

Universidad de Castilla-La Mancha

Escuela Superior de Ingeniería Informática

Departamento de Sistemas Informáticos

Programa Oficial de Postgrado en Tecnologías Informáticas Avanzadas

Trabajo Fin de Máster:

Modelado de retículos difusos avanzados sobre el entorno FLOPER

Julio de 2011

Alumno: Carlos Vázquez Pérez-Íñigo

Directores: Dr. D. Ginés Damián Moreno Valverde
Dr. D. Jaime Penabad Vázquez

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, hemos sido testigos del importante papel que la lógica difusa o borrosa (fuzzy logic) ha jugado en el desarrollo de sofisticadas aplicaciones en campos tan diversos como los sistemas expertos, la teoría de control, la electrónica de consumo, el software médico, etc. Con el objetivo de facilitar el desarrollo de tales aplicaciones, ha surgido el interés por diseñar lenguajes declarativos (en particular, lenguajes lógicos) difusos que incorporen entre sus recursos expresivos el tratamiento de información vaga e imprecisa de forma natural. El presente Máster aspira a contribuir modestamente a estos desarrollos mejorando significativamente el entorno de programación lógica difusa, el sistema FLOPER ("Fuzzy LOgic Programming Environment for Research"), que permita la compilación, ejecución y depuración de este tipo de programas, y que abra las puertas en un futuro cercano a la transformación, especialización y optimización de aplicaciones escritas con este tipo de lenguajes. La investigación que proponemos aquí se enmarca dentro de las líneas prioritarias del grupo de investigación DEC-TAU de la UCLM, centradas en la aplicación de estos nuevos lenguajes a la solución de problemas reales, en aras de agilizar la inserción y rápida difusión de este nuevo tipo de tecnología emergente como una herramienta útil para el desarrollo de software.

La investigación, desarrollo y uso de los lenguajes de programación declarativos ha aumentado considerablemente durante la última década debido a una creciente convicción de que el estilo declarativo de programación puede contribuir esencialmente a la resolución de muchos problemas críticos en la producción de software, particularmente en lo que se refiere a la productividad y fiabilidad. Esto se debe, fundamentalmente, a que el estilo declarativo soporta un nivel de abstracción que facilita conceptualmente la tarea de programar. Si la complejidad de un lenguaje de programación se refleja en la complejidad de su descripción semántica, lo que caracteriza a los lenguajes declarativos es la claridad, sencillez y concisión de sus semánticas, que permiten entender rápidamente las construcciones del lenguaje y el significado de los programas. La razón de esta simplicidad está en la lógica matemática subyacente, que aporta al lenguaje su sintaxis, su semántica y su mecanismo de cómputo.

Asimismo, la lógica difusa está cobrando cada día más interés como soporte de este tipo de lenguajes declarativos, donde además de la integración de los estilos clásicos lógico y funcional puros, se buscan integraciones y combinaciones de paradigmas cada vez más potentes. Pero más allá de este uso particular, uno de los objetivos de esta lógica es proporcionar las bases del razonamiento aproximado que utiliza hipótesis vagas como herramienta para formular el conocimiento. Aunque de naturaleza distinta a la lógica clásica, esta lógica es multivalorada y puede verse como extensión de la lógica bivalente, de la trivalente definida por Łukasiewicz en 1922 y, en general, de la multivalorada. Puede plantearse su objetivo como el intento de construir un modelo de razonamiento que perfile el aspecto cualitativo o aproximado. Tiene gran capacidad para tratar problemas muy complejos o mal definidos, diferenciándose de la lógica ordinaria en el carácter de los predicados, los valores de verdad, la presencia de modificadores lingüísticos,

la posibilidad de varias interpretaciones, etc.

Fue L. A. Zadeh quien inició la teoría del razonamiento aproximado en el contexto de la Inteligencia Artificial al buscar herramientas más eficaces para la construcción de sistemas expertos. A él se debe el llamado Principio de Incompatibilidad, por el que resultan antagónicos la precisión y la complejidad, a la hora de describir la conducta de un sistema. Es por ello que se entiende la poca efectividad de los programas convencionales para modelar el comportamiento humano. La aplicación de la lógica difusa a los sistemas basados en reglas se ha desarrollado principalmente en, de una parte, generalizar el modelo de los factores de certeza y, de otra, en el uso de predicados difusos en la descripción de las reglas y de la realidad. En este segundo aspecto, se buscan métodos cada vez más potentes ante las necesidades que plantea el control de sistemas.

Y si el objetivo inicial de esta lógica y de las reglas del razonamiento era precisar los esquemas de razonamiento humano, en dos décadas se generalizaron las aplicaciones. Desde que se usa por primera vez en 1983 en el control de una máquina de vapor diseñada por Mamdani, y en 1987, en Japón, para el control de una depuradora de agua, se multiplican sus aplicaciones en el control electrónico tanto de procesos industriales complejos como del que se encuentra en los electrodomésticos de consumo. El control borroso, la aplicación más extendida de la lógica borrosa, está sustituyendo a los métodos tradicionales de control que requerían complejos modelos matemáticos, si es que cabía alguno. El sello "fuzzy logic" está resultando muy atractivo en el marketing de venta, a la vez que los artículos electrónicos sin componentes borrosos se quedan desfasados. Por mencionar sólo algunas aplicaciones concretas, podríamos citar el control de: sistemas de calefacción/refrigeración; sistemas de navegación de aeronaves (pilotaje automático); sistemas de frenado automático, de suspensión automática, de control de velocidad; sistemas de definición y enfoque de máquinas fotográficas y de vídeo; sistemas de predicción de terremotos; sistemas de predicción del clima y de fenómenos atmosféricos, etc. En definitiva, la introducción del concepto difuso enriquece la capacidad de representación del conocimiento, lo que queda contrastado con muy abundantes y muy significativas realizaciones prácticas.

Ante estos antecedentes, uno de los desafíos más importantes de las tecnologías software para la sociedad de la información, consiste en la propuesta de nuevos métodos, técnicas, plataformas y herramientas de nueva generación que tengan un comportamiento correcto en entornos cambiantes e imprecisos. En este escenario se necesitan prioritariamente nuevos lenguajes, formalismos y teorías, pero también entornos de programación y herramientas automáticas asociadas, que den soporte sistemático y racional al desarrollo del software. Teniendo en mente estos fines, hemos visto que los lenguajes declarativos disfrutan de una serie de ventajas extras, que sin duda alguna se basan en su fuerte fundamentación en algún tipo de lógica subyacente, lo que se relaciona con la potencia expresiva de las notaciones formales, con correspondencia directa entre la sintaxis (programas) y la semántica (lo que significan), y con los medios que permiten analizar (manual o automáticamente) el texto en un lenguaje formal para extraer conclusiones. La programación lógica difusa se muestra, entonces, como una interesante y creciente área de investigación que aglutina los esfuerzos por introducir la lógica difusa en la programación lógica, incorporando recursos más expresivos en tales lenguajes para trabajar de forma natural con incertidumbre y razonamiento aproximado.

Durante los últimos 4 años y en torno a esta línea de investigación que se está abordando en el grupo DEC-TAU de la UCLM, se han desarrollado varias técnicas de transformación (plegado/desplegado, evaluación parcial, cálculo de reductantes) adaptadas a los nuevos contextos difusos. En particular, el enfoque adoptado se basa en el esquema multi-adjunto, un reciente y extremadamente flexible paradigma de programación lógica difusa para el cual, desafortunadamente, no existen herramientas prácticas implementadas hasta ahora, a excepción del prototipo experimental FLOPER cuyo diseño e implementación constituye el núcleo de este trabajo. Una de las principales características del entorno es su habilidad para generar código Prolog a partir de programas difusos multi-adjuntos. Gracias a esta transformación, el texto resultante puede ser directamente ejecutado sobre cualquier entorno puramente lógico (Sicstus Prolog) de forma completamente transparente al usuario.

1.2. Objetivos

Con la intención de avanzar hacia el desarrollo de un entorno moderno y completo de programación difusa, que incluya utilidades para la compilación, ejecución, optimización, depuración, etc., de este tipo de programas, en este Trabajo Fin de Máster describimos las ampliaciones que hemos efectuado en la herramienta FLOPER consistentes en el enriquecimiento de su sintaxis, para permitirle la manipulación de varios retículos multi-adjuntos, así como en la ampliación de su mecanismo operativo para incluir la fase interpretativa en los árboles de ejecución o desplegado generados. Con este nuevo enriquecimiento, el núcleo del sistema queda perfectamente preparado para extender sus posibilidades de manipulación de programas, abarcando todas las técnicas de optimización, especialización, depuración, etc., que se investigan actualmente en el grupo DEC-TAU.

A modo de resumen, estos son los objetivos propuestos, sobre los que indicamos las principales aportaciones obtenidas a lo largo del Máster:

1. **Definición** de retículos multi-adjuntos en forma de ficheros Prolog con el objetivo de ser fácilmente modelados por usuarios comunes de Prolog y manipulados por el sistema FLOPER ([17, 57, 52]).
2. **Implementación** de la fase interpretativa de los árboles de ejecución ([56, 52]).
3. **Aplicación** de la fase interpretativa incluyendo la noción de paso interpretativo corto ([56]).
4. **Definición** de diversos (retículos) cuyo efecto, al interpretar programas multi-adjuntos sobre ellos, consiste en la adquisición de información relativa a su propia ejecución ([53, 54, 55]).

Durante el desarrollo del Máster, el autor ha disfrutado de una beca de iniciación a la investigación concedida por el Vicerrectorado de Investigación de la UCLM. Además, la integración dentro del grupo DEC-TAU de la UCLM ha sido clave para la confección de las publicaciones y la asistencia a congresos reseñados en el apartado de curriculum. Creemos que la calidad de nuestras investigaciones puede quedar avalada por el nivel de los diferentes foros en las que hemos presentado nuestros desarrollos. Una breve descripción de estos eventos quedaría como sigue, revelando que la vocación de este Máster no es sólo teórica, sino que también aporta un desarrollo práctico muy significativo:

- TPF'10 ([17]). El Taller de Programación Funcional es un evento dentro del ámbito de las jornadas de Programación y Lenguajes (PROLE). En la edición de 2010 participamos en el seminario de experiencias docentes, presentando una aplicación de las DCGs de Prolog que, más adelante, permitirían abordar el trabajo realizado en este Máster.
- ICAI'10 ([56]). En este congreso formalizamos la fase interpretativa en términos de un sistema de transición de estados, incluyendo la implementación de dicho sistema en el sistema FLOPER.
- RULEML'10 y RULEML'11 ([52, 53]). En la edición de 2010 del "International Symposium on Rule Interchange and Applications", celebrado Washington, presentamos un procedimiento práctico para manipular grados de verdad de diversos retículos en FLOPER. Además, en la edición del año siguiente, esbozamos el modo de obtener trazas declarativas de la ejecución de un programa en la respuesta computada difusa, ejemplificando con FLOPER.
- PROLE'10 ([57]). Las jornadas PROLE, cuyas actas se publican con ISBN, surgen en 2001 como un lugar de reunión de los grupos españoles que trabajan en temas ligados a la programación y a los lenguajes de programación. Desde esa fecha nuestro grupo de investigación en programación declarativa siempre ha estado presente en todas sus ediciones. En las jornadas de 2010, presentamos el modo de introducir retículos multi-adjuntos para interpretar programas en FLOPER.
- IWANN'11 ([55]). En la edición de 2011 del "International Work Conference on Artificial Neural Networks" profundizamos en la inclusión de información de la ejecución en las respuestas computadas difusas.

- CMMSE'11([54]). En este congreso internacional sobre métodos computaciones en ciencia e ingeniería, recuperamos una construcción clásica de la matemática (la compleción de Dedekind-MacNeille) para estudiar las singularidades en el caso de los retículos multi-adjuntos a fin de salvar en la práctica su carácter completo, que es decisivo en este marco.

1.3. Organización de la memoria

El presente Trabajo Fin de Máster se encuadra en el módulo III del programa oficial de postgrado en "Tecnologías Informáticas Avanzadas", impartido por el Departamento de Sistemas Informáticos de la Universidad de Castilla-La Mancha. La estructura de la memoria es la siguiente:

- En el Capítulo 2 se presenta el Curriculum Vitae del autor de esta memoria.
- El Capítulo 3 recoge un resumen de los cursos de doctorado realizados por el autor correspondientes a los módulos I y II del Máster en "Tecnologías Informáticas Avanzadas".
- El Capítulo 4 resume el estado del arte de la programación lógica difusa, área en la que se ubica nuestra línea de investigación.
- Los Capítulos 5 y 6 recogen el detalle de nuestros desarrollos. En el primero, además de incluir conceptos clásicos de programación lógica multi-adjunta, aparecen las contribuciones formales que hemos realizado en este estilo de programación lógica difusa. En el segundo, y sobre la base teórica referida, se muestran las técnicas que hemos diseñado para calcular su complejidad y definir los retículos asociados a estos programas difusos. Además, damos cuenta de las múltiples mejoras de la herramienta FLOPER, un entorno avanzado de programación lógica difusa que permite (entre otros recursos) ejecutar y depurar programas multi-adjuntos, así como manipular retículos multi-adjuntos.
- Por último, en el Capítulo 7 se presentan las conclusiones que se obtienen de esta investigación y las direcciones futuras que surgen a partir de ella.

Deseo agradecer a los organismos que han facilitado el desarrollo de este Máster: la UCLM, donde he realizado mis estudios, la Junta de Comunidades de Castilla-La Mancha y el Ministerio de Ciencia e Innovación que han financiado mis trabajos. Quisiera agradecer también la gran ayuda que han supuesto para mí en la confección de esta memoria mis directores de Tesis, el Dr. D. Ginés Moreno Valverde y el Dr. D. Jaime Penabad Vázquez.

Capítulo 2

Curriculum Vitae

2.1. Formación Académica

Título: Ingeniero en Informática

Centro: Escuela Superior de Ingeniería Informática de Albacete (ESII). Universidad de Castilla-La Mancha.

Fecha: Terminación el 7/07/2010

Calificación: 8.6, premio Extraordinario de Fin de Carrera en 2010

Calificación del Proyecto Fin de Carrera: Matrícula de Honor, 10

2.2. Situación Profesional Actual

Categoría profesional y fecha de inicio: Investigador, 8/07/2010

Organismo: Universidad de Castilla-La Mancha

Facultad, Escuela o Instituto: Instituto de Investigación en Informática de Albacete

Dept./Secc./ Unidad estr.: Grupo de Investigación DEC-TAU

Dirección postal: Instituto de Investigación en Informática de Albacete, Campus Universitario, s/n 02071, Albacete

2.3. Idiomas de Interés Científico

	Habla	Lee	Escribe*
Inglés	B1	B2	B1
Alemán	A1	A1	A1

* Según el marco común europeo de referencia para las lenguas.

2.4. Participación en Proyectos de Investigación

Actualmente forma parte del equipo investigador solicitante del proyecto de investigación nacional -en proceso de evaluación por el MICINN- titulado "HASBLOD: Herramientas y Aplicaciones Software Basadas en Lógica Difusa".

1. **Título del proyecto:** ALDDEIA: Aplicaciones de la Lógica Difusa al Desarrollo de Entornos Informáticos Avanzados

   Entidad financiadora: MEC Ref TIN2007-65749

Duración: Del 1 de Octubre de 2007 al 30 de Diciembre de 2011

Investigador principal: Ginés Moreno Valverde

2. **Título del proyecto:** PROLOF: Programación con lógica difusa

   Entidad financiadora: JCCM Ref. PII1I09-0117-4481

   Duración: Desde 1 de Abril de 2009 hasta 31 de Marzo de 2012

   Investigador principal: Pascual Julián Iranzo

2.5. Becas Disfrutadas

- **Organismo que concedió la Beca:** Vicerrectorado de Investigación de la Universidad de Castilla-La Mancha

  Finalidad de la beca: Ayuda de matriculación (curso 2010-2011) en el Máster Oficial Universitario en Tecnologías Informáticas Avanzadas.

  Fecha de inicio y de fin: Curso 2010-2011

  Centro de aplicación de la beca: Escuela Superior de Ingeniería Informática.

- **Organismo que concedió la Beca:** Asociación Española Para la Inteligencia Artificial (AEPIA)

  Finalidad de la beca: Asistencia al 3th International Spring School on Computational Logic, (ISCL'11) celebrado en Bertinoro, Italia.

  Fecha de inicio y de fin: Del 10 al 15 de abril de 2011.

  Centro de aplicación de la beca: Centro Residencial Universitario de Bertinoro (Forli-Cesena), Italia.

2.6. Publicaciones

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez.

  Título: Modeling interpretive steps into the FLOPER environment

  Ref. revista/libro: Proc. of the 2010 International Conference on Artificial Intelligence (ICAI'10). Editores: H.R. Arabnia et al., Páginas: 16-22. CSREA Press. ISBN 1-60132-148-1. AÑO: 2010.

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez.

  Título: A practical management of fuzzy truth-degrees using FLOPER

  Ref. revista/libro: Lecture Notes in Computer Science. Editores: M. Dean, J. Hall, A. Rotolo y S. Tabet. Vol. 6403. Páginas 119-126. ISSN: 0302-9743, ISBN: 978-3-642-16288-6. Springer Verlag, Berlin-Heidelberg. AÑO: 2010

- **Autores:** Juan A. Guerrero, Ginés Moreno, Carlos Vázquez.

  Título: Una implementación del lambda-cálculo en Prolog

  Ref. revista/libro: Actas de las X Jornadas sobre Programación y Lenguajes (PROLE'10), editores: V. M. Gulías, J. Silva y A. Villanueva. Páginas: 7-14 (sección de trabajos asociados al II Taller de Programación Funcional). Editorial: Garceta grupo editorial. ISBN: 978-84-92812-55-4. AÑO: 2010.

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez

  Título: Multi-adjoint lattices for manipulating truth-degrees into the FLOPER system

  Ref. revista/libro: Actas de las X Jornadas sobre Programación y Lenguajes (PROLE'10), editores: Víctor M. Gulías, Josep Silva y Alicia Villanueva. Páginas: 151-161. Editorial: Garceta grupo editorial. ISBN: 978-84-92812-55-4. AÑO: 2010.

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez

  Título: Fuzzy computed answers collecting proof information

Ref. revista/libro: Proc. of the 11th International Work-Conference on Artificial Neural Networks (IWANN'11), special session on "Fuzzy Logic, Soft Computing and Applications". Páginas: 445-452, editorial: Springer, LNCS Lecture Notes in Computer Science. AÑO: 2011

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez

  Título: Dedekind-MacNeille completion and multi-adjoint lattices

  Ref. revista/libro: Proc. of 11th International Conference on Mathematical Methods in Science and Engineering, CMMSE 2011, special session on Mathematical Methods for Computer Science. 15 páginas. ISBN es 978-84-614-6167-7. AÑO: 2011

- **Autores:** Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, Carlos Vázquez

  Título: Declarative Traces Into Fuzzy Computed Answers

  Ref. revista/libro: Lecture Notes in Computer Science (Proc. of the 5th International Symposium on Rules, RULEML 2011). Editores: N. Bassiliades, G. Governatori and A. Pasckhe. Editorial: Springer Verlag, Berlin-Heidelberg (16 páginas, en prensa). AÑO: 2011.

2.7. Aportaciones a Congresos

- **Denominación del evento:** 2010 International Conference on Artificial Intelligence (ICAI'10)
  Lugar de celebración y año: Las Vegas (USA), 12-15 de Julio 2010
  Entidad/grupo organizador: WORLDCOM
  Tipo de participación: Autor

- **Denominación del evento:** Segundo Taller de Programación Funcional (TPF'10)
  Lugar de celebración y año: Valencia (España), 7 de Septiembre de 2010
  Entidad/grupo organizador: SISTEDES
  Tipo de participación: Ponente

- **Denominación del evento:** Décimas Jornadas sobre Programación y Lenguajes (PROLE'10)
  Lugar de celebración y año: Valencia (España), 7-10 de Septiembre de 2010
  Entidad/grupo organizador: SISTEDES
  Tipo de participación: Ponente

- **Denominación del evento:** 4th International Web Rule Symposium (RuleML'10)
  Lugar de celebración y año: Washington (USA), 21-23 de Ocutbre de 2010
  Entidad/grupo organizador: RULEML, the rule markup initiative
  Tipo de participación: Autor

- **Denominación del evento:** 10th International Work Conference on Artificial Neural Networks (IWANN'11)
  Lugar de celebración y año: Torremolinos (España), 8-10 de Junio de 2011
  Entidad/grupo organizador: Spanish Chapter of the IEEE Computational Intelligence Society
  Tipo de participación: Ponente

- **Denominación del evento:** CMMSE'11
  Lugar de celebración y año: Benidorm (España), 26-29 de Junio de 2011
  Entidad/grupo organizador: University of Salamanca
  Tipo de participación: Autor

- **Denominación del evento:** The 5th International Symposium on Rules: Research Based and Industry Focused (RuleML'11)
  Lugar de celebración y año: Barcelona (España), 19-21 de Julio de 2011.
  Entidad/grupo organizador: RULEML, the rule markup initiative
  Tipo de participación: Autor

2.8. Otros Méritos

Pertenencia a sociedades científicas:

- AEPIA, Asociación Española Para la Inteligencia Artificial, desde el 18 de Febrero de 2011

Premios:

- Premio Extraordinario de Fin de Carrera en 2010

Capítulo 3

Asignaturas del Máster en Tecnologías Informáticas Avanzadas

En este Capítulo hacemos un repaso de las asignaturas cursadas durante el desarrollo del Máster en Tecnologías Informáticas Avanzadas, detallando sus objetivos y sus principales aportaciones. Las seis asignaturas se agrupan en dos módulos. Todas se han impartido en el campus de Albacete, aunque "Programación Internet con Lenguajes Declarativos Multiparadigma", al que por su importancia para este trabajo dedicamos un espacio mayor que las demás, se ha impartido simultáneamente en Albacete y Ciudad Real mediante videoconferencia.

3.1. Generación de Documentos Científicos en Informática

Esta asignatura se divide en tres partes claramente diferenciadas. En la primera, de menor duración que las posteriores, se intenta introducir al alumno en el mundo de la investigación, aportando una metodología de investigación que le ayude a desenvolverse con más facilidad en esta nueva faceta de su carrera universitaria hacia el doctorado. En este apartado se dan consejos basados en la propia experiencia de los profesores, que nos serán muy útiles a la hora de escribir un artículo, mandarlo a congresos o revistas, comprender las revisiones que se nos hagan, presentarlo en público, etc. También se han desarrollado contenidos teóricos, entre los que destacan los índices de impacto. Estos índices sirven para medir la relevancia de un artículo, de una revista o de un congreso. También se nos enseña a realizar búsquedas en Internet para encontrar artículos (ordenados por temática, relevancia, autores, etc.) en páginas dedicadas a este ámbito de la investigación.

La segunda parte, que constituye el grueso de la asignatura, está encaminada al uso de LaTeX para la generación de documentos y presentaciones científicas de calidad. LaTeX fue creado por Leslie Lamport y utiliza como motor de composición TeX. El índice que se siguió durante el curso para abordar la mayoría de los aspectos de LaTeX fue:

- Introducción: Introducción a LaTeX. Fuentes de información y distribuciones LaTeX.
- Estructura de un documento.
- Escribiendo en LaTeX: Composición de texto. Entornos de texto.
- Escribiendo fórmulas en LaTeX: Composición de fórmulas matemáticas y elementos flotantes.
- Inclusión de gráficos.
- Automatización de tareas: Bibliografía.

La última parte de la asignatura se dedicó a la rama de la estadística conocida como contraste de hipótesis. Dado que a lo largo de la labor investigadora, muchas veces ideamos algoritmos, programas, elementos hardware, etc., que pretenden mejorar de alguna manera los ya existentes, con las técnicas que aquí se presentan podemos comprobar si realmente nuestras propuestas son más rápidas, más eficientes, etc., que las demás con las que los comparamos.

El trabajo de la asignatura consistió en plantear y resolver adecuadamente dos problemas estadísticos, uno para un test paramétrico y otro para un test no paramétrico. Además, el trabajo debía ser realizado en LaTeX y para ello había que utilizar la mayoría de las características que vimos en su momento.

3.2. Introducción a la Programación de Arquitecturas de Altas Prestaciones

Esta asignatura presenta técnicas de programación de arquitecturas de altas prestaciones, y tiene por objetivo establecer la metodología que permita al programador obtener códigos capaces de resolver problemas de la manera más rápida y eficiente posible. Estas técnicas abarcan diversas clases de sistemas, como arquitecturas monoprocesador, arquitecturas paralelas con memoria compartida, o arquitecturas paralelas con memoria distribuida. Se pretende presentar tanto las ideas básicas de la programación secuencial orientada a bloques como las de la computación paralela. Así dado un cierto código secuencial la forma de hacer su ejecución lo más rápida posibles es, bien optimizando ese código secuencial (programación orientada a bloques) o ejecutando ese código en trozos de forma paralela (paralelización del código secuencial). Técnicas que a su vez son complementarias y que se pueden utilizar juntas para conseguir mejores resultados.

A la hora de optimizar resultados debemos tener en cuenta que las prestaciones de un programa no sólo dependen de lo veloz que sea el procesador donde se ejecuta, sino también de la capacidad del sistema de memoria que alimenta al procesador, por lo que hay que valorar conceptos como latencia (tiempo desde que se pide un dato hasta que dicho dato está disponible), y ancho de banda (velocidad de suministro de datos). Para mejorar estos parámetros se trabaja con la memoria caché, que permite mejorar las prestaciones cuando se reutilizan los datos que contiene (localidad temporal), e incrementado el ancho de banda para aumentar la utilización del procesador (localidad espacial). Así, en problemas con gran cantidad de datos, entra en juego la programación orientada a bloques, que divide el problema en bloques para explotar al máximo la localidad de los datos. El correcto uso de la programación paralela nos ayuda a resolver problemas de una manera más eficiente. Actualmente, la computación paralela tiene infinidad de aplicaciones, tanto en ingeniería y diseño, como en aplicaciones científicas y comerciales, aunque esta arquitectura todavía presenta desafíos y limitaciones al programador.

Otra parte importante de la asignatura es la introducción de diversas librerías de programación de arquitecturas de altas prestaciones orientadas a la resolución de problemas de álgebra lineal numérica. Por ejemplo, a la hora de multiplicar matrices de grandes dimensiones (operación muy usual en cierto tipo de problemas) en lugar de hacer uso de bucles para su resolución (lo que tardaría mucho y consumiría muchos recursos), podemos utilizar instrucciones especialmente diseñadas para estas operaciones que incorporan dichas librerías. Para la consolidación de todos estos conceptos resultaron fundamentales las clases prácticas de extensa duración que se impartieron a lo largo del cuatrimestre. En ellas, además de optimizaciones secuenciales, se profundizó en el uso de MPI para paso de mensajes y en las librerías BLAS, BLACS Y PBLAS.

El trabajo de la asignatura consistió en un trabajo práctico en el que había que utilizar todos los recursos y librerías que aprendimos en las prácticas para desarrollar y optimizar el código que diera respuesta a los problemas planteados. Además de dicho trabajo práctico, se realizó otro trabajo de investigación de carácter teórico, en el que me fue posible relacionar conocimientos adquiridos en mi línea de investigación (programación lógica difusa) con la programación paralela. El resultado fue un estudio completo del lenguaje de programación Parlog, una especie de Prolog paralelo.

3.3. Sistemas Inteligentes Aplicados a Internet

El principal objetivo de esta asignatura consiste en conocer los fundamentos de distintos formalismos relacionados con los sistemas inteligentes, que están en pleno auge en la actualidad, y cómo pueden aplicarse para resolver determinados problemas relacionados con Internet. Se hace especial hincapié en las redes bayesianas, basadas en diferentes implementaciones del teorema de Bayes para la inferencia de conocimiento a partir de datos probabilísticos estructurados en forma de red. La minería de datos permite obtener información útil y altamente agregada a partir de datos masivos para los que no hay otra herramienta teórica. Su aplicación a Internet permite una gran acumulación de conocimiento que se puede aplicar a cualquier faceta de la industria.

La carga de la asignatura se divide en tres módulos o unidades temáticas:

1. Modelado e Inferencia en Redes Bayesianas: En este apartado se ha abordado una introducción a las redes bayesianas detallando los conceptos básicos de los grafos dirigidos, y estudiando el concepto de independencia y d-separación. Se repasa la concepción bayesiana de la inteligencia artificial, considerada como un agente autónomo que maximiza su función de utilidad. Se detallan los tipos de razonamiento (diagnosis, predictivo, intercausal y combinado) y se da una introducción al ciclo de vida de sistemas basados en redes neuronales, como el aportado por KEBN. También se señalan aspectos del modelado de redes bayesianas y conceptos de inferencia.
2. Aprendizaje de Redes Bayesianas: Centrado en el aprendizaje de modelos en redes bayesianas mediante la estimación de parámetros a partir de datos. Se basa en optimización y búsqueda, en el estudio de métricas y en la búsqueda en grafos dirigidos, culminando con aprendizaje estructural y paramétrico con datos perdidos.
3. Recuperación de Información (Web Mining): En este apartado se comprenden las heurísticas y metaheurísticas, deteniéndose especialmente en los algoritmos genéticos y el algoritmo de estimación de distribuciones. Se estudian estos métodos sobre problemas bien conocidos como el del viajante de comercio, para permitir compararlos.

En esta asignatura se han empleado las herramientas Weka y Elvira. La primera posee una extensa colección de algoritmos de Máquinas de conocimiento desarrollados por la universidad de Waikato (Nueva Zelanda) implementados en Java; útiles para ser aplicados sobre datos mediante los interfaces que ofrece o para embeberlos dentro de cualquier aplicación. Además, Weka contiene las herramientas necesarias para realizar transformaciones sobre los datos, tareas de clasificación, regresión, clustering, asociación y visualización. Weka está diseñado como una herramienta orientada a la extensibilidad por lo que añadir nuevas funcionalidades es una tarea sencilla.

El programa Elvira cuenta con un formato propio para la codificación de los modelos, un lector-intérprete para los modelos codificados, una interfaz gráfica para la construcción de redes, con opciones específicas para modelos canónicos (puertas OR, AND, MAX, etc.), algoritmos exactos y aproximados (estocásticos) de razonamiento tanto para variables discretas como continuas, métodos de explicación del razonamiento, algoritmos de toma de decisiones, aprendizaje de modelos a partir de bases de datos, fusión de redes, etc. Elvira está escrito y compilado en Java, lo cual permite que pueda funcionar en diferentes plataformas y sistemas operativos (linux, MS-DOS/Windows, Solaris, etc.). La principal limitación del programa Elvira es que la búsqueda de resultados de investigación a corto plazo dificultó la aplicación de los principios de la metodología de desarrollo de software. El programa aún carece de ayuda en línea y todavía necesita bastante trabajo de depuración.

3.4. Grid Computing

En esta asignatura se entra de lleno en el reciente campo de la computación Grid. Como se nos explica en ella, la computación Grid es una tecnología innovadora que permite utilizar de forma coordinada todo

tipo de recursos (entre ellos cómputo, almacenamiento y aplicaciones específicas) que no están sujetos a un control centralizado. En este sentido es una nueva forma de computación distribuida, en la cual los recursos pueden ser heterogéneos (diferentes arquitecturas, supercomputadores, clusters, etc.) y se encuentran conectados mediante redes de área extensa (por ejemplo Internet).

A lo largo de este curso se ha explicado con minucioso detalle la arquitectura y servicios del Grid, su *middleware* (capa intermedia de servicios muy importante en el Grid), aplicaciones reales del Grid, implementaciones específicas y actuales, y otras cuestiones de vital importancia como el control de recursos, la seguridad o la transferencia de datos. También otras características como la evolución del Grid (actualmente centrado en el ámbito académico) hacia un entorno comercial y cuestiones de rendimiento se han abordado en esta materia.

En esencia, entre las ventajas de la arquitectura de computación Grid que hemos podido apreciar en este curso, destacamos la capacidad de balanceo de sistemas, es decir, la capacidad de reasignarse desde la granja de recursos a donde se necesite. Otra ventaja es la alta disponibilidad: si un servidor falla, se reasignan los servicios en los servidores restantes. Por último es importante la reducción de costes, ya que con esta arquitectura los servicios son gestionados por "granjas de recursos". Ya no es necesario disponer de "grandes servidores" y podremos hacer uso de componentes de bajo coste. Pero la computación Grid presenta todavía notables desafíos. Por un lado, al estar los diferentes recursos gestionados por diferentes instituciones (empresas, universidades, particulares) cada dominio tendrá unas políticas de seguridad propias, por lo que es necesario controlar y respetar estas políticas en el intercambio de recursos por la red. Por otro lado, se hace necesario también definir los nuevos *Grid Services* frente a los servicios Web tradicionales. Aparte de todos estos contenidos, la asignatura presenta otro valor (o desafío) añadido. Este curso se imparte íntegramente en el idioma inglés, ya que a la docencia del profesor Fernando Pelayo se une la del profesor Karim Djemame, de la Universidad de Leeds.

El trabajo (también realizado y presentado en inglés) que realicé, titulado "Cloud Computing: a vendors' vision", versaba sobre lo aspectos relevantes de los diferentes proovedores de recursos *Cloud* (una extensión de la computación Grid) que un usuario de dichos recursos, ya sea un individuo u otra empresa, deben tener en cuenta, como el modo de pago, la regulación de este tipo de tecnologías y las diferentes prestaciones.

3.5. Modelos para el Análisis y Diseño de Sistemas Concurrentes

En esta asignatura se ha tratado en profundidad el concepto de sistema concurrente, y se ha dedicado especialmente a su modelado y diseño. Se han estudiado las lógicas que permiten modelar la concurrencia entre diversos sistemas, destacando los lenguajes LTL, CTL, TCTL. También se ha profundizado en los siguientes modelos formales de concurrencia:

- Redes de Petri. Son una herramienta para el análisis y modelado de sistemas concurrentes. Se basan en un grafo dirigido con dos tipos de nodos, denominados lugares y transiciones, de modo que no se puede dar un lugar adyacente a otro, ni una transición adyacente a otra. Los *tokens* se asocian a los lugares. Estos elementos discurren por la red, reproduciendo el funcionamiento de un sistema concurrente y permitiendo analizar propiedades de alcanzabilidad, vivazidad, seguridad, etc.
- Álgebra de procesos. Se trata de lenguajes de naturaleza algebráica, utilizados para la especificación de modelos de sistemas concurrentes. Los más destacables son:
  - CSP, *Communicating sequential processes*. Fue descrito en 1978 por C. A. R. Hoare.
  - CCS, acrónimo de *Calculus of communicating Systems*. Este lenguaje fue introducido en 1980 por R. Milner. La notación de este lenguaje, más sencilla que la del anterior, permite introducir sus expresiones directamente en un ordenador usando caracteres ASCII.

- Autómatas temporizados. Consisten en autómatas finitos equipados con un conjunto de variables reales positivas denominadas relojes. Este tipo de autómatas se contemplan como una lógica multimodal usada para especificar propiedades de un sistema de transción de estados etiquetados, similares a autómatas.

Se ha usado la herramienta Uppaal. Los sistemas en tiempo real usados por Uppaal se modelan como redes programadas que pueden ampliarse con diferentes tipos de datos (selecciones, enteros, etc.). Esta herramienta sirve como un lenguaje de diseño o de modelado que permite describir el comportamiento de un sistema como redes automatizadas ampliadas con variables de datos y de reloj. El simulador es una herramienta de validación que permite examinar las posibles ejecuciones dinámicas de un sistema durante las fases de diseño y proporciona una sencilla detección de los fallos antes de que este sea verificado por el comprobador de modelos.

El trabajo realizado en esta asignatura ha consistido en la resolución de diversos problemas propuestos por los profesores, incluyendo el modelado de redes de Petri para implementar el algoritmo de comunicación del bit alternante, como caso base. La tarea más apreciable ha consistido en el uso de la herramienta Uppaal, descrita anteriormente, para modelar, simular y analizar diversos sistemas concurrentes.

3.6. Programación Internet con Lenguajes Declarativos Multiparadigma

El resumen de esta asignatura va a ser más extenso que en los casos anteriores, ya que se trata de la materia más relacionada con mi línea de investigación. Su principal objetivo es presentar los fundamentos de los lenguajes declarativos multi-paradigma, que integran las principales características de los lenguajes lógicos y funcionales puros, (y también de éstos con el paradigma difuso). Para ello, se estudian los mecanismos operacionales y las semánticas declarativas, empleadas en las diferentes propuestas de integración. También se detallan lenguajes reales, basados en las distintas aproximaciones estudiadas anteriormente, incidiendo en sus aplicaciones prácticas y con especial énfasis en sus facilidades para la programación Internet.

En los lenguajes declarativos se usa una cierta lógica como lenguaje de programación, y los programas se ven como un conjunto de fórmulas lógicas que resultan ser la especificación del problema a resolver, de manera que la computación se entiende como una forma de inferencia o deducción en dicha lógica. La lógica empleada debe disponer de un lenguaje que sea suficientemente expresivo, además de una semántica operacional y de una semántica declarativa que permita atribuir significado a los programas de forma independiente a su posible ejecución, exigiendo resultados de corrección y completitud.

Dentro del marco declarativo, se trata en primer lugar la programación lógica y la programación funcional. Señalando que la programación lógica tiene una gestión automática de la memoria, su mecanismo de cómputo permite una búsqueda indeterminista de soluciones, computa con datos parcialmente definidos y su relación de entrada-salida no está fijada de antemano. Además la semántica operacional de la programación lógica está basada en la resolución SLD.

En cambio, la programación funcional se basa en el concepto matemático de función y su definición mediante ecuaciones, que constituyen el programa. En ésta, la computación es una reducción determinista de expresiones funcionales para obtener un valor mientras que la semántica operacional está basada en la reducción o ajuste de patrones.

Estos dos planteamientos de programación declarativa tienen sus semejanzas y diferencias, así como ventajas de uno frente a otro según el problema a tratar, por lo que cada estilo tiene algo que ofrecer al otro. Los lenguajes lógicos incorporan unificación y variables lógicas, y los lenguajes funcionales poseen la lógica de la igualdad y de las funciones. Como el objetivo es combinar las mejores características de los lenguajes lógicos y funcionales, han surgido los lenguajes lógico-funcionales. Esta integración se

intenta conseguir mediante dos planteamientos. El primero trata de dotar de lógica al planteamiento funcional, añadiendo variables lógicas y no determinismo a un lenguaje funcional, donde la semántica operacional sería reducción más unificación, lo que llamamos Narrowing, o reducción más espera hasta que las funciones estén suficientemente instanciadas, lo que llamamos Residuación. El otro planteamiento, sería dotar de sentido funcional al planteamiento lógico, añadiendo igualdad a un lenguaje lógico, así como definir funciones y añadir tipos de datos. Con este último enfoque, las semánticas operacionales podrían consistir en des-anidar funciones más SLD (Flattening), E-unificación más resolución SLD, o resolución más nuevas reglas para las funciones (Residuación o Narrowing).

En la parte central de esta materia se introducen extensiones difusas (fuzzy) a la programación declarativa. Ésta es precisamente el área en la cual se centra mi línea de investigación. Podemos decir que los lenguajes tradicionales de la programación lógica no incorporan técnicas que permitan tratar, de manera explícita, la vaguedad o incertidumbre. La Programación Lógica Difusa es un área de investigación que aglutina el esfuerzo de introducir la lógica difusa en la programación lógica. En las últimas décadas, han sido desarrollados varios sistemas de programación lógica difusa; en ellos se reemplaza el mecanismo clásico de inferencia, la SLD Resolución, por una variante difusa que permita razonar con incertidumbre y evaluar grados de verdad. En este sentido hay dos vías principales para lograr la "fuzzyficación":

- La primera aproximación reemplaza el mecanismo de la unificación sintáctica, de la clásica resolución-SLD, por un algoritmo de unificación difuso (unificación por similaridad) basado en ecuaciones de similaridad que se resuelven mediante relaciones difusas. Esta aproximación está representada por lenguajes como LIKELOG.
- En la segunda aproximación, los programas son un subconjunto de cláusulas con un grado de verdad asociado que está explícitamente anotado. El trabajo de computación y propagación de los grados de verdad recae en una extensión del principio de resolución clásico, mientras el mecanismo de unificación (sintáctica) permanece inalterado. Un ejemplo de este tipo de lenguaje es el llamado *lenguaje lógico multi-adjunto*, que es el que hemos elegido para tratar la incertidumbre en mi línea de investigación.

Cabe destacar las diferentes técnicas de optimización vistas a lo largo de la asignatura, y que constituyen el núcleo de la misma. Se pueden distinguir dos grupos de técnicas: aquéllas que se basan en preprocesar un programa, y que devuelven un código optimizado, entre las que se cuentan el plegado/desplegado y el *tupling*, y las que se basan en optimizar el modo de ejecución más que el código en sí, como la tabulación, técnica que explota la metodología de programación dinámica. El uso combinado de estas técnicas puede resultar en ganancias de eficiencia y de completitud, como la tabulación umbralizada con plegado/desplegado.

La parte final de la asignatura se centra en aplicar la programación declarativa a la búsqueda en Internet. Una de las aplicaciones más llamativas consiste en el uso de una variante difusa de los lenguajes XPath y XQuery para la manipulación de código XML. Dicha herramienta, denominada fuzzyXPath, permite realizar la búsqueda de información dentro de un fichero XML utilizando reglas de búsqueda difusas. FuzzyXPath está siendo desarrollado actualmente dentro del grupo DEC-TAU.

El trabajo que realicé en esta materia consistió en la presentación del entorno de programación que hemos implementado para el marco de la lógica difusa (FLOPER, Fuzzy LOgic Programming Environment for Research). En él, como detallaremos ampliamente en esta memoria, se pueden ejecutar programas difusos expresados en el lenguaje referido. Además el sistema es capaz de analizarlos, ejecutarlos y obtener soluciones con grados de verdad asociados, y posee también otras opciones de depuración de estos programas, permite la construcción de árboles de ejecución, etc.

Capítulo 4

Fundamentos de la Programación Lógica Difusa

En este capítulo introducimos conceptos básicos de la programación lógica (4.1), que son de gran relevancia en esta memoria. En la Sección 4.2 se resume el estado del arte de la lógica difusa, que aporta las nociones formales de todos los conceptos básicos (sintácticos, semánticos, operacionales, etc.) de los lenguajes de programación lógica difusa (descritos en 4.3).

4.1. Programación Lógica

La programación lógica consiste esencialmente en la aplicación del conjunto de conocimientos sobre lógica clásica al diseño de lenguajes de programación. La programación declarativa comprende básicamente dos paradigmas de programación: la programación lógica y la programación funcional, donde el primer estilo gira en torno al concepto de predicado, o relación entre elementos y el segunda se basa en el concepto de función (que no es más que una evolución de los predicados), de corte más matemático.

La programación lógica [4, 38, 41, 21] se basa en fragmentos de la lógica de predicados, siendo la aproximación más popular aquella que se basa en la lógica de cláusulas de Horn (HCL, Horn clause logic), que pueden emplearse como base para un lenguaje de programación al poseer una semántica operacional susceptible de ser eficientemente implementable, como es el caso de la resolución SLD. Como semántica declarativa se utiliza una semántica por teoría de modelos, que toma como dominio de interpretación un universo puramente sintáctico: el universo de Herbrand. La resolución SLD es un método de prueba por refutación, que emplea el algoritmo de unificación como mecanismo de base y permite la extracción de respuestas (es decir, el enlace de un valor a una variable lógica).

Algunas de las características de la programación lógica pueden ponerse de manifiesto mediante el siguiente ejemplo de concatenación de dos listas. En programación lógica el problema se resuelve definiendo una relación app con tres argumentos: generalmente, los dos primeros hacen referencia a las listas que se desea concatenar y el tercero a la lista que resulta de la concatenación de las dos primeras. El programa se compone de dos cláusulas de Horn (hacemos uso de la notación que emplea Prolog para la representación de listas, en las que las variables se escriben en mayúsculas, el símbolo "[]" representa la lista vacía y el símbolo "|" es el constructor de listas):

```
app([],X,X)
app([X|Xs], Y, [X|Zs]) :- app(Xs,Y,Zs)
```

Este programa tiene una lectura declarativa clara. La primera cláusula afirma que la concatenación de la lista vacía [] y otra lista X es la propia lista X. Mientras que la segunda cláusula puede entenderse en los siguientes términos: la concatenación de dos listas $[X|Xs]$ e Y es la lista que resulta de añadir el primer elemento X de la lista $[X|Xs]$ a la lista Zs, que se obtiene al concatenar el resto Xs de la primera lista a la segunda Y.

Uno de los primeros hechos que advertimos en el ejemplo, es que no hay ninguna referencia explícita al tipo de representación en memoria de la estructura de datos lista. De hecho, una característica de los lenguajes declarativos en programación lógica, es que proporcionan una gestión automática de la memoria, evitando una de las mayores fuentes de errores en la programación con otros lenguajes. Por otra parte, el programa puede responder, haciendo uso del mecanismo de resolución SLD, a diferentes cuestiones (objetivos) sin necesidad de efectuar ningún cambio en el programa, gracias al hecho de emplearse un mecanismo de cómputo, que permite una búsqueda indeterminista (built-in search) de soluciones. Esta misma característica permite computar con datos parcialmente definidos y hace posible que la relación de entrada/salida no este fijada de antemano.

A continuación hacemos un muy rápido repaso a algunos conceptos técnicos relacionados con el principio de SLD-resolución, cuya extensión al caso difuso serán claves en apartados posteriores. La estrategia de resolución SLD es un caso particular de resolución lineal. Para definir lo que es un paso de resolución SLD, primero definimos lo que es una regla de computación.

Definición 4.1.1 (Regla de Computación) *Se llama regla de computación (o función de selección) φ a una función que, cuando se aplica a un objetivo G, selecciona uno y sólo uno de los átomos de G.*

Definición 4.1.2 (Paso de Resolución SLD) *Dado un objetivo $G = \leftarrow A_1, \ldots, A_j, \ldots, A_n$ y una cláusula $C = \leftarrow B_1, \ldots, B_m$ entonces G' es un resolvente de G y C (por resolución SLD) usando la regla de computación φ si se cumple que:*

1. *$A_j = \varphi(G)$ es el átomo seleccionado por φ*
2. *$\theta = umg(A, A_j)$*
3. *$G' = \leftarrow \theta(A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)$*

También diremos que G' se deriva de G y C (en un paso) y lo representaremos como $G \rightarrow_{SLD} G'$, $G \rightarrow^{\theta}_{SLD} G'$ o $G \rightarrow^{[C,\theta]}_{SLD} G'$ según el grado de información que se desee hacer explícita. En esencia, las condiciones indican que G' se obtiene como resolvente lineal de G y C cuando se explota, de entre los átomos del objetivo G, exactamente A_j.

Definición 4.1.3 *Sea $\mathcal{P}$ un programa definido y G un objetivo. Una derivación SLD para $\mathcal{P} \cup \{G\}$, usando la regla de computación φ, consiste en una secuencia de pasos de resolución SLD*

$$G_0 \rightarrow^{[C_1,\theta_1]}_{SLD} G_1 \rightarrow^{[C_2,\theta_2]}_{SLD} \cdots G_{n-1} \rightarrow^{[C_n,\theta_n]}_{SLD} G_n$$

donde:

1. *Para $i = 1, \ldots, n$, G_i es un resolvente de G_{i-1} y C_i (obtenido por resolución SLD) usando la regla de computación φ*
2. *Cada C_i es una variante de una cláusula $\mathcal{P}$*
3. *Cada θ_i es el unificador más general (umg) obtenido en el correspondiente paso i-ésimo de resolución SLD.*

Definición 4.1.4 (Refutación SLD) *Sea $\mathcal{P}$ un programa definido y G un objetivo. Una refutación SLD para $\mathcal{P} \cup \{G\}$ es una derivación SLD finita cuyo último objetivo es la cláusula vacía.*

Definición 4.1.5 (Respuesta computada) *Sea $\mathcal{P}$ un programa definido y G un objetivo. Sea $G \rightarrow^{\theta_1}_{SLD} G_1 \rightarrow^{\theta_2}_{SLD} \cdots \rightarrow^{\theta_n}_{SLD} \square$ una refutación para $\mathcal{P} \cup \{G\}$. Una respuesta computada para $\mathcal{P} \cup \{G\}$ es la sustitución θ que resulta de la composición, restringida a las variables de G, de la secuencia de umg's $\theta_1, \ldots, \theta_n$.*

4.2. Lógica Difusa

La lógica difusa proporciona una base matemática para modelar la vaguedad y la incertidumbre o imprecisión que encontramos en el mundo real cuando intentamos describir fenómenos que no tienen límites bien definidos o sistemas complejos. La lógica difusa descansa sobre la noción de conjunto difuso [69]. Los conjuntos difusos son objetos matemáticos introducidos para poder formalizar y tratar esas situaciones de vaguedad e incertidumbre o falta de información (principalmente, información imprecisa o difícilmente cuantificable como: el dolor que sufre una persona; la clasificación de las estaturas en altas y bajas o de edades en jóvenes y viejos; el nivel de suciedad del agua en un proceso de aclarado; de temperaturas o presiones altas o bajas, etc.) que aparece en el diseño de sistemas basados en el conocimiento, como los sistemas expertos, o en el desarrollo de sistemas de control. En un contexto clásico, dado un conjunto ordinario $\mathcal{U}$, un subconjunto A del mismo puede definirse mediante su función característica, $\chi_A(x)$, que especifica nítidamente si un elemento x pertenece a A, (es decir, devuelve 1 si x pertenece a A, 0 en caso contrario).

Análogamente, un subconjunto difuso A de $\mathcal{U}$ se caracteriza por una función $\mu_A : \mathcal{U} \rightarrow [0, 1]$. La función μ_A se denomina grado de pertenencia, y el valor $\mu_A(x)$ representa el grado de pertenencia de x en el subconjunto difuso A. Así, mediante el uso de conjuntos borrosos, podemos formalizar de forma más conveniente si un individuo de una edad determinada puede considerarse una persona joven, y con qué grado, o una determinada presión o temperatura puede considerarse alta o baja. De forma más general, la función grado de pertenencia puede definirse de manera que tome valores en un cierto conjunto parcialmente ordenado. Por analogía, el concepto de "grado de verdad" surge el concepto de "grado de pertenencia", así en este tipo de lógica una proposición puede tener grado de verdad $\alpha = \mu_A(x_0) \in [0, 1]$, donde $A = \{x \in \mathcal{U} : A(x)\}$. La evaluación de proposiciones depende de la evaluación de sus componentes más simples que se utilizan como entrada de las funciones de verdad asociadas a las conectivas contenidas en dicha proposición.

Lo anterior resalta que la lógica difusa sigue una aproximación basada en el uso de funciones de verdad, lo que la diferencia de la teoría de la probabilidad (la probabilidad de la intersecciónno viene determinada por las probabilidades de esas proposiciones) [18].

La expresión "lógica difusa" admite, al menos, dos interpretaciones [70, 58, 18]:

1. Por un lado, se refiere al uso de conjuntos borrosos [69] para la manipulación de conocimiento vago (como ya se ha comentado), el uso de relaciones borrosas y la generalización de las conectivas lógicas tradicionales, $\neg$, $\wedge$, $\vee$, mediante las nociones de negación (borrosa), t-normas, y t-conormas respectivamente. Así como otros muchos conceptos relacionados, que la diferencian de la lógica ordinaria: el carácter de los predicados, los valores de verdad, la presencia de modificadores lingüísticos, la posibilidad de varias interpretaciones, etc., siendo muy relevante también la presencia de cuantificadores específicos (casi, algunos, la mayoría, aproximadamente, etc.). .

2. Por otro lado, se refiere, en su formulación más sencilla, a la extensión de la lógica clásica bivalente a una lógica con infinitos valores de verdad en el intervalo cerrado [0, 1]. Es pues una rama de la lógica multivaluada basada en el paradigma de inferencia bajo imprecisión. Este tipo de lógica sigue el espíritu de la lógica clásica al definir nociones semánticas y perseguir la especificación de

sistemas deductivos (axiomáticos) que preserven la corrección y completitud. Esta orientación de la lógica es relativamente reciente, remontándose a los trabajos de Pavelka [60], que introduce una lógica basada en la de Lukasiewicz con valores de verdad que son números racionales. Posteriormente, Novak et. al. [59] desarrollaron esta lógica proporcionando una sintaxis anotada (es decir, pares ⟨fórmula, valor de verdad⟩) y una regla de inferencia basada en una forma de modus ponens residuado (si $\langle A, \upsilon_1\rangle$, y $\langle A \rightarrow B, \upsilon_2\rangle$ entonces se puede inferir $\langle B, \upsilon_1 * \upsilon_2\rangle$, donde "$*$" es la t-norma de Lukasiewicz). Hajek [19] ha desarrollado la denominada lógica difusa básica, que es la lógica de las t-normas continuas, caracterizada por un sistema axiomático con axiomas bien establecidos y que emplea modus ponens residuado y generalización como únicas reglas de inferencia.

La primera de las acepciones ha recibido el nombre de "lógica difusa en sentido amplio" y la segunda "lógica difusa en sentido estricto". En este trabajo estamos interesados en la segunda de las acepciones, debido a que sirve mejor para la fundamentación de los lenguajes de programación lógica. En el futuro, cuando hablemos de "lógica difusa" o "lógica borrosa", estaremos haciendo mención a la concepción "estricta" de la misma.

Para finalizar, nos gustaría hacer mención de algunos de los miembros más relevantes de la comunidad internacional que trabaja en lógica difusa: D. Dubois, M. Fitting, G. Gerla, J. A. Goguen, P. Hájek, R. C. T. Lee, E. Mamdani, M. Mukaidono, V. Novák, L. Paulik, J. Pavelka, H. Prade, P. Vojtas, T. J. Weigert, R. Yager y L. A. Zadeh. Algunos de los miembros más destacados de la comunidad nacional son: Lluis Godó, Enric Trillas y Lluis Valverde.

4.3. Lenguajes de Programación Lógica Difusa

Centrándonos ya en la aplicación de la lógica difusa al diseño de lenguajes declarativos (en especial, lógicos) difusos, digamos que inicialmente la programación lógica pura [4, 38, 41, 21] se ha usado para resolver una amplia gama de problemas y para representación del conocimiento. A pesar de ello, los lenguajes tradicionales de programación lógica no incorporan técnicas que permitan tratar, de manera explícita, la incertidumbre y el razonamiento aproximado, lo que puede constituir una limitación a la hora de tratar muchos problemas del mundo real. La programación lógica difusa es un área de investigación que aglutina los esfuerzos que se realizan para introducir conceptos y técnicas de la lógica difusa en los sistemas de programación lógica, con la finalidad de superar esas limitaciones a las que nos hemos referido.

En las últimas décadas, se han desarrollado varios sistemas de programación lógica difusa reemplazando 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 [39], como el sistema Prolog-Elf [20], el sistema Fril Prolog [6] y el lenguaje F-Prolog [40].

Sin embargo, no hay un método común para integración de conceptos borrosos en la programación lógica: mientras que algunos de estos sistemas sólo contemplan la componente difusa en los predicados introduciendo la noción de similaridad entre ellos [5], otros consideran hechos y/o reglas difusas etiquetando cláusulas de programa con números reales que representan el grado de verdad asociado [68]. Además, tampoco hay unanimidad acerca del tipo de lógica difusa (min-max, Lukasiewicz, intuicionista, etc.) usada para interpretar las diferentes conectivas lógicas de un programa. Así, aunque muchos de los sistemas usan la lógica min-max (para modelar la conjunción y la disyunción), algunos usan la lógica de Lukasiewicz [36]. Otras aproximaciones permiten una interpretación más genérica de las conectivas sustituyendo el principio de resolución por un mecanismo operacional basado en una forma de modus ponens generalizado [68]. Más recientemente se recogen en [67, 44, 43, 45] dos modelos teóricos (diferentes) para la programación lógica difusa que admiten además distintas interpretaciones para las implicaciones. Finalmente, en [14] encontramos un esquema muy flexible en el que, además de introducir la negación y considerar conjuntos difusos asociados a intervalos reales, cada cláusula de programa puede ser interpretada con una lógica diferente.

Todas estas consideraciones (junto con otras relativas al tratamiento semántico de estos lenguajes, que no citamos por brevedad) confirman el estado todavía incipiente del área en tanto en cuanto son muy variadas las aproximaciones al problema al tiempo que ninguna de ellas consigue imponerse sobre las demás. Sin embargo, aunque en el momento presente no es posible hablar de estándares en cuanto a lenguajes lógicos difusos, nosotros hemos detectado dos grandes corrientes (casi antagónicas o complementarias entre sí):

- La primera aproximación está representada por lenguajes que reemplazan el mecanismo de la unificación sintáctica, que está en la base de la resolución SLD clásica, por un algoritmo de unificación borrosa. La idea general es modificar el algoritmo de unificación sintáctica para que pueda soportar nuevos tipos de datos como constantes borrosas y variables lingüísticas, obteniendo un algoritmo de unificación borrosa que nos permita tratar la información imprecisa. En esta línea se encuentran trabajos como el de Virtanen [66] donde se presenta un algoritmo de unificación difuso basado en relaciones de equivalencia borrosas. Rios-Filho et al. [62] hace una distinción entre conocimiento específico y general con el objetivo de direccionar el algoritmo de unificación borrosa, es lo que se llama algoritmo de unificación contextual. Otros trabajos intentan construir algoritmos de unificación borrosa basados en relaciones de compatibilidad borrosas, con el objetivo de incluir el tipo de datos conjunto borroso en el marco de la programación lógica, véase por ejemplo [3]. Por otro lado están aquellos planteamientos, menos expresivos pero más eficientes desde el punto de vista de su implementación, que sustituyen el mecanismo de unificación sintáctico de la resolución SLD clásica, por un algoritmo de unificación débil, basado en relaciones de similaridad [10, 12, 13, 65, 64]. Con ello se obtiene un mecanismo operacional denominado resolución débil o resolución SLD basada en similaridad en [64].

  Mientras que el mecanismo global de resolución permanece básicamente inalterado, la unificación cambia drásticamente contemplando la posibilidad de "igualar" sintácticamente símbolos (de constante, función o predicado) distintos, siempre que exista un cierto grado de similaridad entre ellos. Este algoritmo, proporciona un unificador más general débil, así como un valor numérico, llamado grado de unificación. Intuitivamente, el grado de unificación representa el grado de verdad asociado con la instancia computada de la pregunta. Los programas escritos en esta clase de lenguajes consisten, en esencia, en un conjunto de cláusulas, junto a un conjunto de "ecuaciones de similaridad" que juegan un papel importante durante el proceso de unificación [64]. Esta última línea de trabajo ha llevado a la implementación de dos lenguajes [11, 42], que desgraciadamente no están accesibles al público y de los que se dispone limitada documentación.

- En la segunda aproximación, los programas son simples conjuntos de cláusulas, pero con la novedad de que cada una de ellas lleva asociado un grado de verdad que está explícitamente anotado. Los objetivos son preguntas que se lanzan al sistema en forma de una fórmula (atómica) más una substitución. El trabajo de computación y propagación de los grados de verdad recae en una extensión del principio de resolución clásico, o bien éste se sustituye por una forma de modus ponens generalizado con razonamiento hacia atrás, mientras que el mecanismo de unificación (sintáctica) permanece inalterado. Ejemplos de este tipo de lenguajes son el descrito en [68], o el presentado en [14] donde, además de introducir negación y considerar conjuntos difusos asociados a intervalos reales [37], cada cláusula de programa puede ser interpretada con una lógica diferente. Quizás, entre los lenguajes que se ajustan al esquema anteriormente descrito, el marco más flexible y evolucionado es el basado en la programación lógica multi-adjunta de [67, 44, 43, 45], donde cada programa lleva asociado un retículo de valores de verdad y se permite encapsular distintos tipos de lógicas difusas incluso dentro de una misma cláusula.

  Dado un programa lógico multi-adjunto, los objetivos son evaluados en dos fases computacionales separadas. Durante la fase operacional, se aplican de manera sistemática pasos admisibles. Un paso admisible para un átomo A seleccionado en un objetivo y una regla $\langle H \leftarrow_i B; \upsilon\rangle$ de un programa, tal que existe un unificador más general θ entre A y H, se obtiene al sustituir en el objetivo de partida el átomo A por la expresión $(\upsilon \&_i B)\theta$, donde "$\&_i$" es una conjunción adjunta ligada a la evaluación de la regla de modus ponens. Finalmente, la fase operacional devuelve una substitución computada junto con una expresión donde todos los átomos han sido explotados. Esta última expresión es interpretada posteriormente en un retículo concreto en una fase interpretativa, devolviendo un par $\langle$grado de verdad; substitución$\rangle$ que es el concepto borroso análogo a la noción clásica de respuesta computada usada tradicionalmente en la programación lógica.

Para finalizar, cabe indicar que la comunidad internacional que trabaja en programación lógica difusa aunque amplia es bastante dispersa. En la lista de investigadores más relevantes deberíamos incluir a: F. Arcelli, C. V. Damásio, G. Gerla, L. V. S. Lakshmanan, R. C. T. Lee, Y. Loyer, L. Paulik, E. Y. Shapiro, M. I. Sessa, V. S. Subrahmanian, U. Straccia, P. Vojtas. Muchos de los investigadores de la lista anterior lideran grupos de investigación muy activos en el área, otros se incluyen por la importancia de sus aportaciones. Algunos de los miembros más destacados de la comunidad nacional son: Teresa Alsinet, Luis Godó, Jesús Medina y Manuel Ojeda.

Capítulo 5

Programación Lógica Multi-adjunta

5.1. Introducción

La Programación Lógica Multi-adjunta es un entorno de programación lógica muy rico y flexible en el cual se combinarán la lógica difusa y la programación lógica. De manera informal, un programa lógico multi-adjunto puede verse como un conjunto de reglas con un grado de verdad asociado, y un objetivo es una pregunta al sistema más una sustitución (inicialmente la sustitución identidad, denotada por id). Dado un programa lógico multi-adjunto, los objetivos son evaluados en dos fases computacionales separadas.

Durante la fase operacional, se aplican de manera sistemática pasos admisibles (una generalización de la regla de inferencia clásica modus ponens) mediante un proceso de razonamiento hacia atrás, de manera análoga a la ejecución de pasos de resolución en la programación lógica pura. De modo más preciso, un paso admisible, para un átomo A seleccionado en un objetivo y una regla $\langle H \leftarrow_i B; v\rangle$ de un programa, tal que existe un unificador más general θ entre A y H, se obtiene al sustituir en el objetivo de partida el átomo A por la expresión $(v \&_i B)$, donde $\&_i$ es una conjunción adjunta ligada a la evaluación de la regla de modus ponens. Finalmente, la fase operacional devuelve una sustitución computada junto con una expresión donde todos los átomos han sido explotados. Esta última expresión es interpretada posteriormente en un retículo concreto en la fase que nosotros llamamos interpretativa, devolviendo un par (grado de verdad; sustitución) que es el concepto difuso análogo a la noción clásica de respuesta computada usada tradicionalmente en la programación lógica.

Un programa lógico multi-adjunto, interpretado en un retículo completo, precisa contener un tipo especial de reglas llamadas reductantes a fin de garantizar su completitud (aproximada). Esto introduce severos inconvenientes prácticos a la hora de implementar un sistema eficiente y completo de programación lógica multi-adjunta, ya que pueden dispararse tanto el tamaño de los programas, como los tiempos de ejecución. En general, la noción de reductante no es siempre computable y puede introducir una importante pérdida de eficiencia en las implementaciones prácticas. Por tanto, si se desea desarrollar un sistema completo y eficiente para este entorno de programación, resulta crucial definir técnicas de optimización para el cálculo de reductantes.

5.2. Sintaxis

Esta sección da un breve resumen de los principales rasgos sintácticos de la programación lógica multi-adjunta (remitimos al lector interesado a [30, 61] donde podrá encontrar una formulación más completa).

En cuanto al lenguaje, trabajamos con un lenguaje de primer orden, $\mathcal{L}$, conteniendo variables, símbolos

de función, símbolos de predicado, constantes, cuantificadores $\forall$ y $\exists$, y conectivas varias (arbitrarias) que mejoran la expresividad del mismo:

$$\begin{array}{ll} \wedge_1, \wedge_2, \ldots, \wedge_k & (conjunciones) \\ \vee_1, \vee_2, \ldots, \vee_l & (disyunciones) \\ \leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m & (implicaciones) \\ @_1, @_2, \ldots, @_n & (agregadores) \end{array}$$

En nuestro planteamiento, usamos varias conectivas de implicación ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$) y también otras conectivas que podrían agruparse bajo el nombre de agregadores u operadores de agregación. Se usan para combinar/propagar los valores de verdad a través de las reglas. La definición general de los operadores de agregación subsume los operadores de conjunción (denotados por $\wedge_1, \wedge_2, \ldots, \wedge_k$), de disyunción ($\vee_1, \vee_2, \ldots, \vee_l$) y otros operadores híbridos (usualmente denotados por $@_1, @_2, \ldots, @_n$). Aunque los conectores $\wedge_i$, $\vee_i$ y $@_i$ son operadores binarios, normalmente se generalizan como funciones con un número arbitrario de argumentos. En lo que sigue, escribiremos a menudo $@_i(x_1, \ldots, x_n)$ en vez de $@_i(x_1, @_i(x_2, \ldots, @_i(x_{n-1}, x_n)\ldots))$. Los operadores de agregación son útiles para describir/especificar las preferencias del usuario. Un operador de agregación, cuando se interpreta como una función de verdad, puede tener un significado aritmético, de suma ponderada o, en general, de cualquier aplicación monótona cuyos argumentos son valores de un retículo L completo. Por ejemplo, si un agregador @ es interpretado como $[[@]](x, y, z) = (3x + 2y + z)/6$, le estamos dando la mayor prioridad al primer argumento, luego al segundo, y así sucesivamente. Por definición, la función del grado de verdad para un operador de agregación n-ario $[[@]] : L^n \rightarrow L$ se requiere que sea monótono y que satisfaga $[[@]](\bot, \ldots, \bot) = \bot$, $[[@]](\top, \ldots, \top) = \top$.

Además, nuestro lenguaje $\mathcal{L}$ contiene valores de un retículo multi-adjunto, $(L, \prec, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$, equipado con una colección de pares adjuntos $\langle \leftarrow_i, \&_i \rangle$, donde cada $\&_i$ es un conjuntor ligado a la evaluación del modus ponens. En general, el conjunto de grados de verdad L puede ser un retículo completo aunque, a efectos de una mejor legibilidad, en los ejemplos tomaremos L como el intervalo cerrado [0, 1] (que es un retículo totalmente ordenado o cadena). La definición posterior introduce formalmente este concepto de retículo multi-adjunto, en los términos que se recogen en [61]:

Definición 5.2.1 *Sea $(L, \leq)$ un retículo. Un* retículo multi-adjunto *es una $n-$upla $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ que cumple las siguientes condiciones:*

1. *$(L, \leq)$ es acotado (es decir, existe el ínfimo, $\bot$, y el supremo, $\top$, de $(L, \leq)$).*
2. *$(L, \leq)$ es completo*[1]*, es decir, para todo subconjunto $X \subset L$ existen $inf(X)$, $sup(X)$*[2]*.*
3. *$\top \ \&_i \ v = v \ \&_i \ \top = v$, $\forall v \in L$, $i = 1, \ldots, n$.*
4. *cada par $(\leftarrow_i, \&_i)$, con $i = 1, \ldots, n$, es un* par adjunto*, es decir:*
   a) *la operación $\&_i$ es creciente en ambos argumentos*
   b) *la operación $\leftarrow_i$ es creciente en el primer argumento y decreciente en el segundo argumento.*
   c) propiedad adjunta*: $\forall x, y, z \in L$, $x \leq (y \leftarrow_i z) \Leftrightarrow (x \&_i z) \leq y$.*

Una $\mathcal{L}$-expresión es una expresión bien formada compuesta por valores y conectivas de $\mathcal{L}$, así como de símbolos de variable y operadores primitivos (por ejemplo, símbolos aritméticos como *,+, min, etc.). En lo que sigue, asumimos que la función de verdad de cualquier conectiva @ en L es dada por su correspondiente definición de conectiva, es decir, una ecuación de la forma $@(x_1, \ldots, x_n) = E$, donde E es una $\mathcal{L}$-expresión que no contiene símbolos de variable aparte de $x_1, \ldots, x_n$. Esta definición de $\mathcal{L}$-expresión la utilizaremos en el apartado 5.3.2 para la formalización de la fase interpretativa.

[1] En verdad, si $(L, \leq)$ es completo, entonces es acotado y podríamos reformular la definición evitando exigir explícitamente el carácter acotado.

[2] Téngase en cuenta que la existencia de $inf(X)$, $sup(X)$ está garantizada cuando X es finito, dado que L es un retículo.

Una **regla** es una fórmula lógica $H \leftarrow_i B$, donde H es una fórmula atómica (habitualmente llamada cabeza) y B (cuerpo) es una fórmula construida con fórmulas atómicas $B_1, \ldots, B_n$, $(n \geq 0)$, grados de verdad de L y conjunciones, disyunciones, agregadores y conjunciones adjuntas. Las reglas cuyo cuerpo es $\top$ se llaman hechos (habitualmente, representaremos un hecho como una regla con cuerpo vacío). Un objetivo es un cuerpo planteado como una pregunta al sistema. Las variables de las reglas están universalmente cuantificadas.

A grandes rasgos, un programa lógico multi-adjunto es un conjunto de pares $\langle R; \alpha \rangle$, donde R es una regla y α es el grado de verdad (un valor de L) que expresa la confianza del usuario del sistema acerca de dicha regla. Observemos que los grados de verdad están asignados axiomáticamente (por ejemplo) por un experto. Por abuso de lenguaje, en ocasiones nos referiremos al par $\langle R; \alpha \rangle$ como a una "regla". Existe una notación alternativa que permite expresar una regla $\langle R; \alpha \rangle$ como R *with* α. La siguiente definición precisa la idea expuesta anteriormente.

Definición 5.2.2 *Un programa lógico multi-adjunto es un conjunto $\mathcal{P}$ de reglas de la forma $\langle A \leftarrow_i \mathcal{B}; v \rangle$ verificando:*

i) A es una fórmula atómica (llamada cabeza).

ii) $\mathcal{B}$ es una fórmula arbitraria, llamada cuerpo, construida con fórmulas atómicas $B_1, \ldots, B_n, n \geq 0$ y cualesquiera conjunciones, disyunciones, agregadores, grados de verdad $\alpha \in L$ y conjunciones adjuntas $\&_i$.

iii) $v \in L$ es el grado de verdad de la fórmula lógica $A \leftarrow_i \mathcal{B}$, donde $(L, \leq)$ es el retículo multi-adjunto asociado al programa $\mathcal{P}$.

5.3. Semántica Procedural

La semántica procedural del lenguaje lógico multi-adjunto $\mathcal{L}$ puede describirse contemplando dos fases: una fase operacional seguida de una fase interpretativa. En esta sección establecemos una separación clara entre ambas fases.

5.3.1. Fase Operacional

El mecanismo operacional usa una generalización del modus ponens tal que, dado un objetivo A y una regla del programa $\langle H \leftarrow_i B; v \rangle$, si hay una sustitución $\theta = umg(\{A = H\})$, sustituimos el átomo A por la expresión $(v \&_i B)\theta$. En lo que sigue, escribiremos $C[A]$ para denotar una fórmula donde A es una subexpresión (frecuentemente un átomo) que aparece arbitrariamente en el contexto $C[]$ (posiblemente vacío). Además, la expresión $C[A/H]$ significa el reemplazamiento de A por H en el contexto $C[]$. Usaremos también $Var(s)$ para referirnos al conjunto de variables distintas que aparecen en el objeto sintáctico s, mientras que $\theta[Var(s)]$ denota la sustitución obtenida a partir de θ restringiendo su dominio, $Dom(\theta)$, a $Var(s)$.

Definición 5.3.1 (Paso Admisible) *Sea $\mathcal{Q}$ un objetivo y σ una sustitución. El par $\langle \mathcal{Q}; \sigma \rangle$ es un estado y denotamos por $\mathcal{E}$ el conjunto de estados. Dado un programa $\mathcal{P}$, una computación admisible se formaliza como un sistema de transición de estados, cuya relación de transición $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ es la menor relación que satisface las siguientes reglas admisibles (donde consideramos en todo caso que A es el átomo seleccionado en $\mathcal{Q}$):*

Regla 1:

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v\&_i B])\theta; \sigma\theta \rangle \text{ si } \begin{cases} A \text{ es el átomo seleccionado en } \mathcal{Q}, \\ \theta = umg(\{A' = A\}), \\ \langle A' \leftarrow_i B; v \rangle \in \mathcal{P}, \text{ y } B \text{ es no vacío.} \end{cases}$$

Regla 2:

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle \text{ si } \begin{cases} A \text{ es el átomo seleccionado en } \mathcal{Q}, \\ \theta = umg(\{A' = A\}), \\ \langle A' \leftarrow_i; v \rangle \in \mathcal{P} \end{cases}$$

Regla 3:

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle \text{ si } \begin{cases} A \text{ es el átomo seleccionado en } \mathcal{Q}, \\ \text{No hay una regla en } \mathcal{P} \text{ cuya cabeza unifique con } A \end{cases}$$

Ejemplo 5.3.2 *En este ejemplo concretamos la aplicación de cada una de las reglas anteriores.*

1. *Sea* $\langle p(a) \leftarrow_{prod} p(f(a)); 0.7 \rangle$ *una regla. Si lanzamos el objetivo que mostramos a continuación, podemos dar un paso admisible usando la* ***Regla 1*** *con dicha regla:*

$$\langle (p(b) \ \&_G \ p(X)) \ \&_G \ q(X); id \rangle \rightarrow_{AS1} \langle (p(b) \&_G \ 0.7 \ \&_p rod \ p(f(a))) \ \&_G \ q(a); \{X/a\} \rangle$$

2. *ea* $\langle p(a) \leftarrow_{prod}; 0.7 \rangle$ *una regla. Si lanzamos el objetivo que mostramos a continuación, podemos dar un paso admisible usando la* ***Regla 2*** *con dicha regla:*

$$\langle (p(b) \ \&_G \ p(X)) \ \&_G \ q(X); id \rangle \rightarrow_{AS2} \langle (p(b) \&_G \ 0.7) \ \&_G \ q(a); \{X/a\} \rangle$$

3. *Si ninguna regla unifica con* $p(X)$*:*

$$\langle (p(b) \ \&_G \ p(X)) \ \&_G \ q(X); id \rangle \rightarrow_{AS3} \langle (p(b) \&_G \ 0) \ \&_G \ q(X); id \rangle$$

Las fórmulas involucradas en pasos admisibles de computación se renombran antes de que sean usadas. Obsérvese además, que la **Regla 3** está contemplada para evitar las (posibles) derivaciones admisibles fallo. Cuando se necesite, debemos usar los símbolos $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ y $\rightarrow_{AS3}$ para distinguir entre los distintos pasos de computación dados aplicando una de las reglas admisibles en concreto. Además, cuando sea necesario, cada regla del programa que ha sido usada en el correspondiente paso será anotada como un súperíndice del símbolo $\rightarrow_{AS}$.

Definición 5.3.3 (Derivación admisible) *Sea* $\mathcal{P}$ *un programa y sea* $\mathcal{Q}$ *un objetivo. Una* ***derivación admisible*** *es una secuencia de cero o más pasos de derivación* $\langle \mathcal{Q}; id \rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \theta \rangle$*. Si* $\mathcal{Q}'$ *es una fórmula que no contiene átomos, el par* $\langle \mathcal{Q}'; \sigma \rangle$*, donde* $\sigma = [Var(\mathcal{Q})]$*, se llama* ***respuesta computada admisible*** *(a.c.a., de admissible computed answer) para la derivación.*

Ilustramos todos los conceptos referidos en el presente capítulo en el siguiente ejemplo.

Ejemplo 5.3.4 *Sea* $\mathcal{P}$ *el siguiente programa y* $([0,1], \leq)$ *el retículo multi-adjunto asociado.*

$$\begin{array}{ll}\mathcal{R}_1 : \langle p(X) \leftarrow_P \&_G(q(X), @_{aver}(r(X),\ s(X))) & ;\ 0.9\rangle \\ \mathcal{R}_2 : \langle q(a) \leftarrow & ;\ 0.8\rangle \\ \mathcal{R}_3 : \langle r(X) \leftarrow & ;\ 0.7\rangle \\ \mathcal{R}_4 : \langle s(X) \leftarrow & ;\ 0.5\rangle \end{array}$$

Las etiquetas P y G se refieren a la lógica del producto y de Gödel, respectivamente, mientras que $@_{aver}$ es el agregador dado por la media aritmética. Esto es $[[\&_P]](x,y) = x \cdot y$, $[[\&_G]](x,y) = min(x,y)$, y $[[@_{aver}]] = (x+y)/2$. En la siguiente derivación admisible para el programa $\mathcal{P}$ y el objetivo $\leftarrow p(X)$, subrayaremos la expresión seleccionada en cada paso admisible:

$$\begin{array}{ll}\langle \underline{p(X)}; id\rangle & \rightarrow^{R_1}_{AS1} \\ \langle \&_P(0.9, \&_G(\underline{q(X_1)}, @_{aver}(r(X_1), s(X_1)))); \{X/X_1\}\rangle & \rightarrow^{R_2}_{AS2} \\ \langle \&_P(0.9, \&_G(0.8, @_{aver}(\underline{r(a)}, s(a)))); \{X/a\}\rangle & \rightarrow^{R_3}_{AS2} \\ \langle \&_P(0.9, \&_G(0.8, @_{aver}(0.7, \underline{s(a)}))); \{X/a\}\rangle & \rightarrow^{R_4}_{AS2} \\ \langle \&_P(0.9, \&_G(0.8, @_{aver}(0.7, 0.5))); \{X/a\}\rangle & \end{array}$$

Por tanto, la a.c.a. asociada a esta derivación admisible es $\langle \&_P(0.9, \&_G(0.8, @_{aver}(0.7, 0.5))); \{X/a\}\rangle$.

5.3.2. Fase Interpretativa

Si desplegamos todos los átomos del objetivo, aplicando tantos pasos admisibles como sean necesarios durante la fase operacional, entonces éste se transforma en una fórmula sin átomos que puede ser interpretada directamente en el retículo multi-adjunto L. Esto justifica las siguientes nociones de paso de computación interpretativo, derivación interpretativa y respuesta computada interpretativa.

Definición 5.3.5 (Paso Interpretativo) *Sea $\mathcal{P}$ un programa, $\mathcal{Q}$ un objetivo y σ una sustitución. Formalizamos la noción de computación interpretativa como un sistema de transición de estados, cuya relación de transición $\rightarrow_{IS} \subseteq (\mathcal{E} \times \mathcal{E})$, donde $\mathcal{E}$ es el conjunto de estados, se define como la más pequeña relación binaria que satisface:*

$$\langle \mathcal{Q}[@(r_1, r_2)]; id\rangle \rightarrow_{IS} \langle \mathcal{Q}[@(r_1, r_2)/[[@]](r_1, r_2)]\rangle$$

donde $[[@]]$ es la función de verdad de la $@$ en el retículo $(L; \leq)$ asociado a $\mathcal{P}$.

Ejemplo 5.3.6 *Siendo la función de verdad asociada al operador producto $\&_P$, con el siguiente objetivo podemos dar un paso interpretativo del modo:*

$$\langle 0.8 \&_{luka} ((\underline{0.7 \&_P 0.9}) \&_G 0.7); \theta\rangle \rightarrow_{IS} \langle 0.8 \&_{luka} ((\underline{0.63}) \&_G 0.7); \theta\rangle$$

Definición 5.3.7 (Derivación Interpretativa) *Sea $\mathcal{P}$ un programa y $\langle \mathcal{Q}; \sigma\rangle$ una a.c.a., es decir, $\mathcal{Q}$ es un objetivo que no contiene átomos. Una **derivación interpretativa** es una secuencia de cero o más pasos interpretativos $\langle \mathcal{Q}; \sigma\rangle \rightarrow^*_{IS} \langle \mathcal{Q}'; \sigma\rangle$. Cuando $\mathcal{Q}' = r$, $r \in L$, siendo $(L; \leq)$ el retículo asociado a $\mathcal{P}$, el estado $\langle r, \sigma\rangle$ se dice que es una **respuesta computada difusa** (f.c.a., fuzzy computed answer) para la derivación.*

*Frecuentemente, llamaremos **derivación completa** a la secuencia de pasos admisibles/interpretativos de la forma $\langle \mathcal{Q}; id\rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \sigma\rangle \rightarrow^*_{IS} \langle r; \sigma\rangle$ (a veces denotaremos también $\langle \mathcal{Q}; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\rangle$) donde $\langle \mathcal{Q}'; \sigma[Var(\mathcal{Q})]\rangle$ y $\langle r; \sigma[Var(\mathcal{Q})]\rangle$ son, respectivamente, la a.c.a. y la f.c.a. para la derivación.*

Ejemplo 5.3.8 *Ahora completamos la derivación previa del Ejemplo 5.3.4 al ejecutar los pasos interpretativos necesarios para obtener la respuesta computada con respecto al retículo* $\langle [0,1], \leq \rangle$.

$$\begin{array}{ll}
\langle \&_P(0.9, \&_G(0.8, \underline{@_{aver}(0.7, 0.5)})); \{X/a\}\rangle & \rightarrow_{IS} \\
\langle \&_P(0.9, \underline{\&_G(0.8, 0.6)}); \{X/a\}\rangle & \rightarrow_{IS} \\
\langle \underline{\&_P(0.9, 0.6)}; \{X/a\}\rangle & \rightarrow_{IS} \\
\langle 0.54; \{X/a\}\rangle &
\end{array}$$

Por tanto la respuesta computada difusa (f.c.a.) para esta derivación ya completa es el par $\langle 0.54; \{X/a\}\rangle$. *Denominaremos* D_1 *a esta derivación completa en los siguientes apartados.*

Capítulo 6

Gestión de retículos y costes computacionales en FLOPER

6.1. Introducción

Las técnicas que hemos visto a lo largo de este trabajo, como la generación de código Prolog a partir de código difuso, representación del código difuso a bajo nivel, generación de árboles de desplegado, etc., se han llevado a la práctica a través de la implementación de una herramienta prototipo llamada FLOPER.

FLOPER (Fuzzy LOgic Programming Environment for Research) es un ambicioso proyecto que se está llevando a cabo en esta universidad dentro del grupo de investigación DEC-TAU y que tiene como objetivo final la construcción de un lenguaje de programación lógico difuso. Para el desarrollo del sistema se ha trabajado con SICStus Prolog v3.12.7, aunque no hay ningún problema de compatibilidad con versiones más antiguas. El sistema FLOPER no necesita una instalación previa ya que es capaz de ejecutarse únicamente introduciendo la siguiente expresión: `?- consult(floper.pl)`. Puede descargarse la herramienta FLOPER de la página http://dectau.uclm.es/floper/, que incluye gran cantidad de información sobre el sistema.

A continuación, mostraremos la interfaz de esta herramienta y comentaremos las opciones más interesantes que nos ofrece este sistema. En cada una de ellas, significaremos las aportaciones o mejoras que hemos realizado a lo largo de este Máster.

6.2. Menú Principal

Al iniciar la aplicación FLOPER se mostrará un menú que contiene los distintos comandos que se pueden ejecutar y una descripción breve de los mismos. Este menú aparecerá cada vez que se ejecute y se procese completamente una orden.

Como vemos en la Figura 6.1 la interfaz esta dividida en tres menús diferentes. El primero (`PROGRAM MENU`) encuadra las opciones referentes al programa, como pueden ser cargar/salvar un programa, analizarlo, etc. El segundo (`GOAL MENU`) refleja las opciones relativas a los objetivos: introducción de los mismos, ejecución de objetivos, generación de árboles de desplegado a partir de ellos y profundidad de dichos árboles. El último submenú (`TRANSFORMATION MENU`) se refiere a técnicas de transformación (plegado/desplegado, evaluación parcial, etc.) para los programas difusos.

Estas opciones del último menú no han sido implementadas hasta el momento, pero ya están integradas

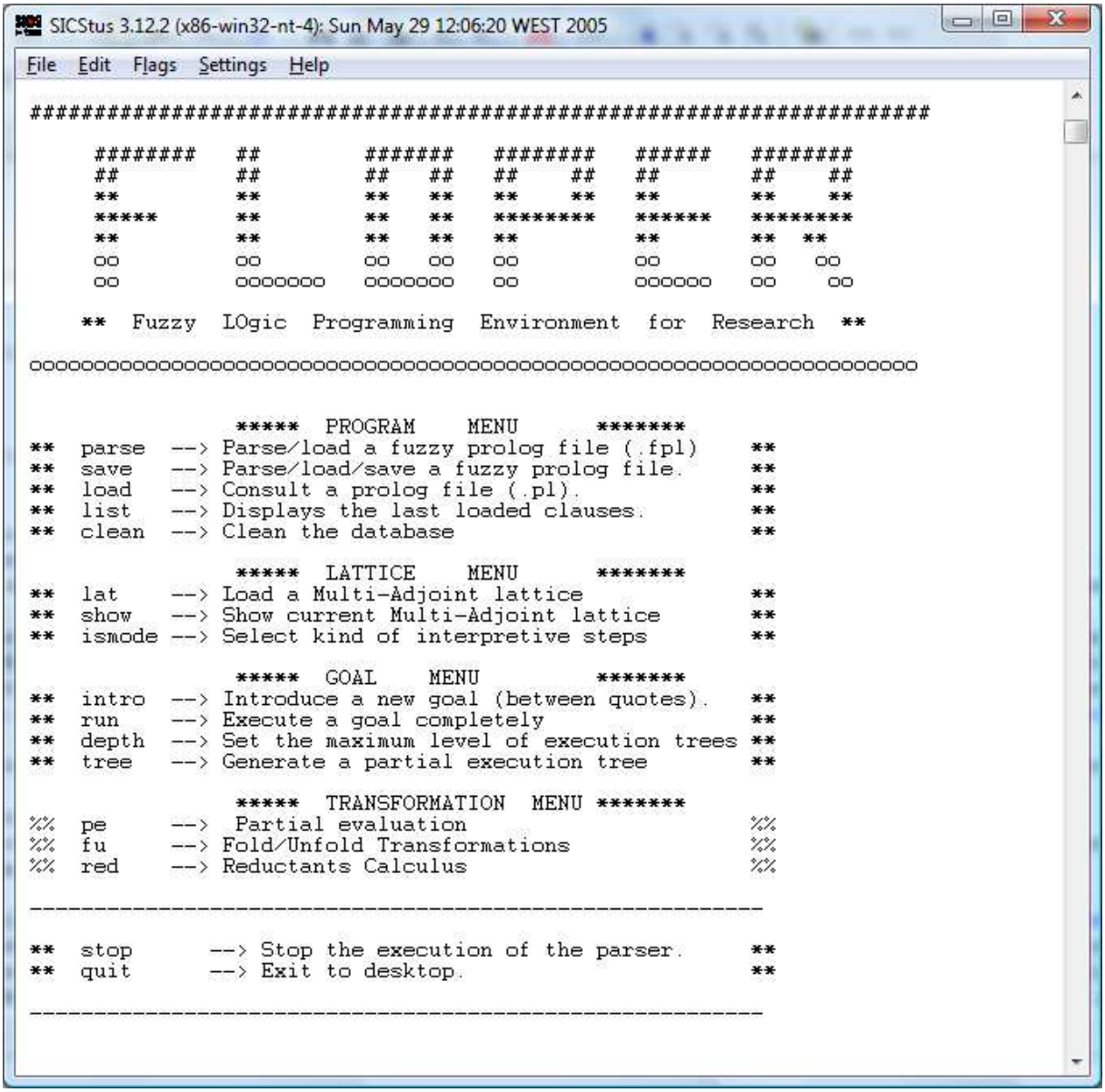

Figura 6.1: Interfaz de FLOPER

en la interfaz para su uso futuro. Fuera de estos menús se encuentran opciones más generales como son la de terminación de la ejecución del analizador (Stop) y cierre del programa Sictus (Quit).

6.2.1. Opción 'load'

El comando 'load' se encarga de consultar un fichero Prolog estándar (con extensión '.pl'), es decir, cargarlo en la base de datos. Normalmente, este fichero contiene un conjunto de cláusulas que implementan agregadores, predicados de usuario, etc. Sin embargo, las conectivas lógicas originales del Producto, Gödel y Łukasiewicz expresadas en código Prolog vistas en apartados anteriores, están definidas en el fichero "num.pl", el cual se carga automáticamente por el sistema al comienzo de cada sesión de trabajo.

6.2.2. Opción 'parse'

Ésta es una de las opciones básicas del sistema. Se encarga de analizar un programa difuso de entrada (incluido en un fichero con extensión '.fpl') y decir si es sintácticamente correcto o no. Se usa para ello el mecanismo de las DCG's (*Definite Clause Grammars*), del mismo modo que se describe en [17]. Además, si el programa difuso es correcto, se realiza a la vez el proceso de generación de código.

Como ya hemos comentado en esta memoria, el sistema genera dos códigos diferentes a partir del mismo programa difuso de entrada. Por un lado, se genera código Prolog puro equivalente al código difuso de entrada, que nos sirve para hacer ejecuciones completas de objetivos; y por otro lado se genera una representación a bajo nivel del código de entrada, muy útil para procesos de depuración y optimización. Para simultanear el proceso de análisis con el de generación de código, cada predicado de análisis usado por las reglas DCG's, ha sido aumentado con dos argumentos extras (dos variables) las cuales se utilizan para contener el código Prolog generado y la representación a bajo nivel generada después de analizar el correspondiente fragmento de código difuso respectivamente. También se admite la presencia de cláusulas Prolog puras dentro de un fichero '.fpl', precediéndolo con un símbolo '$'.

Cabe destacar las significativas actualizaciones llevadas a cabo en este Máster en la nueva versión del analizador sintáctico y, consecuentemente, en el lenguaje de aceptación de FLOPER. Estas actualizaciones conllevan principalmente una ampliación del lenguaje MALP que simplifica su escritura. A continuación se detallan los cambios principales:

Ampliación del concepto de grado de verdad

En las primeras versiones de FLOPER, el sistema estaba restringido al retículo [0,1]. Actualmente puede usarse un elemento de cualquier retículo multi-adjunto que se asocie al programa. Los grados de verdad se definen sintácticamente de un modo parecido al de un término y pueden ser:

- Números
- Literales
- Listas
- Funciones

Relajación de la especificidad de las implicaciones

Anteriormente, FLOPER exigía especificar la implicación de cada regla del programa, esto es, era necesario indicar mediante una etiqueta la implicación concreta que se quería emplear (por ejemplo, `<godel`). Ahora se permite abstraer la implicación: no es necesario indicar exactamente cuál se desea usar. Así, si no es relevante, el programador pasará a escribir `<-` en la implicación. FLOPER escogerá una implicación arbitraria de las que se hayan definido en el retículo.

Permitir código Prolog puro entre llaves

Para aumentar la expresividad de los programas MALP, se permite ahora la inclusión de código Prolog (conjunciones de átomos) dentro del cuerpo de una regla. Un ejemplo de regla válida incluye ahora:

```
p(X) <prod &godel(q(X), {r(X,_), s(X,_)}) with 1.
```

A efectos semánticos se considerará que el código entre llaves se sustituye por $\top$ en caso de satisfacerse, o por $\bot$ en caso contrario, siendo $\top = sup(L)$ y $\bot = inf(L)$.

Permitir reglas sin pesos explícitos

En la nueva versión de FLOPER deja de ser obligatorio indicar el peso de cada regla (su grado de verdad), introducido por la palabra clave '`with`'. En su lugar, el programador puede omitir totalmente el grado de verdad cuando éste coincida con el elemento $\top$ del retículo asociado. Así, la regla:

```
p(X) <prod &godel(q(X), @aver(r(X), s(X))).
```

equivale a

```
p(X) <prod &godel(q(X), @aver(r(X), s(X))) with 1.
```

Permitir variables en los grados de verdad

En la versión original de FLOPER, un grado de verdad debía ser una constante numérica. Al ampliar esta definición para incluir toda clase de términos Prolog, se ha permitido la posibilidad de incluir variables dentro de un peso, con la única restricción de que el grado de verdad no debe ser él mismo una variable. Por ejemplo, sí podrá ser un término que contiene entre sus parámetros una variable, como se indica a continuación.

```
p <- q(X) &prod tv(X) with tv(1).
```

6.2.3. Opción 'list'

Esta selección muestra un listado del conjunto de cláusulas Prolog cargadas de un fichero '.pl' así como aquéllas obtenidas después de compilar un fichero '.fpl'. Naturalmente, el programa difuso original contenido en este último fichero también se muestra. Es importante destacar que se muestra la traducción a código Prolog del código difuso, pero no la representación a bajo nivel que el analizador también realiza de éste. Esta representación se guarda en un fichero intermedio, ya que creemos que sólo es interesante para el implementador. En la Figura 6.2 podemos apreciar el resultado de ejecutar la opción 'list'. El programa que se muestra ha sido previamente analizado mediante la opción 'parse' de FLOPER (es el mismo programa que utilizamos para el Ejemplo 5.3.4 del Capítulo 5).

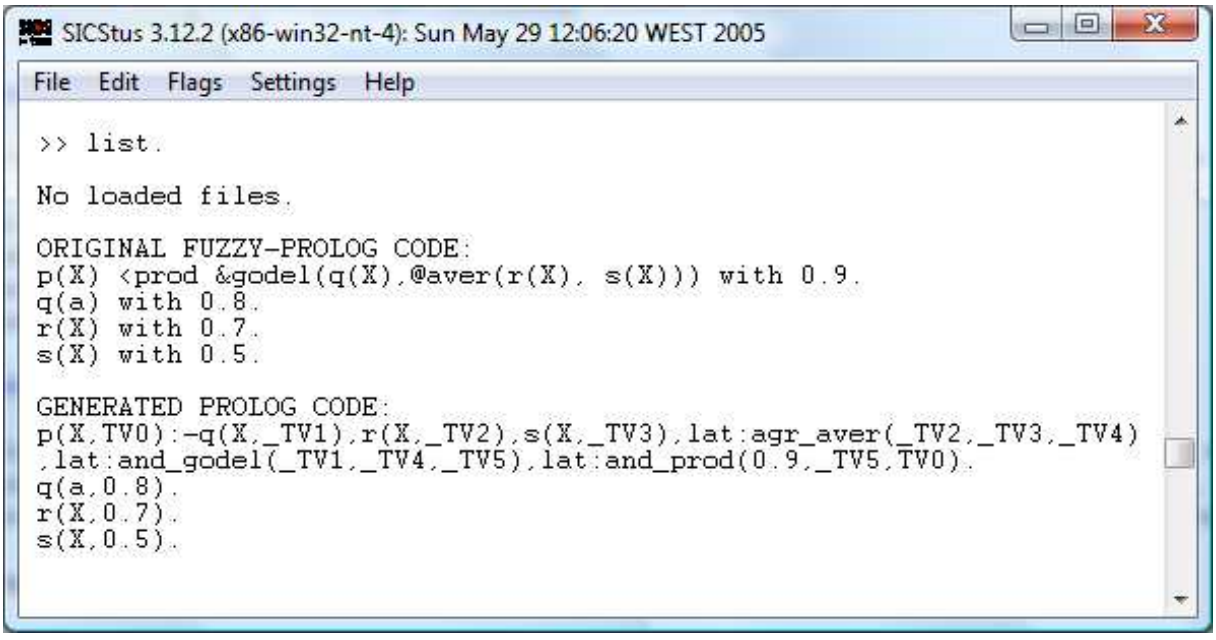

Figura 6.2: Ejecución de la opción 'list'

6.2.4. Opción 'save'

Salva el código Prolog resultante en un fichero. Aquí queremos señalar que el conjunto de cláusulas obtenido durante el proceso de compilación son también automáticamente asertadas en la base de datos del intérprete Prolog, el cual obviamente también contiene las cláusulas que implementan la propia herramienta. Esta acción nos ayudará en el desarrollo de la siguiente opción.

6.2.5. Opción 'run'

Su misión es ejecutar un objetivo difuso (introducido por teclado mediante la opción 'intro'). Esta opción de ejecución traduce código multi-adjunto a Prolog para hacer ejecuciones completas del programa y objetivo difusos introducidos. A continuación mostramos un ejemplo de la salida que ofrece la opción 'run' en FLOPER cuando ejecutamos el objetivo p(X) sobre el programa del Ejemplo 5.3.4 (que daba tres soluciones):

```
>> run.
[Truth_degree=0.54,X=a]
```

Como vemos, la respuesta computada difusa es la esperada, la ejecución ha sido correcta y completa.

6.3. Medidas de Costes y Opción 'tree'

Con la opción 'tree' se realiza la generación de árboles de desplegado de un programa. En concreto, gracias a la representación a bajo nivel que se hace del código difuso de entrada, tanto de los programas como de los objetivos, podemos generar el árbol de ejecución asociado. El nivel de profundidad de este árbol puede ser modificado por el usuario mediante la opción de FLOPER 'depth'. El objetivo para el que se desee generar el árbol debe ser previamente introducido por teclado mediante la opción 'intro'.

El tipo de información que muestra este comando depende de la opción 'ismode' seleccionada por el usuario. En la Sección 6.3.1 se dará cuenta de las diferentes opciones 'ismode' y se mostrarán los árboles correspondientes para cada una.

Un modo sencillo para estimar el coste computacional requerido para realizar una derivación consiste en contar el número de pasos computacionales ejecutados en la misma. Así, dada la derivación D, definimos su:

- *Coste operacional*, $\mathcal{O}_{\mathcal{C}}(D)$, como el número de pasos admisibles ejecutados en D.
- *Coste interpretativo*, $\mathcal{I}_{\mathcal{C}}(D)$, como el número de pasos interpretativos ejecutados en D.

Obsérvese que los costes operacional e interpretativo de la derivación D del Ejemplo 5.3.4 y 5.3.8 son, $\mathcal{O}_{\mathcal{C}}(D_1) = 4$ y $\mathcal{I}_{\mathcal{C}}(D_1) = 3$, respectivamente. Intuitivamente, $\mathcal{O}_{\mathcal{C}}$ cuenta el número de átomos explotados durante la derivación. De forma similar, $\mathcal{I}_{\mathcal{C}}$ estima el número de conectivos evaluados. Sin embargo, no los cuenta totalmente: $\mathcal{I}_{\mathcal{C}}$ sólo tiene en cuenta aquellos conectivos que aparecen en los cuerpos de las reglas del programa, pero no los que se anidan recursivamente en la definición de otros conectivos. El siguiente ejemplo lo pone de manifiesto:

Ejemplo 6.3.1 *Sea $\mathcal{P}$ el programa del Ejemplo 5.3.4, asociado al retículo* $([0,1], \leq, \leftarrow_P, \&_P, \leftarrow_G, \&_G)$, *donde* $\&_P(x_1, x_2) \triangleq x_1 \cdot x_2$, $\&_G(x_1, x_2) \triangleq min(x_1, x_2)$ *y* $@_{aver}(x_1, x_2) \triangleq (x_1 + x_2)/2$.

Una versión simplificada de la regla $\mathcal{R}_1$ de $\mathcal{P}$, cuyo cuerpo sólo contiene un símbolo de agregación, es: $\langle \mathcal{R}_1^* : \quad p(X) \leftarrow_P @_1(q(X), r(X), s(X)); 0.9\rangle$, *donde* $@_1(x_1, x_2, x_3) \triangleq \&_G(x_1, @_{aver}(x_2, x_3))$. *Obsérvese que $\mathcal{R}_1^*$ tiene el mismo significado que $\mathcal{R}_1$, aunque tenga una sintaxis distinta. La diferencia se encuentra en el conjunto de conectivos que aparecen explícitamene en el cuerpo de cada una. Ahora usamos $\mathcal{R}_1^*$ en lugar de $\mathcal{R}_1$ para generar la derivación D_1^*, que termina con la misma f.c.a. que D_1.*

$$
\begin{array}{ll}
\langle \underline{p(X)}; id\rangle & \rightarrow_{AS1}^{R_1} \\
\langle \&_P(0.9, @_1(\underline{q(X_1)}, r(X_1), s(X_1))); \{X/X_1\}\rangle & \rightarrow_{AS2}^{R_2} \\
\langle \&_P(0.9, @_1(0.8, \underline{r(X_1)}, s(X_1))); \{X/a\}\rangle & \rightarrow_{AS2}^{R_3} \\
\langle \&_P(0.9, @_1(0.8, 0.7, \underline{s(X_1)})); \{X/a\}\rangle & \rightarrow_{AS2}^{R_4} \\
\langle \&_P(0.9, \underline{@_1(0.8, 0.7, 0.5)}); \{X/a\}\rangle & \rightarrow_{IS} \\
\langle \underline{\&_P(0.9, 0.6)}; \{X/a\}\rangle & \rightarrow_{IS} \\
\langle 0.54; \{X/a\}\rangle &
\end{array}
$$

Obsérvese que, ya que se han explotado los mismos átomos que en la derivación D_1, por lo que $\mathcal{O}_{\mathcal{C}}(D_1^*) = \mathcal{O}_{\mathcal{C}}(D_1) = 4$. Sin embargo, aunque los conectivos $\&_G$ y $@_{aver}$ han sido evaluados en ambas derivaciones, en D_1^* dichas evaluaciones no se cuentan explícitamente como pasos interpretativos y, consecuentemente, no incrementan el coste interpretativo $\mathcal{I}_{\mathcal{C}}$. Esto se refleja en el resultado anormal $\mathcal{I}_{\mathcal{C}}(D_1) = 3 > 2 = \mathcal{I}_{\mathcal{C}}(D_1^*)$. Nótese que D_1^* no es una versión optimizada de D_1, incluso aunque la medida errónea $\mathcal{I}_{\mathcal{C}}$ parezca indicar lo contrario.

Este problema se señaló inicialmente en [29], donde se propuso una solución preliminar mediante la asignación de pesos a los conectivos que tuvieran en cuenta el número de operadores primitivos involucrados en la definición del propio conectivo @. Más aún, en [48] se mejoró la noción de *peso de conectivo* para tener en cuenta también el número de llamadas recursivas a conectivos difusos (directa o indirectamente) realizadas en la definición de @.

Una manera diferente de resolver el problema se presenta en [47] donde, en lugar de usar pesos de conectivos, se opta por un método más "visual", ya implementado en FLOPER, basado en la redefinición de la fase interpretativa.

Definición 6.3.2 (Paso interpretativo corto) *Sea $\mathcal{P}$ un programa, $\mathcal{Q}$ un objetivo y σ una sustitución. Asumimos que la L-expresión $\Omega(r_1, \ldots, r_n)$ aparece en $\mathcal{Q}$, donde Ω es un operador primitivo o un conectivo definido en el retículo $(L, \leq)$ asociado a $\mathcal{P}$, y $r_1, \ldots, r_n$ son elementos de L. Formalizamoos la noción de* paso interpretativo corto *como un sistema de transición de estados, cuya relación de transición $\overset{SIS}{\rightsquigarrow} \subseteq (\mathcal{E} \times \mathcal{E})$ es la menor relación que satisface las siguientes* reglas de interpretación corta *(donde siempre consideraremos que $\Omega(r_1, \ldots, r_n)$ es la L-expresión seleccionada en $\mathcal{Q}$ y $\mathcal{E}$ es el conjunto de estados):*

1) $\langle Q[\Omega(r_1, \ldots, r_n)]; \sigma\rangle \overset{SIS}{\rightsquigarrow} \langle Q[\Omega(r_1, \ldots, r_n)/E']; \sigma\rangle$, *si Ω es un conectivo definido como $\Omega(x_1, \ldots, x_n) \triangleq E$ y E' se obtiene de la L-expresión E reemplazando cada variable (parámetro formal) x_i por su correspondiente valor (parámetro real) r_i, $1 \leq i \leq n$, que es, $E' = E[x_1/r_1, \ldots, x_n/r_n]$.*

2) $\langle Q[\Omega(r_1, \ldots, r_n)]; \sigma\rangle \overset{SIS}{\rightsquigarrow} \langle Q[\Omega(r_1, \ldots, r_n)/r]; \sigma\rangle$, *si Ω es un operador primitivo que, una vez evaluado con sus parámetros $r_1, \ldots, r_n$, produce el resultado r.*

A partir de aquí usaremos los símbolos $\overset{\text{SIS1}}{\rightsquigarrow}$ y $\overset{\text{SIS2}}{\rightsquigarrow}$ para distinguir entre pasos de computación ejecutados aplicando la regla *interpretativa corta* específica. Más aún, cuando usemos la expresión *derivación interpretativa*, nos referiremos a una secuencia de *pasos interpretativos cortos* (de acuerdo a la definición anterior) en lugar de a una secuencia de *pasos interpretativos* (de acuerdo a la Definición 5.3.5). Obsérvese

que este hecho supone también una ligera revisión de la Definición 5.3.7, que no afecta la esencia de la definición de "respuesta computada difusa": la aplicación repetida de ambos tipos de pasos interpretativos en un estado dado sólo afecta a la longitud de las derivaciones correspondientes, pero ambas llegan a los mismos estados finales (que contienen las correspondientes respuestas computadas difusas).

Ejemplo 6.3.3 *Recordando de nuevo la a.c.a. obtenida en el Ejemplo 5.3.4, se puede llegar a la respuesta computada difusa* $\langle 0.63; \{X/a\}\rangle$ *(conseguida en el Ejemplo 5.3.8 por medio de pasos interpretativos) mediante la generación de la siguiente derivación interpretativa* D_2 *basada en pasos interpretativos cortos:*

$$
\begin{array}{ll}
\langle \&_P(0.9, \&_G(0.8, \underline{@_{aver}(0.7, 0.5)})); \{X/a\}\rangle & \overset{SIS1}{\rightsquigarrow} \\
\langle \&_P(0.9, \&_G(0.8, \underline{(0.7 + 0.5)/2)}); \{X/a\}\rangle & \overset{SIS2}{\rightsquigarrow} \\
\langle \&_P(0.9, \&_G(0.8, \underline{1.2/2})); \{X/a\}\rangle & \overset{SIS2}{\rightsquigarrow} \\
\langle \&_P(0.9, \underline{\&_G(0.8, 0.6)}); \{X/a\}\rangle & \overset{SIS1}{\rightsquigarrow} \\
\langle \&_P(0.9, \underline{min(0.8, 0.6)}); \{X/a\}\rangle & \overset{SIS2}{\rightsquigarrow} \\
\langle \underline{\&_P(0.9, 0.6)}; \{X/a\}\rangle & \overset{SIS1}{\rightsquigarrow} \\
\langle \underline{0.9 \cdot 0.6}; \{X/a\}\rangle & \overset{SIS2}{\rightsquigarrow} \\
\langle 0.54; \{X/a\}\rangle &
\end{array}
$$

Regresando al Ejemplo 6.3.1, podemos recontruir la fase interpretativa de la derivación D_1^* *en términos de pasos interpretativos cortos, generando así la siguiente derivación interpretativa* D_2^**. En primer lugar, aplicando un paso* $\overset{SIS1}{\rightsquigarrow}$ *en la L-expresión* $\&_P(0.9, \underline{@_1(0.8, 0.7, 0.5)})$*, se transforma en el subobjetivo* $\&_P(0.9, \&_G(0.8, @_{aver}(0.7, 0.5)))$*, y los estados posteriores de la derivación interpretativa coinciden con los de la derivación* D_2 *previa.*

En este momento, es necesario tener en cuenta las medidas de coste relativas a las derivaciones D_1, D_1^*, D_2 y D_2^*. En primer lugar, obsérvese que el cose operacional de todas ellas coincide, lo que era de esperar. Sin embargo, mientras que $\mathcal{I}_{\mathcal{C}}(D_1) = 3 > 2 = \mathcal{I}_{\mathcal{C}}(D_1^*)$, tenemos ahora que $\mathcal{I}_{\mathcal{C}}(D_2) = 7 < 8 = \mathcal{I}_{\mathcal{C}}(D_2^*)$. Esta aparente contradicción puede confundirnos a la hora de decidir qué regla del programa ($\mathcal{R}_1$ o $\mathcal{R}_1^*$) es más apropiada. El uso de la Definición 6.3.2 en las derivaciones D_2 y D_2^* es la clave para resolver el problema, como veremos a continuación. En el Ejemplo 6.3.1 justificamos que contar simplemente el número de pasos interpretativos realizados en la Definición 5.3.5 puede producir resultados anormales, pues la evaluación de conectivos con complejidades diferentes eran medidas (erróneamente) con el mismo coste computacional.

Afortunadamente, la noción de paso interpretativo corto hace visible en la propia derivación todos los conectivos y operadores primitivos que aparecen en las derivaciones (posiblemente anidadas de forma recursiva) de un conectivo que aparezca en cualquier estado de la derivación. Como hemos visto, en D_2 se han expandido en tres pasos interpretativos cortos de tipo 1 las definiciones de tres conectivos, como $@_{aver}, \&_{\mathtt{G}}$ y $\&_{\mathtt{P}}$, y se han aplicado cuatro pasos interpretativos cortos de tipo 2 para resolver operadores primitivos, com $+$, min, $/$ y $*$. El mismo esfuerzo computacional se ha realizado en D_2^*, aunque aquí también se ha aplicado un paso corto de tipo 1 más para expandir el conectivo $@_1$. Esto justifica por qué $\mathcal{I}_{\mathcal{C}}(D_2) = 7 < 8 = \mathcal{I}_{\mathcal{C}}(D_2^*)$ y contradice las medidas erróneas del Ejemplo 6.3.1: el esfuerzo interpretativo realizado en las derivaciones D_1 y D_2 (ambas usando la regla $\mathcal{R}_1$), es ligeramente inferior que el realizado en las derivaciones D_1^* y D_2^* (que usan la regla $\mathcal{R}_1^*$), y no al contrario.

La precisión de la nueva medida de coste y de la forma de realizar computaciones interpretativas es crucial al comparar la ejecución de programas obtenidos mediante técnicas de transformación como el marco plegado/desplegado descrito en [30, 15]. En este sentido, en lugar de medir el coste absoluto de las derivaciones realizadas en un programa, interesan más las ganancias/pérdidas de eficiencia relativas producidas en la transformación. Por ejemplo, mediante la aplicación de la llamada *operación de agregación* descrita en [15], podemos transformar la regla $\mathcal{R}_1$ en $\mathcal{R}_1^*$ y, para realizar transformaciones alternativas (plegado, desplegado, etc.), si el programa degenera con respecto al original (como ocurre

en este caso), necesitamos una medida de coste apropiada. Este hecho tiene una importancia capital para descubrir situaciones que pueden aparecer en secuencias de transformación degeneradas, como la generación de definiciones de agregadores muy anidados. Por ejemplo, sea la siguiente definición de conectivos: $@_{100}(x_1, x_2) \triangleq @_{99}(x_1, x_2)$, $@_{99}(x_1, x_2) \triangleq @_{98}(x_1, x_2), \ldots$, y, finalmente, $@_1(x_1, x_2) \triangleq x_1 * x_2$.

Al tratar de resolver dos expresiones de la forma $@_{99}(0.9, 0.8)$ y $@_1(0.9, 0.8)$, las medidas de coste basadas en el número de pasos interpretativos ([32])y los pesos de los pasos interpretativos ([29]) asignarían 1 unidad de coste interpretativo a ambas derivaciones. Afortunadamente, la nueva aproximación descrita aquí puede distinguir claramente entre ambos casos, ya que el número de pasos cortos de tipo 1 realizados en cada uno es bastante distinto (100 y 1, respectivamente).

6.3.1. Opción 'ismode'

La opción "`ismode`" permite elegir entre tres niveles de detalle al visualizar las computaciones interpretativas llevadas a cabo mostradas en "árboles de evaluación". A continuación mostramos un ejemplo de cómo se muestra un árbol de desplegado en este entorno, utilizando el mismo programa que en el Ejemplo 5.3.4, y tomando como objetivo 'p(a)', para cada opción:

Modo 'large'. Esta opción permite omitir la fase interpretativa del árbol, mostrando sólo la f.c.a. de la derivación:

```
>> tree.
R0 < p(X), {} >
    R1 < &prod(0.9,&godel(q(X1),@aver(r(X1),s(X1)))), {X/X1} >
        R2 < &prod(0.9,&godel(0.8,@aver(r(a),s(a)))), {X/a,X1/a} >
            R3 < &prod(0.9,&godel(0.8,@aver(0.7,s(a)))), {X/a,X1/a,X11/a} >
                R4 < &prod(0.9,&godel(0.8,@aver(0.7,0.5))), {X/a,X1/a,X11/a,X16/a} >
                    result < 0.7200000000000001, {X/a,X1/a,X11/a,X16/a} >
```

Modo 'medium'. Con esta opción, en el árbol se realizan pasos interpretativos tal como se establece en la Definición 5.3.5:

```
>> tree.
R0 < p(X), {} >
    R1 < &prod(0.9,&godel(q(X1),@aver(r(X1),s(X1)))), {X/X1} >
        R2 < &prod(0.9,&godel(0.8,@aver(r(a),s(a)))), {X/a,X1/a} >
            R3 < &prod(0.9,&godel(0.8,@aver(0.7,s(a)))), {X/a,X1/a,X11/a} >
                R4 < &prod(0.9,&godel(0.8,@aver(0.7,0.5))), {X/a,X1/a,X11/a,X16/a} >
                    is < &prod(0.9,&godel(0.8,0.85)), {X/a,X1/a,X11/a,X16/a} >
                        is < &prod(0.9,0.8), {X/a,X1/a,X11/a,X16/a} >
                            is < 0.7200000000000001, {X/a,X1/a,X11/a,X16/a} >
```

Modo 'small'. Esta opción muestra el árbol de derivación utilizando los pasos interpretativos cortos, dados por la Definición 6.3.2:

```
>> tree.
R0 < p(X), {} >
    ....
        sis1 < &prod(0.9,&godel(0.8,#prod(#add(0.7,0.5),0.5))), {X/a,X1/a,X11/a,X16/a} >
            sis2 < &prod(0.9,&godel(0.8,#prod(1.2,0.5))), {X/a,X1/a,X11/a,X16/a} >
                sis2 < &prod(0.9,&godel(0.8,0.6)), {X/a,X1/a,X11/a,X16/a} >
                    sis1 < &prod(0.9,#min(0.8,0.6)), {X/a,X1/a,X11/a,X16/a} >
```

```
sis2 < &prod(0.9,0.6), {X/a,X1/a,X11/a,X16/a} >
    sis1 < #prod(0.9,0.6), {X/a,X1/a,X11/a,X16/a} >
        sis2 < 0.54, {X/a,X1/a,X11/a,X16/a} >
```

En la Figura 6.3 se muestra la ejecución de la opción 'tree' tal como se presenta en la interfaz de texto de FLOPER.

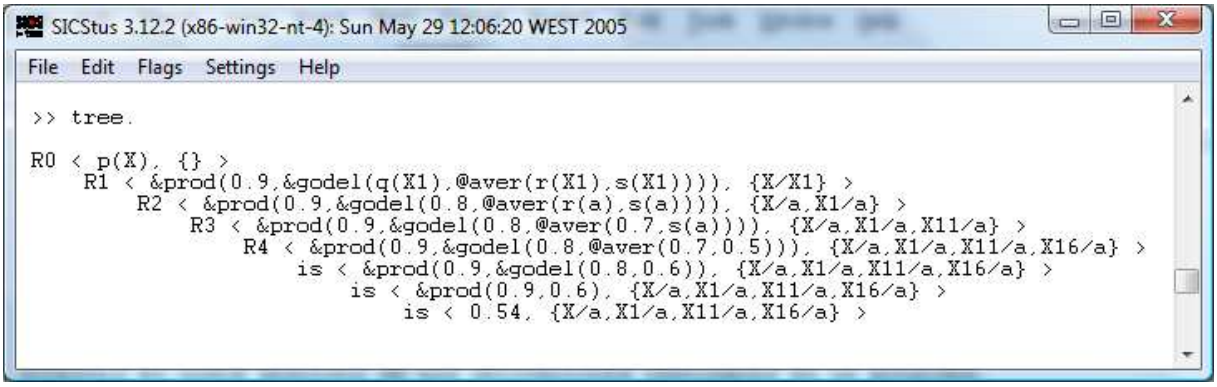

Figura 6.3: Ejemplo de ejecución de la opción 'tree'

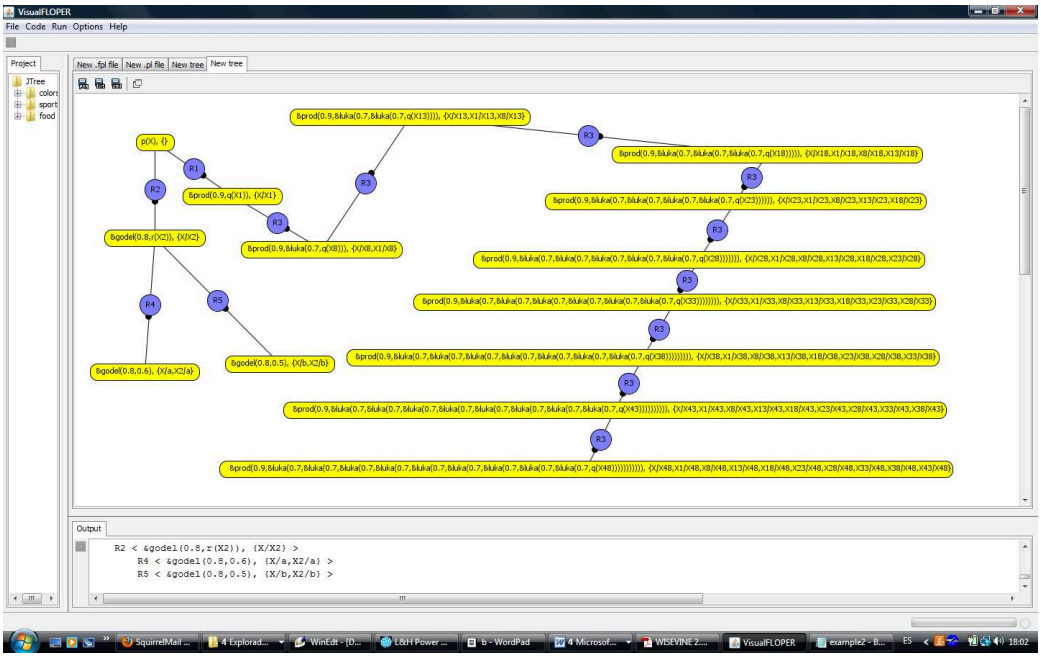

Figura 6.4: Opción 'tree' desde la interfaz gráfica de FLOPER

6.4. Gestión de Retículos

Como se ha detallado en el Capítulo 5, un programa MALP se interpreta durante su ejecución en un retículo multi-adjunto. Se ha utilizado un sencillo método para modelar estos retículos en la herramienta FLOPER. Todos los elementos relevantes de un retículo pueden ser encapsulados en un fichero Prolog, que contenga un conjunto de predicados que definan los elementos del retículo. Estos elementos son:

- `member/1`. Este predicado se satisface cuando se llama con un parámetro que sea un grado de verdad válido para el retículo. En el caso de retículos finitos es recomendable implementar `members/1` para que devuelva una lista de todos los grados de verdad. Por ejemplo, en el caso Booleano, ambos predicados pueden ser modelados con: `member(0). member(1).` y `members([0,1]).`
- `bot/1` y `top/1`. Responden con los elementos mínimo y máximo (respectivamente) del retículo. Para el retículo Booleano serían implementados como `bot(0).` y `top(1).`
- `leq/2`. Modela la relación de orden entre los elementos del retículo. Obviamente, sólo se debería satisfacer cuando se invoca con dos elementos del retículo, siendo el primero menor o igual que el segundo. Para el retículo Booleano, este predicado se implementa con los hechos `leq(0,X).` y `let(X,1).`
- Por Último, dado el conjunto de conectivos del retículo, de la forma $\&_{label_1}$ (conjunción), $\vee_{label_2}$ (disyunción), o $@_{label_3}$ (agregador), con aridades n_1, n_2 y n_3, respectivamente, se deben proveer las cláusulas que definan los predicados conectivos:

  "`and_`*label*$_1$`/(n`$_1$`+1)`", "`or_`*label*$_2$`/(n`$_2$`+1)`" y "`agr_`*label*$_3$`/(n`$_3$`+1)`"

  donde el tercer argumento de cada predicado debe contener el resultado de la evaluación del conectivo. Por ejemplo, en el caso Booleano, la conjunción clásica se puede modelar con estos dos hechos:

  `and_bool(0,_,0). and_bool(1,X,X).`

El retículo Booleano, cuyos elementos han sido detallados, puede residir en un fichero con el nombre "bool.pl". Obsérvese que al trabajar con programas MALP cuyas reglas sigan la forma:

$$\text{“}A \leftarrow_{bool} \&_{bool}(B_1, \ldots, B_n) \ \ with \ \ 1\text{”}$$

siendo A y B_i átomos, engloba correctamente la estructura y el funcionamiento de los programas Prolog clásicos, donde las cláusulas tienen la forma "$A : -B_1, \ldots, B_n$".

Por otro lado, y siguiendo el estilo Prolog comentado previamente, se ha incluido en el fichero "num.pl" las cláusulas mostradas en la Figura 6.5. Aquí se ha modelado un retículo más flexible (que se usará más extensamente en los ejemplos) que permite trabajar con grados de verdad en el espacio infinito (nótese que esta condición impide la implementación del predicado "`members/1`" de los números reales entre 0 y 1, permitiendo así utilizar los operadores de conjunción y disyunción obtenidos de las tres lógicas difusas típicas descritas anteriormente (las lógicas de *Łukasiewicz*, *Gödel* y del *product*), así como la descripción del agregador *average*.

Obsérvese también que se han incluido definiciones para predicados auxiliares, cuyos nombres empiezan siempre con el prefijo "`pri_`". Estos predicados permiten describir operadores aritméticos primitivos (en nuestro caso, $+$, $-$, $*$, $/$, *min* y *max*) en estilo Prolog, siendo apropiadamente llamados por las cláusulas que definen conectivos.

Hasta ahora se han considerado dos retículos totalmente ordenados, guardados en los archivos "bool.pl" y "num.pl". Sin embargo, gracias a la capacidad expresiva que se permite en los ficheros de los retículos, pueden considerarse otros parcialmente ordenados, en los que no siempre se pueden comparar dos elementos cualesquiera. Considérese el retículo multi-adjunto (parcialmente ordenado) de la Figura 6.6, en el que la conjunción y la implicación son las de la lógica intuicionista de *Gödel*, tal que la conjunción de *Gödel* queda expresada como

```
member(X) :- number(X),0=<X,X=<1.  %% no members/1 (infinite lattice)

bot(0).              top(1).             leq(X,Y) :- X=<Y.

and\_luka(X,Y,Z) :- pri_add(X,Y,U1),pri_sub(U1,1,U2),pri_max(0,U2,Z).
and_godel(X,Y,Z):- pri_min(X,Y,Z).
and_prod(X,Y,Z) :- pri_prod(X,Y,Z).

or_luka(X,Y,Z)  :- pri_add(X,Y,U1),pri_min(U1,1,Z).
or_godel(X,Y,Z) :- pri_max(X,Y,Z).
or_prod(X,Y,Z)  :- pri_prod(X,Y,U1),pri_add(X,Y,U2),pri_sub(U2,U1,Z).

agr_aver(X,Y,Z) :- pri_add(X,Y,U),pri_div(U,2,Z).

pri_add(X,Y,Z)  :- Z is X+Y.    pri_min(X,Y,Z) :- (X=<Y,Z=X;X>Y,Z=Y).
pri_sub(X,Y,Z)  :- Z is X-Y.    pri_max(X,Y,Z) :- (X=<Y,Z=Y;X>Y,Z=X).
pri_prod(X,Y,Z) :- Z is X * Y.  pri_div(X,Y,Z) :- Z is X/Y.
```

Figura 6.5: Retículo Multi-adjunto que modela el intervalo real [0,1] ("num.pl").

$$\&_{\mathtt{G}}(x,y) \triangleq inf(x,y)$$

donde debe observarse la sustitución de "min" por "inf".

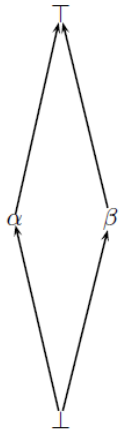

Figura 6.6: Retículo 'four.pl'

```
member(bottom). member(alpha).
member(beta). member(top).
members([bottom,alpha,beta,top]).

bot(bottom). top(top).
leq(bottom,X). leq(X,top). leq(X,X).

and_godel(X,Y,Z) :- pri_inf(X,Y,Z).
pri_inf(bottom,X,bottom):-!.
pri_inf(alpha,X,alpha):-leq(alpha,X),!.
pri_inf(beta,X,beta):-leq(beta,X),!.
pri_inf(top,X,X):-!.
pri_inf(X,Y,bottom).
```

Para la implementación del retículo mostrado en la Figura 6.6, obsérvese el código Prolog que le acompaña, donde se han introducido cinco cláusulas definiendo el nuevo operador primitivo "`pri_inf/3`", que debe devolver el *ínfimo* de un conjunto de dos elementos. Deben tenerse en cuenta los siguientes aspectos:

- Dado que los grados de verdad α y β (o sus correspondientes representaciones como términos Prolog, "`alpha`" y "`beta`") no son comparables, cualquier llamada a objetivos como '`?- leq(alpha,beta).`" o "`?- leq(beta,alpha).`" siempre fallará.
- Un objetivo de la forma "`?- pri_inf(alpha,beta,X).`", o "`?- pri_inf(beta,alpha,X).`", en lugar de fallar, devolverá el resultado correspondiente "`X=bottom`".
- Obsérvese que la implementación del predicado "`pri_inf/1`" es necesaria para implementar la definición dada de "`and_godel/3`".

El retículo de la Figura 6.6 es un ejemplo que muestra la flexibilidad del sistema FLOPER para utilizar retículos.

6.4.1. Opción 'lat'

En cada momento sólo puede haber un retículo multi-adjunto cargado en FLOPER, y todos los programas que se evalúen en el sistema serán interpretados en dicho retículo. La opción 'lat' permite al usuario establecer el retículo multi-adjunto en el que se interpretará su programa. Como se ha adelantado en la Sección 6.4, los retículos multi-adjuntos están definidos en ficheros Prolog. La opción 'lat' pide al usuario que introduzca la ruta al fichero donde está definido el retículo correspondiente:

```
>> lat.
Current lattice: num.pl
Introduce new lattice:
```

FLOPER almacena el retículo como un conjunto de cláusulas Prolog, alojándolas en el módulo 'lat'. De ese modo, al cambiarlo, el sistema puede eliminar el retículo anterior limpiamente borrando todo el contenido del módulo 'lat' y cargando en él el programa Prolog correspondiente al nuevo. Durante la carga se realizan determinados chequeos para corroborar la presencia de los predicados obligatorios ('member', 'bot', 'top' y 'leq'), así como la presencia de 'members', que indicaría que el retículo es finito. Una vez realizado el chequeo, muestra una salida del siguiente estilo:

```
member/1 OK
bot/1 OK
WARNING: top/1 is not defined
leq/2 OK
Infinite lattice
```

En esta salida, FLOPER indica que los predicados 'member', 'bot' y 'leq' están presentes en el retículo, pero que no ha encontrado el predicado 'top'. Tampoco ha encontrado el predicado 'members', por lo que supone que el retículo es infinito. Si hubiera encontrado dicho predicado hubiera mostrado `Finite lattice` en su lugar. En la Figura 6.7 se muestra la ejecución de esta opción desde el interfaz de texto de FLOPER.

6.4.2. Opción 'show'

Mediante la opción 'show', del submenú de retículos, se puede requerir al sistema que muestre por pantalla todas las cláusulas que definen el retículo. Internamente, FLOPER realiza una consulta Prolog 'listing'

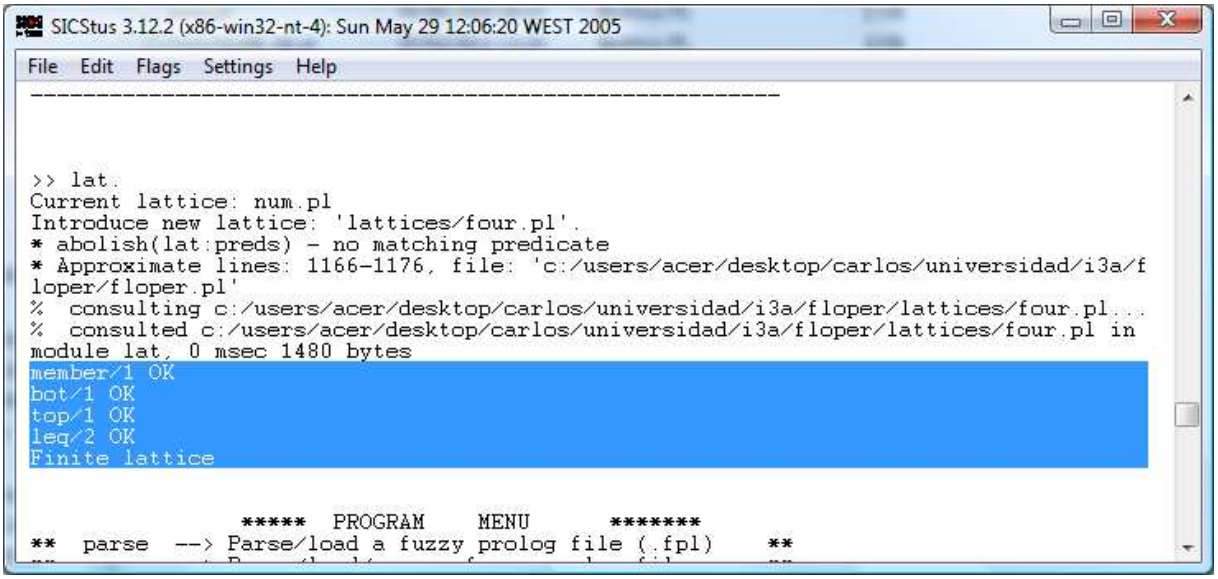

Figura 6.7: Ejemplo de la ejecución de la opción 'lat'

sobre el contenido del módulo 'lat'. Por ello, todas las cláusulas van precedidas por la expresión '`lat:`'.

A continuación se muestra un ejemplo de la ejecución de la opción 'show' tras cargar el retículo 'four', representado en la Figura 6.6.

```
>> show
     .
Current lattice: lattices/four.pl
lat:and_godel(A, B, C) :-
        lat:pri_inf(A, B, C).

lat:bot(bottom).
lat:top(top).

lat:leq(bottom, _).
lat:leq(_, top).
lat:leq(A, A).

lat:member(bottom).
lat:member(alpha).
lat:member(beta).
lat:member(top).

lat:members([bottom,alpha,beta,top]).
lat:preds([p([a,n,d,'_',g,o,d,e,l],[3])]).

lat:pri_inf(bottom, _, bottom) :- !.
lat:pri_inf(alpha, A, alpha) :-
        lat:leq(alpha, A), !.
lat:pri_inf(beta, A, beta) :-
        lat:leq(beta, A), !.
lat:pri_inf(top, A, A) :- !.
lat:pri_inf(_, _, bottom).

lat:testable.
```

6.5. Respuestas Extendidas vía 'lat' y 'run'

En la Sección 6.4 se ha descrito brevemente la forma en la que se modelan los retículos multi-adjuntos para ser cargados en el sistema FLOPER y permitir, así, interpretar sobre ellos los programas multi-adjuntos. Se ha resaltado la flexibilidad de este modelo con el ejemplo de un retículo parcialmente ordenado de cuatro elementos, "`top`", "`alpha`", "`beta`" y "`bottom`". En esta sección vamos un paso más allá, utilizando retículos más complejos con el objetivo de obtener información de la evaluación de un objetivo en la respuesta computada difusa.

6.5.1. Respuestas Computadas Difusas con Información Extendida

El retículo que hemos escogido para extender la información de las respuestas computadas difusas está compuesto de elementos que comtienen dos componentes: un grado de verdad junto a una etiqueta que pueda llevar información sobre las reglas del programa y los conectivos difusos usados en su ejejución.

Para cargarlo en FLOPER, se debe definir en Prolog el nuevo retículo. Sus elementos pueden ser expresados como términos de la forma "`info(Fuzzy_Truth_Degree, Label)`". A continuación se muestra la implementación de dicho retículo:

```
member(info(X,_)):-number(X),0=<X,X=<1.

bot(info(0,'')).
top(info(1,'')).

leq(info(X1,_),info(X2,_)) :- X1 =< X2.

and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
             pri_prod(X1,Y1,Z1,DatPROD),pri_app(X2,Y2,Dat1),
             pri_app(Dat1,'&PROD.',Dat2),pri_app(Dat2,DatPROD,Z2).

pri_app(X,Y,Z):-name(X,L1),name(Y,L2),append(L1,L2,L3),name(Z,L3).

pri_append([],X,X).
append([A|B],C,[A|D]):-append(B,C,D).
```

Obsérvese que al implementar (por ejemplo) el operador conjunción de la lógica del producto, en el segundo componente de la noción extendida de grado de verdad, se ha incluido la etiqueta `'&PROD.'` junto a las etiquetas de los argumentos, indicando que se ha dado un paso interpretativo sobre la conjunción del producto. Por supuesto, en el programa difuso se debe tener en cuenta el uso de etiquetas asociadas a las reglas del programa. Véase la modificación que debe acometerse en las reglas del programa del Ejemplo 5.3.4:

```
p(X) <prod &godel(q(X), @aver(r(Y), s(X))) with info(0.9,'RULE1,').
q(a) with info(0.8,'RULE2,').
r(X) with info(0.7,'RULE3,').
s(X) with info(0.5,'RULE4,').
```

Al ejecutar el objetivo `p(X)` mediante la opción 'run' se obtiene:

```
>> run.
[Truth_degree=info(0.54,RULE1,RULE2,RULE3,RULE4,@AVER,&GODEL,&PROD,),X=a]
```

A continuación emplearemos un programa $\mathcal{P}_2$ diferente en el que el objetivo `p(X)` tiene dos soluciones.

```
p(X) <prod q(X,Y) &godel r(Y) with info(0.8,'RULE1,').
q(a,Y) <prod s(Y) with info(0.7,'RULE2,').
q(b,Y) <luka r(Y) with info(0.8,'RULE3,').
r(Y) with info(0.7,'RULE4,').
s(b) with info(0.9,'RULE5,').
```

Tras ejecutar el objetivo mencionado, se obtienen las dos respuestas computadas, que incluyen la secuencia de reglas del programa explotadas y los conectivos evaluados :

```
[ Truth_degree=info(0.504,   RULE1.RULE2.RULE5.&PROD.
                             RULE4.&GODEL.&PROD.}),        X=a]

[ Truth_degree=info(0.4,     RULE1.RULE3.RULE4.&LUKA.
                             RULE4.&GODEL.&PROD.),         X=b]
```

Puede verse en la Figura 6.8 este resultado tal como se muestra en la interfaz de texto de FLOPER, donde se han resaltado en azul las líneas pertinentes.

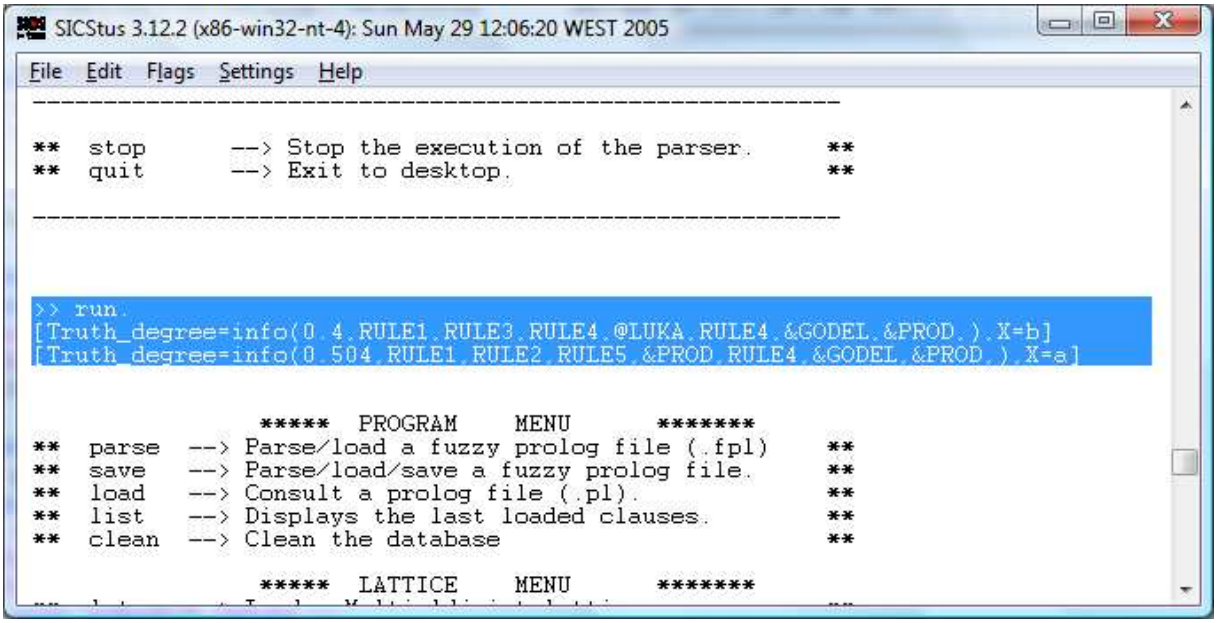

Figura 6.8: Ejemplo de ejecución de $\mathcal{P}_2$

6.5.2. Trazas Declarativas usando FLOPER

Como ya se ha dicho, y se detalla en [2, 49, 52, 56, 57], el traductor ha sido implementado mediante las DCG's (*Definite Clause Grammars*) del lenguaje Prolog, ya que es una notación útil para representar reglas gramaticales. Una vez que se carga la aplicación en el intérprete de Prolog (en nuestro caso, Sicstus Prolog v.3.12.5), muestra un menú que incluye opciones para cargar, traducir, listar y guardar programas difusos, así como para ejecutar objetivos difusos.

Estas acciones están basadas en la traducción del código difuso a código Prolog estándar. La clave de la traducción consiste en extender cada átomo con un argumento extra, denominado *variable de verdad*,

de la forma "$\texttt{\_TV}_\texttt{i}$", para contener el grado de verdad obtenido tras la subsecuente evaluación del átomo. Por ejemplo, la primera cláusula de nuestro programa se traduce como

```
p(X,_TV0)  :-  q(X,_TV1),
               r(X,_TV2),
               s(X,_TV3),
               agr_aver(_TV2,_TV3,_TV4),
               and_godel(_TV1,_TV4,_TV5),
               and_prod(0.9,_TV5,_TV0).
```

Más aún, la segunda cláusula de nuestro programa en el Ejemplo 5.3.4, pasa a ser el hecho Prolog "`q(a,0.8)`", mientras que un objetivo difuso como "`p(X)`", es traducido al objetivo Prolog puro: "`p(X,Truth_degree)`" (obsérvese que la última variable de grado de verdad no es anónima ahora) por lo que el intérprete de Prolog devuelve la respuesta computada difusa deseada [$\texttt{Truth\_degree} = 0.54, \texttt{X} = \texttt{a}$].

El conjunto anterior de opciones basta para ejecutar programas difusos (la opción "`run`" también usa las cláusulas contenidas en el fichero "num.pl", que representa el retículo por defecto): todas las computaciones internas (incluyendo la compilación y ejecución) son derivaciones puras de Prolog donde las entradas (programas difusos y objetivos) y las salidas (respuestas computadas difusas tienen siempre un sabor difuso, produciendo así al usuario final la ilusión de estar trabajando con una herramienta puramente difusa.

Por otro lado, en [52, 57] detallamos que hemos equipado recientemente a FLOPER con nuevas opciones, denominadas "`lat`" y "`show`", para permitir al usuario cambiar y mostrar el retículo multi-adjunto asociado a un programa dado. Supóngase que "new_num.pl" contiene el mismo código Prolog que "num.pl" con la excepción de la definición relativa al operador de la media. Ahora, en lugar de computar la media de dos grados de verdad, la nueva versión computa la media entre los resultados obtenidos después de aplicar a ambos elementos las disyunciones descritas por Gödel y Łukasiewicz, es decir: $@_{\texttt{aver}}(x_1, x_2) \triangleq (\vee_{\texttt{G}}(x_1, x_2) + \vee_{\texttt{L}}(x_1, x_2)) * 0.5$.

La cláusula Prolog correspondiente, definida en "new_num.pl" puede ser

```
agr_aver(X,Y,Z)  :-  or_godel(X,Y,Z1),
                     or_luka(X,Y,Z2),
                     pri_add(Z1,Z2,Z3),
                     pri_prod(Z3,0.5,Z).
```

y ahora, seleccionando de nuevo la opción "`run`" (sin cambiar ni el programa ni el objetivo), el sistema debería mostrar la nueva respuesta computada difusa: [$\texttt{Truth\_degree} = 0.72, \texttt{X} = \texttt{a}$].

Consideremos ahora el denominado *dominio de cualificación* $(\mathcal{W}, \leq)$ $\mathcal{W}$ usado en el marco QLP (*Qualified Logic Programming*) de [63], cuyos elementos pretenden representar costes de pruebas, medidas como la profundidad ponderada de árboles de prueba (aunque próximo a MALP, el esquema QLP permite un menor repertorio de conectivos en el cuerpo de las reglas del programa). A efectos de estimar los pasos de computación ejecutados, puede tomarse $\mathcal{W} = \mathbb{N} \cup \{\infty\}$, con el orden $\leq$ el invertido del usual (de modo que $\infty = inf(\mathcal{W})$ y $0 = sup(\mathcal{W})$.

Usando de nuevo la opción "`lat`" de FLOPER podemos asociar este retículo $\mathcal{W}$ al programa anterior después de cambiar los pesos de cada regla del programa a 1 (la idea subyacente es que el uso de cada regla del programa en una derivación implica la aplicación de un paso admisible). Más aún, ya que en este retículo la operación aritmética "+" define una conjunción (*t-norma*), usamos la definición del conjunto de conectivos que aparecen en el programa *mapeado* a "+" (es decir, $\&_{\texttt{P}}(x, y) \triangleq x + y$, $\&_{\texttt{G}}(x, y) \triangleq x + y$ y $@_{\texttt{aver}}(x, y) \triangleq x + y$). Ahora, para el objetivo "$p(X)$" podemos generar una derivación admisible similar a la vista en el Ejemplo 5.3.4, pero terminando con $\langle \&_{\texttt{P}}(1, \&_{\texttt{G}}(1, @_{\texttt{aver}}(1, 1))); \{X/a\}\rangle$ Y dado que $\&_{\texttt{P}}(1, \&_{\texttt{G}}(1, @_{\texttt{aver}}(1, 1))) = +(1, +(1, +(1, 1))) = 4$, la respuesta computada difusa, o f.c.a.,

$\langle 4; \{X/a\}\rangle$ indica que el objetivo "$p(X)$" se satisface cuando X es a, tras aplicar 4 pasos admisibles, como esperábamos.

Además, podemos concebir un retículo más interesante expresado como el *producto cartesiano*. Ahora, cada regla del programa tiene asociado un peso que es un par con dos componentes, un grado de verdad y una medida de coste. Para ser cargado en FLOPER, debemos definir en Prolog el nuevo retículo, cuyos elementos pueden ser expresados, por ejemplo, como términos de la forma

```
info(Fuzzy_Truth_Degree,Cost_Number_Steps)
```

Las cláusulas que definen los predicados requeridos para manipularlos son:

```
member(info(X,Y)) :- number(X), 0=<X, X=<1, number(Y), Y=<0.

leq(info(X1,Y1),info(X2,Y2)) :- X1=<X2, Y1>=Y2.
top(info(1,0)).
bot(info(0,inf))

and_godel(info(X1,Y1),info(X2,Y2),info(X3,Y3)) :-  pri_min(X1,X2,X3),
                                                   pri_add(Y1,Y2,Y3).
```

Finalmente, si los pesos asignados a las reglas de nuestro ejempmlo ejecutable son "`info(0.9,1)`" para $\mathcal{R}_1$, "`info(0.8,1)`" para $\mathcal{R}_2$, "`info(0.7,1)`" para $\mathcal{R}_3$ y "`info(0.5,1)`" para $\mathcal{R}_4$, entonces, para el objetivo "$p(X)$" podemos obtener la f.c.a. $\langle \texttt{info}(0.54, 4); \{X/a\}\rangle$ con el significado obvio de que necesitamos 4 pasos admisibles para probar que la consulta se satisface con un grado de verdad 0.56 grado de verdad cuando X cambia por a.

Un paso mmás consiste en diseñar un retículo todavía más complejo que permita asociar a las reglas del programa trazas declarativas. Sus elementos deben tener dos componentes, incluyendo grados de verdad y "etiquetas" que recogen información sobre las reglas del programa, conectivos difusos y operadores primitivos usados en la ejecución. Para cargar el nuevo retículo en FLOPER debemos definirlo de nuevo como un programa Prolog, cuyos elementos serán expresados ahora como términos de la forma "`info(Fuzzy_Truth_Degree, Label)`", como se muestra en la Figura 6.9 (obsérvese que la versión compleja del *conectivo media* es llamado aquí `agr_aver2` e invoca la versión simple de `agr_aver`).

Vemos aquí que al implementar, por ejemplo, el operador de conjunción de la lógica del Producto, en el segundo componente de nuestra noción extendida de grado de verdad, hemos unido las etiquetas de sus argumentos con la etiqueta `'&PROD'`. Por supuesto, en el programa difuso debemos tener en cuenta el uso de las etiquetas asociadas a las reglas del programa. Por ejemplo, el conjunto de reglas de nuestro ejemplo (donde usamos la versión compleja de la media, es decir, `@aver2` en la primera regla) deben tener la forma

```
p(X) <prod &godel(q(X),@aver2(r(X),s(X))) with info(0.9,'RULE1.').
q(a) with   info(0.8,'RULE2.').
r(X) with   info(0.7,'RULE3.').
s(X) with   info(0.5,'RULE4.').
```

Ahora, el lector puede comprobar fácilmente que, después de ejecutar el objetivo `p(X)`, obtenemos las respuestas computadas difusas deseadas que incluyen la traza declarativa respecto a reglas del programa, llamadas a conectivos y operadores primitivos evaluados hasta encontrar la respuesta computada difusa:

```
>> run.

[Truth_degree=info(0.72,  RULE1.RULE2.RULE3.RULE4.
```

```
                    @AVER2.|GODEL.#MAX.|LUKA.

                    #ADD.#MIN.@AVER.#ADD.#DIV.

                    &GODEL.#MIN.&PROD.#PROD.),

                    X=a]
```

En esta respuesta computada difusa obtenemos el grado de verdad 0.72 y la sustitución $\{X/a\}$ asociada a nuestro objetivo. También obtenemos la secuencia de reglas de programa explotadas al ejecutar los pasos admisibles, así como los propios conectivos difusos evaluados durante la fase interpretativa, y el conjunto de operadores primitivos (de la forma #*label*) invocados.

```
member(info(X,_)):-number(X),0=<X,X=<1.

bot(info(0,_)).

top(info(1,_)).

leq(info(X1,_),info(X2,_)):- X1 =< X2.

and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)) :-

                    pri_prod(X1,Y1,Z1,DatPROD),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'&PROD.',Dat2),pri_app(Dat2,DatPROD,Z2).


or_godel(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-

                    pri_max(X1,Y1,Z1,DatMAX),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'|GODEL.',Dat2),pri_app(Dat2,DatMAX,Z2).


or_luka(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-

                    pri_add(X1,Y1,U1,DatADD),pri_min(U1,1,Z1,DatMIN),
                    pri_app(X2,Y2,Dat1),pri_app(Dat1,'|LUKA.',Dat2),
                    pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatMIN,Z2).


agr_aver(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-

                    pri_add(X1,Y1,Aux,DatADD),pri_div(Aux,2,Z1,DatDIV),
                    pri_app(X2,Y2,Dat1),pri_app(Dat1,'@AVER.',Dat2),
                    pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatDIV,Z2).


agr_aver2(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-

                    or_godel(info(X1,''),info(Y1,''),Za),
                    or_luka(info(X1,''),info(Y1,''),Zb),
                    agr_aver(Za,Zb,info(Z1,Dat3)),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'@AVER2.',Dat2),pri_app(Dat2,Dat3,Z2).
```

```
and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)) :-
                    pri_prod(X1,Y1,Z1,DatPROD),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'&PROD.',Dat2),pri_app(Dat2,DatPROD,Z2).


or_godel(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                    pri_max(X1,Y1,Z1,DatMAX),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'|GODEL.',Dat2),pri_app(Dat2,DatMAX,Z2).


or_luka(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                    pri_add(X1,Y1,U1,DatADD),pri_min(U1,1,Z1,DatMIN),
                    pri_app(X2,Y2,Dat1),pri_app(Dat1,'|LUKA.',Dat2),
                    pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatMIN,Z2).


agr_aver(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                    pri_add(X1,Y1,Aux,DatADD),pri_div(Aux,2,Z1,DatDIV),
                    pri_app(X2,Y2,Dat1),pri_app(Dat1,'@AVER.',Dat2),
                    pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatDIV,Z2).


agr_aver2(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                    or_godel(info(X1,''),info(Y1,''),Za),
                    or_luka(info(X1,''),info(Y1,''),Zb),
                    agr_aver(Za,Zb,info(Z1,Dat3)),pri_app(X2,Y2,Dat1),
                    pri_app(Dat1,'@AVER2.',Dat2),pri_app(Dat2,Dat3,Z2).


pri_add(X,Y,Z,'#ADD.')  :- Z is X+Y.

pri_sub(X,Y,Z,'#SUB.') :- Z is X-Y.

pri_prod(X,Y,Z,'#PROD.'):- Z is X * Y.

pri_div(X,Y,Z,'#DIV.') :- Z is X/Y.

pri_min(X,Y,Z,'#MIN.')  :- (X=<Y,Z=X;X>Y,Z=Y).

pri_max(X,Y,Z,'#MAX.')  :- (X=<Y,Z=Y;X>Y,Z=X).

pri_app(X,Y,Z)          :- name(X,L1),name(Y,L2),append(L1,L2,L3),name(Z,L3).

append([],X,X).

append([X|Xs],Y,[X|Zs]):-append(Xs,Y,Zs).
```

Figura 6.9: Retículo multi-adjunto que modela grados de verdad con etiquetas.

Capítulo 7

Conclusiones y trabajos futuros

La hipótesis de partida de este trabajo se basa en que la lógica difusa supone un marco teórico, formal y matemáticamente muy desarrollado, que ya se ha aplicado con notable éxito en distintas facetas que mejoran nuestra vida cotidiana, desde la electrónica de consumo y el control de procesos industriales, al desarrollo de aplicaciones software muy flexibles y potentes, con capacidad de adaptación a contextos en los que prima la presencia de la vaguedad. Si bien estos productos finales se han beneficiado de investigaciones precedentes en en ámbito teórico, no ha ocurrido lo mismo con los lenguajes y herramientas de programación que se han utilizado como medio para desarrollarlos, lo que constituye una evidente paradoja. Por otra parte, creemos que los lenguajes declarativos tienen una sintaxis y una semántica simples, lo que facilita la tarea de escribir, manipular y razonar sobre los programas. Por consiguiente, surge la necesidad de diseñar lenguajes, herramientas y entornos declarativos basados directamente en la lógica borrosa, para amplificar su impacto y apoyar el desarrollo racional y sistemático de tales aplicaciones.

A lo largo de este Máster hemos consolidado y mejorado de forma notable el diseño, implementación y aplicación del entorno de programación lógica difusa FLOPER, cuyo desarrollo actual y futuro se enmarca dentro del proyecto ALDDEIA (financiado por el Ministerio de Ciencia e Innovación con referencia TIN 2007-65749) y que se adecúa así a las prioridades del Programa Nacional de Tecnologías Informáticas (TIN), especialmente en lo concerniente a la prioridad temática 2 sobre "tecnologías de soporte y desarrollo de software". Por todo lo expuesto en esta memoria, es fácil comprobar que nuestra propuesta se ajusta textualmente a "el estudio de los lenguajes de programación, como el estudio de la lógica subyacente, se relaciona con la potencia expresiva de las notaciones formales, con correspondencia directa entre la sintaxis (programas) y la semántica (lo que significan), y con los medios mediante los cuales se puede analizar (manual o automáticamente) el texto en un lenguaje formal para extraer conclusiones. En consonancia con el objetivo general del Programa Nacional de Tecnologías Informáticas "desarrollo de tecnologías software imprescindibles para la sociedad de la información del mañana: nuevos métodos, técnicas y herramientas, nuevas tecnologías, nuevas plataformas, [...] y posibilitar la construcción de un software de nueva generación", nosotros hemos dado un paso más allá en esta dirección en torno al distintivo *fuzzy*, cuyas garantías de novedad y relevancia hemos detallado.

En este escenario donde se necesitan prioritariamente nuevos lenguajes, teorías, entornos de programación, técnicas formales y herramientas automáticas asociadas, que mejoren la productividad y den soporte sistemático y racional al desarrollo del software, con este trabajo intentamos, en nuestro grupo, paliar estas carencias, mediante la construcción de entornos avanzados que den soporte al desarrollo de aplicaciones basadas en lógica difusa, como es el caso de FLOPER. Una primera versión aparece descrita tanto en [50, 46], y la herramienta está dispoinible en la página web habilitada al efecto, "http://dectau.uclm.es/floper/", desde la cual, además, puede descargarse libremente.

Centrándonos en las aportaciones realizadas en este Máster, a día de hoy el sistema permite la traducción (mediante dos técnicas diferentes) de programas lógicos multi-adjuntos a código Prolog estándar [1, 2]. Con este proceso de compilación ya es posible ejecutar (un subconjunto de) programas lógicos

multi-adjuntos e incluso depurar y evaluar parcialmente objetivos mediante la construcción de árboles de desplegado.

Además, hemos reformulado la fase interpretativa de la semántica procedural permitiendo, por un lado, alcanzar un grado de rigor formal comparable al de la fase operacional y, por otro, facilitar la estimación de su coste computacional. Para solventar el problema se ha modelado a bajo nivel de la noción de paso interpretativo corto, replicando los diferentes casos de los pasos admisibles pero en un contexto interpretativo. En este sentido, se utilizan dos tipos de paso interpretativo corto: el primero es capaz de expandir las definiciones de los conectivos que aparecen en un objetivo de partida sobre el objetivo siguiente. El segundo se limita a efectuar operaciones aritméticas sobre los operadores primitivos que aparecen en un objetivo. La combinación de ambos tipos de pasos interpretativos permite observar con total precisión el comportamiento interno (con fuertes repercusiones a la hora de estimar su coste computacional) de la fase interpretativa y visualizar su evolución a lo largo de una derivación. Para modelar la noción de paso interpretativo corto, se ha partido de la primera versión, presentada en [32, 33], donde los pasos interpretativos se definen en términos de un rígido sistema de transición de estados que no contempla la expansión de las definiciones de las conectivas sobre los nuevos estados. El nuevo concepto induce una nueva medida de coste computacional mucho más precisa que la inicialmente descrita en [25, 29]. Pensamos que la nueva aproximación puede tener importantes aplicaciones en los cálculos de ganancia de eficiencia asociados a las transformaciones que estudiamos en nuestro grupo.

Otra de las aportaciones más relevantes de este trabajo ha consistido en la definición de retículos multi-adjuntos para poder ser cargados y manipulados por la herramienta FLOPER. Para ello, se ha especificado un retícculo como un fichero Prolog que contiene unos determinados predicados que definen el conjunto de datos, la relación de orden, el ínfimo, el supremo y cláusulas que definan los conectivos del retículo. Se han implementado diversos retículos, como el intervalo real $[0, 1]$ y el retículo "four". La aportación más ddestacable a este respecto consiste en el modelado de retículos que permiten obtener información sobre la ejecución de un objetivo ([53, 55]).

En cuanto a desarrollos futuros, son muchos los avances que se pueden llevar a cabo sobre esta plataforma si realmente queremos confeccionar lenguajes y herramientas prácticas de corte declarativo cuya eficiencia sea comparable a la de los lenguajes imperativos (tal y como lo han conseguido el lenguaje funcional Haskell o el lenguaje lógico Prolog), siendo éste un requisito indispensable para hacer posible su uso en entornos industriales de desarrollo de aplicaciones. Algunas de estas extensiones futuras son:

1. A nivel de codificación, todavía es necesario calibrar las estructuras de datos que permitan traducir programas difusos a Prolog. Las experiencias adquiridas en [2, 51, 49] confirman las bondades de una doble generación de código basada por un lado en texto Prolog directamente ejecutable (lo que permitirá resolver objetivos difusos desde un intérprete PROLOG de forma completamente transparente al usuario) y por otro en una representación de más bajo nivel que permita la generación de árboles parciales (con sus capacidades depurativas inmediatas).

2. La implantación de interfaces "amigables" que permitan interactuar con la herramienta de la forma lo más cómoda posible, es un aspecto que debe abordarse cuanto antes.

3. Permitir trabajar con programas que incorporen características de la programación lógico-funcional, o que admitan otras nociones alternativas de *difuminación* como es el caso de las relaciones de similaridad.

4. Evitar la traducción de código difuso por compilación a Prolog mediante la ejecución/interpretación directa de objetivos o incluso plantear la generación de código para una máquina virtual basadas en alguna extensión del diseño de la WAM, que constituye un estándar para la implementación eficiente de lenguajes lógicos.

5. Gran parte del material investigador que el grupo DEC-TAU ha generado en los últimos años admite su implantación futura sobre FLOPER, como es el caso de las reglas de transformación de plegado/desplegado para programas lógicos multi-adjuntos descritas en [30, 33, 15, 16], las medidas de coste para estimar la eficiencia de los cómputos (tanto operacionales como interpretativos)

propuestas en [25, 29, 47] y la semántica operacional por tabulación para programas lógicos multi-adjuntos que se trata en [22, 23]. Todas estas actuaciones podrían llevarse a cabo añadiendo nuevos menús a partir de la opción de generación de árboles de desplegado implementada sobre FLOPER.

Pese al camino recorrido hasta la fecha, todavía queda mucho por hacer en cuanto al diseño, implementación y estandarización de este tipo de lenguajes, lo que abre varias líneas de investigación entre las que podría situarse la futura tesis doctoral del autor de esta memoria. Si bien el marco de programación lógica multi-adjunta tiene, sintáctica y operacionalmente, un diseño casi impecable, nosotros hemos encontrado ciertas lagunas o problemas cuya resolución podría abordarse como sigue:

- A nivel semántico, todos estos lenguajes proponen una extensión borrosa del modelo mínimo de Herbrand [41, 31], como significado declarativo de los programas, acompañado muchas veces de otra semántica de punto fijo equivalente. Nosotros nos planteamos superar esta barrera mediante la definición de semánticas más ricas (por ejemplo, en la línea de la semántica de respuestas computadas, o s-semánticas [8, 9, 7]). En el caso concreto de los lenguajes de programación lógica multi-adjunta queremos definir semánticas por desplegado más potentes que las basadas en la obtención del conjunto de éxitos básicos, cuyo desarrollo se beneficiará de nuestra experiencia en la definición de reglas difusas de desplegado, documentada en [34, 35, 30, 32, 33]. Además, creemos que es factible diseñar un nuevo operador de consecuencias inmediatas cuya iteración continuada devuelva una semántica por punto fijo equivalente a la semántica obtenida por desplegados continuados de un programa.
- Es bien sabido que la aproximación multi-adjunta no disfruta de resultados generales de completitud, y que esta importante propiedad tan sólo se puede recuperar mediante la confección de reglas específicas llamadas reductantes para todos y cada uno de los objetivos que se pretenden evaluar. Además, cada reductante sólo es válido para un único objetivo, que debe ser básico (ground), es decir, sin variables, siendo el conjunto de objetivos básicos normalmente infinito, lo que no sólo hace inviable está noción a nivel práctico, sino que incluso la compromete seriamente a nivel teórico. En cuanto a criterios de eficiencia, no sólo la construcción a la manera clásica de los propios reductantes, sino su uso en tiempo de ejecución, degrada drásticamente el tamaño de los programas y la eficiencia de las computaciones. En [28, 24, 26, 27] hemos experimentado con la definición de reductantes más eficientes construidos en torno a técnicas de evaluación parcial umbralizada. Conocer las propiedades formales de esos nuevos reductantes es de sumo interés. En cualquier caso, superar también en este contexto el caso básico supone un importante desafío que queremos abordar en el futuro.

Bibliografía

[1] J.M. Abietar, P.J. Morcillo, y G. Moreno. Building a Fuzzy Logic Programming Tool. In E. Pimentel, editor, Proc. of VII Jornadas sobre Programación y Lenguajes (PROLE'07), Zaragoza, September 12-14, pages 215-222. Thomson-Paraninfo, 2007.

[2] J.M. Abietar, P.J. Morcillo and G. Moreno. Designing a software tool for fuzzy logic programming. In T.E. Simos and G. Maroulis, editors, Proc. of the International Conference of Computational Methods in Sciences and Engineering (ICCMSE'07), Vol. 2 (Computation in Modern Science and Engineering), pages 1117-1120. American Institute of Physics (distributed by Springer), 2007.

[3] T. Alsinet and L. Godó. Fuzzy Unification Degree. In Proc. of II International Workshop on Logic Programming and Soft Computing, in conjunction with 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98), Manchester, pages 23-43, 1998.

[4] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics. Elsevier, Amsterdam. The MIT Press, Cambridge, Mass., 1990.

[5] F. Arcelli and F. Formato. Likelog: A logic programming language for flexible data retrieval. In Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, pages 260-267. ACM, Artificial Intelligence and Computational Logic, 1999. Electronic Edition (DOI: 10.1145/ 298151.298348).

[6] J. F. Baldwin, T. P. Martin and B. W. Pilsworth. Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence. John Wiley and Sons, Inc., 1995.

[7] A. Bossi, M. Gabbrielli, G. Levi and M. Martelli. The s-semantics approach: Theory and applications. Journal of Logic Programming, 19-20:149-197, 1994.

[8] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. Theoretical Computer Science, 69(3):289-318, 1989.

[9] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. Information and Computation, 103(1):86-113, 1993.

[10] F. A. Fontana and F. Formato. A similarity-based resolution rule. International Journal of Intelligent Systems, 17(9):853-872, 2002.

[11] F. A. Fontana and F. Formato. Likelog: A logic programming language for flexible data retrieval. In Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, pages 260-267, 1999.

[12] F. Formato, G. Gerla and M. I. Sessa. Extension of logic programming by similarity. In M. C. Meo and M. V. Ferro, editors, Joint Conference on Declarative Programming (APPIA-GULP-PRODE), pages 397-410, 1999.

[13] F. Formato, G. Gerla and M. I. Sessa. Similarity-based unification. Fundamentos de Informática, 41(4):393-414, 2000.

[14] S. Guadarrama, S. Muñoz and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. Fuzzy Sets and Systems, Elsevier, 144(1):127-150, 2004.

[15] J. Guerrero and G. Moreno. Optimizing fuzzy logic programs by unfolding, aggregation and folding. Electronic Notes in Theoretical Computer Science, vol. 219, pages 19-34, 2008.

[16] J.A. Guerrero and G. Moreno. Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding. In J. Visser and V. Winter, editors, Electronic Notes in Theoretical Computer Science, pages 19-34. Elsevier, 2008.

[17] J.A. Guerrero, G. Moreno and C. Vázquez. Una implementación de Lambda-Cálculo en Prolog. Actas de las X Jornadas sobre Programación y Lenguajes (PROLE'10), editores: V. M. Gulías, J. Silva y A. Villanueva. Páginas: 7-14 (sección de trabajos asociados al II Taller de Programación Funcional). Editorial: Garceta grupo editorial. ISBN: 978-84- 92812-55-4. 2010.

[18] P. Hajek. Fuzzy logic. In E. N. Zalta, editor, The Stanford Encyclopedia of Philosophy (Summer 2008 Edition). The MRL and the CSLI, Stanford University, 2006.

[19] P. Hajek. Metamathematics of fuzzy logic. Kluwer. Dordrecht, 1998.

[20] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, Proc. of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85). Los Angeles, pages 701-703. Morgan Kaufmann, 1985.

[21] P. Julián and M. Alpuente. Programación Lógica. Teoría y Práctica. Pearson Prentice Hall, Madrid, 2007.

[22] P. Julián, J. Medina, G. Moreno and M. Ojeda. Combining tabulation and thresholding techniques for executing multi-adjoint logic programs. In L. Magdalena, M. Ojeda-Aciego and J.L. Verdegay, editors, In Proc. of the 12th International Conference on Information Processing and Management of Uncertainty in Knoledge-based Systems (IPMU'08), June 22-27, 2008, Málaga, pages 550-512. University of Málaga, 2008.

[23] P. Julián, J. Medina, G. Moreno and M. Ojeda. Thresholded Tabulation in a Fuzzy Logic Setting. In J. Almendros, editor, Proc. of VIII Jornadas sobre Programación y Lenguajes (PROLE'08), Gijón, October 7-10, page 15. Thomson-Paraninfo, 2007.

[24] P. Julián, G. Moreno and J. Penabad. An Improved Reductant Calculus Using Partial Evaluation Thecniques. Fuzzy Sets and Systems, Vol. 160, Elsevier, pages 162-181, 2009.

[25] P. Julián, G. Moreno and J. Penabad. Cost Measures for Fuzzy Logic Computations. In E. Pimentel, editor, Proc. of VII Jornadas sobre Programación y Lenguajes (PROLE'07), Zaragoza, September 12-14, pages 103-110. Thomson-Paraninfo, 2007.

[26] P. Julián, G. Moreno and J. Penabad. Eficient Reductants Calculi using Partial Evaluation Techniques with Thresholding. In P. Lucio, editor, Actas de las VI Jornadas sobre Programación y Lenguajes (PROLE'06), Sitges, Octubre 10-12, pages 275-289. Universidad Politécnica de Cataluña, 2006.

[27] P. Julián, G. Moreno and J. Penabad. Efficient reductants calculi using partial evaluation techniques with thresholding. In P. Lucio, editor, Electronic Notes in Theoretical Computer Science, vol. 188, pages 77-90. Elsevier, 2007.

[28] P. Julián, G. Moreno and J. Penabad. Evaluación Parcial de Programas Lógicos Multi-adjuntos y Aplicaciones. In A. Fernández, editor, Proc. of Campus Multidisciplinar en Percepción e Inteligencia (CMPI'06), Albacete, July 10-14, pages 712-724. Universidad de Castilla-La Mancha, 2006.

[29] P. Julián, G. Moreno and J. Penabad, Measuring the interpretive cost in fuzzy logic computations. In Proc. of Applications of Fuzzy Sets Theory, 7th International Workshop on Fuzzy Logic and Applications (WILF'07), Camogli, July 7-10, F. Masulli, S. Mitra and G. Pasi, editors,Springer Verlag, LNAI 4578, 2007, pages 28-36.

[30] P. Julián, G. Moreno and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach, Fuzzy Sets and Systems. vol. 154, pages 16-33, 2005.

[31] P. Julián, G. Moreno and J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In J. Cabestany, F. Sandoval, A. Prieto and J. M. Corchado, editors, Proc. of the 10th InternationalWork-Conference on Artificial Neural Networks (IWANN'09), Vol. 5517. Salamanca, Junio 10-12, pages 253-260. Springer Verlag, 2009.

[32] P. Julián, G. Moreno and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. In F. López-Fraguas, editor, Proc. of V Jornadas sobre Programación y Lenguajes (PROLE'05), Granada, September 14-16, pages 239-248. Universidad de Granada, 2005.

[33] P. Julián, G. Moreno and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. Journal of Universal Computer Science, 12:1679-1699, 2006.

[34] P. Julián, G. Moreno and J. Penabad. Unfolding Fuzzy Logic Programs. In Proc. of the 4th International Conference on Intelligent Systems Design and Applications (ISDA'04), Sponsored by IEEE. Budapest, August 26-28, pages 595-600, 2004.

[35] P. Julián, G. Moreno and J. Penabad. Unfolding-based Improvements on Fuzzy Logic Programs. In Salvador Lucas, editor, Electronic Notes in Theoretical Computer Science, vol. 137, pages 69-103. Elsevier, 2005.

[36] F. Klawonn and R. Kruse. A Łukasiewicz logic based Prolog. Mathware and Soft Computing, 1(1):5-29, 1994.

[37] G. J. Klir and B. Yuan. Fuzzy Sets and Fuzzy Logic. Prentice-Hall, 1995.

[38] R. Kowalski. Problems and Promises of Computational Logic. In J.W. Lloyd, editor, Computational Logic. Springer Verlag, Berlin, 1990.

[39] R.C.T. Lee. Fuzzy Logic and the Resolution Principle. Journal of the ACM, 19(1):119-129, 1972.

[40] D. Li and D. Liu. A fuzzy Prolog database system. John Wiley and Sons, Inc., 1990.

[41] J.W. Lloyd. Foundations of Logic Programming. Springer Verlag, Berlin, 1987. Second edition.

[42] V. Loia, S. Senatore and M. I. Sessa. Similarity-based SLD resolution and its implementation in an extended prolog system. In FUZZ-IEEE, pages 650-653, 2001.

[43] J. Medina, M. Ojeda-Aciego and P. Vojtás. A procedural semantics for multi-adjoint logic programming. Progress in Artificial Intelligence (EPIA'01), Springer Verlag, LNAI, 2258(1):290-297, 2001.

[44] J. Medina, M. Ojeda-Aciego and P. Vojtás. Multi-adjoint logic programming with continuous semantics. Proc. of Logic Programming and Non-Monotonic Reasoning (LPNMR'01), Springer Verlag, LNAI, 2173:351-364, 2001.

[45] J. Medina, M. Ojeda-Aciego and P. Vojtás. Similarity-based unification: a multi-adjoint approach. Fuzzy Sets and Systems, 146(1):43-62, 2004.

[46] P.J. Morcillo and G. Moreno. FLOPER, a Fuzzy Logic Programming Environment for Research. In J. Almendros, editor, Proc. of VIII Jornadas sobre Programación y Lenguajes (PROLE'08), Gijón, October 7-10, pages 259-263. Thomson-Paraninfo, 2007.

[47] P. Morcillo and G. Moreno. Modeling interpretive steps in fuzzy logic computations. In Proc. of the 8th International Workshop on Fuzzy Logic and Applications (WILF'09). Palermo, June 9-12, V. D. Gesù, S. Pal and A. Petrosino, editors, Springer Verlag, LNAI 5571, 2009, pages 44-51.

[48] P. J. Morcillo and G. Moreno. On cost estimations for executing fuzzy logic programs. In Proc. of the 2009 International Conference on Artificial Intelligence (ICAI'09), July 13-16, 2009, Las Vegas, Nevada, USA, H. R. Arabnia, D. de la Fuente and J. A. Olivas, editors, CSREA Press, pages 217-223, 2009.

[49] P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER tool. In N. Bassiliades, G. Governatori and A. Paschke, editors, Proc. of the 2nd International Symposium on Rule Representation, Interchange and Reasoning on the Web (RuleML'08), Orlando, October 30-31, pages 119-126 Springer Verlag, LNCS 3521, 2008.

[50] P.J. Morcillo and G. Moreno. The Fuzzy Logic Programming Environment FLOPER. In M. Leuschel, editor, Proc. of 15th International Symposium on Formal Methods (FM'08), Posters and Research Tools, May 26-30, Turku, 2008.

[51] P.J. Morcillo and G. Moreno. Using FLOPER for Running/Debugging Fuzzy Logic Programs. In L. Magdalena, M. Ojeda-Aciego and J.L. Verdegay, editors, In Proc. of the 12th International Conference on Information Processing and Management of Uncertainty in Knoledge-based Systems (IPMU'08), June 22-27, 2008, Málaga, pages 481-488. University of Málaga, 2008.

[52] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In M. Dean et al., editor, Proc. of the 6th International Symposium on the Rule Interchange and Applications (RuleML'10), Washington, October 21-23, pages 20-34. Springer Verlag. LNCS 6403, 2010.

[53] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. Declarative Traces Into Fuzzy Computed Answers. In M. Dean and S. Tabet, editors, Proc. of the 7th International Symposium on the Rule Interchange and Applications (RuleML'11), Barcelona, July 19-21, 15 pages. Springer Verlag, 2011 (accepted).

[54] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. Dedekind-MacNeille completion and multi-adjoint lattices. In V. Vigo-Aguiar, editor, Proc. of International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'11), Vol. II. Benidorm, June 26-29, pages 846-858, 2011.

[55] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. Fuzzy computed answers collecting proof information. In C. Cabestany, I. Rojas and G. J. Caparrós, editors, Proc. of 11th International Work-Conference on Artificial Neural Networks (IWANN'11). Torremolinos, June 8-10, pages 445-452. Springer Verlag, 2011.

[56] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. Modeling interpretive steps into the FLOPER environment. In H.R. Arabnia et al., editor, Proc. of the 12th International Conference on Artificial Intelligence (ICAI'10), July 12-15, Las Vegas, pages 16-22. CSREA Press, 2010.

[57] P.J. Morcillo, G. Moreno, J. Penabad and C. Vázquez. Multi-Adjoint Lattices for Manipulating Truth-Degrees into the FLOPER System. In V. Gulías and J. Silva and A. Villanueva, editor, Proc. of X Jornadas sobre Programación y Lenguajes (PROLE'10), Valencia, Spain, September 7-10, pages 151-161. Garceta grupo editorial, 2010.

[58] H.T. Nguyen and E.A. Walker. A First Course in Fuzzy Logic. Chapman and Hall/CRC, Boca Ratón, Florida, 2000.

[59] V.Novak, I. Perfilieva and J. Mockor. Mathematical principles of fuzzy logic. Dordrecht. Kluwer, 2000.

[60] J. Pavelka. On fuzzy logic I, II, III. Zeitschrift fur Math. Logik und Grundlagen der Math, 25:45-52, 119-134, 447-464, 1979.

[61] J. Penabad. Desplegado de programas lógicos difusos. Universidad de Castilla-La Mancha. Tesis Doctoral. 2010.

[62] L.G. Rios-Filho and S.A. Sandri. Contextual Fuzzy Unification. In Proc. of 5th International Fuzzy Systems Association Congress (IFSA'95), Sao Paulo, pages 81-84, 1995.

[63] M. Rodriguez-Artalejo and C. Romero-Díaz. Quantitative logic programming revisited. In J. Garrigue and M. Hermenegildo, editors, Proc. of Functional and Logic Programming (FLOPS'08), pages 272-288. Springer LNCS 4989, 2008

[64] M.I. Sessa. Approximate reasoning by similarity-based SLD resolution. Fuzzy Sets and Systems, 275:389-426, 2002.

[65] M. I. Sessa. Flexible querying in deductive database. In A. Di Nola and G. Gerla, editors, School on Soft Computing at Salerno University: Selected Lectures 1996-1999, pages 257-276. Springer Verlag, 2000.

[66] E. Virtanen. Fuzzy Unification. Proc. of the International Conference on Information Processing and Management of Uncertainty in Knoledge-based Systems (IPMU'94), Paris, pages 1147-1152, 1991.

[67] P. Vojtás. Fuzzy Logic Programming. Fuzzy Sets and Systems, Elsevier, 124(1):361-370, 2001.

[68] P. Vojtás and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In R. Dyckhof et al, editor, Proc. of Extensions of Logic Programming (ELP'96) Leipzig, pages 289-301. LNCS 1050, Springer Verlag, 1996.

[69] L. A. Zadeh. Fuzzy Sets. Information and Control, 8(3):338-353, 1965.

[70] L. A. Zadeh. In R. J. Marks II, editor, Fuzzy logic technology and applications. IEEE Publications, 1994.