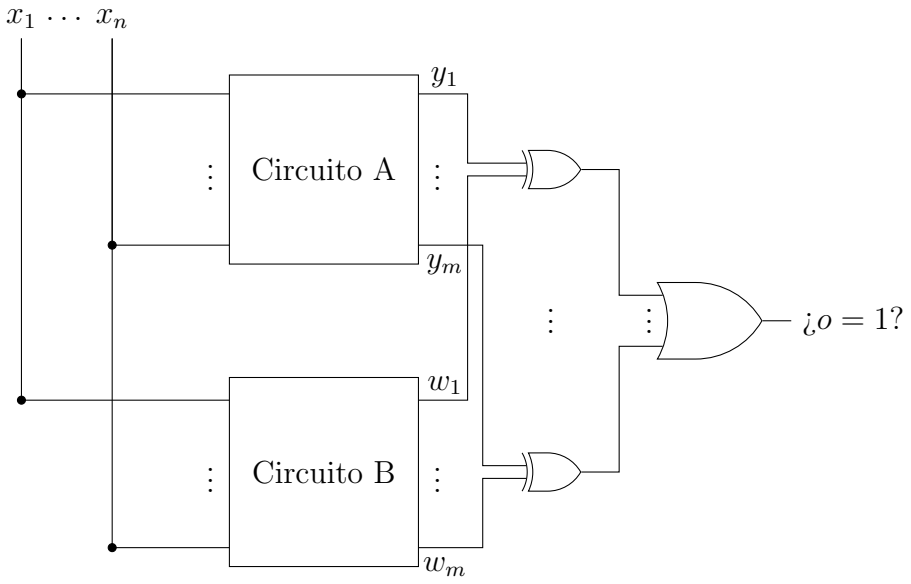


Figura 5.1: Circuito de comprobación de equivalencia (*miter*)

5.1.2. Implementación

Expresaremos cada uno de los circuitos combinacionales en FASILL como predicados de aridad 2, donde el primer argumento es una lista que contiene las entradas, y el segundo argumento es una lista que contiene las salidas.

El predicado incorporado `truth_degree/2` de FASILL nos permite evaluar un objetivo y obtener el grado de verdad asociado a este como un término del lenguaje.

Ejemplo 5.1. A continuación se muestra una posible codificación de los circuitos representados en la Figura 5.2 como los predicados `a/2` y `b/2` en el lenguaje FASILL:

```
1 a([X0,X1,X2,X3], [Y0,Y1]) :-
2     truth_degree(
3         @not('&'(X0,
4             @not('&'(
5                 @not('&'(X1,X2)),
6                 @not('&'(X2,X3))
7             ))
```

```
8           ))
9       , Y0),
10      truth_degree(@not('&'(X2,X3)), Y1).
11
12  b([X0,X1,X2,X3], [Y0,Y1]) :-
13      truth_degree(
14          @not('&'(X0,
15              '|'(
16                  '&'(X1,X2),
17                  '&'(X2,X3)
18              )
19          ))
20      , Y0),
21      truth_degree(@not('&'('&'(X2,X3),X3)), Y1).
```

Ahora podemos comprobar la salida de estos circuitos ante determinadas entradas. Por ejemplo, para la asignación $(0, 1, 1, 1)$, el objetivo `Xs = [false,true,true,true], a(Xs,Ys), b(Xs, Ws)` devuelve la respuesta computada difusa $\langle$`true`; $\{Y_s/[true, true], W_s/[true, true]\}\rangle$. Además, podemos pasar constantes simbólicas como entradas a los circuitos, y obtener las L^s-expresiones correspondientes en forma de términos. Por ejem-

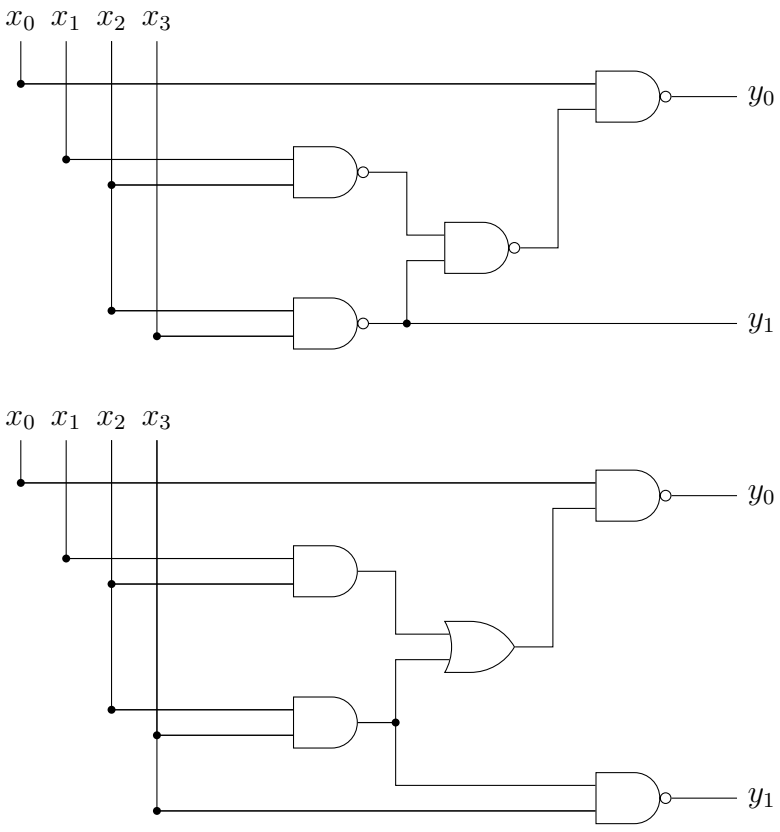


Figura 5.2: Dos circuitos combinacionales implementando la misma función

plo, para la asignación simbólica (x_0, x_1, x_2, x_3), el objetivo `a([#x0, #x1, #x2, #x3], Ys)` devuelve la respuesta computada difusa $\langle$`true`; $\{Y_s/[@_{not}(^0\&^0(\#x_0, @_{not}(^0\&^0(@_{not}(^0\&^0(\#x_1, \#x_2)), @_{not}(^0\&^0(\#x_2, \#x_3))))))), @_{not}(^0\&^0(\#x_2, \#x_3))]\}\rangle$.

Para comprobar la equivalencia entre dos circuitos, codificaremos el circuito de equivalencia representado en la figura 5.1. El predicado `miter/3` toma dos circuitos (dos átomos que representan el nombre de los predicados de cada circuito) y una lista de entradas, y comprueba si alguna de las salidas de ambos circuitos es distinta para la entrada proporcionada. Este predicado se evalúa con grado de verdad `false` cuando todas las salidas son iguales ante la entrada proporcionada, o con grado de verdad `true` en cualquier otro caso.

El predicado incorporado `call/n` de FASILL permite invocar un predicado, especificado por el primer argumento, siendo los parámetros de este predicado el resto de argumentos. El predicado `zip_xor/2` toma dos listas y devuelve una lista combinando elemento a elemento las dos primeras con el término $@_{xor}$. El predicado `fold_or/2` toma una lista y la reduce a un único término en forma de disyunción, tomando `false` como valor inicial.

```
1  zip_xor([],[],[]).
2  zip_xor([X|Xs],[Y|Ys],[@xor(X,Y)|Zs]) :- zip_xor(Xs,Ys,Zs).
3
4  fold_or([],false).
5  fold_or([X|Xs],'|'(X,Ys)) :- fold_or(Xs,Ys).
6
7  miter(Ca,Cb,Xs) :-
8      call(Ca,Xs,Ys),
9      call(Cb,Xs,Ws),
10     zip_xor(Ys,Ws,XOR),
11     fold_or(XOR,OR),
12     OR.
```

Ejemplo 5.2. Con el predicado `miter/3` podemos comprobar si los circuitos implementados en el Ejemplo 5.1 proporcionan alguna salida distinta ante una entrada determinada. Si lanzamos el objetivo `miter(a, b, [false, true, true, true])` obtenemos la respuesta computada difusa $\langle$`false`; $\{\}\rangle$, es decir, todas las salidas de ambos circuitos son idénticas ante la entrada $(0, 1, 1, 1)$.

Nótese que, si evaluamos este predicado con una entrada simbólica, ahora obtendremos una respuesta computada difusa simbólica, ya que `miter/3` se evalúa con el grado de verdad que representa el término devuelto como disyunción de todas las disyunciones exclusivas de las salidas de ambos circuitos. Por ejemplo, para el objetivo `miter(a, b, [#x0,#x1,#x2,#x3])`, obtenemos la sfca $\langle$`'&'(true,'&'(true,'&'(true,'&'(true,`

`'|'(@xor(@not('&'(#x1, @not('&'(@not('&'(#x2, #x3)), @not('&'(#x3, #x4)))))), @not('&'(#x1, '|'('&'(#x2,#x3), '&'(#x3,#x4))))), '|'(@xor(@not('&'(#x3,#x4)),@not('&'('&'(#x3,#x4),#x4))),false))))))`; $\{\}\rangle$. Se observa que ahora la L^s-expresión forma parte del grado de verdad de la respuesta computada, en lugar de formar parte de la sustitución.

Una vez implementado el predicado `miter/3`, es fácil detectar la equivalencia entre dos circuitos cualesquiera mediante el calibrado de programas. Para ello, sólo es necesario un caso de prueba de la forma:

$$\texttt{true -> miter(C}_\texttt{A}\texttt{, C}_\texttt{B}\texttt{, [\#x}_1\texttt{, ..., \#x}_\texttt{m}\texttt{]).}$$

Este caso de prueba le indica al sistema que queremos encontrar una combinación de entradas $(x_1, \ldots, x_m)$ para los circuitos C_A y C_B tal que la salida del predicado `miter/3` sea `true`, es decir, tal que alguna de las salidas y_i de ambos circuitos sea distinta. Si los circuitos son equivalentes, el sistema no será capaz de encontrar tal asignación de valores, y devolverá una sustitución simbólica arbitraria con una desviación de 1.0 (puesto que habrá fallado con el único caso de prueba que hemos introducido, que será evaluado a `false`). En caso contrario, devolverá una sustitución simbólica para la que alguna de las salidas es distinta en ambos circuitos, con una desviación de 0.0.

Ejemplo 5.3. Comprobemos la equivalencia de los circuitos mostrados en la Figura 5.2. Para ello, lanzamos el proceso de calibrado con el programa del Ejemplo 5.1 y con el siguiente caso de prueba:

```
1 true -> miter(a, b, [#x0, #x1, #x2, #x3]).
```

El entorno FASILL proporciona la siguiente salida, que demuestra que los circuitos son equivalentes, ya que la desviación de la sustitución simbólica encontrada es de 1.0:

```
1 substitution: {#x0/false,#x1/false,#x2/false,#x3/false}
2 deviation: 1.0
```

Ejemplo 5.4. Introduzcamos ahora un tercer circuito combinacional, C_C, no equivalente a los dos anteriores, representado en FASILL por el siguiente predicado `c/2`:

```
1 c([X0,X1,X2,X3], [Y0,Y1]) :-
2     truth_degree(@not('&'(X0,X1)), Y0),
3     truth_degree(@not('|'(X2,X3)), Y1).
```

Lanzamos ahora el proceso de calibrado para comprobar la equivalencia de los circuitos `a/2` y `c/2`. En este caso, el sistema encuentra una combinación de las entradas, $(0, 0, 0, 1)$, para la que ambos circuitos producen salidas diferentes, ya que la desviación es de 0.0:

Tabla 5.1: Tabla de verdad de los circuitos combinacionales C_A, C_B y C_C

Entradas				C_A		C_B		C_C	
x_0	x_1	x_2	x_3	y_0	y_1	w_0	w_1	v_0	v_1
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	1	1	1	0
0	0	1	1	1	0	1	0	1	0
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0
0	1	1	0	1	1	1	1	1	0
0	1	1	1	1	0	1	0	1	0
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	0
1	0	1	1	0	0	0	0	1	0
1	1	0	0	1	1	1	1	0	1
1	1	0	1	1	1	1	1	0	0
1	1	1	0	0	1	0	1	0	0
1	1	1	1	0	0	0	0	0	0

```
1 substitution: {#x0/false,#x1/false,#x2/false,#x3/true}
2 deviation: 0.0
```

Para esta combinación, el circuito `a/2` produce las salidas $(1, 1)$, mientras que el circuito `c/2` produce las salidas $(1, 0)$.

En la Tabla 5.1 se muestran todas las combinaciones posibles de entrada para los tres circuitos combinacionales ejemplificados, `a`, `b` y `c`, junto a sus salidas. En esta tabla se observa que, efectivamente, los circuitos `a` y `b` son equivalentes entre sí, pero el circuito `c` produce salidas distintas para varias de las combinaciones de entrada (entre ellas las que arroja el sistema de calibrado de FASILL).

5.1.3. Resultados experimentales

En la Tabla 5.2 se resumen las medias de los tiempos de ejecución[1] del algoritmo de calibrado, en milisegundos, al comprobar la equivalencia de diversos circuitos combinacionales variando el número de entradas de los mismos.

[1] Cada celda contiene la media tras 100 ejecuciones utilizando un ordenador de sobremesa equipado con un procesador AMD Opteron™ @ 1593 MHz y 2.00 GB RAM.

Tabla 5.2: Tiempo de ejecución (en milisegundos) del algoritmo de calibrado en FASILL y Z3 para la equivalencia de circuitos combinacionales en función del número de entradas

# Entradas	FASILL	Z3
4	45	36
5	930	37
6	2260	40
7	5670	41
8	12200	42
9	29340	43
10	65480	45

Se puede observar que el aumento de constantes simbólicas (que es igual al número de entradas de los circuitos) supone un incremento exponencial en el tiempo de ejecución del método de calibrado del entorno FASILL, mientras que utilizando Z3 el incremento es lineal.

5.2. Regresión lineal

5.2.1. Descripción

Uno de los aspectos más relevantes de la estadística es el análisis de la relación o dependencia entre variables. Frecuentemente, resulta de interés conocer el efecto que una o varias variables pueden causar sobre otra, e incluso predecir valores en una variable a partir de otras [3].

Los métodos de regresión estudian la construcción de modelos para representar la dependencia entre una variable dependiente Y y las variables explicativas X_i. Un modelo de regresión lineal puede ser descrito de la siguiente manera:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n + \epsilon \tag{5.2}$$

siendo:

- $\beta_0, \ldots, \beta_n$ parámetros fijos desconocidos;
- $X_1, \ldots, X_n$ variables explicativas no estocásticas (regresores);
- y ϵ una variable aleatoria inobservable.

En aplicaciones prácticas disponemos de una muestra de observaciones de estas variables, y el modelo anterior sugiere que la relación entre estas se satisface para cada una de las observaciones correspondientes [47].

La ecuación 5.2 indica que la variable aleatoria Y se genera como combinación lineal de las variables explicativas, salvo en una perturbación aleatoria ϵ. El problema que abordamos es el de estimar los parámetros desconocidos $\beta_0, \ldots, \beta_n$. Para ello contamos con una muestra de k observaciones de la variable aleatoria Y, y de los correspondientes valores de las variables explicativas X_i. Debemos encontrar entonces aquellos valores de $\beta_0, \ldots, \beta_n$ que hagan mínimos los errores de estimación.

Ejemplo 5.5. Un viejo estudio [53] midió la frecuencia de chirridos (pulsos por segundo) de los grillos 15 veces, a diferentes temperaturas. En la Tabla 5.3 se muestran los datos resultantes, donde la primera columna representa la temperatura en grados centígrados, y la segunda columna representa el nivel de ruido en decibelios.

Tabla 5.3: Conjunto de datos `cricket.csv`

Temp.	Ruido	Temp.	Ruido	Temp.	Ruido
20.00	88.59	15.50	75.19	15.00	79.59
16.00	71.59	14.69	69.69	17.20	82.59
19.79	93.30	17.10	82.00	16.00	80.59
18.39	84.30	15.39	69.40	17.00	83.50
17.10	80.59	16.20	83.30	14.39	76.30

Se quiere estudiar la relación existente entre la temperatura y el nivel de ruido generado por los grillos. Utilizando estos datos para crear un modelo de regresión, obtenemos—mediante el método de los mínimos cuadrados[2]—la recta $y = 25.23 + 3.29x$ mostrada en la Figura 5.3 junto al diagrama de dispersión de los datos.

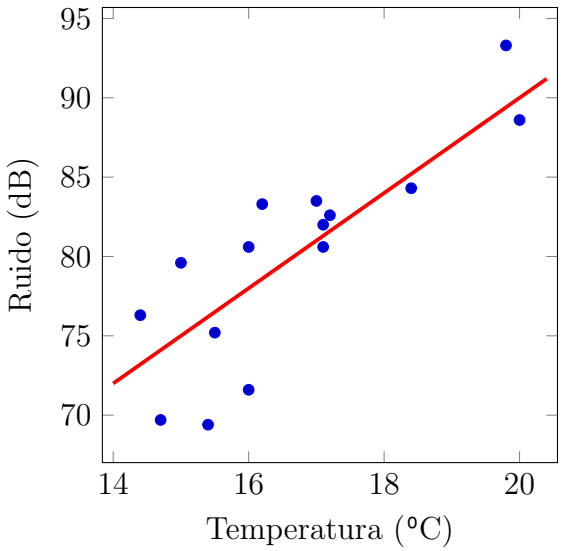


Figura 5.3: Diagrama de dispersión del conjunto de datos `cricket.csv`

[2]Técnica numérica que minimiza la suma de las diferencias al cuadrado de un conjunto de pares.

5.2.2. Implementación

Expresaremos el modelo de regresión como un programa FASILL con una única cláusula que defina la ecuación 5.2 mediante la combinación de las conectivas $|_{add}$ y $\&_{prod}$ del retículo real (véase el Apéndice A.1.3), donde los parámetros β_i serán originalmente constantes simbólicas:

```
y(X1,...,Xn) <- #b0 |add (#b1 &prod X1) |add ... |add (#bn &prod Xn).
```

Este predicado toma n variables explicativas como entrada y se evalúa con un grado de verdad que es combinación lineal de estas entradas. Para encontrar los parámetros β_i que mejor se ajustan a los datos que tenemos, calibraremos este programa introduciendo un caso de prueba por cada muestra del conjunto de datos, donde el valor de la variable dependiente será el grado de verdad esperado del caso de prueba.

Ejemplo 5.6. Calibremos el modelo del Ejemplo 5.5. Dado que el conjunto de datos tiene únicamente una variable explicativa, la temperatura, el programa tendrá dos constantes simbólicas:

```
chirps(Temperature) <- #b0 |add (#b1 &prod Temperature).
```

Cada muestra del conjunto de datos `cricket.csv` se convierte en un caso de prueba, obteniéndose así el siguiente conjunto de casos de prueba:

```
88.6 -> chirps(20.0).
71.6 -> chirps(16.0).
93.3 -> chirps(19.8).
84.3 -> chirps(18.4).
80.6 -> chirps(17.1).
75.2 -> chirps(15.5).
69.7 -> chirps(14.7).
82.0 -> chirps(17.1).
69.4 -> chirps(15.4).
83.3 -> chirps(16.2).
79.6 -> chirps(15.0).
82.6 -> chirps(17.2).
80.6 -> chirps(16.0).
83.5 -> chirps(17.0).
76.3 -> chirps(14.4).
```

Tras ejecutar el proceso de calibrado, el entorno FASILL proporciona la siguiente salida:

```
substitution: {#b0/42.92, #b1/2.28}
deviation: 43.23
```

Podemos observar que estos valores, $\beta_0 = 42.92$ y $\beta_1 = 2.28$, no son los mismos que los obtenidos en el Ejemplo 5.5. Esto es debido a que en el ejemplo anterior se ha minimizado la distancia utilizando el error cuadrático medio, mientras que aquí se ha utilizado el error absoluto medio, que es la noción de distancia definida en el retículo. En la Figura 5.4 se muestra la recta $y = 42.92 + 2.28x$ junto al diagrama de dispersión de los datos, donde la línea discontinua representa el modelo del Ejemplo 5.5.

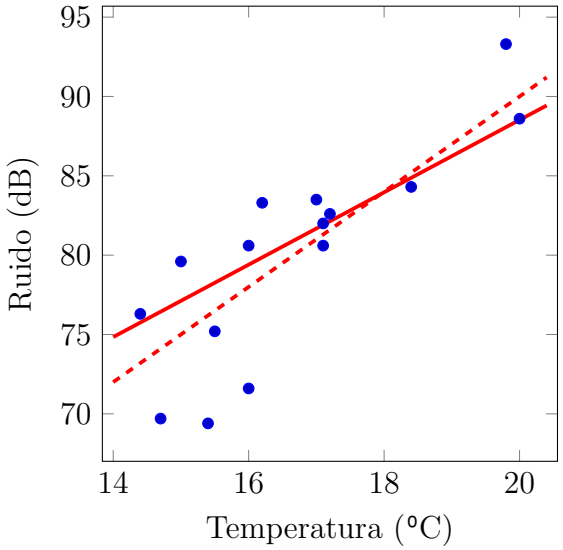


Figura 5.4: Modelo de regresión lineal del Ejemplo 5.6

5.2.3. Resultados experimentales

Dado que los elementos del retículo real no están acotados en un intervalo, resulta complicado realizar una discretización de los mismos adecuada para todo problema de regresión, y aunque fuese posible, el algoritmo de calibrado de FASILL no sería capaz de resolver estos problemas en un tiempo razonable, puesto que tendría que comprobar una gran cantidad de combinaciones. Por lo tanto, en esta sección únicamente mediremos el rendimiento de la nueva técnica de calibrado con Z3.

En la Tabla 5.4 se resumen las medias de los tiempos de ejecución[3] del algoritmo de calibrado, en segundos, al ajustar modelos de regresión variando el número de variables explicativas de los mismos y el número de casos de prueba. Observamos que tanto el incremento de variables explicativas como el incremento de casos de prueba suponen un crecimiento exponencial del tiempo de ejecución del algoritmo de calibrado en Z3, siendo más determinante el número de variables explicativas del modelo que el número de casos de prueba.

[3]Cada celda contiene la media tras 10 ejecuciones utilizando un ordenador de sobremesa equipado con un procesador AMD Opteron™ @ 1593 MHz y 2.00 GB RAM.

Tabla 5.4: Tiempo de ejecución (en segundos) del algoritmo de calibrado en Z3 para regresión lineal en función del número de variables explicativas y casos de prueba

		# Variables explicativas				
		1	2	3	4	5
# Casos de prueba	20	0.35	1.55	6.27	11.38	33.48
	30	2.22	6.45	27.05	49.40	72.34
	40	6.01	25.32	107.68	165.52	378.15
	50	14.04	43.60	184.35	583.45	1547.56
	60	29.80	150.85	619.22	1094.20	3319.37
	70	40.30	213.32	794.55	2452.84	7532.63
	80	51.08	311.07	1058.35	4261.00	17317.25
	90	106.15	625.05	1658.60	7688.81	40313.90
	100	215.10	874.47	3189.72	11204.84	85873.49

5.3. Web semántica

5.3.1. Descripción

La *web semantica* [9] es una extensión de la "World Wide Web" guiada por estándares del *W3C*[4] que promueven formatos de datos comunes y protocolos de intercambio en la web, como RDF o SPARQL. Dotar a la web actual de significado tiene como efecto inmediato que la búsqueda de información sea más eficiente y adecuada a las necesidades del usuario.

El *marco de descripcion de recursos* (RDF) proporciona un modelo de datos simple para expresar declaraciones utilizando tuplas de la forma $(sujeto, predicado, valor)$. Un conjunto de sentencias RDF utiliza un vocabulario particular que define las propiedades y los tipos de datos que son significativos para la aplicación en cuestión [9]. Por otro lado, SPARQL es un lenguaje de consulta de RDF—esto es, un lenguaje de consulta semántica para bases de datos—capaz de recuperar y manipular datos almacenados en formato RDF [57].

Ejemplo 5.7. SPARQL permite el acceso a información disponible en la web a través de diversas plataformas, como DBpedia[5], que proporciona acceso a toda la información de Wikipedia. La siguiente consulta extrae el título de la película y el nombre de las compañías distribuidoras de las películas en las que ha participado el actor Christian Bale, ordenadas ascendentemente por título:

[4] "World Wide Web Consortium", consorcio internacional que genera recomendaciones y estándares que aseguran el crecimiento de la "World Wide Web" a largo plazo.

[5] http://wiki.dbpedia.org

Tabla 5.5: Muestra de la respuesta a la consulta SPARQL del Ejemplo 5.7

?title	?company
Exodus: Gods and Kings	Twentieth Century Fox Film Corporation
The Big Short	Paramount Pictures Corporation
The Dark Knight Trilogy	Warner Bros. Entertainment Inc.
The Fighter	Paramount Pictures Corporation
The Fighter	The Weinstein Company, LLC

```
1 SELECT ?title ?company
2 WHERE {
3     ?film dbo:starring dbr:Christian_Bale ;
4           foaf:name ?title ;
5           dbo:distributor ?distributor .
6     ?distributor foaf:name ?company .
7 }
8 ORDER BY ASC(?title)
```

En la Tabla 5.5 se muestra un resumen de las películas y compañías recuperadas de DBpedia para esta consulta.

Tanto RDF como SPARQL han sido diseñados para consultar información clara y precisa. No obstante, algunos contextos requieren tratar con incertidumbre o conocimiento impreciso. Con este objetivo se diseñó el lenguaje de consulta FSA-SPARQL, una extensión de SPARQL que permite tratar con predicados difusos y agregar grados de verdad mediante conectivas difusas [1].

Ejemplo 5.8. La siguiente consulta FSA-SPARQL extrae el nombre de las películas que sean consideradas buenas y que pertenezcan al género de suspense, donde el grado de verdad debe ser mayor de 0.7:

```
1 SELECT ?name ?rank
2 WHERE {
3     ?movie movie:name ?name .
4     ?movie f:type (movie:genre movie:Thriller ?c) .
5     ?movie f:type (movie:quality movie:Good ?r) .
6     BIND(l:WMEAN(0.5,?r,?c) as ?rank)
7     FILTER (?rank > 0.7)
8 }
```

El grado de verdad de esta consulta se calcula agregando los grados de verdad (línea 6) de los dos predicados difusos que intervienen: que la película sea del género de suspense

Tabla 5.6: Respuesta a la consulta FSA-SPARQL del Ejemplo 5.8

?name	?rank
The American	0.75
The Silence of the Lambs	0.75

(línea 4), y que la película sea buena (línea 5). Para ello, se utiliza el operador $@_{wmean}$, que representa la media ponderada (por el primer argumento) de los dos grados de verdad, cuya función de verdad es $\dot{@}_{wmean}(x, y, z) \ , \ xy + (1 - x)z$. Nótese que no todos los predicados tienen porqué ser difusos, como por ejemplo, el nombre de la película. En la Tabla 5.6 se muestran las películas recuperadas para esta consulta junto a su grado de verdad.

El objetivo ahora es calibrar este tipo de consultas difusas para seleccionar los grados de verdad y las conectivas más adecuadas en base a información previa proporcionada por el usuario, como podría ser el valor de verdad esperado para determinadas películas ante la consulta del Ejemplo 5.8.

5.3.2. Implementación

FSA-SPARQL es una extensión de SPARQL en la que las tuplas RDF son reemplazadas por tuplas RDF difusas. Además, es posible utilizar conectivas difusas—inspiradas por las lógicas del producto, de Gödel y de Łukasiewicz—y modificadores lingüísticos en las consultas.

Para poder calibrar las consultas FSA-SPARQL, primero es necesario codificarlas como programas FASILL. En [2] se detalla el proceso de traducción de FSA-SPARQL a FASILL, que puede ejecutarse automáticamente desde el entorno FLOPER online a través de la URL http://dectau.uclm.es/floper/fsa-sparql, tal y como se muestra en la Figura 5.5. En el Apéndice A.1.4 se describe el retículo utilizado en esta sección, el cual es una adaptación del retículo unitario en el intervalo $[0, 1]$ para hacerlo compatible con el lenguaje FSA-SPARQL y sus conectivas.

Ejemplo 5.9. La consulta FSA-SPARQL mostrada en el Ejemplo 5.8 se traduce al siguiente programa FASILL, donde el predicado `rdf/3` se utiliza para recuperar la información de las tuplas RDF difusas:

```
1 query(NAME,RANK):-
2     rdf(MOVIE,'http://www.movies.org#name',NAME),
3     rdf(MOVIE,'http://www.fuzzy.org#type',X0),
4     rdf(X0,'http://www.fuzzy.org#onProperty',
5         'http://www.movies.org#genre'),
6     rdf(X0,'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
```

```
7          'http://www.movies.org#Thriller'),
8       rdf(X0,'http://www.fuzzy.org#truth',C),
9       rdf(MOVIE,'http://www.fuzzy.org#type',X1),
10      rdf(X1,'http://www.fuzzy.org#onProperty',
11          'http://www.movies.org#quality'),
12      rdf(X1,'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
13          'http://www.movies.org#Good'),
14      rdf(X1,'http://www.fuzzy.org#truth',R),
15      0.5^^'http://www.w3.org/2001/XMLSchema#decimal'=A0_1,
16      R=A1_1,
17      C=A2_1,
18      VAR1=RANK,
19      'http://www.lattice.org#WMEAN'(A0_1,A1_1,A2_1,VAR1),
20      RANK=A6,
21      0.7^^'http://www.w3.org/2001/XMLSchema#decimal'=B6,
22      { A6>B6 }.
```

A este predicado traducido a partir de la consulta original, `query/2`, hay que añadirle las cláusulas RDF necesarias y la definición de los predicados que representan las conectivas difusas, como por ejemplo `'http://www.lattice.org#WMEAN'/4`. Además, por eficiencia, es conveniente desactivar los pasos de fallo de la semántica de FASILL mediante la directiva `:- set_fasill_flag(failure_steps, false)`. Si ejecutamos el objetivo difuso $\mathcal{Q}$ = `query(Name, Rank) & Rank` en el entorno FLOPER, obtenemos las siguientes fca's, que se corresponden con las respuestas de la Tabla 5.6:

$$\mathcal{Q}_1^{\emptyset} = \langle \texttt{0.75\^\^'http://www.w3.org/2001/XMLSchema\#decimal'}, \{ Name/{}^{\emptyset}\mathit{The\ American}^{\emptyset}\texttt{\^\^}^{\emptyset}\mathit{http://www.w3.org/2001/XMLSchema\#string}^{\emptyset}, Rank/0.75\texttt{\^\^}^{\emptyset}\mathit{http://www.w3.org/2001/XMLSchema\#decimal}^{\emptyset}\}\rangle$$

$$\mathcal{Q}_2^{\emptyset} = \langle \texttt{0.75\^\^'http://www.w3.org/2001/XMLSchema\#decimal'}, \{ Name/{}^{\emptyset}\mathit{The\ Silence\ of\ the\ Lambs}^{\emptyset}\texttt{\^\^}^{\emptyset}\mathit{http://.../XMLSchema\#string}^{\emptyset}, Rank/0.75\texttt{\^\^}^{\emptyset}\mathit{http://www.w3.org/2001/XMLSchema\#decimal}^{\emptyset}\}\rangle$$

Ejemplo 5.10. Supongamos que queremos calibrar la consulta del Ejemplo 5.9 para que "Ocean's Eleven" sea considerada una buena película de suspense. Introducimos una constante simbólica v^{s1} en el peso de la conectiva $@_{wmean}$, cuyo valor era 0.5 (dándole el mismo peso a cada grado de verdad). Además, introducimos otra constante simbólica en el filtro, v^{s2}, cuyo valor era 0.7. Ahora, ejecutamos el proceso de calibrado con el siguiente conjunto de casos de prueba, donde únicamente tendremos en cuenta los posibles valores $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ a la hora de generar las sustituciones simbólicas:

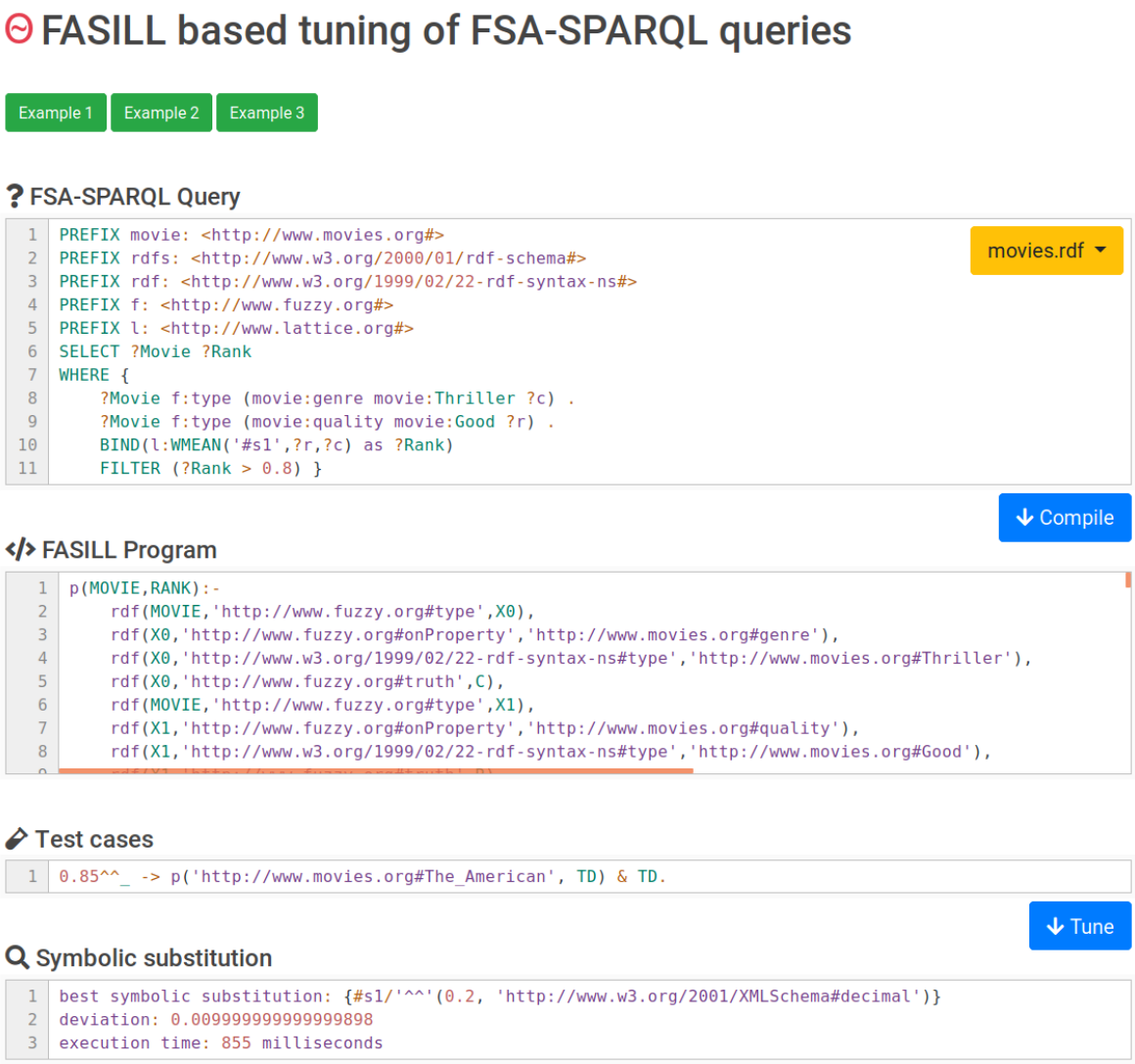


Figura 5.5: Captura de pantalla de la herramienta online para calibrar consultas FSA-SPARQL mediante FASILL

```
0.9^^_ -> query('The American'^^_, Rank) & Rank.
0.8^^_ -> query('The Silence of the Lambs'^^_, Rank) & Rank.
0.5^^_ -> query('Ocean''s Eleven'^^_, Rank) & Rank.
0.0^^_ -> query('Casablanca'^^_, Rank) & Rank.
```

Obtenemos la siguiente sustitución simbólica $\Theta = \{v^{s1}/0.4, v^{s2}/0.1\}$ por parte del entorno FASILL con una desviación de 0.2, donde el peso debería ser 0.4 y el filtro 0.1:

```
substitution: {
    #s1/0.4^^'http://www.w3.org/2001/XMLSchema#decimal'),
    #s2/0.1^^'http://www.w3.org/2001/XMLSchema#decimal')
}
deviation: 0.2
```

Es decir, tras el proceso de calibrado, la consulta FSA-SPARQL original del Ejemplo 5.8 queda finalmente de la siguiente forma:

```
1 SELECT ?name ?rank
2 WHERE {
3     ?movie movie:name ?name .
4     ?movie f:type (movie:genre movie:Thriller ?c) .
5     ?movie f:type (movie:quality movie:Good ?r) .
6     BIND(l:WMEAN(0.4,?r,?c) as ?rank)
7     FILTER (?rank > 0.1)
8 }
```

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Este Trabajo Fin de Máster 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 nuevos mecanismos de calibrado automático para las extensiones simbólicas de los lenguajes difusos mencionados.

Se detallan los formalismos subyacentes de la lógica difusa y la programación lógica, así como de las técnicas de 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 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 el director del trabajo. Con este Trabajo Fin de Máster 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 [2, 18, 19, 39, 40, 41, 42, 43].

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

1. Creación de un nuevo paquete para el intérprete SWI-Prolog, para analizar y escribir expresiones, guiones, declaraciones de teorías y declaraciones de lógicas en SMT-LIB.

2. Implementación del nuevo método de calibrado basado en el uso de solucionadores de satisfacibilidad módulo teorías, para programas sMALP y sFASILL, y su integración en la herramienta online.

3. Descripción de aplicaciones reales de las técnicas de calibrado de programas lógicos difusos a problemas no triviales como son el aprendizaje automático, la verificación de circuitos o la web semántica.

4. Medición y comparación del tiempo de ejecución de los diversos métodos de calibrado, mostrando las ventajas que los solucionadores SMT aportan a las técnicas de calibrado originales del entorno FLOPER.

6.2. Trabajo futuro

Actualmente, la extensión simbólica del lenguaje FASILL solo permite introducir grados de verdad y conectivos simbólicos en las reglas del programa. En el futuro, nos proponemos permitir que las relaciones de similitud acepten también constantes simbólicas, esto es, grados de verdad 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 similitud, también es necesario modificar el proceso de unificación débil de sFASILL. Esta mejora podría suponer un gran impacto en el calibrado de programas sFASILL, debido a que un elemento simbólico en una relación de similitud (por ejemplo en la t-norma asociada) puede afectar a muchas más reglas de un programa que una constante simbólica en una sola cláusula.

En este trabajo hemos adaptado algunas teorías de SMT al calibrado de programas lógicos difusos, como la teoría de los reales, mediante la construcción de los retículos apropiados en SMT-LIB. No obstante, todavía faltan por explorar otras teorías, como la teoría de los vectores de bits o la teoría de los tipos de datos inductivos, que podrían resultar interesantes en determinadas aplicaciones.

Por último, nos hemos centrado, asimismo, en la integración de la nueva técnica de calibrado sobre el entorno FLOPER online, pero, debido al volumen de tareas abordadas, no ha sido posible integrarla también en el nuevo entorno gráfico de escritorio de FLOPER. Por tanto, en el futuro nos proponemos integrar este mecanismo en el nuevo editor implementado en SWI-Prolog, ampliando su interfaz con el paquete gráfico XPCE.

Bibliografía

[1] J. M. Almendros-Jiménez, A. Becerra-Terón, y G. Moreno. A fuzzy extension of SPARQL based on fuzzy sets and aggregators. En *2017 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2017, Naples, Italy, July 9-12, 2017*, páginas 1–6, 2017. 76

[2] J. M. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, y J. A. Riaza. Tuning Fuzzy SPARQL Queries in a Fuzzy Logic Programming Environment. En *2019 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2019*, páginas 1–6, 2019 (sometido). I, 77, 81

[3] B. Vega. Ensayo de Regresión Lineal; Aplicación sencilla del método en el análisis de variables. Disponible en: https://en.calameo.com/read/000981011fa66d1f3904e. Accedido el 11 de enero de 2019. 71

[4] L. Bachmair, A. Tiwari, y L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003. 16

[5] C. Barrett, D. L. Dill, y J. R. Levitt. A decision procedure for bit-vector arithmetic. En *Proceedings of the 35th annual Design Automation Conference*, páginas 522–527. ACM, 1998. 18

[6] C. Barrett, R. Sebastiani, S. A. Seshia, y C. Tinelli. *Satis ability Modulo Theories, Handbook of Satis ability, chapter 12*. IOS Press, 2008. 15, 16

[7] C. Barrett, I. Shikanian, y C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satis ability, Boolean Modeling and Computation*, 3:21–46, 2007. 18

[8] C. Barrett, A. Stump, y C. Tinelli. The smt-lib standard: Version 2.0. En *Proceedings of the 8th International Workshop on Satis ability Modulo Theories (Edinburgh, England)*, volumen 13, página 14, 2010. 35, 39, 41

[9] T. Berners-Lee, J. Hendler, y O. Lassila. The semantic web. *Scienti c american*, 284(5):34–43, 2001. 75

[10] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, y B. Brady. Deciding bit-vector arithmetic with abstraction. En *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 358–372. Springer, 2007. 18

[11] D. Cyrluk, O. Möller, y H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. En *International Conference on Computer Aided Veri cation*, páginas 60–71. Springer, 1997. 18

[12] J. H. Davenport y J. Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1-2):29–35, 1988. 17

[13] M. Davis, G. Logemann, y D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 14

[14] M. Davis y H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. 13

[15] D. Detlefs, G. Nelson, y J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005. 35, 37

[16] B. Dutertre y L. De Moura. A fast linear-arithmetic solver for dpll (t). En *International Conference on Computer Aided Veri cation*, páginas 81–94. Springer, 2006. 38

[17] J. A. Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1969. 22

[18] J. A. Guerrero, F. Mendieta, G. Moreno, J. Penabad, y J. A. Riaza. Testing properties of fuzzy connectives and truth degrees with the latticemaker tool. En *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA*, páginas 1–8. IEEE, 2017. I, 81

[19] J. A. Guerrero, G. Moreno, J. A. Riaza, y J. Sánchez. Smart Design of Similarity Relations for Fuzzy Logic Programming Environments. En *2018 IEEE Symposium Series on Computational Intelligence, Foundations of Computational Intelligence, FOCI 2018, USA*, páginas 220–227. IEEE, 2018. I, 81

[20] 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

[21] M. Johnson. Two ways of formalizing grammars. *Linguistics and Philosophy*, 17(3):221–240, 1994. 56

[22] P. Julián. *Logica Simbolica para Informaticos*. Ra-Ma, 2004. 5, 6, 7, 8, 9

[23] P. Julián y M. Alpuente. *Programacion logica. Teor a y practica*. Pearson, 2007. 9, 10, 11, 19, 20, 21, 22

[24] P. Julián, G. Moreno, y J. Penabad. On the declarative semantics of multi-adjoint logic programs. En *Proc. of 10th International Work-Conference on Arti cial Neural Networks, IWANN'09*, páginas 253–260. Springer-Verlag, LNCS 5517, 2009. 30

[25] P. Julián, G. Moreno, y J. Penabad. Thresholded semantic framework for a fully integrated fuzzy logic language. *J. Log. Algebr. Meth. Program.*, 93:42–67, 2017. 27

[26] P. Julián, G. Moreno, y J. Penabad. FASILL: Fuzzy Correct Answers and Soundnes. En *2018 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, páginas 1–8, 2018. 27

[27] 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 Programacion y Lenguajes, PROLE'14*, volumen 173, páginas 71–86, Cádiz, España, 2015. EPTCS. 27, 28, 29, 30, 33

[28] 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. 30

[29] N. Karmarkar. A new polynomial-time algorithm for linear programming. En *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, páginas 302–311. ACM, 1984. 17

[30] A. Kolesárová y M. Komorníková. Triangular norm-based iterative compensatory operators. *Fuzzy Sets and Systems*, 104(1):109 120, 1999. 25

[31] R. C. T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972. 24, 27

[32] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin. Segunda edición, 1987. 27

[33] S. Malik y G. Weissenbacher. Boolean satisfiability solvers: techniques and extensions. *Software Safety and Security-Tools for Analysis and Veri cation, NATO Science for Peace and Security Series*, 2012. 11, 12, 13

[34] J. Marques-Silva. Practical applications of boolean satisfiability. En *2008 9th International Workshop on Discrete Event Systems*, páginas 74–80, May 2008. 66

[35] J. Marques-Silva y T. Glass. Combinational equivalence checking using satisfiability and recursive learning. En *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, páginas 145–149, March 1999. 65

[36] Y. V. Matijasevič. Diophantine representation of recursively enumerable predicates. En *Studies in Logic and the Foundations of Mathematics*, volumen 63, páginas 171–177. Elsevier, 1971. 17

[37] J. Medina, M. Ojeda-Aciego, y P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004. 30

[38] Microsoft Research. Getting started with z3: A guide. https://rise4fun.com/Z3/tutorial, 2018. 41

[39] G. Moreno, J. Penabad, y J. A. Riaza. On Similarity-Based Unfolding. En *Scalable Uncertainty Management - 11th International Conference, SUM 2017, Granada, Spain, October 4-6, 2017, Proceedings*, volumen 10564, páginas 420–426. Springer, 2017. I, 63, 81

[40] G. Moreno, J. Penabad, y J. A. Riaza. Symbolic unfolding of multi-adjoint logic programs. En *Trends in Mathematics and Computational Intelligence*, páginas 43–51. Springer, Cham, 2019. I, 63, 81

[41] G. Moreno, J. Penabad, J. A. Riaza, y G. Vidal. Symbolic Execution and Thresholding for Efficiently Tuning Fuzzy Logic Programs. En *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volumen 10184, páginas 131–147. Springer, 2016. I, 47, 49, 63, 81

[42] G. Moreno y J. A. Riaza. An online tool for tuning fuzzy logic programs. En *Rules and Reasoning - International Joint Conference, RuleML+RR 2017, London, UK, July 12-15, 2017, Proceedings*, volumen 10364, páginas 184–198. Springer, 2017. I, 47, 63, 81

[43] G. Moreno y J. A. Riaza. An online tool for unfolding symbolic fuzzy logic programs. En *Actas de las XVIII Jornadas sobre Programacion y Lenguajes, PROLE'18*, páginas 1–17, Sevilla, España, 2018. Disponible en: https://biblioteca.sistedes.es/submissions/descargas/2018/PROLE/2018-PROLE-005.pdf. I, 63, 81

[44] L. De Moura y N. Bjørner. Relevancy propagation. Technical report, Technical Report MSR-TR-2007-140, Microsoft Research, 2007. 38

[45] L. De Moura y N. Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008. 35, 38

[46] L. De Moura y N. Bjørner. Z3: An efficient smt solver. En *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 337–340. Springer, 2008. 35, 36

[47] J. Neter, M. H. Kutner, C. J. Nachtsheim, y W. Wasserman. *Applied linear statistical models*, volumen 4. Irwin Chicago, 1996. 71

[48] H. T. Nguyen y E. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida. Tercera edición, 2006. 24, 25

[49] D. C. Oppen. Reasoning about recursively defined data structures. En *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 151–157. ACM, 1978. 18

[50] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981. 17

[51] J. Pavelka. On fuzzy logic I, II, III. *Zeitschrift fur Mathemathische Logik und Grundlagen der Mathematik*, 25:45–52, 119–134, 447–464, 1979. 22

[52] J. Penabad. *Desplegado de Programas Logicos Difusos*. Tesis doctoral, Universidad de Castilla-La Mancha, 2010. 22, 26, 27

[53] G. W. Pierce. *The Songs of Insects; with Related Material on the Production, Propagation, Detection, and Measurement of Sonic and Supersonic Vibrations*. Harvard University Press, 1946. 72

[54] J. A. Riaza. *Implementacion de tecnicas de desplegado difuso sobre el entorno FLOPER*. Trabajo Fin de Grado, Universidad de Castilla-La Mancha, Junio 2017. III, 6, 47

[55] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965. 13

[56] B. Schweizer y A. Sklar. *Probabilistic Metric Spaces*. North-Holland, New York, 1983. 24

[57] T. Segaran, C. Evans, y J. Taylor. *Programming the Semantic Web: Build Flexible Applications with Graph Data*. .º'Reilly Media, Inc.", 2009. 75

[58] M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Elsevier, TCS*, 275(1-2):389–426, 2002. 28, 29, 30, 31

[59] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell system technical journal*, 28(1):59–98, 1949. 14

[60] G. Steele. *Common LISP: the language*. Elsevier, 1990. 40

[61] N. Suzuki y D. Jefferson. Verification decidability of presburger array programs. *Journal of the ACM (JACM)*, 27(1):191–205, 1980. 18

[62] G. S. Tseitin. On the complexity of derivation in propositional calculus. En *Automation of reasoning*, páginas 466–483. Springer, 1983. 12

[63] M. S. Ying. A logic for approximated reasoning. *Journal of Symbolic Logic*, 59(3):830–837, 1994. 28

[64] L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428, 1965. 22

[65] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965. 22, 23

[66] 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. 23

Contenido del CD

El disco adjunto a esta memoria contiene el código implementado en este Trabajo Fin de Máster:

- El directorio `./prolog-smtlib/` contiene el repositorio completo del paquete de SWI-Prolog para analizar y escribir expresiones, guiones, declaraciones de lógicas y declaraciones de teorías en lenguaje SMT-LIB, el cual incluye el código fuente, documentación del mismo, y todas las versiones publicadas hasta la fecha.
- El directorio `./fasill/` contiene el repositorio completo del proyecto del lenguaje FASILL, el cual incluye el código fuente, documentación del mismo, retículos y pruebas unitarias del lenguaje. La principal aportación de este trabajo, el calibrado de programas mediante la herramienta Z3, puede encontrarse en el fichero `./fasill/src/tuning-smt.pl`.
- El directorio `./sat/` contiene el repositorio completo de un solucionador de satisfacibilidad booleana para fórmulas proposicionales en forma normal clausal, implementado en C, el cual incluye el código fuente, documentación del mismo, y pruebas unitarias. Este proyecto se inició con el objetivo de estudiar y comprender los principales algoritmos de SAT.

El código fuente de estos proyectos se encuentra bajo la licencia BSD de 3 cláusulas, y los repositorios pueden consultarse online a través de las siguientes URLs:

- https://github.com/jariazavalverde/prolog-smtlib
- https://github.com/jariazavalverde/fasill
- https://github.com/jariazavalverde/sat

Apéndice A

Retículos completos

En esta sección se muestra la implementación en Prolog y en SMT-LIB de los retículos utilizados en este trabajo. Las Figuras A.1, A.2 y A.3 representan gráficamente los diferentes retículos como grafos acíclicos dirigidos, donde existe un arco desde un nodo x a un nodo y si $x \leq y$.

A.1. Retículos completos en Prolog

A.1.1. Retículo lógico

Este retículo en el dominio lógico es un conjunto que contiene exactamente dos elementos, cuyas interpretaciones incluyen *falso* y *verdadero* (donde $falso \leq verdadero$), representados por los términos `false` y `true`, respectivamente. Se utiliza la definición de *distancia discreta* mostrada en el Ejemplo 3.4 para calibrar programas con este retículo.

Es importante reseñar que las conectivas $@_{not}$ y $@_{xor}$ aquí definidas no satisfacen la condición de monotonía requerida en la Definición 2.22 para ser consideradas agregadores, formalmente. No obstante, se incluyen en el retículo por comodidad a la hora de trabajar con las técnicas de calibrado.

```
1  % Elements
2  member(false).
3  member(true).
4  members([false,true]).
5
6  % Distance
7  distance(X, X, 0.0).
8  distance(_, _, 1.0).
9
10 % Ordering relation
```

```
11 leq(false, _).
12 leq(_, true).
13
14 % Supremum and infimum
15 bot(false).
16 top(true).
17
18 % Binary operations
19 and_bool(true, true, true).
20 and_bool(_, _, false).
21 or_bool(false, false, false).
22 or_bool(_, _, true).
23
24 % Aggregators
25 agr_xor(X, X, false).
26 agr_xor(_, _, true).
27 agr_not(false, true).
28 agr_not(true, false).
29
30 % Default connectives
31 tnorm(bool).
32 tconorm(bool).
```

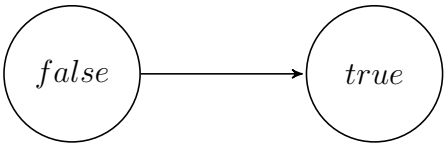


Figura A.1: Representación del retículo lógico

A.1.2. Retículo unitario

Este retículo contiene los elementos del intervalo unitario $[0, 1]$ con la relación de orden usual de los números reales, y está equipado con las conjunciones y disyunciones pertenecientes a las lógicas del producto, Gödel y Łukasiewicz, además de otros agregadores como $@_{aver}$ y $@_{geom}$, y modificadores lingüísticos como $@_{very}$ (véase la Figura 2.2).

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

```
6  distance(X,Y,Z) :- Z is abs(Y-X).
7
8  % Ordering relation
9  leq(X,Y) :- X =< Y.
10
11 % Supremum and infimum
12 bot(0.0).
13 top(1.0).
14
15 % Binary operations
16 and_prod(X,Y,Z) :- Z is X*Y.
17 and_godel(X,Y,Z) :- Z is min(X,Y).
18 and_luka(X,Y,Z) :- Z is max(X+Y-1,0).
19 or_prod(X,Y,Z) :- U1 is X*Y, U2 is X+Y, Z is U2-U1.
20 or_godel(X,Y,Z) :- Z is max(X,Y).
21 or_luka(X,Y,Z) :- Z is min(X+Y,1).
22
23 % Aggregators
24 agr_aver(X,Y,Z) :- Z is (X+Y)/2.
25 agr_very(X,Y) :- Y is X*X.
26
27 % Default connectives
28 tnorm(godel).
29 tconorm(godel).
```

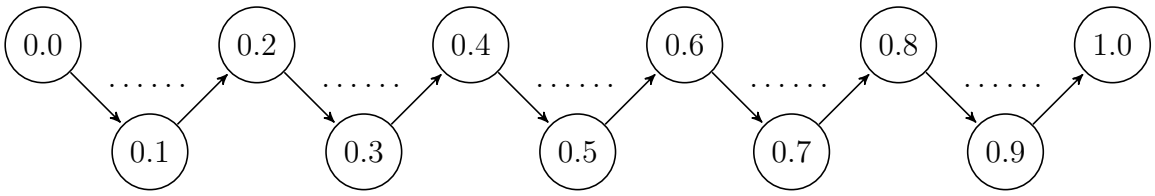


Figura A.2: Representación del retículo unitario

A.1.3. Retículo real

Este retículo contiene los elementos de la recta real extendida, que está formada por todos los elementos de R más los elementos $+\infty$ (infinito positivo) y $-\infty$ (infinito negativo). En este retículo sólo consideramos los elementos pertenecientes a R para calibrar, con la noción de distancia usual.

Tal y como ocurre con otras conectivas en retículos anteriores, aquí la conectiva $\&_{prod}$ no satisface la condición de monotonía requerida en la Definición 2.20 para ser

considerada una norma triangular, formalmente. No obstante, es incluida en el retículo también por comodidad a la hora de trabajar con las técnicas de calibrado.

```
% Elements
member(X) :- number(X).
member(-inf).
member(+inf).
members([-1.0, -0.5, 0.0, 0.5, 1.0]).

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

% Ordering relation
leq(_, +inf).
leq(-inf, _).
leq(X, Y) :- X =< Y.

% Supremum and infimum
bot(-inf).
top(+inf).

% Binary operations
or_add(_, +inf, +inf).
or_add(+inf, _, +inf).
or_add(_, -inf, -inf).
or_add(-inf, _, -inf).
or_add(X, Y, Z) :- Z is X+Y.

and_prod(X, +inf, X).
and_prod(+inf, X, X).
and_prod(X, -inf, X).
and_prod(-inf, X, X).
and_prod(X, Y, Z) :- Z is X*Y.

% Default connectives
tnorm(prod).
tconorm(add).
```

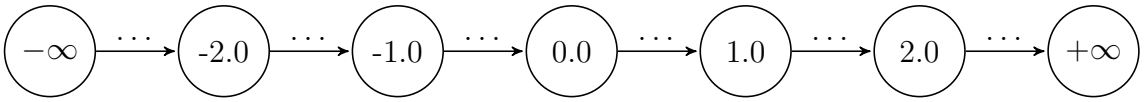


Figura A.3: Representación del retículo real

A.1.4. Retículo FSA-SPARQL

Este retículo es una modificación del retículo unitario, cuyos elementos en el intervalo $[0, 1]$ van asociados a su tipo, `http://www.w3.org/2001/XMLSchema#decimal`, mediante el operador infijo `^^/2` declarado al principio del retículo con prioridad 200. Además, incluye otros agregadores adicionales utilizados comúnmente en las consultas FSA-SPARQL, como por ejemplo la media ponderada $@_{wmean}$.

```
1  :- op(200, xfx, ^^).
2
3  % Elements
4  member(X^^'http://www.w3.org/2001/XMLSchema#decimal') :-
5      number(X), 0 =< X, X =< 1.
6  members([
7      0.0^^'http://www.w3.org/2001/XMLSchema#decimal',
8      0.1^^'http://www.w3.org/2001/XMLSchema#decimal',
9      0.2^^'http://www.w3.org/2001/XMLSchema#decimal',
10     0.3^^'http://www.w3.org/2001/XMLSchema#decimal',
11     0.4^^'http://www.w3.org/2001/XMLSchema#decimal',
12     0.5^^'http://www.w3.org/2001/XMLSchema#decimal',
13     0.6^^'http://www.w3.org/2001/XMLSchema#decimal',
14     0.7^^'http://www.w3.org/2001/XMLSchema#decimal',
15     0.8^^'http://www.w3.org/2001/XMLSchema#decimal',
16     0.9^^'http://www.w3.org/2001/XMLSchema#decimal',
17     1.0^^'http://www.w3.org/2001/XMLSchema#decimal'
18 ]).
19
20 % Distance
21 distance(X^^T,Y^^T,Z) :- Z is abs(Y-X).
22
23 % Ordering relation
24 leq(X^^T,Y^^T) :- X =< Y.
25
26 % Supremum and infimum
27 bot(0.0^^'http://www.w3.org/2001/XMLSchema#decimal').
28 top(1.0^^'http://www.w3.org/2001/XMLSchema#decimal').
29
```

```
% Conjuctions
and_prod(X^^T,Y^^T,Z^^T) :- Z is X*Y.
and_god(X^^T,Y^^T,Z^^T) :- Z is min(X,Y).
and_luk(X^^T,Y^^T,Z^^T) :- Z is max(X+Y-1,0).

% Disjunctions
or_prod(X^^T,Y^^T,Z^^T) :- U1 is X*Y, U2 is X+Y, Z is U2-U1.
or_god(X^^T,Y^^T,Z^^T) :- Z is max(X,Y).
or_luk(X^^T,Y^^T,Z^^T) :- Z is min(X+Y,1).

% Aggregators
agr_mean(X^^T,Y^^T,Z^^T) :- Z is 0.5*X+0.5*Y.
agr_wmean(W^^T,X^^T,Y^^T,Z^^T) :- Z is W*X+(1-W)*Y.
agr_wsum(U^^T,X^^T,V^^T,Y^^T,Z^^T) :- Z is U*X+V*Y.
agr_wmax(U^^T,X^^T,V^^T,Y^^T,Z^^T) :-
    Z is max(min(U,X)+min(V,Y)).
agr_wmin(U^^T,X^^T,V^^T,Y^^T,Z^^T) :-
    Z is min(max(1-U,X)+max(1-V,Y)).
agr_very(X^^T,Z^^T) :- Z is X*X.
agr_more_or_less(X^^T,Z^^T) :-Z is sqrt(X).
agr_close_to(X^^T,L^^T,A^^T,Z^^T) :- Z is 1/(1+((X-L)/A)^2).
agr_at_least(X^^T,L^^T,A^^T,Z^^T) :- X =< A, !, Z is 0.0.
agr_at_least(X^^T,L^^T,A^^T,Z^^T) :-
    A < X, X < L,!, Z is (X-A)/(L-A).
agr_at_least(X^^T,L^^T,A^^T,Z^^T) :- L =< X, !, Z is 1.0.
agr_at_most(X^^T,L^^T,A^^T,Z^^T) :- X >= A, !, Z is 0.0.
agr_at_most(X^^T,L^^T,A^^T,Z^^T) :-
    A > X, X > L, !, Z is (A-X)/(A-L).
agr_at_most(X^^T,L^^T,A^^T,Z^^T) :- X =< L,!, Z is 1.
agr_gt(X^^T, Y^^T, Z^^T) :- X > Y -> Z = 1.0 ; fail.

% Default connectives
tnorm(god).
tconorm(god).
```

A.2. Retículos completos en SMT-LIB

A.2.1. Retículo lógico

```
;; Boolean lattice
```

```
2
3  ;; distance
4  (define-fun lat!distance ((x Bool) (y Bool)) Real
5      (ite (= x y) 0.0 1.0))
6
7  ;; &bool
8  (define-fun lat!and!bool!2 ((x Bool) (y Bool)) Bool
9      (and x y))
10
11 ;; |bool
12 (define-fun lat!or!bool!2 ((x Bool) (y Bool)) Bool
13     (or x y))
14
15 ;; @xor
16 (define-fun lat!agr!xor!2 ((x Bool) (y Bool)) Bool
17     (xor x y))
18
19 ;; @not
20 (define-fun lat!agr!not!1 ((x Bool)) Bool
21     (not x))
22
23 ;; domain of symbolic conjunctions (arity 2)
24 (define-fun dom!sym!and!2 ((s String)) Bool
25     (or
26         (= s "and_bool")))
27
28 ;; domain of symbolic disjunctions (arity 2)
29 (define-fun dom!sym!or!2 ((s String)) Bool
30     (or
31         (= s "or_bool")))
32
33 ;; domain of symbolic aggregators (arity 1)
34 (define-fun dom!sym!agr!1 ((s String)) Bool
35     (or
36         (= s "agr_not")))
37
38 ;; domain of symbolic aggregators (arity 2)
39 (define-fun dom!sym!agr!2 ((s String)) Bool
40     (or
41         (= s "agr_xor")))
42
```

```
43 ;; eval symbolic conjunctions (arity 2)
44 (define-fun call!sym!and!2 ((s String) (x Bool) (y Bool))
      Bool
45     (lat!and!bool!2 x y))
46
47 ;; eval symbolic disjunctions (arity 2)
48 (define-fun call!sym!or!2 ((s String) (x Bool) (y Bool)) Bool
49     (lat!or!bool!2 x y))
50
51 ;; eval symbolic aggregatos (arity 1)
52 (define-fun call!sym!agr!1 ((s String) (x Bool)) Bool
53     (lat!agr!not!1 x))
54
55 ;; eval symbolic aggregators (arity 2)
56 (define-fun call!sym!agr!2 ((s String) (x Bool) (y Bool))
      Bool
57 (lat!agr!xor!2 x y))
```

A.2.2. Retículo unitario

```
1  ;; Unit lattice
2
3  ;; max
4  (define-fun max ((x Real) (y Real)) Real
5      (ite (>= x y) x y))
6
7  ;; min
8  (define-fun min ((x Real) (y Real)) Real
9      (ite (<= x y) x y))
10
11 ;; member
12 (define-fun lat!member ((x Real)) Bool
13     (and (<= 0.0 x) (<= x 1.0)))
14
15 ;; distance
16 (define-fun lat!distance ((x Real) (y Real)) Real
17     (abs (- y x)))
18
19 ;; |luka
20 (define-fun lat!or!luka!2 ((x Real) (y Real)) Real
21     (min (+ x y) 1))
```

```
22
23 ;; |prod
24 (define-fun lat!or!prod!2 ((x Real) (y Real)) Real
25     (- (+ x y) (* x y)))
26
27 ;; |godel
28 (define-fun lat!or!godel!2 ((x Real) (y Real)) Real
29     (max x y))
30
31 ;; &luka
32 (define-fun lat!and!luka!2 ((x Real) (y Real)) Real
33     (max (- (+ x y) 1) 0))
34
35 ;; &prod
36 (define-fun lat!and!prod!2 ((x Real) (y Real)) Real
37     (* x y))
38
39 ;; &godel
40 (define-fun lat!and!godel!2 ((x Real) (y Real)) Real
41     (min x y))
42
43 ;; @aver
44 (define-fun lat!agr!aver!2 ((x Real) (y Real)) Real
45     (/ (+ x y) 2))
46
47 ;; @very
48 (define-fun lat!agr!very!1 ((x Real)) Real
49     (* x x))
50
51 ;; domain of symbolic conjunctions (arity 2)
52 (define-fun dom!sym!and!2 ((s String)) Bool
53     (or
54         (= s "and_godel")
55         (= s "and_luka")
56         (= s "and_prod")))
57
58 ;; domain of symbolic disjunctions (arity 2)
59 (define-fun dom!sym!or!2 ((s String)) Bool
60     (or
61         (= s "or_godel")
62         (= s "or_luka")
```

```
63          (= s "or_prod")))
64
65  ;; domain of symbolic aggregators (arity 1)
66  (define-fun dom!sym!agr!1 ((s String)) Bool
67      (or
68          (= s "agr_very")))
69
70  ;; domain of symbolic aggregators (arity 2)
71  (define-fun dom!sym!agr!2 ((s String)) Bool
72      (or
73          (= s "agr_aver")))
74
75  ;; eval symbolic conjunctions (arity 2)
76  (define-fun call!sym!and!2 ((s String) (x Real) (y Real))
        Real
77      (ite
78          (= s "and_godel")
79          (lat!and!godel!2 x y)
80          (ite
81              (= s "and_luka")
82              (lat!and!luka!2 x y)
83              (lat!and!prod!2 x y))))
84
85  ;; eval symbolic disjunctions (arity 2)
86  (define-fun call!sym!or!2 ((s String) (x Real) (y Real)) Real
87      (ite
88          (= s "or_godel")
89          (lat!or!godel!2 x y)
90          (ite
91              (= s "or_luka")
92              (lat!or!luka!2 x y)
93              (lat!or!prod!2 x y))))
94
95  ;; eval symbolic aggregatos (arity 1)
96  (define-fun call!sym!agr!1 ((s String) (x Real)) Real
97      (lat!agr!very!1 x))
98
99  ;; eval symbolic aggregators (arity 2)
100 (define-fun call!sym!agr!2 ((s String) (x Real) (y Real))
        Real
```

```
101 (lat!agr!aver!2 x y))
```

A.2.3. Retículo real

```
 1 ;; Real lattice
 2
 3 ;; distance
 4 (define-fun lat!distance ((x Real) (y Real)) Real
 5     (abs (- y x)))
 6
 7 ;; |add
 8 (define-fun lat!or!add!2 ((x Real) (y Real)) Real
 9     (+ x y))
10
11 ;; &prod
12 (define-fun lat!and!prod!2 ((x Real) (y Real)) Real
13     (* x y))
14
15 ;; domain of symbolic conjunctions (arity 2)
16 (define-fun dom!sym!and!2 ((s String)) Bool
17     (or
18         (= s "and_prod")))
19
20 ;; domain of symbolic disjunctions (arity 2)
21 (define-fun dom!sym!or!2 ((s String)) Bool
22     (or
23         (= s "or_add")))
24
25 ;; eval symbolic conjunctions (arity 2)
26 (define-fun call!sym!and!2 ((s String) (x Real) (y Real))
      Real
27     (lat!and!prod!2 x y))
28
29 ;; eval symbolic disjunctions (arity 2)
30 (define-fun call!sym!or!2 ((s String) (x Real) (y Real)) Real
31 (lat!or!add!2 x y))
```