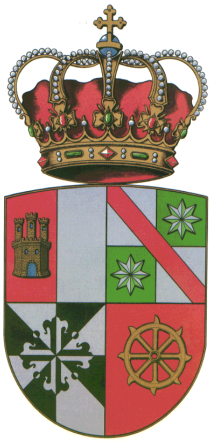
University of Castilla-La Mancha

Department of Computing Systems



Foundations and Applications of Fuzzy Logic Programming with Weights and Similarity Relations

TESIS DOCTORAL

Presentada por:

Carlos Vázquez Pérez-Íñigo

Dirigida por:

Ginés Moreno Valverde (UCLM)

Fundamentos y Aplicaciones de la Programación Lógica Difusa con Pesos y Relaciones de Similaridad

Carlos Vázquez Pérez-Íñigo

Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha

Memoria presentada para optar al título de:

Doctor en Informática

Dirigida por:

Ginés Moreno Valverde

Tribunal de lectura:

Presidente:	**Germán Vidal Oriola**	**U. Politécnica de Valencia**
Vocal:	**Luis Tomás Bolívar**	**Umea University**
Secretario:	**Juan Ángel Aledo Sánchez**	**U. de Castilla-La Mancha**

Albacete, 5 de octubre de 2015.

Summary

Logic is the science of reasoning, an integral part of all branches of knowledge that allows to model a great deal of complex systems and address difficult problems in both science and industry. Since the revolutionary work of L. A. Zadeh in the sixties, a new kind of logic, called *fuzzy logic*, has emerged. Although in the field of computer engineering the use of fuzzy logic is relatively new, it became a very promising area with agile solutions and bridging the distance between the world of crisp boundaries of the machine and the vague and imprecise reasoning of humans. With respect to fuzzy logic programming languages, there is no standard, but a constellation of languages that can be grouped in those focused on similarity relations –which exploit the fuzzy similarity between different elements– and those with weighted rules –which modulate the truth of rules through fuzzy degrees.

Our goal in this Thesis is to widen the semantics of fuzzy logic programming to increment the class of problems it can cope with. To do so, we focus on MALP, a very flexible framework of fuzzy logic programming in the second group above mentioned. It is characterised by the use of a large quantity of fuzzy connectives to link statements, the use of adjoint pairs to model *modus ponens* and its ability to cope with truth degrees defined beyond the usual $[0, 1]$ interval, more exactly, truth degrees belonging to any multi-adjoint lattice. We provide interesting results for this language, like the proof of the multi-adjointness of the cartesian product of multi-adjoint lattices or the use of special lattices to carry information about the execution of goals. Furthermore, as one of the main aims of this thesis, we have radically enhanced the expressiveness of MALP, first by dropping the necessity of the adjoint property –obtaining language X-MALP–, and finally by implementing the capabilities of Bousi~Prolog (a similarity-based language developed in our group). The result, FASILL, integrates the two main families of fuzzy logic programming languages mentioned above.

In order to test our results in MALP –and, afterwards, in FASILL–, since 2005 the tool $\mathcal{FLOPER}$ has been under development in our group. Its last capabilities, that I have implemented, include the possibility of loading different lattices, the implementation of the interpretative phase, and a significant expansion of the language it is based on. Indeed, while $\mathcal{FLOPER}$ was designed to operate MALP programs, now it is able to cope with X-MALP and FASILL programs performing similarity unification. Finally, an online version of $\mathcal{FLOPER}$ has been provided in the link `http://dectau.uclm.es/floper/?q=sim/test`, where the user can test the tool with no need to install any software.

During the development of this thesis we had the opportunity to apply the tools we were refining to some real-life problems in both fundamental and applied research. The first one was related to fuzzyXPath, a fuzzy version of the well known language XPath. The goal consisted on analysing large execution trees –directly provided as XML documents by $\mathcal{FLOPER}$– to explore derivation trees and obtain information such as infinite branches, solutions of the goals, unreachable code, etc. We have also addressed the field of SMT (Satisfiability Modulo Theory) by defining a method to use MALP and $\mathcal{FLOPER}$ to emulate an SMT tool. At last, but not least, we have provided a fuzzy solution to a cloud admission control problem. In particular, our system allowed to perform overbooking of tasks in a safe way by pondering the risks of any decision. As a result, it highly increases resource utilization with very little overpass of capacity.

To conclude, the work developed under this thesis has been shared with the research community through more than twenty publications –twelve of which are indexed in DBLP–, including international journals and conferences.

Agradecimientos

En estas líneas quisiera mostrar mi agradecimiento a todas las personas que me han brinado su ayuda y apoyo en todos los aspectos, y que han hecho posible que complete este proyecto vital que es una tesis doctoral.

En primer lugar, quiero dar las gracais a todos los españoles, pues son quienes han aportado su esfuerzo para dotar la generosidad de las instituciones. Gracias, también, a estas instituciones:

- al Ministerio que primero fue el de Educación, y más tarde el de Educación, Cultura y Deporte, que me ha ayudado a través del programa FPU;
- a dicho ministerio y al Ministerio de Economía y Competitividad por financiar los proyectos de investigación ALDDEIA y DAMAS, respectivamente;
- y a la Junta de Comunidades de Castilla-La Mancha por financiar el proyecto PROLOF.

También doy gracias a la organización EUSFLAT (EUropean Society for Fuzzy Logic And Technology), asociación a la que pertenezco (con número 823), por concederme una beca para asistir al *10th International Workshop on Fuzzy Logic and Applications (WILF'13)*, en Italia, experiencia que ha resultado muy fructífera a nivel académico y personal.

Agradezco también al Vicerrectorado de Investigación y Política Científica de la Universidad de Castilla-La Mancha por la concesión en 2013 de la beca CYTEMA-PUENTE, gracias a la cual pude realizar una estancia en Umea (Suecia), y entablar una fructífera colaboración con unos investigadores a los que no puedo menos que recordar aquí, como son Patrik Eklund y Luis Tomás.

Quiero agradecer aquí especialmente el trabajo que se han tomado en ayudarme a los profesores de la UCLM Jaime Penabad y Juan Antono Guerrero, a quienes me

enorgullece llamar compañeros; y, muy especialmente, a Ginés Moreno, mi director de tesis, por su guía, cercanía y ayuda inconmensurables.

Finalmente, doy gracias a mi familia por su apoyo incondicional. A mi madre, Elisa, y a mi hermano, Antonio, por su paciencia y por el ánimo que me han dado todos estos años, y a mi padre, José Antonio, por su ayuda con los vericuetos del idioma inglés.

Esta tesis no habría sido posible sin todos vosotros.

La ciencia se construye a partir de aproximaciones
que gradualmente se acercan a la verdad.
Isaac Asimov.

Un sueño sólo puede triunfar sobre la realidad si se le da la oportunidad.
Stanislaw Lem

Índice general

Capítulo 1

Introduction

The modern researching tendencies in *logic programming* (PL) are focused on the enrichment of the expressiveness and flexibility of this paradigm by assimilating concepts from fuzzy logic. In the last decade, the DEC-Tau research group of the University of Castilla-La Mancha, to which I belong, has produced multiple works on this line, shaping the framework of the *multi-adjoint logic programming* (MALP), that extends the classical notion of clause by adding fuzzy weights, using fuzzy connectives and characterizing the notion of truth inside a lattice in richer domains than the interval $[0, 1]$.

Fuzzy logic applies to the field of imprecise or vague statements we use to describe complex systems with no clear boundaries. It was first formulated by Lofti Zadeh [Zad65b, Zad65a] and widened by Goguen [Gog69] and Pavelka [Pav79], in order to incorporate to formal logic the imprecise predicates of the common language, and build an approximated type of reasoning. The perception of reality is full of concepts that do not admit a strict categorisation [TT89], as *tall*, *big*, *many*, *slowly*, *young*, *healthy*, *relevant*, *much greater than*, *kind*, among others. In the framework of fuzzy logic, such concepts determine fuzzy sets, i.e., identify classes of objects for which the transition of membership to non-membership is gradual and not crisp. In its simplest formulation, it constitutes an extension of classical bivalent logic to a logic with infinite truth values in the closed interval $[0, 1]$ with the usual ordering relation. It is, therefore, an extension of the polyvalent logic systems that shares with classical logic the search for soundness and completeness of the systems it studies [Haj06], although with different objectives.

Logic programming (LP), on the other hand, was originated in the research on Automatic Theorem Proving. Since the works of [Her30, Rob65, Kow74, War83], logic programming reaches maturity at the beginnings of the eighties. In essence, logic programming (for which [Llo87, Apt90, Apt97, JA07] are fundamental references) is based on a subset of predicate logic, concretely in Horn clauses, that are used as a basis for a programming language together with an operational semantics, SLD-resolution, for which there is an efficient implementation. The declarative semantics of LP can be defined in many ways. An illustrative example is model theory, whose domain is a purely syntactic universe: the Herbrand universe.

Fuzzy logic programming (FLP) arose as an extension of LP in the same sense that fuzzy logic extends classical logic. FLP is defined formally as a part of fuzzy logic focused on the study of fuzzy theories or fuzzy programs, that are a set of fuzzy logic expressions in a first order language directly executable on a computer. There are still neither standards nor a unified framework, but different approaches that take divergent paths. Part of the work of this thesis focuses on the unification of different paths of precursor researches with respect to the fuzzification of the logic paradigm, as detailed in Chapter 6. Many of the different approaches of fuzzy logic programming replace the classical inference mechanism, SLD-resolution, by a fuzzy variant that allows dealing with uncertainty and evaluating truth degrees. Taking this into account, and following the classification provided in [Rub11], it is possible to establish two main tendencies:

- The first one includes truth degrees with facts, rules and goals and, therefore, it modifies the resolution mechanism to perform operations over those degrees, but unification remains intact;
- The second one modifies the unification algorithm and preserves the resolution mechanism by handling truth values separately.

We conclude this introduction detailing the importance of the theory of lattices and, more specifically, the description of lattices that can be associated to programs, as well as the retrieval of basic results. Lattices are (in our scope) ordered sets where the formulas of programs are interpreted in some contexts of fuzzy logic programming. In particular, we are interested in the lattices used in multi-adjoint logic programming, developed in [MOV04, MOV01d, MOV01c], since this is the language we focused on in our research.

In essence, the notion of multi-adjoint lattice consists of an ordered lattice L

equipped with a set of connectives, like implications, conjunctions, disjunctions and aggregators, with the particularity that each implication symbol has an adjoint conjunction used to model the inference rule of *modus ponens* in a fuzzy environment. For instance, we expose now some of the *adjoint pairs* (conjunctions and implications) in the lattice $([0,1], \leq)$ (that we denote by $\mathcal{V}$). The labels `P`, `G` and `L` stand, respectively, for *Product logic* (1), *Gödel logic* (2) and *Łukasiewicz logic* (3). Each logic is suited to model *pessimistic*, *optimistic* and *realistic* scenarios, respectively.

Furthermore, in the MALP environment, each program is associated to its own multi-adjoint lattice, and each program rule is "weighted" by an element of L. Also, the components in the bodies are linked by connectives in the lattice. Those rules are similar to PROLOG clauses [Llo87].

Until the beginning of this thesis very few lattices (basically, the classical $[0,1]$ interval) were used to model truth degrees in MALP programs, ignoring a great deal of benefits that the research on lattices could bring to the expressiveness of programs and answers. This led us to dedicate part of the work of this thesis to the search of flexible and expressive lattices to be used in MALP programs.

In order to produce MALP programs and help their debugging, the $\mathcal{FLOPER}$ tool has been developed in our group (with a mayor participation of the author of this thesis). $\mathcal{FLOPER}$ has been revealed to the research community in more than twenty publications (ten of which are indexed in DBLP) and presentations in important national/international congresses of the areas of declarative programming, formal methods, artificial intelligence, rule-based systems, fuzzy logic, etc., as *IJCM, FOCI, IWANN, WILF, RuleML, FM, ICCMSE, IPMU, PROLE, CLAI* and *ICAI*, that publishes in (among others) prestigious journals as *Taylord&Francis, Springer, IEEE* o *CSREA-Press*.

1.1. Goals and motivations

The main goal of this thesis is to expand the semantics of fuzzy logic programming in order to cope with a wider class of problems. In particular, we focus largely on the multi-adjoint language (MALP) approach, since it is a very flexible and sound framework in whose development our research group has had an important role. Concretely, the contributions of our group to this language include the definition of its declarative semantics via model theory and the proof of the equivalence with the fix-point semantics, the incorporation of an interpretative phase to its operational

semantics, the refinement of a computational cost measure, the definition of a great deal of multi-adjoint lattices for multiple purposes (from reporting the computational steps to carry information of a number of other lattices in the form of a cartesian product of lattices) and the development of a practical environment, $\mathcal{FLOPER}$, for testing MALP programs.

The enhancement on the expressiveness of fuzzy logic programming can be addressed in two ways. The first one consists on providing lattices to represent richer versions of the notion of truth degree. These lattices can be tested easily in our $\mathcal{FLOPER}$ tool, but should not be restricted to a multi-adjoint framework. On the other hand, the language itself can be enriched by abandoning the adjoint constrain. One of the goals of this thesis is to provide a fuzzy logic programming language more expressive and powerful than MALP, that is, a language whose class of programs exceed that of MALP. More ambitiously, we plan to detail a language wide enough to integrate the main families of fuzzy logic programming, i.e. , the languages based on weighted rules (like MALP itself) and the languages based on similarity relations (like Bousi~Prolog, entirely developed in our group). This language should be ultimately testable in the $\mathcal{FLOPER}$ environment.

Although the theoretical research is very important, in the last term, the final goal of researching is to solve real-life problems. Thus, a relevant objective of this thesis is to provide solutions to problems in a purely fuzzy logic programming way, in other words, to bring the notion of truth degrees –possibly out of the simpler $[0, 1]$ range– and approximate reasoning to approach industry –or other research– problems.

With respect to $\mathcal{FLOPER}$, it is a very important tool for this thesis since it allows testing all our developments, whether it is a purely theoretical or a practical solution to an industry problem. For this reason, we aim to maintain this tool updated and, even more, to allow the system to manage and execute programs in the new fuzzy logic languages to be detailed in this thesis.

Our goals, then, can be summarized as follows:

1. Enhance the expressiveness of MALP by using refined lattices.
2. Provide a language more expressive than MALP.
3. Keep the $\mathcal{FLOPER}$ tool updated for managing and executing fuzzy logic programs and for testing MALP-based solutions.
4. Provide solutions based on fuzzy logic programming to real life problems.

5. Integrate the main families of fuzzy logic programming into one language that copes with weighted rules and equality by similarity.

1.2. Contributions

The contributions of this thesis are summarized as follows:

- A proof of the multi-adjointness of rich lattices, like the String lattice (the set of all words for certain alphabet with operation "append" as a conjunction) and the cartesian product of lattices (lattices whose elements are tuples containing elements of other lattices) have been provided [MMPV12c, MMPV12b]. This is an important contribution for the MALP framework since it enriches the class of programs to be modelled in it, as well as allows to report different information about the execution of a goal.

- The richer notion of equality for fuzzy logic, called *similarity-based strict equality*, has been researched ([MPV12b]). We have also created a method to build MALP rules modeling this equality notion from similarity equations ([MPV14b, JMV15]). This work that can be tested freely through the url `http://dectau.uclm.es/sse/`

- We have provided a preliminary description of fuzzy logic via category theory (see [EGH$^+$13a, EGH$^+$13b], for which the early experience of [GMV10] was helpful). In particular, we have addressed the definition of fuzzy data and fuzzy sort, and made an online tool to test this experimental definitions that can be accessed in the url `http://dectau.uclm.es/FLUS/`.

- With respect to the formal description of the MALP language, we have redefined various of its concepts by means of fuzzy sets [MPV14c], thus obtaining a language more deeply routed in the fuzzy notion. Aside from the core definition of the language, we have provided the notion of *logical consequence* for MALP, and we have related it with the concept of *fuzzy correct answer* [MPV12a].

- By means of the FuzzyXPath tool –ultimately developed using MALP and $\mathcal{FLOPER}$–, interesting results can be obtained about MALP derivations.

This can be used further to automatically detect phenomena such as infinite branches, unused program rules and unreachable code, among others [ALMV14, ALMV13].

- MALP has been successfully used in the area of SMT for automatically proving fuzzy propositional formulae [BMVV15, BMVV13]. We have even linked this development with the previous one in [ABL$^+$15], with the aim of automatically searching information about the satisfiability of an SMT formula using FuzzyXPath.

- We have provided a useful solution for resource overbooking in cloud infrastructures that notably enhances resource utilization [VTMT13]. Furthermore, once a work is accepted into a cloud infrastructure, we have designed a fuzzy algorithm to find the most suitable core to allocate it [VMTT15].

- With respect to the $\mathcal{FLOPER}$ tool (for which we provide a full description in [MV14]), we describe the implementation of mutli-adjoint lattices into it [MMPV10a, MMPV10c], as well as the implementation of its interpretative phase [MMPV10b]. It is now able to manage the new X-MALP and FASILL languages the same way it did with MALP [IMPV15, JMPV14]. This tool is also able to perform declarative traces of answers [MMPV11c, MMPV11a].

- Also, an online instance of the environment has been provided under the name "FLOPER online" through the link `http://dectau.uclm.es/floper/?q=sim/test`.

- In the research field, we have provided the X-MALP [MPV14a] language (a step further beyond our work on [MPV13], where we found a MALP subclass of programs independent of the andoint property), that inherits MALP syntax but whose operational and declarative semantics allows it not to be restricted by the adjoint property, that is, to cope with a class of lattices –concretely, complete lattices– much wider than multi-adjoint lattices. Furthermore, incorporating similarity-based equality into X-MALP we obtained the language FASILL [IMPV15, JMPV14], that can be viewed as the integration of the two main families of fuzzy logic programming languages.

1.3. Organization of the memory

This memory is organized as follows. In the first place, after this introductory chapter where we have provided the goals, motivations and contributions of this work, we detail in Chapter 2 the state of the art corresponding to the various researches on fuzzy logic programming. We focus on the idea that fuzzy logic programming emerges from logic programming as fuzzy logic emerges from classical logic. We finish the state of the art detailing the notion of *multi-adjoint lattice* in Section 2.4, that is essential in the thesis, since a great deal of it has been developed under the MALP language, which makes use of this kind of lattices.

With respect to the MALP language, just mentioned, we dedicate Chapter 3 to describe its numerous concepts. We describe its basic notions and syntax, its operational and declarative semantics, and other concepts related with the kind of answers to get from a program and contemplating more expressive lattices to model truth degrees. MALP has have a major role in this thesis since the majority of the research performed here are based on this language. Furthermore, the ulterior research on richer languages takes MALP as a starting point.

In order to test our programs and results in our theoretical works, we have made use of the $\mathcal{FLOPER}$ tool, that has been under development in our group since 2005. This tool itself has been deeply reconstructed in this thesis to better help our researches. Since it is the MALP programming environment par excellence and its implementation issues are also worth noting, we dedicate Chapter 4 to describe $\mathcal{FLOPER}$.

One of our goals is to apply fuzzy logic programming to provide a different point of view to solve real-life problems (both in the theoretical and practical areas). In Chapter 5 we detail the four most relevant applications of the MALP language. These applications include:

- Debugging of execution trees using FuzzyXPath.
- Using MALP as an SMT prover.
- A cloud overbooking tool using MALP.
- A cloud allocation tool using MALP.

We address the goal of paradigm integration in Chapter 6, where we describe, step by step, the developing process of the FASILL language from MALP and another language of our group, Bousi~Prolog.

We end this memory with Chapter 7, where we summarize the work performed and provide an insight of future research lines.

With respect of the language used throughout this memory, we use English for chapters 1 and 7, as it is mandatory in the international thesis program. Spanish has been used in chapters 3, 4, 5 and 6. To balance the use of languages we have also used English in Chapter 2.

Capítulo 2

State of the art

This chapter describes the basis on which the work in this thesis is based. Firstly we describe the field of fuzzy logic, that was originated by the works [Zad65b, Zad65a, Gog69, Pav79] and we use as an extension of the polyvalent logic systems [Pav79, Hó8, NPM99]. In this logic theory it is possible to define a wide variety of operators, like t-norms, t-conorms and aggregators [DP84, DP85, DP86, FY94, FR92, Yag93a, Yag93b, Yag94b, Yag94a, CBM99, CKKM02], and there are many definitions for implications [TCC00, CF95]. After its description, a historical background is provided [Zad96].

Then, we detail the main notions of *logic programming* [Her30, Rob65, Kow74, War83, Llo87, Apt90, Apt97, JA07], before addressing the area of *fuzzy logic programming* [Hin86, MBP87, LL90, IK85]. We review the more prominent languages in this field; in particular, those based on weighted rules (Prolog-Elf [IK85], FProlog [MBP87], Fuzzy Prolog [MSD89], f-Prolog [LL90], RFuzzy and the multi-adjoint logic programming language MALP [MOV01d, MOV01c, MOV01b, MOV01a]), and the other based on similarities (Likelog[FGS00], SiLog[LSS01] based on [Ses02], and Bousi∼Prolog [JRG08, JRG09a, JR09b, JR10a]).

We end this chapter by providing the definition of multi-adjoint lattices [MOV01d, MOV01c]. Multi-adjoint lattices are a very general kind of structure that generalizes others like the residuated lattices [Dil39, Ger04, NPM99], that can be used to produce declarative traces of the computations [MMPV12c, MMPV12b, MMPV11a, MMPV11c, MMPV10c].

2.1. Fuzzy logic

Fuzzy logic applies to the field of imprecise or vague statements we use to describe complex systems with unclear boundaries. In this sense, if classical logic was defined as the science that studies the laws, ways and types of reasoning, fuzzy logic could be defined in the same way as the science that studies the laws, ways and types of approximated reasoning.

Fuzzy logic was first formulated by Lofti Zadeh [Zad65b, Zad65a] and widen by Goguen [Gog69] and Pavelka [Pav79], in order to incorporate to formal logic the imprecise predicates of the common language, and build an approximated type of reasoning.

For [Zad96], creator of this discipline and the one who introduced the term "fuzzy", there are two meanings for the concept of fuzzy logic. In a broad sense, as generally understood, fuzzy logic refers to the use of fuzzy sets for manipulating imprecise knowledge. Therefore, it defines a theory of classes with non-sharp boundaries. This approximation differs clearly from ordinary logic in the use of fuzzy relations, the generalisation of traditional logic connectives such as '$\neg$', '$\wedge$' y '$\vee$' by their fuzzy counterparts, the fuzzy negations, t-norms and t-conorms, as well as other concepts like truth values, the presence of linguistic modifiers, etc. In this wider conception of the fuzzy logic, truth values can be fuzzy themselves, independently of the vagueness of the predicates. In practice, this means that the evaluation of a relation does not give a value (e.g., a number between 0 and 1), but the characteristic function that defines the relation itself.

In the other hand, strictly speaking, fuzzy logic refers to a logic system that formalizes approximated reasoning. In its simplest formulation, it constitutes an extension of classical bivalent logic to a logic with infinite truth values in the closed interval $[0, 1]$ with the usual ordering relation. It is, therefore, an extension of the polyvalent logic systems that shares with classical logic the search for soundness and completeness of the systems it studies [Haj06], although with different aims. This orientation of logic is relatively recent. It dates back to the works of [Pav79], who, together with [H9́8, NPM99], constitutes the fundamental references in this second definition.

Traditionally, imprecise reasoning has not been appropriately addressed. According to [TAT95], since the beginnings of classical logic there have been only insufficient solutions: some has tried to gain precision against the imprecise (Frege or Russel), and others tried to isolate the imprecise to carefully avoid it (Plato, Hume

or some contemporary strategies). This limitation in the traditional tools motivates the investigation on fuzzy sets and, in parallel, fuzzy logic. Classical logic, classical set theory and probability are not well suited to address the vagueness, imprecision, uncertainty, lack of specificity, inconsistency and complexity of the real world.

In the crisp (bivalent) ambit of classical mathematics there is no room for vagueness and partial truth. In this framework, all statement has to admit a precise definition that divides the objects of the considered universe into to subsets: one for those that satisfy it, and the other for those that do not satisfy it; and there is no possibility of dubious cases.

In frontal opposition to this world of clear borders, the perception of reality is full of concepts that do not admit a strict categorisation [TT89], as *tall*, *big*, *many*, *slowly*, *young*, *healthy*, *relevant*, *much greater than*, *kind*, among others. In the framework of fuzzy logic such concepts determine fuzzy sets, i.e., identify classes of objects for which the transition of membership to non-membership is gradual and not crisp.

Furthermore, the effort to acquire knowledge relating the real world is giving way to the effort to know aspects of the knowledge itself. Currently the delimitation of the scope and soundness of the information is as relevant (if not more) as its mere acquisition: it is necessary to know to which extent do we know something, i.e., assign a truth degree to it. Uncertainty is inextricably bounded to information. Even though there are different types of uncertainty, the one produced by the imprecision and subjectivity of human thinking is the most relevant [Zad08]. In many occasions it is convenient to sacrifice part of the precise information available to have a more vague but useful information in order to cope efficiently with the complexity of real world. Many of the usual concepts share an imprecise nature, i.e., they are not clearly delimited but they are significant. Fuzzy logic and fuzzy set theory offer a natural method to deal with vagueness and imprecision.

In [Zad65b] the author introduces for the first time a theory of fuzzy sets, that are sets with non precise borders and whose membership function gives a degree. One of the main goals of this theory is to provide a basis for approximate reasoning that uses vague hypotheses as a tool for formulating knowledge. Although its nature is different from other logics, in opinion of [TAT95], this logic of infinite truth values can be seen as an extension of the bivalent logic, of the trivalent logic defined by Łukasiewicz in 1922 and, in general, of the multivalued logic [Ack67, Cha58].

In other words, fuzzy logic can be seen as a reasoning model that takes into account qualitative or approximated aspects and that has a great ability to manage

very complex or poorly defined problems. Its natural basis is conformed by the above mentioned fuzzy sets, that are the mathematical columns that sustain fuzzy predicates and allow to perform the logic calculations needed to perform inferences.

This logic is one of the most interesting and recent theories to model an abundant number of systems for which classical logic, multivalent logics and the probability framework are insufficient or inappropriate. For this problems, fuzzy logic offers symbols and operators that operates with the notion of vagueness, and inference rules that preserve, delimit and transmit the truth values from the hypothesis to the thesis. Next, we define some of the basic notions of fuzzy logic as collected in the thesis of Jaime Penabad [Pen10].

2.1.1. Fuzzy sets, aggregators and fuzzy implications

[Zad65b] introduces a notion of fuzzy set through which the concepts of fuzzy interpretation, fuzzy logic operations and linguistic modifiers, among others ([PG98]) are formalized. Consider ordinary sets like

$$A = \{x \in \mathbb{Z} : x \text{ is prime}\},\ A = \{x \in \mathbb{N} : x \text{ is even}\},\ A = \{x : x \text{ is mortal}\}$$

for which the membership relation is discrete, i.e., an element (of the corresponding universe) belongs or does not belong to the set:

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

In the case of fuzzy sets, the membership is associated to a degree; is the case of sets like:

$$A = \{x \in \mathbb{Z} : x \text{ is big}\}$$

$$A = \{x : x \text{ is a warm day}\}$$

$$A = \{x : x \text{ is a developed country}\}$$

in which the nature of the property (predicate) that characterizes them is not clear (as it is in ordinary sets), but fuzzy. Therefore, we cannot say that the elements of the universe satisfy or not a certain predicate, but they satisfy it at some degree. These sets are formalised providing this new notion of membership, as we do next.

A fuzzy set A, in a universe $\mathcal{U}$, is expressed as:

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

where the application

$$\mu_A : \mathcal{U} \to [0,1]$$

is the membership degree function.

In other words, a fuzzy set is determined by a function μ_A. For each $x \in \mathcal{U}$, $\mu_A(x) \in [0,1]$ is a real number that indicates the compatibility of x with the characteristic (predicate) that defines the set A.

It is also possible to consider that the ordinary membership is determined by the characteristic function

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

that is,

$$x \in A \Leftrightarrow \chi_A(x) = 1, \quad x \notin A \Leftrightarrow \chi_A(x) = 0, \quad \forall x \in \mathcal{U}$$

The fuzzy membership, given by μ_A, is a generalisation of the classical one, since χ_A is a particular case of μ_A.

From this easy but relevant observation follows that the notion of fuzzy set extends the one of classical set. That is, an ordinary set is a fuzzy set. Particularly, the universe $\mathcal{U}$ (that we take as ordinary in the definition of A) and the empty set $\varnothing$ are fuzzy. Indeed,

$$\mu_\varnothing = \chi_\varnothing \text{ such that } \mu_\varnothing(x) = \chi_\varnothing(x) = 0, \ \forall x \in \mathcal{U}$$

$$\mu_\mathcal{U} = \chi_\mathcal{U} \text{ such that } \mu_\mathcal{U}(x) = \chi_\mathcal{U}(x) = 1, \ \forall x \in \mathcal{U}$$

Once characterized a fuzzy set by its membership degree function, μ_A, all its properties are referred to this, so the content, complementary, operations, etc., are expressed in terms of the corresponding membership functions.

From a semantic point of view, the essential definition of fuzzy logic is the one of interpretation, which associates to each (atomic) formula an element usually taken[1] in the real interval $[0,1]$. We detail now how an interpretation gives a truth degree to a fuzzy proposition through the concept of fuzzy set.

Given a predicate $A(x)$ in a universe $\mathcal{U}$ and an element $x_0 \in \mathcal{U}$, the formula $A(x_0)$ is interpreted as true with truth degree $\mu_A(x_0)$. In that case we write:

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

[1] In a more general way it can be taken from a certain ordered set [Zad08], as we consider later in this chapter.

Then we say that the proposition $A(x_0)$ holds with degree $\mu_A(x_0)$, that is the membership degree of x_0 to the fuzzy set A. And that set is, necessary, the set associated to predicate $A(x)$:

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

We can assume that the previous definition is the formalisation (interpretation) of predicate $A(x)$ through the fuzzy set A. Indeed, it is legitimate to define it in these terms: "x_0 satisfies predicate $A(x)$ with degree $\mu_A(x_0)$, that is, the membership degree of x_0 to the fuzzy set $A = \{x \in \mathcal{U} : A(x)\}$."

Later in this chapter, by means of linguistic modifiers, we provide a meaning to consider fuzzy propositions as *false*, *very true*, *very false*, *more or less true*, etc. This way we also incorporate the fuzzy nature to the concept of interpretation: particularly, it is possible to associate a different interpretation to each predicate modifier (like *no*, *too*, *a little*, *approximately*, etc.) to be formalized. This possibility provides another differentiating element of fuzzy logic.

Before proceeding, it is mandatory to distinguish the vagueness of a statement (that affects to the statement itself) and uncertainty (that affects its compliance). In other words, this is not a possibilistic logic that considers the nonrandom uncertainty of non fuzzy propositions.

We detail now the syntax of this paradigm beyond fuzzy sets. The syntax of fuzzy logic has not too many novelties with respect to the interpretation of connectives. Once an elemental expression has been interpreted, the compounded expressions takes their values using ad hoc formulae [Lee72]. Thus, for instance, the conjunction is usually defined by the formula

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

where $A(x), B(y)$ are some predicates in universes $\mathcal{U}$, $\mathcal{V}$, respectively, and $x_0 \in \mathcal{U}, y_0 \in \mathcal{V}$.

If we take predicates $A(x), B(x)$ over the same universe $\mathcal{U}$ and they define, respectively, fuzzy sets $A, B \subset \mathcal{U}$, it also follows

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

where $\mu_{A \cap B}(x_0)$ is the membership degree of x_0 to the intersection set $A \cap B$. That is, it is allowed to define the fuzzy conjunction by means of the corresponding intersection of sets.

Generally speaking, the truth function of the fuzzy conjunction can be defined also by the wide range of functions known as triangular norms, introduced by [SS83] to model distances in probabilistic metric spaces (defined by K. Menger in 1942) and the semigroups of distribution functions.

We define now these functions in the interval $[0, 1]$.

Definition 2.1.1 ([NW06]). *An operation $T : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ is a triangular norm or t-norm if, and only if, it verifies*

i) *is commutative, i.e., $T(x, y) = T(y, x), \forall x, y \in [0, 1]$.*

ii) *is associative, i.e., $T(x, T(y, z)) = T(T(x, y), z), \forall x, y, z \in [0, 1]$.*

iii) $T(x, 1) = x, \forall x \in [0, 1]$.

iv) *is monotonic in each component, i.e.*[2]*, if $x_1 \leq x_2$, then $T(x_1, y) \leq T(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.*

Analogously, disjunction is usually characterized by the expression

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

and if we consider predicates $A(x), B(x)$ on the same universe $\mathcal{U}$ defining, respectively, the fuzzy sets $A, B \subset \mathcal{U}$, we also have

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

where $\mu_{A \cup B}(x_0)$ is the membership degree of x_0 to the union set $A \cup B$. Consequently, this logic operation is associated to the union of sets. More precisely, it is possible to formalize fuzzy disjunction (of propositions and also of predicates) by the union of fuzzy sets.

Furthermore, as in the conjunction, the (truth function of the) fuzzy disjunction can be defined by the wide range of functions called t-conorms, characterized in the following way in the interval $[0, 1]$.

Definition 2.1.2 ([NW06]). *An operation $S : [0, 1] \times [0, 1] \longrightarrow [0, 1]$ is a triangular conorm, or t-conorm, if, and only if, it verifies*

i) *is commutative, i.e., $S(x, y) = S(y, x), \forall x, y \in [0, 1]$.*

[2] From the given characterization (only for the first component) follows also the monotonicity in the second one using conditions *i)* and *iv)*.

ii) *is associative, i.e.,* $S(x, S(y, z)) = S(S(x, y), z), \forall x, y, z \in [0, 1]$.

iii) $S(x, 0) = x, \forall x \in [0, 1]$.

iv) *is monotonic in each component, i.e.*[3]*, if* $x_1 \leq x_2$*, then* $S(x_1, y) \leq S(x_2, y)$, $\forall x_1, x_2, y \in [0, 1]$.

If T is a t-norm in $[0, 1]$, then $S(x, y) = 1 - T(1 - x, 1 - y)$ defines a t-conorm and S is said to derive from T. More generally, given a t-norm T and a strong negation[4] N, then function $S_N : [0, 1] \times [0, 1] \longrightarrow [0, 1]$, defined as $S_N(x, y) = N(T(N(x), N(y)))$, is a t-conorm called N-dual of T.

By the elemental properties of negation we have $T(x, y) = N(S_N(N(x), N(y)))$, that is, T is the N-dual t-norm of S_N. Given a t-conorm S and a strong negation N, the function $T_N : [0, 1] \times [0, 1] \longrightarrow [0, 1]$, defined as $T_N(x, y) = N(S(N(x), N(y)))$, is a t-norm called N-dual t-norm of S.

Again, since N is a negation, $S(x, y) = N(T_N(N(x), N(y)))$, that is, S is the N-dual t-conorm of T_N.

Concluding, we say that T and S are N-dual if $\forall x, y \in [0, 1]$ it holds:

$$T(x, y) = N(S(N(x), N(y))) \qquad S(x, y) = N(T(N(x), N(y)))$$

Particularly, taking the usual negation $N(x) = 1 - x$, T and S are dual if $\forall x \in [0, 1]$ it holds:

$$T(x, y) = 1 - S(1 - x, 1 - y) \qquad S(x, y) = 1 - T(1 - x, 1 - y)$$

We present below basic pairs of basic t-norms and t-conorms (dual) ([CFF97]):

- Zadeh's (or the Minimum/Maximum) defined by

$$T(x, y) = min\{x, y\} \qquad S(x, y) = max\{x, y\}$$

- Łukasiewicz's defined by

$$T(x, y) = max\{x + y - 1, 0\} \qquad S(x, y) = min\{x + y, 1\}$$

[3] The monotonicity in the second component follows also from *i)* and *iv)*.

[4] A strong negation in $[0, 1]$ is a function $N : [0, 1] \longrightarrow [0, 1]$ that is continuous, strictly decreasing and $N(0) = 1, N(N(x)) = x$.

- Of the Product, defined by

$$T(x,y) = xy \qquad S(x,y) = x + y - xy$$

- Weak/Strong, defined by

$$T(x,y) = \begin{cases} min\{x,y\}, & \text{if } max\{x,y\} = 1 \\ 0, & \text{otherwise} \end{cases} \qquad S(x,y) = x + y - xy$$

- Hamacher's, defined for each $\gamma \geq 0$ by

$$T_\gamma(x,y) = \frac{xy}{\gamma + (1-\gamma)(x+y-xy)} \qquad S(x,y) = \frac{x+y-(2-\gamma)xy}{1-(1-\gamma)xy}$$

- Yager's, defined for each $p > 0$ by

$$T_p(x,y) = 1 - min\{1, \sqrt[p]{(1-x)^p + (1-y)^p}\} \quad S_p(x,y) = min\{1, \sqrt[p]{x^p + y^p}\}$$

It is common to use t-norms and t-conorms to produce new connectives [Miz89a, Miz89b, Tur92, FC98, DSMK07, KMP04]. T-norms and t-conorms are particular cases of aggregation operators[5] (studied by [DP84, DP85, DP86], [FY94, FR92] and [Yag93a, Yag93b, Yag94b, Yag94a]) and, also, certain combinations of them originate new aggregation operators [CBM99, CKKM02].

It is possible to produce aggregators (see [Lin65, Miz89b, Tur92, MTK99, JM03, Jen04, Jen06]) by convex combinations of a t-norm T and a t-conorm S, that is, produce the aggregator $@(x,y) = \alpha T(x,y) + (1-\alpha)S(x,y)$, that preserves symmetry and idempotence.

Aggregators are common in the development of multiple intelligent systems, as is the case of neuronal networks, fuzzy controllers, expert systems and, specially, in decision theory. Aggregators allow the efficient and flexible combination of information [HHV96], which has become a main task in multiple-criteria decision problems where it is necessary to process a great deal of information of different quality and precision.

The most general definition for the aggregation operator, in the interval $[0,1]$, is the one given in [KK99], that we reproduce here.

Definition 2.1.3. *An aggregation operator* $@$ *is an application* $@ : [0,1]^n \longrightarrow [0,1]$ *that fulfils:*

[5] See Definition 2.1.3 for a characterization of the former.

i) $@(0,\ldots,0)=0, @(1,\ldots,1)=1$ *(boundary conditions)*

ii) $\forall(x_1,\ldots,x_n),(y_1,\ldots,y_n)\in[0,1]^n$,
$(x_1,\ldots,x_n)\leq(y_1,\ldots,y_n)$[6] $\Longrightarrow @(x_1,\ldots,x_n)\leq @(y_1,\ldots,y_n)$ *(monotonicity)*

Sometimes other conditions are required together with the ones mentioned above, such as continuity, symmetry and idempotence. Particularly, @ is symmetric if, and only if, for all permutation σ of $\{1,\ldots,n\}$ and all n-uple $(x_1,\ldots,x_n)\in[0,1]^n$ the next holds: $@(x_1,\ldots,x_n)=@(x_{\sigma(1)},\ldots,x_{\sigma(n)})$; also, @ is idempotent (i.e., $@(x,\ldots,x)=x$) if and only if for all n-uple $(x_1,\ldots,x_n)\in[0,1]^n$, $min\{x_1,\ldots,x_n\}\leq @(x_1,\ldots,x_n)\leq max\{x_1,\ldots,x_n\}$ holds.

Some well known examples of aggregation operators are t-norms and t-conorms (previously detailed), Quasi-Linear Weighted Means [Acz48, Yag94b] (if they are, also, symmetric, they give the quasi-arithmetic mean, like the arithmetical average, the geometric average, and harmonic and quadratic means), OWA operators [Yag88] (arithmetic average is also a particular case of these operators), the extended aggregation functions [MC97], and the γ-operators of [ZZ80], among others.

We end this section addressing a fundamental element in (fuzzy) logic: implication. Fuzzy implication constitutes the most interesting composed operation of fuzzy logic (as well as in classical logic), since it allows to perform logical inferences and deduce theorems from axioms. Its truth function allows different non-equivalent formulations; there are, thus, many different fuzzy implications that not always extend the usual (classical) implication [TCC00, CF95].

The usual way to interpret fuzzy implication

$$A(x_0)\Rightarrow B(y_0)$$

is given by the formula

$$\mathcal{I}(A(x_0)\Rightarrow B(y_0))=max\{min\{\mathcal{I}(A(x_0)),\mathcal{I}(B(y_0))\},1-\mathcal{I}(A(x_0))\},$$

where $A(x),B(y)$ are arbitrary predicates in universes $\mathcal{U}$, $\mathcal{V}$ respectively and $x_0\in\mathcal{U}, y_0\in\mathcal{V}$. If predicates $A(x),B(y)$ define fuzzy sets $A\subset\mathcal{U}, B\subset\mathcal{V}$, it follows

$$\mathcal{I}(A(x_0)\Rightarrow B(y_0))=max\{min\{\mu_A(x_0),\mu_B(y_0)\},1-\mu_A(x_0)\};\quad x_0\in\mathcal{U}, y_0\in\mathcal{V}$$

This truth function for fuzzy implication, provided by Zadeh, generalizes the classical implication.

[6] Where $(x_1,\ldots,x_n)\leq(y_1,\ldots,y_n)$ if, and only if, $x_i\leq y_i, i=1,\ldots,n$.

Mamdani and Larsen provide other interesting examples of fuzzy implication, whose interpretations we present here[7]

$$\text{Mamdani}: \quad \mathcal{I}(A(x_0) \Rightarrow B(y_0)) = min\{\mathcal{I}(A(x_0)), \mathcal{I}(B(y_0))\}$$

$$\text{Larsen}: \quad \mathcal{I}(A(x_0) \Rightarrow B(y_0)) = \mathcal{I}(A(x_0)) \cdot \mathcal{I}(B(y_0))$$

We present now the fuzzy implication in the most general way. First, we consider that, given a Boole algebra $(A, \wedge, \vee, ', 0, I)$, an operation $\rightarrow$: $A \times A \longrightarrow A$ is an implication if for each $x, y \in A$, it holds $x \wedge (x \rightarrow y) \leq y$. It is well known that $p \rightarrow q = (p \wedge q')'$, $p \rightarrow q = (p \wedge q) \vee (p' \wedge q) \vee (p \wedge q')$ are implications and, by the properties of the Boole algebra, we have that $(p \wedge q) \vee (p' \wedge q) \vee (p \wedge q') = (p' \wedge q') \vee q = p' \vee (p \wedge q) = p' \vee q$, and also $(p \wedge q')' = p' \vee q$.

If, in the fuzzy context, we choose a t-norm T instead of a conjunction $\wedge$, a t-conorm S instead of a disjunction $\vee$ and a strong negation N instead of a negation $'$, we obtain the following models of fuzzy implication [TCC00]:

$$\begin{aligned} J_1(x,y) &= N(T(x, N(y))) \\ J_2(x,y) &= S(N(x), y) \\ J_3(x,y) &= S(N(x), T(x,y)) \\ J_4(x,y) &= S(T(N(x), N(y)), y) \end{aligned}$$

With respect to its characterization, a fuzzy implication in the real interval $[0,1]$ is defined by the following truth function[8] [TCC00, TV85].

Definition 2.1.4. *An implication function J is an application $J : [0,1]^n \longrightarrow [0,1]$ that fulfils:*

(1) If $x_1 \leq x_2$, then $J(x_1, y) \geq J(x_2, y)$

(2) If $y_1 \leq y_2$, then $J(x, y_1) \leq J(x, y_2)$

(3) $J(0, y) = 1$

(4) $J(1, y) = y$

(5) $J(y, J(x, z)) = J(x, J(y, z)), \forall x, y, z \in [0,1]$

[7] Both implications are very used in the field of fuzzy control.

[8] For convenience, we omit from now on all mention to the logic expressions involved in the hypotheses and the theses of the implication, and we refer only to the truth function of the implication connective.

It is also frequent to require some of the next conditions

(6) $J(x, 0) = N(x)$

(7) J is continuous

(8) $J(x, y) = J(N(y), N(x))$, for some strong negation N

(9) $J(x, x) = 1$

(10) $x \leq y$ if, and only if, $J(x, y) = 1$

There are three main classes of (truth functions of) implications [CFF97, ACT95]:

- S-implications defined by

$$x \longrightarrow y = S(N(x), y)$$

  where S is a t-conorm and N is a negation in $[0, 1]$. These implications come from the equivalence, in binary logic, of formulae $p \rightarrow q$ and $p' \vee q$. Some of these S-implications are given by

  - Łukasiewicz, defined by $x \longrightarrow y = min\{1 - x + y, 1\}$
  - Kleene-Dienes, defined by $x \longrightarrow y = max\{1 - x, y\}$

- R-implications defined by residuation of a continuous t-norm T, like

$$x \longrightarrow y = sup\{z \in [0, 1] : T(x, z) \leq y\}$$

  These implications come from Gödel logic and, among them, we have the ones given by:

  - Gödel, defined by $x \longrightarrow y = \begin{cases} 1, & \text{if } x \leq y \\ y, & \text{if } x > y \end{cases}$
  - Łukasiewicz, defined by $x \longrightarrow y = min\{1 - x + y, 1\}$

Also, the following are admitted as implications:

- T-norm implications, defined through a t-norm T as

$$x \longrightarrow y = T(x, y)$$

  This group includes the Mamdani implication, used in theory of fuzzy control, and the implication from the product t-norm.

- QM-implications [TCC00, Yin02], characterized by $Q : [0,1]^2 \longrightarrow [0,1]$ given by $Q(x,y) = S(N(x), T(x,y))$ from a t-norm T, a t-conorm S and a strong negation N.

  Some QM-operators (that is the name that we should use in this case since these do not determine, in general, an implication) are

  - $Q_1(x,y) = S(1-x, T(x,y)) = max\{1-x, y\}$, where T is the t-norm of Łukasiewicz and S its dual t-conorm.
  - $Q_2(x,y) = S(1-x, T(x,y)) = 1 - x + x^2 y$, where T is the Product t-norm and S its dual t-conorm.
  - $Q_3(x,y) = \begin{cases} 1, & \text{if } y = 1 \\ y, & \text{if } x = 1 \\ 1-x, & \text{otherwise} \end{cases}$

Note that in classical logic the S-implication $p' \vee q$ and the QM-implication $p' \vee (p \wedge q)$ are equivalent and define the ordinary logic implication, although they are different as fuzzy operators.

In order to perform fuzzy inference, it is essential the property of the fuzzy (or generalized) modus ponens, that was first proposed by L. A. Zadeh and that propagates the truth degrees of the premises to the conclusion by means of a composition of fuzzy relations. $^{\circ}$:

$$\left.\begin{array}{c} \text{If } A(x) \text{ then } B(y) \\ \text{and} \\ A'(x) \end{array}\right\} \text{ then } B'(y)$$

where $A(x)$, $A'(x)$ are arbitrary fuzzy predicates (in an arbitrary universe $\mathcal{U}$) as well as $B(y)$, $B'(y)$ (in a universe $\mathcal{V}$). Such predicates are associated to the corresponding fuzzy sets A, A', B, B'.

The truth degree for this expression makes use of the composition rule

$$\left.\begin{array}{c} \text{If } A(x) \\ \text{and} \\ R(x,y) \end{array}\right\} \text{ then } (A \circ R)(y)$$

being:

$$\mu_{A \circ R}(y) = max\{min\{\mu_A(x), \mu_R(x,y)\}\}, x \in \mathcal{U}$$

where R is a binary fuzzy relation over $\mathcal{U}\times\mathcal{U}$, and $\circ$ denotes the (unary) composition of the fuzzy set A –characterized by predicate $A(x)$ in the universe $\mathcal{U}$– and the fuzzy relation R.

In contrast to ordinary logic, the antecedent $A(x)$ is not required to coincide with the previous $A'(x)$, and this fuzzy modus ponens can be seen as a particular case of the composition rule, where the relation R is the fuzzy cartesian product $A \times B$.

A fuzzy modus ponens appears, for instance, in [VP96], in the language f-Prolog that we describe in Section 2.3, although in its formalisation fuzzy relations are not involved.

While negation is the only modifier for classical predicates, a fuzzy predicate $A(x)$ can also be modified by

not $A(x)$, very $A(x)$, a little $A(x)$, more or less $A(x)$, approximately $A(x)$,...

Indeed, it is possible to formalize these so-called predicate modifiers, that correspond to adverbs and shapes the use of property $A(x)$. For instance, we can define modifier *very* in this way (supposing that $A(x)$ is defined in $\mathcal{U}$ and $x_0 \in \mathcal{U}$).

$$\mathcal{I}(very\ A(x_0)) = [\mathcal{I}(A(x_0))]^2 \quad \text{or equivalently} \quad \mathcal{I}(very\ A(x_0)) = [\mu_A(x_0)]^2$$

and for the modifier approximately

$$\mathcal{I}(approx\ A(x_0)) = [\mathcal{I}(A(x_0))]^{1/2} \quad \text{or equivalently} \quad \mathcal{I}(approx\ A(x_0)) = [\mu_A(x_0)]^{1/2}$$

It is very useful, for approximate reasoning, the logic concept of linguistic variable, developed by [Zad75]. A linguistic variable is a set of terms of natural (or formal) language expressions that can be taken as linguistic labels in the considered context. These labels are fuzzy sets over a domain. As an example of linguistic variable, consider variable *speed* with values: low, medium, high, among others, defined over the domain of kilometers per hour. Other example is the *truth*, with values: very true, a little true, false, very false, etc., that is, the values associated to the corresponding modifiers previously formalized.

Finally, aside from the already observed differences with classical logic, as the vague character of predicates, the infinite truth values, the presence of linguistic modifiers and the different interpretations, it is noteworthy the presence of specific quantifiers (see [DP80]) like: nearly all, some, the majority, quite a little.

2.1.2. Applications

Fuzzy logic (FL) constitutes a model for reasoning that allows to deal with complex problems, poorly defined problems or problems for which there are no precise mathematical model. Thanks to this kind of logic it has been possible to model an solve situations traditionally considered untreatable from the point of view of classical logic. In the last decades fuzzy logic has been used in a growing variety of instruments, machines, software and diverse fields of daily life. This proliferation of applications has diminished the initial distrust to this kind of logic.

In fact, since the basic notions of fuzzy logic where stablished, by the first time, by the paper of [Zad65b] on fuzzy sets; and in spite of the enthusiasm of some researchers, like the mathematicians R. Bellman and G. Moisil, to adopt the new ideas; the main tendency was skeptical, even hostile, towards the new theory. Currently, while some controversy still remains, the value of its contribution to multiple applications has consolidated this paradigm.

Professor Zadeh's original intention was to provide a formalism to handle the imprecision and vagueness of human thinking, expressed linguistically, although afterwards much of the merit of fuzzy logic has focused on the field of automatic control of processes. This is due to the fuzzy "boom" in Japan, that began in 1987 and reach its peak at the beginnings of the nineties. Indeed, aside from the relevant seminary EE.UU.-Japan on fuzzy sets and its applications held in Berkeley in 1974, other important milestone for the development of this logic was the congress IFSA (International Fuzzy Systems Association) of Tokyo that year (1987). In that congress, Matsushita announced the first consumer product based on fuzzy logic (a showerhead). Simultaneously, in other field, the Sendai underground was launched. It used a controller based on fuzzy logic, and is considered as one of the most successful applications of this logic [VZ96].

Since then, a large amount of consumer products use fuzzy technology, many of them using the label "fuzzy" as a symbol of quality and high performance. As early as in 1974, professor Mamdani experimented successfully with a fuzzy controller in a steam machine, while the first real implantation of such a controller was in performed in 1980 by F. L. Smidth & Co. in a cement plant in Denmark. In 1983, Fuji applied fuzzy logic to the control of chemical injection for a water treatment plant, for the first time in Japan. In 1987, OOMRON developed the first commercial fuzzy controllers with the professor Yamakawa. From then on, fuzzy control has been successfully applied to many branches of technology, as we see with examples

of specific applications at the end of this section. Its success lies in the conceptual and developmental simplicity of these control applications.

Worldwide, Japan is, as we have seen, the country where fuzzy logic and its applications has been best welcome. Professors K. Asai, J. Tanaka and T. Terano were precursors in 1968 with their works on fuzzy automata and learning systems. In Europe, the interest for fuzzy logic began in the seventies and the most significant contributions focuses on theoretical developments. With respect to Spain, professor E. Trillas began a research on fuzzy logic and its applications and, in opinion of [Zad96], thanks to his contributions Spain is a leading country in Europe in this field.

With respect to its application fields, one of the most important ones is control theory. Indeed, the application of fuzzy logic to control has been natural, and the "fuzzy" label was introduced initially linked to this area. The evolution of fuzzy control has been spectacular. Its growth has been quick because fuzzy control application are easy to make since they only require fuzzy rules of the form "if then" to handle commands. Fuzzy rules, usually fine-tuned by experts, are fuzzy implications involving fuzzy propositions (simple of compound) [DHR96, PDH97]. Also, fuzzy controllers are simple and sound and, in mane cases, there are no possibility to use traditional controllers since there are no mathematical model (or it is non practical), as states Ebrahim Mamdani (pioneer in fuzzy control) in his work [Mam93].

L. A. Zadeh creates also the theory of approximate reasoning of fuzzy logic in the context of artificial intelligence in his search for more efficient tools for building expert systems (other main field of FL).

In [Zad73] the principle of incompatibility is enunciated. It states that complexity and precision are antagonistic when describing the behaviour of a system, so conventional programs have little effectiveness to model human behaviour. Addressing this problem, it suggests, in one hand, to represent (imprecise) information by means of fuzzy sets; and in the other, the inference over imprecise information, based on the use of fuzzy implication and the most relevant property: generalized modus ponens, formalized as the unary composition of a fuzzy set in a fuzzy relation (the so-called inference compositional rule). This composition property of two fuzzy relations allows to apply fuzzy logic to fuzzy control and, then, the development of reasoning systems and their implementation. The concrete implication is to be chosen carefully since it is essential to the system. The effectiveness of different implication functions for reproducing human reasoning and ease inference methods has

been quite studied in the literature.

Fuzzy logic emerges in the search of professor Zadeh, as an answer to fuzzy logic because of two main aspects: it represents formally the imprecise knowledge and manages adequately the uncertainty in some expert systems. These characteristics gives to FL great relevancy in the field of knowledge engineering.

So, an expert system is a system based on knowledge plus information from the experts on the domain [PS05]. Its goal is the resolution of problems in this domain, applying reasoning techniques over the information in their base of knowledge. Despite probability theory is the classical formal model to represent uncertainty, it has not been universally accepted in the design of expert systems to address uncertainty. It is well known that it requires a large collection of data and operations to be appropriate [SB75, Ada76]. Many methods and specialized extensions of probability calculus has been developed to overcome these limitations, suchs as certainty factors of MYCIN[9] [SB75], the subjective Bayesian [DHN90], the theory of evidence of Dempster-Shafer [Sha76] and the theory of *endorsements* [Coh85]. In contrast to the previous methods, that lack a well known semantic framework, L. A. Zadeh proposed a formal logic based on theory of fuzzy sets that is very adequate for dealing with uncertainty.

Fuzzy Rule-Based Systems (FRBSs) are an extension of classical systems for representing knowledge based on rules [CHHM01]. As those, FRBSs are composed of conditional rules of the form "if then", with the particularity that the antecedent and consequent are fuzzy expressions. We list next some advantages of fuzzy expert systems:

- They are an easy way of codifying a non-linear system.
- They correspond well with the schemes of human thinking over a large amount of mathematical problems.
- They are efficient (the run quickly) on conventional computers.
- They run extremely quick on specialized hardware.

The application of fuzzy logic to rule-based systems has focused mainly in, in one hand, generalize the model of certainty factors and, in the other, the use of fuzzy predicates in the description of rules and reality.

[9]MYCIN is an expert system written in Prolog.

Other field of application to highlight is, finally, the contribution of fuzzy logic to *soft computing*. The emergence of neurocomputation and genetic algorithms (in the mid 80s) had a significant impact on the development of fuzzy logic. Probability and fuzzy logic can be used together in the methodologies of neurocomputation and genetic algorithms. This suggests to [Zad96] the concept of *soft computing*, understood as "a kind of society of fuzzy logic, neurocomputation and probabilistic reasoning". In this scope, fuzzy logic provides a methodology to deal with imprecision, approximate reasoning and computation with words. The most important of *soft computing* is that it suggests the possibility of using fuzzy logic, neurocomputation and genetic algorithms combined instead of isolated. One of the most relevant combinations currently is the "neuro-fuzzy" systems (for an introduction, see [Ngu02]). The growing use of *soft computing* has brought an important contribution for the conception, design and development of intelligent systems.

A part of this field is considered by many the new challenge for fuzzy logic: the Internet. As stated by professor José Ángel Olivas from the University of Castilla-La Mancha, the use of fuzzy technologies is mandatory to address the massive amount of data, retrieve information, and control and manage the net. This intuition coincides also with the new path that, according to professor Zadeh, should follow fuzzy logic. The first encounter on fuzzy logic and the Internet (FLINT 2001) held at the University of Berkely on summer 2001 and organized by Zadeh itself is proof of this. The main idea that arose is the tendency towards *Computing with words*, by means of techniques of *soft computing* (that includes fuzzy logic, neuronal networks and evolutionary computation). These terms, coined by professor Zadeh, materializes in many research lines like:

- A new generation of search engines on the Internet, using techniques of *soft computing* to enhance the current (lexicographic) search to a conceptual search.

- Advanced techniques to describe user profiles that allow a more intelligent use of the Internet.

- Semantic web, where users could delegate tasks on the software, that will be able to process, reason, combine information and perform logic deductions to solve daily problems.

And much more new fields of application of *soft computing*which already are producing promising results.

To end this section we relate some of the multiple specific applications of this logic (in the field of fuzzy control and expert systems, mainly).

- Consumer electronic products: intelligent washing machines of Panasonic or Bosch (Matsuhita Electronic Industrial), microwave ovens, termic systems, video recorders, televisions, image stabilizing systems in photographic and video cameras of Sony, Sanyo, Cannon (Matsuhita) and automatic focus systems in photographic cameras.

- Systems: automatic pilotage systems for airplanes, maneuvering control for lift or trains (underground of Senadi, Japan, 1987), water treatment systems, automotive systems (ABS of Mazda and Nissan, automatic speed control, climate control, automatic driving systems), industrial combustion control systems, traffic controllers, heating/cooling systems (Mitsubishi air conditioning, *rice-cooker*), climate prediction systems, atmospheric prediction systems and writing recognition systems.

- Software for: clinical diagnosis (CADAG, Adlssnig, Arita, OMRON), security (Yamaichi, Hitachi), linguistic translation, data understanding, informatics technology and fuzzy data bases for storing and querying imprecise information (use of language FSQL).

To summarize, and attending the opinion of [VZ96], fuzzy logic has applications, mainly, in two very different fields: the first, control theory applications, and the other, expert system development. Internet and *soft computing* could be third and fourth fields. According to him, we are entering an era of intelligent systems that will have a deep impact on the way we communicate, take decisions and use machines, and fuzzy logic (together with *soft computing* and fuzzy declarative languages, in our opinion), will play an important role in bringing the era of intelligent systems.

With respect to the work performed in this thesis, its practical applications are part of other promising line of application of fuzzy logic: the design and enhancement of declarative languages that allow to codify easily applications with fuzzy taste in the referred fields. Concretely, we focused on enhancements related to the procedural semantics of one of the most interesting paradigms in fuzzy logic programming, from our point of view, that is the multi-adjoint logic. In the next section we provide a brief view of logic programming to expose, afterwards, in a more detailed way, the most relevant aspects of fuzzy logic programming.

2.2. Logic programming

Logic programming (LP) was originated in the research on Automatic Theorem Proving. Since the works of [Her30, Rob65, Kow74, War83], logic programming reaches maturity at the beginnings of the eighties. In essence, logic programming (for which [Llo87, Apt90, Apt97, JA07]) are fundamental references) is based on a subset of predicate logic, concretely in Horn clauses, that are used as the core of a programming language together with an operational semantics, SLD-resolution, for which there are an efficient implementation.

Its main feature is, indeed, the use of logic as programming language. More precisely, a program in LP is conceived as a formal theory in a certain logic, and computation is understood as a logic deduction in this logic.

The base logic has to include the following elements (see [Jul00]):

- A language expressive enough to address an interesting field of application,
- An operational semantics, that is, a calculation mechanism to execute programs,
- A declarative semantics to provide a meaning to programs independently of their possible execution, and
- Results of soundness and completion to assure that the computed result coincides with what is considered true according to the notion of truth given by the declarative semantics.

Also, this declarative semantics specifies the meaning of the syntactic objects of the language by means of its translation to elements and structures in a known (generally mathematical) domain.

The operational semantics in LP is based on a method of proof by refutation called SLD-resolution, that is an instance of the resolution strategy. SLD-resolution is based on the unification algorithm and allows the retrieval of answers, i.e., the link of a value to a logical variable. It is a refinement of Robinson's resolution, that was first described by [Kow74], and whose name comes from "Selective Linear Definite clause resolution". Besides, it is a sound and complete method for the referred logic.

The declarative semantics of LP can be defined in many ways. An illustrative example is model theory, whose domain is a purely syntactic universe: the Herbrand universe.

In essence, a logic program is a set of Horn clauses. A clause has the form $A \leftarrow B_1, \ldots, B_n$, and can be considered as a part of the definition of a routine. A clause of the form $\leftarrow C_1, \ldots, C_k$ is a goal, and each C_k can be understood as a call to a routine. To execute a program is to query a goal. If the goal is $\leftarrow C_1, \ldots, C_k$, a computation step implies unifying some C_j with the head A of a clause $A \leftarrow B_1, \ldots, B_n$, thus obtaining:

$\leftarrow (C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)\theta$

where θ is a unifier substitution. Unification is, then, a mechanism for the argument passing, data selection and data building. The computation ends when the goal transits to an empty clause and there are no more literals inside of it to solve.

We introduce now some basic notions of logic programming that we use in further chapters. These concepts are addressed with more detail in [Llo87, JA07].

Let $\mathcal{V}$ be an infinite set of variables and Σ a set of function symbols f/n, each one of them with an arity n associated. $\mathcal{T}(\Sigma, \mathcal{V})$[10] and $\mathcal{T}(\Sigma)$ stand, respectively, for the set of terms and ground terms (terms with no variables) built upon $\Sigma \cup \mathcal{V}$ and Σ. The set of variables in an expression E is denoted by $\mathcal{V}ar(E)$. A term, then, is said to be ground if $\mathcal{V}ar(t) = \varnothing$.

Definition 2.2.1 ([JA07])**.** *A substitution σ is an application $\sigma : \mathcal{V} \longrightarrow \mathcal{T}$ that assigns to each variable x the set of variables $\mathcal{V}$ of a first order language $\mathcal{L}$, a term $\sigma(x)$ of the set of terms $\mathcal{T}$.*

It is usual to require $\sigma(x) \neq x$ only for a finite number of variables and, also, to express the substitution in terms of sets, identifying (in some sense) the application σ to the set of images. That is, we write $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$, where $t_i = \sigma(x_i)$ is different from x_i and each pair x_i/t_i is called "binding" or substitution element.

The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} : \sigma(x) \neq x\} = \{x_1, \ldots, x_n\}$ is said to be the domain of σ, and its range is $\mathcal{R}an(\sigma) = \{\sigma(x) : x \in \mathcal{D}om(\sigma)\} = \{t_1, \ldots, t_n\}$. Additionally, we represent by id the identity substitution, that can be understood as the set of empty bindings, so $\mathcal{D}om(id) = \varnothing$, that is, $id(x) = x$ for all $x \in \mathcal{V}$. Also, σ is said to be ground if the terms t_i are ground (the include no variables).

Definition 2.2.2. *Given an expression E and a substitution σ, $\sigma(E)$ is called* instance *and is the result of applying σ over E, replacing simultaneously all instance of x_i in E by the corresponding term t_i, being x_i/t_i an element of substitution σ.*

[10] Occasionally we only write $\mathcal{T}$.

Usually the previous instance is written $E\sigma$ instead of $\sigma(E)$. Whenever a substitution applies to the more general formulae of language $\mathcal{L}$, and not only to expressions in a clausal language, it is convenient to rename the bound variables before applying the substitution (see, [Jul04]).

Definition 2.2.3. *Given the substitutions* $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$, $\theta = \{y_1/s_1, \ldots, y_m/s_m\}$, *the composition* $\sigma \circ \theta$[11] *is the substitution determined from the set* $\sigma \circ \theta = \{x_1/\theta(t_1), \ldots, x_n/\theta(t_n), y_1/s_1, \ldots, y_m/s_m\}$, *removing the bindings* $x_i/\theta(t_i)$ *such that* $x_i = \theta(t_i)$ *and removing from* θ *the bindings* y_j/s_j *such that* $y_j \in \{x_1, \ldots, x_n\}$.

This composition verifies, over an expression E, that $(\sigma \circ \theta)(E) = \sigma(\theta(E))$ is associative and the identity substitution is the (two-sided) identity element. In the other hand, given σ, θ with $\mathcal{V}ar(\sigma) \cap \mathcal{V}ar(\theta) = \varnothing$, the union $\sigma \cup \theta$ is defined by the union set of both, that is, $(\sigma \cup \theta)(x) = \sigma(x), x \in \mathcal{D}om(\sigma)$ and $(\sigma \cup \theta)(x) = \theta(x), x \in \mathcal{D}om(\theta)$.

A substitution ρ is called renaming substitution or, simply, renaming, if there is ρ^{-1} (inverse substitution) such that $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$. Two expressions E_1, E_2 are variant if there are renaming substitutions ρ, ρ' such that $E_1 = \rho(E_2)$ and $E_2 = \rho'(E_1)$.

Composition of substitutions induces the usual preorder among substitutions: $\theta \leq \sigma$ if, and only if, there is γ that $\sigma = \theta \circ \gamma$, and we say that θ is a more general substitution than σ. This preorder induces a partial preorder over terms given by $t \leq t'$ if there is γ that $t' = t\gamma$.

Two terms t and t' are variants (one another) if there is a renaming ρ that $t\rho = t'$. Given a substitution θ and a set of variables $W \subseteq \mathcal{V}$, we denote by $\theta_{\restriction W}$ the substitution obtained from θ by restricting $Dom(\theta)$ only to the variables W. We write $\theta = \sigma\ [W]$ if $\theta_{\restriction W} = \sigma_{\restriction W}$, and $\theta \leq \sigma\ [W]$ denotes the existence of a substitution γ such that $\theta \circ \gamma = \sigma\ [W]$.

In the next definition, we address the concept of unification, that is fundamental in logic programing and automatic proving ([JA07]). In an intuitive way, to unify two expressions is to make them syntactically equal by applying over them a substitution called unifier (i.e., both expressions become equal to the instances resulting from them trough some substitution).

Definition 2.2.4. *A substitution* θ *is a unifier of the expressions* E_1, E_2 *if, and only if,* $\theta(E_1) = \theta(E_2)$.

[11] Occasionally we write only $\sigma\theta$ instead of $\sigma \circ \theta$ to abbreviate.

We can extend this definition in a very natural way to an infinite number of expressions $E_1, \ldots, E_n$, and we use, then, the unifier of the set $S = \{E_1, \ldots, E_n\}$.

Definition 2.2.5 ([JA07]). *A unifier σ of a set of expressions S is the most general unifier for S if, and only if, any other unifier θ is such that $\sigma \leq \theta$.*

We write *mgu* for the *most general unifier* of a set of expressions. The *mgu* always exists and is unique (not taking renaming into account, see [LMM88]).

To end this brief summary to logic programming, we state that the strength of this paradigm reside in its declarative component, that allows the construction of software by specifying "what" to compute instead of "how" to compute it, task delegated to the control system. Furthermore, since LP is based upon logic, it is well suited for representing knowledge and to obtain new information from the represented information. By contrast, it is not possible to represent vagueness or imprecise knowledge in LP, in principle, due to its rigid way to answer queries. This characteristic can be considered a limitation when modelling certain problems. So, in order to extend a framework as rich as logic programming to overcome this limitation, we have to exploit methods, techniques or tools to handle imprecision in an efficient way through a computer; these tools have also to adapt to the framework to be extended, that is, the extension of the framework has to be natural and, in absence of imprecision, it has to preserve the properties of the original framework. As seen in the previous section, fuzzy logic is a mathematical tool that fulfils those constraints.

Then, from the fusion of logic programming and fuzzy logic comes fuzzy logic programming, that handles imprecision and vagueness in a natural way, thus addressing that limitation of LP by integrating the well-established concepts of fuzzy logic. As we see in the next section, there are two main approaches in this paradigm. One possibility consists on the implementation of the same language enhanced to deal with imprecision [Hin86]; The other consists on extending the original language to allow the aforementioned goal [MBP87, LL90, IK85].

We focus in the next section on detailing in depth the most important notions of fuzzy logic programming, studying also the main approaches to this paradigm and classifying them by different criteria.

2.3. Fuzzy logic programming

Fuzzy logic programming (FLP) arose as an extension of LP in the same sense that fuzzy logic extends classical logic. FLP is defined formally as a part of fuzzy logic

focused on the study of fuzzy theories or fuzzy programs, that are a set of fuzzy logic expressions in a first order language directly executables in a computer.

This style of programming applies to areas where the high level of abstraction and expressiveness of traditional declarative languages is required, but those are not able to neither model vague or imprecise scenarios nor formulate approximate reasoning. To this end, new expressive resources from fuzzy logic are incorporated, as the ones mentioned in 2.1.

The area of fuzzy logic programming is in a relatively incipient state, although it is being consolidated by a growing net of researchers that provide maturity in the theory aspects as well as in the practice ones. However there are still neither standards nor a unified framework, but different approaches that take divergent paths. Part of the work of this thesis focuses on the unification of different paths of precursor research with respect to the fuzzification of the logic paradigm, as detailed in Chapter 6.

Due to this variety of schemes on FLP, it is possible to establish many classifications, analysing the procedural mechanism to deal with vagueness (Section 3.2), the extension of syntactic unification, the extension of SLD-resolution, or other considerations (see Subsection 2.3.1) where we include interesting concepts as the implementation or not of the negation in this area, or different fuzzy logics.

Many of the different approaches of fuzzy logic programming replace the classical inference mechanism, SLD-resolution, by a fuzzy variant that allows to deal with uncertainty and evaluate truth degrees. Taking this into account, and following the classification provided in [Rub11], it is possible to establish to main trends:

- The first one includes truth degrees together with facts, rules and goals and, therefore, it needs to modify the resolution mechanism to perform operations over those degrees, and unification remains intact.

- The other modifies the unification algorithm and preserves the resolution mechanism by handling truth values separately.

Thus, there is no common method to "fuzzify" the resolution principle of Prolog (see [VGM02]): the majority of these languages implement the fuzzy resolution principle introduced by [Lee72] (extended by [Muk82] and [WTL93]), like the system Prolog-Elf [IK85], Fril Prolog [BMP95] and the language F-Prolog [LL90]. Other fuzzy languages like Likelog, considered in [AF99a], or Bousi∼Prolog ([JRG08]) only contemplate the fuzzy component of predicates by introducing the notion of si-

milarity. [AG93] implements a modality of resolution conceived to manage the truth values of clauses as intervals (each boundary represents a truth and a false degree), [LL90] uses fuzzy expressions incorporating semantic hedges, [MSD89] includes expressions with an associated confidence obtained from its truth degree, and it sets the confidence of the resolvent from the confidence of the original clauses, and other systems, like [VP96, MOV01d, MOV01c, MOV04, MO04], where there are fuzzy facts and/or fuzzy rules labelling clauses with real numbers (or, more generally, elements of a lattice) representing its associated truth degree.

To summarize, in these languages there are many methods to fuzzify the knowledge, to represent it and to handle it. The soundness and completenes properties for the different types of procedural semantics has been proposed related to an appropriate declarative semantics that, in many cases, has been conceived as a fuzzy extension of the classical least Herbrand model [Llo87].

We detail now with more depth the two main trends in fuzzy logic programming with respect to the procedural mechanism, and their most representative languages, as indicated in [Rub11].

FLP extending SLD-resolution

In general, in this approach programs are a subset of clauses with an associated truth degree that is explicitly annotated. Computation and truth propagation is performed through a procedural semantic that is an extension of the classical resolution principle, while the (syntactic) unification mechanism remains untouched.

Thus, to represent vagueness in this framework, each fact, rule and goal is associated to a truth degree (for simplicity we detail here only a reduced framework of FLP that extends resolution. For a more detailed formalization, see [Voj01] or [MOV01d]). More precisely, a fuzzy logic program consists of tree parts: a fuzzy fact of the form $p(t_1, \ldots, t_n) \leftarrow [f]$, where p is a predicate symbol, each t is a term and f is a truth degree associated to $p(t_1, \ldots, t_n)$; a fuzzy rule of the form $A \leftarrow [\alpha] \langle B_1, \alpha_2 \rangle, \ldots, \langle B_n, \alpha_n \rangle$, where the truth degree of each (sub)goal B_i is α and the value of all conditions is $\Delta(\alpha, \beta)$, being Δ a t-norm or a fuzzy conjunction (see Section 2.1.1 in this chapter); and $\beta = \Delta_{i=1}^{n}(\alpha_i)$ a fuzzy goal of the form $\leftarrow [c] B_1, \ldots, B_n$, being c a constant that indicates the maximum truth degree to reach in the inference; or with the form $\leftarrow [\mathcal{F}] B_1, \ldots, B_n$, being $\mathcal{F}$ a variable to store the finally computed truth degree.

These kind of frameworks of FLP keep intact the unification mechanism and extend only the SLD-resolution mechanism to let the truth degrees associated to

each atom transit from the antecedents to the consequents until succeed or fail. Therefore, supposing a set of fuzzy clauses $\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}$:

$$\begin{aligned} \mathcal{C}_1 &\equiv \quad A \leftarrow [\alpha_0] B_1, \ldots, B_n \\ \mathcal{C}_2 &\equiv \quad B_1^{'} \leftarrow [\alpha_1] \\ &\ldots \\ \mathcal{C}_{n+1} &\equiv \quad B_n^{'} \leftarrow [\alpha_n] \end{aligned}$$

where there is a fuzzy fact $\mathcal{B}_i^{'} \leftarrow [\alpha_i]$ for each antecedent in $\mathcal{C}_1$, i.e., $B_i^{'} = B_i$. Then, we can infer the fuzzy fact $\alpha \leftarrow [\Delta_{i=0}^{n} \alpha_i]$. Now, in case of a success, together with the output, a truth degree is provided.

After these brief general considerations about fuzzy logic languages that extend the SLD-resolution, we enumerate now some of the most interesting languages in this approach:

- **Prolog-ELF**. This language [IK85] is a PROLOG system resulting from the *Fifth Generation of Programming Languages of Japan*. A Prolog-ELF program is a set of clauses associated to a truth degree in the interval $[0, 1]$. The system is based on the fuzzy resolution of [Lee72], clauses are $+A - B_1, \ldots, -B_n$ or $+A$ and goals of the form: $-B_1, \ldots, -B_n$.

  Truth values are assigned through this notation: $\alpha. + A - B_1, \ldots, -B_n$ or $assert(\alpha : +A - B_1, \ldots, -B_n)$. This notation has been adopted instead of the classical one $(A : -B_1, \ldots, B_n)$ because, according to the authors, there are many interpretations in fuzzy logic for $\neg A \vee B$, and they do not always correspond to the implication. Variables in Prolog-Elf begin by the character "$*$", and commentaries by ":". Prolog-Elf allows also to define fuzzy sets by means of special predicates that act as a membership function, i.e., that return a value between 0 and 1. Truth values in the body of a rule are combined using the t-norm "minimum" for conjunction, the t-conorm "maximum" for disjunction, and $1 - x$ for negation. Since it is based on the work of Lee, truth values of positive literals have to be in the interval $(0.5, 1]$.

- **FProlog**. FProlog [MBP87] is the name of the first implementation of Fril. The first version of this language was developed at the end of the seventies as a continuation of the works of Baldwin on fuzzy relations. The second version was released in the first years of the eighties. Both versions can be considered related languages. FProlog uses Lisp syntax instead of PROLOG syntax.

In order to provide the system with deductive capabilities, a PROLOG interpreter was integrated in its structure. The system could work as an autonomous PROLOG system. FProlog also allowed the definition of fuzzy relations, with a truth degree associated to each tuple. To compute the truth degree it uses fuzzy counterparts of conjunction, disjunction and negation.

- **Fuzzy Prolog**. Was created by Mukaidono [MSD89]. It extends Lee's approach to allow truth values in the interval $[0, 1]$. This work brought its first proposal about a Fuzzy Prolog, consolidated in [MSD89] by the introduction for the first time of the notion of *fuzzy first order predicate*, whose variables can be membership functions over a fuzzy subset (this idea has been acquired by more modern systems, like the fuzzy module of Ciao-Prolog of [GMV04]).

  The most recent contribution of Mukaidono is a fuzzy PROLOG based on the Łukasiewicz implication (LbFP). In LbFP there are no changes in the unification process. Facts and rules are extended with truth values in the interval $[0, 1]$ where 1 represents truth and 0 unknown or absurd. Truth values in the body of a rule are combined using the "minimum" t-norm for the conjunction, the "maximum" t-conorm for the disjunction and $1 - x$ for the negation. For a certain solution this framework takes the maximum of the computed values. Resolution in LbFP takes the shape of a tree in contrast with the classical linear resolution.

- **f-Prolog**. f-Prolog was created by [LL90]. An f-Prolog program differs only from a PROLOG program in the truth degree that some facts carry and in the degree of the implication, that is associated to the rules. An f-Prolog program is a sequence of f-facts, f-rules and an f-goal. An f-fact $\mathcal{A}$ is f. An f-rule has the form $A - [f] - B_1, \ldots, -B_n$, where A, B_i are atoms and the degree of the conclusion A is the product of α and the minimum of the truth values associated to each B_i. An f-goal is an expression $-[F] - Q_1, \ldots, -Q_n$ being F a variable. The system also allows to specify the minimum degree that a goal has to satisfy: for instance, it is possible to query the system solutions with more than 0.8 truth degree.

- **Rfuzzy**. RFuzzy language is an extension and enhancement of the fuzzy module Ciao-Prolog presented in [GMV04]. This module uses the union of subsets of truth degrees to handle imprecision. These truth degrees are combined by means of aggregation operators. Additional arguments can be used in each

clause for managing these degrees. The implementation is based on a process of compilation that translates fuzzy programs to PROLOG code. Furthermore, this module uses the notion of fuzzy first order predicate introduced by Mukaidono in [MSD89].

One of the problems the authors of RFuzzy points out to the module Ciao is its complexity [MCS09]. This complexity is due to the use of real intervals to represent truth degrees. Besides, answers are associated to constraints, and the management of variables that carry truth degrees is problematic. RFuzzy reduces this complexity in some aspects: it uses real numbers instead of intervals to represent truth values; it answers with direct values instead of with constraints; and it removes the necessity of including variables to manage truth degrees.

- **Multi-adjoint language**. In multi-adjoint logic programming (MALP in brief) [MOV01d, MOV01c, MOV01b, MOV01a], each program is associated with a certain lattice that provides truth degrees and allows to encapsulate different types of fuzzy logics inside each rule. Given a MALP program, goals are evaluated in two separated computational phases. During the resolution phase, the fuzzy counterpart of SLD-derivation steps, called admissible steps, are applied. This first phase produces a substitution and an expression where all atoms have been exploited. This last expression is interpreted afterwards (in the so-called interpretative phase) in the multi-adjoint lattice associated to the program, thus obtaining a pair (*truth degree; substitution*) that is the fuzzy analogous to the classical notion of computed answer traditionally used in LP.

  Furthermore, it is noteworthy that the framework based on similarity presented in [Ses02] can be emulated by means of a certain multi-adjoint lattice [MOV04], particularly, the real interval $[0, 1]$ with the t-norm of Gödel, and extending the original program with a set of rules defining a similarity relation (in a similar way to the extension of first order logic with equality axioms). This illustrates the generality and expressiveness of the multi-adjoint logic language.

  From our point of view, the features of this language makes it one of the most powerful and interesting in the area of fuzzy logic programming. By this reason we have chosen it as a framework to develop our research in this field. In Chapter 3 we expose in depth the syntax, semantics and further basic notions of the multi-adjoint approach.

Extension of unification in FLP

The second trend in FLP is represented by languages that replaces the syntactic unification mechanism (of classical SLD-resolution) by a fuzzy unification algorithm. While the global mechanism of resolution remains untouched, unification changes dramatically since it can "unify" different predicates (provided they have some similarity degree).

The main goal of this framework is to obtain more flexible unification mechanisms. There are, at least, three approaches:

- Semantic unification ([AG98]): each constant is associated to a meaning (e.g., a fuzzy subset) and the unification is based on that meaning (e.g., by a *matching* procedure);
- Unification based on distance edition ([GS00]): the unification degree is based on a syntactic similarity between two symbols (e.g., house, mouse);
- Weak unification: it is the fuzzy extension of classical unification.

From now on, we focus on weak unification, since both semantic unification and unification based on distance edition can be implemented on weak unification.

Generally speaking, weak unification consists on the substitution of the syntactic equality by a similarity (or proximity) relation. This similarity relation $\mathcal{R}$ relates symbols of the alphabet of a first order language $\mathcal{L}$, that is, $\mathcal{R} = \mathcal{R}_{\mathcal{P}} \cup \mathcal{R}_{\mathcal{F}} \cup \mathcal{R}_{\mathcal{V}}$ where $\mathcal{R}_{\mathcal{V}}$ is a similarity on $\mathcal{V}$ defined by (1) $\mathcal{R}_{\mathcal{V}}(x, y) = 0$ if $x \neq y$ being $x, y \in \mathcal{V}$; (2) $\mathcal{R}_{\mathcal{V}}(x, x) = 1$. Furthermore, $\mathcal{R}_{\mathcal{F}}$ is such that, given two symbols $f, g \in \mathcal{F}$ with the same arity n, belonging to the alphabet of $\mathcal{L}$, $\mathcal{R}_{\mathcal{F}}$ relates both f and g, and that relation is quantified by an approximation degree in a way that can be read as "f is similar to g with approximation degree α", with $\alpha \in (0, 1]$.

A fuzzy program, in this framework, is composed by a set of clauses and a relation between the symbols in the alphabet of $\mathcal{L}$. This kind of languages, in FLP, need to modify the unification algorithm so it copes with the given relations. In this framework the unification algorithm does not end when two terms differ in some symbol, but it searches a relation between them. Furthermore, SLD-resolution has to be adapted to allow the composition of the approximation degrees obtained in each resolution steps. Then, supposing a set of clauses $\mathcal{C}_1, \ldots, \mathcal{C}_{n+1}$:

$$\begin{aligned}
\mathcal{C}_1 \equiv \quad & A \leftarrow [\alpha_0] B_1^{'}, \ldots, B_n^{'} \\
\mathcal{C}_2 \equiv \quad & B_1 \leftarrow \\
\ldots & \\
\mathcal{C}_{n+1} \equiv \quad & B_n \leftarrow
\end{aligned}$$

where there is a fact $\mathcal{B}_i^{'} \leftarrow$ similar to an antecedent in $\mathcal{C}_1$, i.e.,

$$\mathcal{R}(\mathcal{B}_1^{'}, \mathcal{B}_1) = \alpha_1, \ldots, \mathcal{R}(\mathcal{B}_n^{'}, \mathcal{B}_n) = \alpha_n$$

Then we infer $\langle A, \beta \rangle$, with $\beta = \Lambda_{i=1}^{n} \alpha_i$.

We present now the basic features of the main fuzzy logic languages based on similarity.

- LIKELOG: this language, whose name is an acronym of "Likeness in Logic", is the practical implementation of the works of [FGS00]. This language is based on the theoretical concept of cloud. A cloud is a set of elements that can be considered similar with respect to their meaning [AF99a]. According to the authors, LIKELOG is an environment of logic programming implemented in *Dec-10 Standard Prolog*. The language is able to handle flexible queries and answers. Its syntax $\mathcal{L}'$ is an extension of a first order language $\mathcal{L}$ with no function symbols where similarity is introduced by a fuzzy relation $\mathcal{R} = \mathcal{R}_{\mathcal{C}} \cup \mathcal{R}_{\mathcal{V}} \cup \mathcal{R}_{\mathcal{P}}$ on the symbols of the alphabet, and it fulfils the reflexive, symmetric and transitive properties. Its procedural semantics is a fuzzy extension of the unification algorithm associated to the resolution, while its fixpoint semantics is given by an extension of the classical least Herbrand model. One of the applications of LIKELOG is its flexibility on handling answers and queries in deductive databases.

- SiLog: this language ([LSS01]) is based on the work [Ses02]. The SiLog system is implemented on a W-Prolog interpreter built in Java (for which an online reference is `http://waitaki.otago.ac.nz/~michael/wp/`). It is composed by two parts: an inference engine and a similarity manager. The inference engine handles the weak unification, while the similarity manager generates a similarity from the quotient sets (families of λ-cuts) from the elements included in a dictionary incorporated by the user. The similarity manager ensures that the built relations fulfil the definition of similarity, which means that the extension is a little rigid and limited. Similarity relations are addressed independently to the program.

- Bousi~Prolog ([JRG08, JRG09a, JR09b, JR10a]): this language is one of the most recent and interesting languages in this tendency. This language is the practical implementation of the theoretical works of [JR06b, JR06a]. Its procedural semantics [JR09a] adapts the principle of SLD-resolution and its unification algorithm is based on proximity relations (i.e., a binary fuzzy reflexive and symmetric relation on a set, not necessarily transitive). It generalizes the procedural mechanism of Sessa, based on similarity relations, and enhances the expressive power of the resulting language. The properties of reflexivity and symmetry are very appropriate to express the value of "proximity" between elements of a domain. The authors of this language can model problems in cases where the transitive restriction of the similarity relation is an obstacle.

  The syntax of Bousi~Prolog is, basically, Prolog syntax enriched by a constructor symbol $\sim$ used to describe proximity relations (in fact, fuzzy binary relations that are translated automatically to proximity or similarity relations) by means of what authors call a proximity equation. Those equations are expressions of the form *symbol* $\sim$ *symbol* = *proximity value*. This operator is the fuzzy counterpart of the syntactic unification operator '=' of Prolog. Furthermore, the language supports the use of fuzzy sets [JRG09b, JRG10, JR10b], integrating them easily in the core of the system by the use of fuzzy relations, not affecting the procedural semantics.

  On the other hand, the authors of Bousi~Prolog have created UNICORN [JR09c], a programming environment that allows to edit, compile and run Bousi~Prolog programs, according to the principle of weak SLD-resolution described for this language. The complete implementation of this tool consists of 900 lines of Prolog code, approximately, and it includes more useful options for programmers. A more complete specification of the tool can be found in `http://dectau.uclm.es/bousi/`.

2.3.1. Other considerations

To end this section on FLP, we include here important concepts about this paradigm, as presented in the PhD thesis [Pen10]. For example, together with the classification provided in previous sections, it is also possible to differentiate between languages based on annotations from languages based on implications (see [LS01b, LS05]).

- Languages based on annotations (see [KL88, KS92, NS91, NS92, Lu96, Cao00,

KLV02]), admit rules of the form $A : f(\beta_1, \ldots, \beta_n) \longleftarrow B_1 : \beta_1, \ldots, B_n : \beta_n$ whose meaning can be understood as "the truth degree of A is, at least, $f(\beta_1, \ldots, \beta_n)$, if the truth degree of each atom B_i is at least β_i", $1 \leq i \leq n$, being f a computable function and β_i a constant of a variable over an appropriate domain (set of truth values).

- In contrast, those languages based on implications (see [DMO04a, DM04, LS94, LS01b, MOV01d, MOV01c, MOV01b, MOV01a, MO02, MOV04, MO04, DMO07, vE86, Voj01]) have rules of the form $\langle A \leftarrow @(B_1, \ldots, B_n);\ v \rangle$ where v is the truth degree associated to formula $A \leftarrow @(B_1, \ldots, B_n)$, where @ is a connective that combines atomic expressions B_i.

  Computationally, given an interpretation $\mathcal{I}$, truth values $\mathcal{I}(B_i)$ are determined by the truth function of connective @ and, afterwards, propagated to atom A in the head of the rule. Also, truth degrees can be taken from a lattice, that is, the application $\mathcal{I}$ can be mapped to values on a certain lattice. [DMO04a, KLV04, LS01b, Voj01] show that the majority of the frameworks that manages imprecision can be implemented in this context.

On the other hand, it is noteworthy that the majority of approaches do not include any kind of non-monotonic reasoning, neither admit negation. It is not the case of [DM01a, LS06, LS02a, LS02b, LS03, Str05a, Str05b], where the domain of truth values is a lattice. The language of [VGM02, GMV04] also admits negation, but here truth degrees are real intervals [KY95], which are very adequate to formalize linguistic variables as "age" or "speed".

With respect to negation, it is the most relevant logic concept not originally coped by fuzzy logic programming. This is because its inclusion leads to a significant complexity. However, negation plays an important role on representing knowledge and many of its applications cannot be emulated by positive programs. Negation is also useful in the management of databases, program composition, reasoning by default, etc. [Ger05] can, in the context of control techniques, address positive and negative information if truth values are taken from a bilattice.

Furthermore, as already stated in this introductory chapter, there is no consensus about which fuzzy logic corresponds to which context. The majority of the systems uses a min-max logic (to model conjunction an disjunction), but some systems uses Łukasiewicz logic [KK94]. Other approaches allow a more generic interpretation of connectives [VP96], and the multi-adjoint framework allows also different logics to

model connectives.

Finally, with respect to the set where the interpretation of formulae take place[12], there are fuzzy logic programs interpreted in:

1. The real interval $[0, 1]$, as the case of [MSD89, vE86, Sha83, VP96, Voj01, AF99a, KLV04].

2. A lattice, as [MOV01d, MOV01c, MOV01b, MO02, MOV04, MO04] (multi-adjoint lattice) and also [DM00, DM01b, DM02, DMO04b] (residuated lattice).

3. A bilattice [Gin88, Fit91, Ger05, LS04, Str05b], trilattice [LS01a] or, more generally, a multilattice [MOR05, Mor06b, MOR06a, MOR07c, MOR07b, MOR07a].

4. A set of intervals, like [VGM02, GMV04, LS01a, Luk01, AG93].

5. A qualification domain, as the case of [CRR08, RR08b, RR09].

To conclude, fuzzy logic programming is a research area in constant expansion and promising perspectives, whose application in the form of fuzzy logic languages can greatly help to codify systems with fuzzy features in the fields of most solid implantation of fuzzy logic, as seen in Section 2.1.2 (the construction of expert systems, control applications, *soft computing*, etc.).

As already stated, from our point of view the most interesting context is the multi-adjoint programming, due to its great generality (it can implement many of the other fuzzy logic schemes), its high level of expressiveness and is clearly defined procedural semantics. We detail in Chapter 3 the essential concepts of the multi-adjoint framework.

2.4. Multi-adjoint lattices

To conclude this initial chapter, we include this section devoted to the theory of lattices and, more specifically, the description of lattices that can be associated to programs, as well as some basic results. Lattices are (in our scope) ordered sets where the formulae of programs are interpreted in some contexts of fuzzy logic programming.

[12] In other contexts more general structures have been used to interpret formulae, like algebraic domains (see [RZ01, Sco82]).

In particular, we are interested in the lattices used in multi-adjoint logic programming, developed in [MOV04, MOV01d, MOV01c], since this is the language we focused on in our research. In this section we formally define the notion of multi-adjoint lattice, and we proof theoretically the multi-adjointness of diverse lattices, ensuring they fulfil all the required properties. This work was presented in [MMPV12a] and [MMPV12b]. In Chapter 4 we detail the way these lattices are introduced in our fuzzy logic programming tool (FLOPER), while Chapter 5 reports some of the powerful applications that the use of the different multi-adjoint lattices we present now allows.

In essence, a multi-adjoint lattice consists in an ordered lattice L equipped with a set of connectives, like implications, conjunctions, disjunctions and aggregators, with the particularity that each implication symbol has an adjoint conjunction used to model the inference rule of *modus ponens* in a fuzzy environment. For instance, we expose now some of the *adjoint pairs* (conjunctions and implications) in the lattice $([0,1], \leq)$ (that we denote by $\mathcal{V}$). The labels `P`, `G` y `L` stand for, respectively, *Product logic* (1), *Gödel logic* (2) and *Łukasiewicz logic* (3). Each logic is suited to model *pessimistic*, *optimistic* and *realistic* scenarios, respectively.

$$
\begin{array}{llll}
(1)\ \&_{\mathtt{P}}(x,y) \triangleq x * y & & \leftarrow_{\mathtt{P}}(x,y) \triangleq \text{mín}\{1, x/y\} \\
(2)\ \&_{\mathtt{G}}(x,y) \triangleq \text{mín}\{x,y\} & & \leftarrow_{\mathtt{G}}(x,y) \triangleq \begin{cases} 1 & \text{si } y \leq x \\ x & \text{otherwise} \end{cases} \\
(3)\ \&_{\mathtt{L}}(x,y) \triangleq \text{máx}\{0, x+y-1\} & & \leftarrow_{\mathtt{L}}(x,y) \triangleq \text{mín}\{x-y+1, 1\}
\end{array}
$$

Furthermore, in the context of the multi-adjoint logic programming language (or MALP), each program is associated to its own multi-adjoint lattice, and each program rule is "weighted" by an element of L. Furthermore, the components in the bodies are linked by connectives in the lattice. Those rules are similar to PROLOG clauses [Llo87]. See, for instance, the following propositional MALP program (where $@_{\mathtt{aver}}$ means evidently the classical aggregator of the arithmetical average):

$$
\begin{array}{llll}
p & \leftarrow_{\mathtt{P}} & @_{\mathtt{aver}}(q,r) & with \quad 0.9 \\
q & \leftarrow & & with \quad 0.8 \\
r & \leftarrow & & with \quad 0.6
\end{array}
$$

the last two rules[13] assign directly truth degrees 0.8 and 0.6 to propositional variables q and r, respectively; the execution of p through the first rule (whose *body* "calls"

[13] In Section 3.1 we describe the syntax of MALP programs.

the other rules) consists on the evaluation of the expression "$\&_{\mathtt{P}}(0.9, @_{\mathtt{aver}}(0.8, 0.6))$", that returns the final truth degree 0.63.

Now we focus on the theoretical results that guarantee that certain lattices (presented afterwards) are multi-adjoint lattices, as well as the cartesian product of them. This kind of more sophisticated lattices can be associated to MALP programs not only to obtain the truth degrees of (possible) answers, but also to keep track of the run time in terms of computational steps performed in order to reach each answer. This feature is detailed in Section 3.6 of this memory.

We provide now the formal definitions of adjoint pair and multi-adjoint lattice from [MOV01d] and [MOV01c]:

Definition 2.4.1. *Let $(L, \leq)$ a lattice. A* multi-adjoint lattice *is an n-uple $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ that fulfils these conditions:*

1. *$(L, \leq)$ is bounded (that is, there are the infimum $\bot$ and supremum $\top$ elements of $(L, \leq)$).*
2. *$(L, \leq)$ is complete*[14]*, i.e., for all subset $X \subset L$ there are $inf(X)$, $sup(X)$*[15]*.*
3. *$\top \,\&_i\, v = v \,\&_i\, \top = v$, $\forall v \in L$, $i = 1, \ldots, n$.*
4. *each pair $(\leftarrow_i, \&_i)$, with $i = 1, \ldots, n$, is an* adjoint pair*, that is*
   a) *the operation $\&_i$ is non-decreasing in both arguments*[16]*.*
   b) *the operation $\leftarrow_i$ is non-decreasing in the first argument and non-increasing in the second one.*
   c) adjoint property*: $\forall x, y, z \in L$, $x \leq (y \leftarrow_i z) \Leftrightarrow (x \&_i\, z) \leq y$.*

With respect to the adjoint property, from now on x represents the truth degree of the rule $y \leftarrow_i z$, so this rule holds (the corresponding interpretation satisfies such rule) if, and only if, the truth degree y of the head is greater or equal the one of the adjoint conjunction of z (the truth degree of the hypothesis) with x (the truth

[14] Indeed, if $(L, \leq)$ is complete, then it is bounded and we can reformulate the definition not requiring explicitly the bounded feature.

[15] Take into account that the existence of $inf(X)$, $sup(X)$ is guaranteed when X is finite, since L is a lattice.

[16] We do not require neither commutativity (see[KLM$^+$02], for example, where the absence of commutativity is a particular feature of one of the application context for the multi-adjoint logic programming) nor associativity for this connective.

degree of the rule). The so-called *adjoint property* is the most relevant feature in this context (in contrast with many other approaches), that justifies many of the crucial results of soundness, completeness, applicability, etc.

The notion of adjoint pair was first introduced in the logic context by [Pav79], which takes truth degrees for a category, and the relation between each conjunction and its associated implication as a functor in that category. This notion is a new example of the concept of adjunction defined in 1950 by Kan in the general theory of categories.

As reported in [MOV04], residuated lattices (see [Dil39, Ger04, NPM99]) are examples of multi-adjoint lattices in which there is only one adjoint pair and the conjunction of the pair determines a monoid structure (with identity element $\top$) in the underlying lattice.

Also, [DMO04c, DMO04a, DMO07] contemplates a multi-adjoint programming framework (of typified multi-adjoint logic programs) where in a very suggestive way many multi-adjoint lattices are combined. This flexibility allows to associate to each program rule two (many, in general) truth degrees, one of them representing, e.g., the certainty factor and the other the doubt factor of the rule.

Finally, recall that symbols $\&_i$ (adjoint conjunction) and $\alpha \in L$ (truth degree in L) must belong to the language $\mathcal{L}$ so it is possible to define a procedural mechanism for the multi-adjoint logic language (see Section 3.2).

Note here that the notion of *qualification domain* used in the framework of QLP (*Qualified Logic Programming*) described in [RD08], plays a role similar to the one of multi-adjoint lattices in MALP. A qualification domain is a structure $\langle D, \sqsubseteq, \bot, \top, \circ \rangle$ such that $\langle D, \sqsubseteq, \bot, \top \rangle$ is a lattice with supremum ($\top$) and imfimum ($\bot$) elements and where the so-called *attenuation operation* "$\circ$" is a conjunction. Given two elements $d, e \in D$, $d \sqcap e$ is the *inf{d,e}*, while $d \sqcup e$ represents the *sup{d,e}*. We also write $d \sqsubset e$ as an abbreviation of $d \sqsubseteq e \wedge d \neq e$. The attenuation operator $\circ$ fulfils the conditions below:

1. $\circ$ is associative, commutative and monotonous with respect to $\sqsubseteq$.
2. $\forall d \in D, d \circ \top = d$.
3. $\forall d \in D, d \circ \bot = \bot$.
4. $\forall d, e \in D - \{\bot, \top\}, d \circ e \sqsubset e$.
5. $\forall d, e_1, e_2 \in D, d \circ (e_1 \sqcap e_2) = d \circ e_1 \sqcap d \circ e_2$.

It should be noted that the required properties in QLP and MALP are similar. However, the distributivity (*5.*) and the adjunction are quite different features.

There is, though, the possibility of interchanging concepts between both frameworks, as is the boolean qualification domain $\mathcal{B} = (\{0,1\}, \leq, 0, 1, \wedge)$, whose aspect as a multi-adjoint lattice is $(\{0,1\}, \leq, \leftarrow, \wedge)$, where $\wedge$ is the boolean conjunction and $\leftarrow$ its adjoint implication (i.e., the usual logic implication). Other example is the qualification domain of the *uncertainty values of Van Emden* used in QLP $\mathcal{U} = ([0,1], \leq, 0, 1, \times)$, for which there are many possible adjoint pairs, as considered at the beginning of the section. This lattice will be fully used in many examples in this memory, as well as the so-called *domain of weights*, $\mathcal{W} = (\mathbb{N} \cup \{\infty\}, \geq, \infty, 0, +)$, whose detailed description we also expose further.

In general, every qualification domain $\langle D, \sqsubseteq, \bot, \top, \circ \rangle$ whose attenuation operation $\circ$ conforms an adjoint pair with a given implication operator $\leftarrow_\circ$, can be expressed as the multi-adjoint lattice $(D, \sqsubseteq, \leftarrow_\circ, \circ)$. Anyway, we end this brief comparison highlighting that the variety of connectives admitted in multi-adjoint lattices is much greater than in the qualification domain (where for a given program, all rules must use the same attenuation operator), which justifies the greater expressive power of MALP w.r.t. QLP.

We focus now on the classical notion of cartesian product of multi-adjoint lattices, that inherits the multi-adjoint character by the next result:

Theorem 2.4.2. *If $L_1, \ldots, L_n$ are multi-adjoint lattices, then their cartesian product $L = L_1 \times \cdots \times L_n$ is also a multi-adjoint lattice.*

Proof. *With no loss of generality, we detail the result for $n = 2$. We consider, then, multi-adjoint lattices $(L_1, \leq_1, \&_1, \leftarrow_1)$ and $(L_2, \leq_2, \&_2, \leftarrow_2)$, each of them equipped with only one adjoint pair. $L = L_1 \times L_2$ has lattice structure with order induced in the product $L = L_1 \times L_2$ given by: $(x_1, y_1) \leq (x_2, y_2) \Leftrightarrow x_1 \leq_1 x_2,\ y_1 \leq_2 y_2$. Also, being $\top_1 = sup(L_1),\ \bot_1 = inf(L_1),\ \top_2 = sup(L_2)$ and $\bot_2 = inf(L_2)$, we have that $(\top_1, \top_2) = sup(L)$ y $(\bot_1, \bot_2) = inf(L)$, from which follows that the cartesian product L is a bounded lattice is L_1 and L_2 are also bounded.*

Analogously, it is easy to justify that $L_1 \times L_2$ is complete if L_1 y L_2 are complete.

Finally, from adjoint pairs $(\&_1, \leftarrow_1)$, $(\&_2, \leftarrow_2)$ of L_1 and L_2, respectively, it is possible to define the following connectives in L: $(x_1, y_1)\&(x_2, y_2) \triangleq (x_1 \&_1 x_2, y_1 \&_2 y_2)$ and $(x_1, y_1) \leftarrow (x_2, y_2) \triangleq (x_1 \leftarrow_1 x_2, y_1 \leftarrow_2 y_2)$, and it is easy to justify that the form an adjoint pair in $L_1 \times L_2$ (in particular, they fulfil the adjoint property).

Analogously, it is also possible to define new connectives (conjunctions, disjunctions and aggregations) in the cartesian product $L_1 \times L_2$ from the corresponding pairs of operators in L_1 and L_2.

Once provided that the cartesian product of multi-adjoint lattices is also a multi-adjoint lattice, we consider again the so-called *domain of weighted values* $\mathcal{W}$ using the paradigm QLP (*Qualified Logic Programming*) of [RD08, CRR08, RR09, RR08a]). As previously noted, although it shares similarities with MALP, the QLP scheme admits a lesser repertoire of connectives in the body of program rules. Its elements are intended to represent the *cost of the proof*, i.e., measures of the weighted depth of the proof trees. In essence, and as proven in the contribution [MMPV12a], $\mathcal{W}$ is a lattice $(\mathbb{N} \cup \{\infty\}, \leq)$, where $\leq$ is the inverted ordering (with $\infty \leq d$ for all $d \in \mathbb{N}$), so ∞ is the imfimum and 0 is the supremum. Furthermore, in this lattice the arithmetic operation "+" is a t-norm, since it fulfils the properties of t-norms (see [NW06]). Also, we can obtain the residual implication of the t-norm "+", defined in general by $y \leftarrow z \triangleq sup\{t \in \mathcal{W} : t + z \leq y\}$ that, in this particular case, has the following expression:

$$y \leftarrow z \triangleq \begin{cases} y - z, & si\ y \leq z \\ 0, & otherwise \end{cases}$$

It is not difficult to prove that $(+, \leftarrow)$ conforms an adjoint pair in $(\mathbb{N} \cup \{\infty\}, \leq)$, and that $(\mathcal{W}, \geq, \leftarrow, +)$ is, in fact, a multi-adjoint lattice.

If we call $\mathcal{V}$ the multi-adjoint lattice $([0, 1], \leq)$, equipped with two adjoint pairs modelling the implication and conjunction symbols from the logics of *Łukasiewicz* and *product*; then, for the cartesian product $\mathcal{V} \times \mathcal{W}$ the maximum is $(1, 0)$ and the infimum is $(0, \infty)$, and it also has the following definitions for the conjunction (among others) connectives whose names refer to *extensions* of Product and Łukasiewicz logics:

$$(v_1, w_1)\ \&_{\text{P}+}\ (v_2, w_2) \triangleq (v_1 * v_2, w_1 + w_2)$$

$$(v_1, w_1)\ \&_{\text{L}+}\ (u_2, w_2) \triangleq (\text{máx}\{0, v_1 + v_2 - 1\}, w_1 + w_2)$$

In short, the cartesian product $\mathcal{V} \times \mathcal{W}$ is a multi-adjoint lattice. In practice, we use that lattice to compute the number of computational steps performed to reach an answer in multi-adjoint logic programs. As already stated, this feature is better explained in Section 3.6.

Furthermore, we go a step beyond simply counting the number of steps in an execution. Indeed, we can know more detailed information about the set of program

rules and connectives evaluated to obtain a solution, although it is necessary the use of the new lattice $\mathcal{S}$ to do so. This lattice is based on *strings*, or *labels* (that is, sequences of characters), so we generate the cartesian product $\mathcal{V} \times \mathcal{S}$. To this end, firstly we prove that not only is $\mathcal{S}$ a multi-adjoint lattice, but also that the concatenation operation, usually called *append* in many programming languages, can play the role of an adjoint conjunction in that lattice. The results of this research were presented by the first time in [MMPV12b].

To solve both questions, we analyse many alternatives to establish an ordering relation in $\mathcal{S}$, as the classical lexicographical order (that is usual to order words in dictionaries), or an ordering based on prefixes, *sub-strings*, etc. (for instance, 'ab' and 'bc' are respectively a prefix and a sub-string of 'abcd'). Unfortunately, neither of these orderings allow *append* to act as a t-norm.

However, we conceived an alternative for which $\mathcal{S}$ can be seen as a multi-adjoint lattice. The key point is to translate each string of characters into a unique number by associating to each character its corresponding ASCII code. Thus, it is possible to establish a bijective application $[\ ] : \mathcal{S} \to \mathbb{N}$.

Let $A = \{a_0, \ldots, a_{n-1}\}$ be a set called *alphabet*, whose elements can be seen as symbols. A string s over A is a finite sequence of elements of this set, that is, $s = a_1 \ldots a_m$, where $a_i \in A$, $i = 1, \ldots, m$. The set of strings over A, denoted by S, can be formally defined as $S = \cup_{k \in \mathbb{N}} A^k$. This definition of S guarantees its numerable character, i.e., S contains a (numerable) infinite number of *strings* $a_1 \ldots a_m$, formed by elements a_1 of A. Although the mentioned numerable character is obtained from well known set-theoretical results, Theorem 2.4.3 justifies it by formalising a bijection (as well as its reverse) of S on $\mathbb{N}$, which have a great relevance in the multi-adjoint scope.

Each element of A^k is a string of k elements (of length k) that can be seen as a word; in that case, S can be understood as the set of all words with a finite length. This interpretation is very interesting since each formal language with alphabet A is the set of formulae built from elements of words of S that, also, fit tu the syntactic rules for that language. So all language with alphabet A is, consequently, a subset of S.

Furthermore, in the set S we define the concatenation, called $\cdot$, as the operation

$$\cdot : A^p \times A^q \to A^{p+q}$$

$$(s, t) \mapsto s \cdot t$$

so, given $s = a_1 \dots a_p, t = b_1 \dots b_q$, then $s \cdot t = a_1 \dots a_p b_1 \dots b_q$.

Let $cod : A \to Im(cod)$ and $asc : Im(cod) \to A$ be primitive functions, so $cod(a_i) = i$ and $asc(i) = a_i$. It is trivial to prove that cod is the inverse of asc and reciprocally. The symbol $''$ represents the empty string (with length 0). We define the application $[\ ] : S \to \mathbb{N}$ through rules R_1 and R_2:

$$\begin{aligned} R_1 :&\quad [''] = 0 \\ R_2 :&\quad [a_1 \dots a_m] = (cod(a_1) + 1)n^{m-1} + \dots + (cod(a_m) + 1)n^0 \end{aligned}$$

where $a_i \in A, \forall i$, and $a_1 \dots a_m \in S$. Rule R_2 can be rewritten as:

$$R_2^* : \quad [s.a] = [s]n + cod(a) + 1$$

where $s.a$ means a string obtained by adding symbol a at the end of string s, $s.a = a_1 \dots a_m.a = a_1 \dots a_m a$, and n is the number of elements of A.

To illustrate the definition of this application, consider the alphabet $\{a, b, c\}$ and the string $s = cab$. Then $[cab] = (cod(c) + 1)3^2 + (cod(a) + 1)3 + (cod(b) + 1) = 3 * 9 + 1 * 3 + 2 = 32$.

Theorem 2.4.3. *Application $[\ ]$, defined by R_1 y R_2 (o R_2^*), is bijective.*

Proof. *Indeed, it is an application since each string $s \in S$ is associated to a natural number. In order to prove the bijective character of $[\ ]$, we prove here that its reverse function is the application $<>: \mathbb{N} \to S$, defined as:*

$$\begin{aligned} &< 0 >='' \\ &< m >=< \lfloor (m-1)/n \rfloor > .asc((m-1)\,\%n) \end{aligned}$$

where $\lfloor\rfloor : \mathbb{R} \to \mathbb{N}$ is the function integer part, so $\lfloor (m-1)/n \rfloor$ is the integer part of the quotient, and $(m-1)\,\%n$ is the remainder n of the integer $m-1$.

We see that the compositions $(<> \circ [\ ]) : S \to S$ and $([\ ] \circ <>) : \mathbb{N} \to \mathbb{N}$ coincide with the identity application (id_S y $id_{\mathbb{N}}$, respectively), which justifies that $[\ ]$ is the reverse of $<>$ and reciprocally.

Note in the first place that $(<> \circ [\ ])(s) = s, \forall s \in S$.

1. $(<> \circ [\ ])('') =< [''] >=< 0 >=''$
2. $(<> \circ [\ ])(s.a) =$
   $< [s.a] >=< [s]n + cod(a) + 1 >=$
   $< \lfloor ([s]n + cod(a) + 1 - 1)/n \rfloor > .asc(([s]n + cod(a) + 1 - 1)\,\%n) =$

$$< \lfloor([s]n + cod(a))/n\rfloor > .asc(([s]n + cod(a))\,\%n) =$$
$$< \lfloor[s]n/n\rfloor > .asc(cod(a)\,\%n) =< \lfloor[s]\rfloor > .asc(cod(a)) =$$
$$< [s] > .a$$

That is, $< [s.a] >=< [s] > .a$. *Consequently, given* $s \in S$, *if* $s = a_1 \dots a_m$, *then* $< [a_1 \dots a_m a] > = < [a_1 \dots a_m] > .a = \cdots =< [''] > .a_1 \dots . a_m.a = s.a = id_S(s.a)$, *where* $id_S : S \to S$ *is the identity application of* S, *so* $(<> \circ [\,]) = id_S$.

We also prove that $([]\circ <>)(j) = j, \forall j \in \mathbb{N}$:

1. $([]\circ <>)(0) = [< 0 >] = [''] = 0$

Since $j > 0$ *can be expressed as* $xn + y$, *where* $x, y \in \mathbb{N}, 0 < y \leq n$, *if* $j > 0$, *we have that*

2. $([\,]\circ <>)(j) =$
   $([\,]\circ <>)(xn + y) = [< xn + y >] =$
   $[< \lfloor(xn + y - 1)/n\rfloor > .asc((xn + y - 1)\,\%n)] =$
   $[< \lfloor(xn + y - 1)/n\rfloor >]n + cod(asc((xn + y - 1)\,\%n)) + 1 =$
   $[< \lfloor xn/n\rfloor >]n + cod(asc(y - 1)) + 1 =$
   $[< x >]n + (y - 1) + 1 =$
   $[< x >]n + y$

So, $[< xn + y >] = [< x >]n + y, 0 < y \leq n$. *Since* $j \in \mathbb{N}$, *if* $j \neq 0$, *can be expressed as* $j = y_1 n^{m-1} + \cdots + y_{m-1}n^1 + y$, *with* $y_1, \dots, y_{m-1} \in \{1, \dots, n\}$, *we have that* $[< j >] = [< y_1 n^{m-1} + \cdots + y_{m-1}n^1 + y >] = [< y_1 n^{m-2} + \cdots + y_{m-1}n^0 >]n + y = \cdots = [< 0 >]n + y_1 n^{m-1} + \cdots + y_{m-1}n^1 + y = j$. *Then,* $[<>] = id_{\mathbb{N}}$, *by 1. and 2.*

Consequently, if $[\,]$ *is the reverse function of* $<>$ *(and reciprocally), both of them are bijective functions, as wanted.*

By the previous theorem, sets $\mathbb{N}$ and S are bijective and, hence, the corresponding ordered structures are isomorphic lattices (both of them multi-adjoint lattices).

In Example 2.4.4 we illustrate these applications with a reduced alphabet, while in Example 2.4.5 we work with a very common environment to provide an idea on the usefulness of this approach.

Example 2.4.4. *Consider the alphabet* $A = \{a, b, c\}$. *Then, the string of characters associated to the integer* 32 *is* $< 32 >=< \lfloor 31/3\rfloor > .asc(32\,\%3) =< 10 > .asc(1) =< \lfloor 9/3\rfloor > .asc(9\,\%3).b =< 3 > .asc(0).b =< \lfloor 2/3\rfloor > .asc(2\,\%3).a.b =< 0 > .asc(2).a.b ='' .c.a.b = cab$

Example 2.4.5. *Consider now the following* strings *(using ASCII code as alphabet): $s = sea$ and $t = son$. We apply the operation* append *over them, thus obtaining its associated integer.*

- $[s] =$
  $[sea] = (cod(s) - 1)128^2 + (cod(e) - 1)128 + (cod(a) - 1) =$
  $(115 - 1)128^2 + (101 - 1)128 + (97 - 1) =$
  1880672
- $[t] =$
  $[son] = (cod(s) - 1)128^2 + (cod(o) - 1)128 + (cod(n) - 1) =$
  $(115 - 1)128^2 + (111 - 1)128 + (110 - 1) =$
  1881965
- $[s \cdot t] =$
  $[season] = (cod(s) - 1)128^5 + (cod(e) - 1)128^4 + (cod(a) - 1)128^3 + (cod(s) - 1)128^2 + (cod(o) - 1)128 + (cod(n) - 1) =$
  $(115-1)128^5+(101-1)128^4+(97-1)128^3+(115-1)128^2+(111-1)128+109 =$
  3944056928109

We define now the ordering relation on S that compares strings $s, t \in S$:

$$s \leq t \iff [s] \leq [t]$$

Then, the supremum of S is the empty string $''$. The image of the application of *append* to strings *sea* and *son* from Example 2.4.5, this is $s \cdot t = season$ is less or equal than s and t, that is (by the definition of S), $[s \cdot t] \leq [s]$ and $[s \cdot t] \leq [t]$. By Theorem 2.4.3, we have that S is bijective with $\mathbb{N}$. Then, the completion of S, $\mathcal{S} = S \cup \{inf(S)\}$, is bijective with the completion of $\mathbb{N}$, $(\mathbb{N} \cup \{inf(\mathbb{N})\})$, expressed also as $\mathcal{W}$ ([RD08, RR09]). Furthermore, since we know that $(\mathcal{W}, \leq) = (\mathbb{N} \cup \{inf(\mathbb{N})\}, \leq)$ is a multi-adjoint lattice, then so is $\mathcal{S}$[17].

Once proven that $\mathcal{S}$ is a multi-adjoint lattice by using a bijective application to the multi-adjoint set $\mathcal{W}$, by Theorem 2.4.2 we know that the cartesian product $\mathcal{V} \times \mathcal{S}$ is also a multi-adjoint lattice. This lattice is very useful to generate declarative execution traces of a multi-adjoint logic program with more detail that the ones generated with $\mathcal{V} \times \mathcal{W}$. Now, together with the length of the derivation, we also report explicitly the rules and the connectives evaluated until reaching each (possible) answer.

[17] It can be easily checked that $\&_{\texttt{append}}$ is actually an adjoint conjunction in lattice $\mathcal{S}$.

To summarize, in this Section we have justified the multi-adjoint character of many lattices, thus allowing the development of interesting applications in MALP. These applications include the debugging and generating of execution traces, as we describe more precisely in Section 3.6.

Capítulo 3

Programación lógica multi-adjunta

En el presente capítulo introducimos el paradigma de programación lógica multi-adjunta, o MALP (véase su formulación original en [MOV01d, MOV01c], o bien una descripción basada en conjuntos difusos, en [MPV14c]), sobre el que hemos realizado el grueso de nuestra investigación a lo largo de esta tesis. Dicho lenguaje tiene como propósito, al igual que otros modelos junto a los que se enmarca, la introducción de conceptos propios de la lógica difusa dentro de la bien conocida programación lógica.

El motivo principal por el que hemos centrado nuestra investigación en el marco de la la programación multi-adjunta es que ésta constituye un marco muy general capaz de subsumir otros lenguajes lógico difusos, lo que lo hace muy atractivo para realizar diversas investigaciones –desde transformación automática de programas a la demostración de propiedades– que, a su vez, podrían ser aplicadas a estos lenguajes, considerados como instancias del lenguaje multi-adjunto.

Esta generalidad se traduce también en un alto nivel de expresividad, que facilita enormemente la implementación de problemas del mundo real.

A las características de generalidad y expresividad se suma la de contar con una semántica operacional clara, requisito fundamental para la definición de técnicas de transformación de programas que están siendo desarrolladas en nuestro grupo de investigación. Cabe añadir que nuestro grupo ha participado activamente en la definición del propio lenguaje MALP. En [JMP06c] y [JMP09c] se diseñó la fase inter-

pretativa y se formuló su semántica declarativa por teoría de modelos, y en el trabajo [MPV12a] se estableció la relación entre la noción de *respuesta computada difusa* y *consecuencia lógica* para este marco. En [MM09c, MM09a, MM09b, MMPV10b] se define y refina la noción de coste computacional para MALP. Además, en los trabajos [MMPV11c, MMPV11b, MMPV11a, MPV12b], de los que soy coautor, se elaboran retículos que permiten documentar la ejecución de un objetivo.

Estructuramos la descripción del lenguaje MALP del siguiente modo. En primer lugar, en la Sección 3.1 detallamos su sintaxis, seguida por su semántica operacional en la Sección 3.2. A continuación, en la Sección 3.3 abordamos su semántica declarativa, tanto por modelo mínimo como por punto fijo, y probamos su equivalencia entre sí y con la semántica operacional. Después, analizamos las medidas de costes computacionales en la fase interpretativa y proponemos una nueva regla para realizar dicha fase –los pasos interpretativos *cortos*–. En la sección 3.7 referimos el trabajo realizado en [MPV12b, MPV14b, JMV15], en el que desarrollamos un método para introducir la noción de (igualdad basada en) similaridad en el lenguaje MALP, en primer lugar, que más adelante automatizamos y extendimos al lenguaje FASILL, que subsume al primero. Finalmente, tras un estudio sobre la inclusión de trazas declarativas en las respuestas computadas mediante retículos cartesianos [MMPV12a], terminamos el capítulo mediante un esbozo de la descripción de la lógica difusa a partir de la Teoría de Categorías [EGH$^+$13a, EGH$^+$13b].

3.1. Comparación con otros lenguajes

La programación lógica multi-adjunta (MALP), introducida en [MOV01d, MOV01c], constituye una generalización de la programación lógica monótona y residuada descrita en [DM00, DS00, DM01b, DM02, DM04], la cual extiende a los programas lógicos probabilísticos híbridos de [DS00], los programas de bases de datos deductivas probabilísticas de [LS01a], los programas lógicos probabilísticos ordinarios de [Luk01] y los programas del marco de deducción cuantitativa de [vE86] (que, por su parte, extienden a los programas de [DLP91]).

Esto hace de MALP un paradigma extremadamente flexible dentro de la programación lógica difusa. Como lenguaje basado en reglas asociadas a un grado de verdad, también mejora antiguas aproximaciones (véase, por ejemplo, el sistema Prolog–Elf de [IK85], el sistema Fril de [BMP95] y variantes difusas de Prolog propuestas en [LL90, VP96, GMV04]).

Como se ha indicado, el marco multi-adjunto no es el único en su campo. El creciente interés por modelos de razonamiento capaces de trabajar con información "imperfecta" ha propiciado el desarrollo de un gran número de propuestas para la integración del razonamiento aproximado en el contexto de la programación lógica (clásica). De todas las aproximaciones, algunas no quedan subsumidas por el paradigma multi-adjunto. Tal es el caso de la programación lógica basada en similaridad [Ses02, JR06a] y la programación lógica anotada [KS92]. Con respecto al primer tipo, encontramos que en [MOV04] se establecen ciertas correspondencias con MALP, aunque útiles únicamente a nivel teórico. En particular, se ha probado que la semántica producida por los métodos de unificación y resolución basados en similaridad de [Ses02, JR06a] se pueden replicar mediante el mecanismo procedimental de los programas lógicos multi-adjuntos ampliado con reglas con pesos especiales que emulan ecuaciones de similaridad [MPV12b, JMPV14].

Una de las principales limitaciones de la mayoría de los marcos difusos (incluyendo MALP) es la carencia de cualquier forma de negación no monótona, si bien no es el caso del que se presenta en [Str05a, LS05], otro marco muy general cuyos programas lógicos normales permiten capturar, entre otros, el marco de la deducción cuantitativa de [vE86], el marco posibilista de [DLP91], el marco de las bases deductivas de datos de [LS01b] y la programación lógica difusa de [Voj01].

Por todo ello, para los programas lógicos anotados de [KS92] y especialmente para la más reciente versión de [Str05a, LS05], que contempla programas de sintaxis muy sencilla, pasamos a valorar las ventajas e inconvenientes que comporta el lenguaje MALP. Tal y como se recoge en[Pen10], si comparamos esta aproximación con la programación multi-adjunta podemos observar:

- **Expresividad**. Lo más reseñable en este apartado es la capacidad de lenguajes basádos en birretículos ([Fit91]), que son capaces de utilizar la negación no monótona. Mediante este mecanismo estos lenguajes incorporan junto a los grados de certeza, la noción de grado de duda. En contraste, MALP no es capaz de manejar agregadores no monótonos en los cuerpos de las reglas y se basa exclusivamente en grados de certeza. En parte, esta limitación del marco multi-adjunto se ha tratado en [DMO07], donde se han empleado multiretículos y múltiples tipos. No obstante, este resultado sigue siendo menos expresivo que el desarrollado por [Str05a, LS05].
- **Sintaxis**. La sintaxis empleada en [Str05a, LS05] para definir las reglas de programa es $A \leftarrow f(B_1, \ldots, B_n)$, donde A y B_i son, evidentemente, átomos, y

f es un operador asociado a una función de verdad y que, en última instancia, es el resultado de combinar otros operadores del cuerpo de la regla. Como se muestra en [DMO07], esta sintaxis es muy parecida a la empleada por MALP donde, si bien los cuerpos de las reglas tienen, en principio, una forma más general, en [GM08a] se presenta una transformación ("agregación") que adapta las fórmulas de MALP a este formato más sencillo. La diferencia más notable en este aspecto radica en una limitación fundamental de [Str05a], consistente en que no se admiten símbolos de función. Más aún, los autores de dicho marco limitan su atención al caso proposicional, de modo que en este aspecto su sintaxis es mucho menos rica que la de MALP.

- **Semántica Procedimental**. En este apartado surgen las mayores diferencias. Aquí, el modelo de [Str05a, LS05] utiliza un procedimiento de resolución de arriba-abajo para programas lógicos normales (estudiados también en [LS02b, LS05]) sobre retículos y birretículos. La ventaja de este método radica en que se puede adaptar a diversas semánticas, como la de Kripke-Kleene o la semántica Bien-Fundada (véase [LS03, LS04]). Este procedimiento, sin embargo, se aporta en forma de algoritmo. Esto queda lejos de la claridad y formalidad empleada en MALP, que se basa en un sistema de transición de estados, un sistema que facilita enormemente las técnicas de transformación de programas y el estudio de las relaciones entre la semántica de los programas transformados y los originales.

A raíz de estas observaciones, concluimos que el marco de [Str05a, LS05] es una aproximación lógico difusa atractiva, especialmente por su uso de birretículos y la negación no monótona, a la que aún le queda por superar el caso proposicional y permitir símbolos de función. Sería, por tanto, interesante extender nuestros resultados, mediante las técnicas de transformación de programas estudiadas en nuestro grupo de investigación, al maco contemplado en [Str05a, Str05b, LS02b, LS03, LS04].

Por ello, y como detallamos con mayor profusión en la Sección 2.3, cabe afirmar que el lenguaje multi-adjunto es probadamente potente y expresivo en comparación con otros dentro de la programación lógica difusa. No sólo eso, también es capaz de adaptar gran número de otros lenguajes y conservar su semántica. Es, por tanto, el marco (más general) elegido para formular diversas técnicas de transformación de programas, para las que algunas de las novedosas técnicas de mediciones de coste que presentamos en este trabajo resultan cruciales; así como para estudiar sus propiedades de corrección, completitud y eficiencia, propiedades sobre las que

también hemos conseguido importantes avances en el ámbito de esta tesis, ya sea a través de reductantes (generales) para asegurar su completitud, como a través de técnicas de tabulación y desplegado para mejorar su eficiencia.

Sintaxis de la programación lógica multi-adjunta

Ahora que hemos ubicado el esquema multi-adjunto en el contexto de la programación lógica difusa, abordamos la descripción de sus principales rasgos, concretamente, de su sintaxis, tal como se recoge en [MOV01d, MOV01c, MOV01b, MOV01a, MO02, MOV04, MO04, DMO07]. Los programas MALP se basan en un lenguaje de primer orden $\mathcal{L}$ construido a partir de los elementos de una signatura Σ. Esta signatura contiene variables (un número infinito numerable de ellas), símbolos de predicado, símbolos de función, constantes (que pueden considerarse como símbolos de función de aridad 0), cuantificadores ($\forall$ y $\exists$) y un número arbitrario de conectivas (representadas por @). Los elementos del lenguaje, como términos y átomos, se construyen como es común en los lenguajes lógicos. Los átomos se unen entre sí mediante conectivas formando reglas y fórmulas. Las conectivas se clasifican en:

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

De manera general distinguimos entre implicaciones ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$), y las demás conectivas, que agrupamos bajo el nombre de *agregadores* y sirven para combinar átomos y propagar sus grados de verdad. Denotamos los agregadores con $@_1, @_2, \ldots, @_n$, e incluyen conjunciones ($\wedge_1, \wedge_2, \ldots, \wedge_k$), disyunciones ($\vee_1, \vee_2, \ldots, \vee_l$) y cualesquiera otros operadores híbridos –como la media aritmética, en caso de que el retículo asociado al programa sea el intervalo real $[0, 1]$. Los agregadores se distinguen por su aridad (el número de parámetros que aceptan) y por su función de verdad, que es una función definida en un retículo L asociado al programa MALP, definida como $\dot{@}_k : L^n \to L$, para la conectiva $@_k$ de aridad n. Además, la función de verdad de todo agregador es monótona y satisface las condiciones de frontera $\dot{@}(\top, \ldots, \top) = \top$, $\dot{@}(\bot, \ldots, \bot) = \bot$. Se da también que las conjunciones $\wedge_i$ satisfacen $\wedge_i(\top, v) = v = \wedge_i(v, \top), \forall v \in L, \forall i, i = 1, \ldots, k$, y las disyunciones $\vee_i$ satisfacen $\vee_i(\bot, v) = v = \vee_i(v, \bot), \forall i, i = 1, \ldots, l$.

Para dotar de mayor expresividad a esta sintaxis, generalizamos las conectivas $\wedge_i$, $\vee_i$ y $@_i$, que son operadores binarios, para que acepten cualquier número de argumentos. Así, escribimos $@_i(x_1, \ldots, x_n)$ en lugar de $@_i(x_1, @_i\ (x_2, \ldots, @_i(x_{n-1}, x_n)\ldots))$. Mantenemos los paréntesis por no poder garantizar la asociatividad (ni la conmutatividad) de algunas conectivas.

Todo programa MALP está asociado a un retículo multi-adjunto $\langle L, \leq \rangle$ equipado con una colección de pares adjuntos $(\leftarrow_i, \&_i)$, donde cada $\&_i$ es una conjunción[1] ligada a la evaluación del *modus ponens*. Así, el lenguaje $\mathcal{L}$ también admite elementos $\alpha \in L$ y, a modo de conectivas, las conjunciones adjuntas $\&_i$. En la mayoría de los ejemplos de esta tesis utilizamos el intervalo real $[0, 1]$ con el orden usual, si bien a un programa puede asociarse cualquier retículo completo que disponga al menos de un par adjunto $(\&_i, \leftarrow_i)$.

En MALP se llama *regla* a una fórmula del lenguaje $\mathcal{L}$ de la forma $A \leftarrow \mathcal{B}$, donde A es una fórmula atómica (un átomo), llamado *cabeza*, y $\mathcal{B}$, que llamamos *cuerpo*, es una fórmula construida a partir de fórmulas atómicas $B_1, \ldots, B_n$ – $n \geq 0$ - y grados de verdad del retículo, unidos por conjunciones, disyunciones, agregadores y conjunciones adjuntas. En caso de que el cuerpo esté vacío la regla recibe el nombre de *hecho*. Un *objetivo* es una fórmula construida como un cuerpo y planteada como una pregunta al sistema. Las variables que aparezcan en las reglas se suponen universalmente cuantificadas.

El retículo multi-adjunto asociado a un programa multi-adjunto sirve para interpretar las fórmulas del lenguaje. Su definición formal puede verse en el capítulo previo (Definición 2.4.1). Definimos ahora los conceptos de programa lógico multi-adjunto tal y como se recoge en [MOV04], entendido como un conjunto de reglas lógicas asociadas cada una a un elemento del retículo multi-adjunto que es su *grado de verdad*.

Definición 3.1.1. *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 $\&_i$.

[1] Procuramos reservar el símbolo $\&_j$ para estas conjunciones adjuntas a fin de distinguirlas de otras conjunciones del lenguaje.

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 a* $\mathcal{P}$*.*

Llamamos hechos a las reglas cuyo cuerpo es $\top$ (consideraremos en este caso que son reglas con cuerpo vacío).

Definimos el *Universo de Herbrand* $U_{\mathcal{L}}$ de un programa lógico multi-adjunto $\mathcal{P}$ como el conjunto de todos los términos básicos (sin variables) que se pueden construir en su lenguaje $\mathcal{P}$, y su *Base de Herbrand* $B_{\mathcal{L}}$ como el conjunto de todos los átomos que se pueden construir tomando un predicado de la signatura Σ con argumentos en el Universo de Herbrand, esto es, todos los átomos básicos que se pueden construir en $\mathcal{L}$. Con estos dos conceptos, definimos una *interpretación de Herbrand difusa*[2], $\mathcal{I}$, como una aplicación de la base de Herbrand en el retículo multi-adjunto de grados de verdad L, y decimos que el grado de verdad de un átomo básico $A \in B_{\mathcal{L}}$ es un elemento $\mathcal{I}(A) \in L$. Dada una asignación ϑ que asocia variables a términos, la valoración de una fórmula bajo una interpretación se obtiene por inducción estructural sobre la complejidad de esta fórmula:

$$\begin{aligned}
\mathcal{I}(p(t_1, \ldots, t_n))[\vartheta] = &\quad \mathcal{I}(p(t_1\vartheta, \ldots, t_n\vartheta)),\\
\mathcal{I}(@(A_1, \ldots, A_n))[\vartheta] = &\quad \dot{@}(\mathcal{I}(A_1)[\vartheta], \ldots, \mathcal{I}(A_n)[\vartheta]),\\
\mathcal{I}(A \leftarrow \mathcal{B})[\vartheta] = &\quad \mathcal{I}(A)[\vartheta] \dot{\leftarrow} \mathcal{I}(\mathcal{B})[\vartheta],\\
\mathcal{I}((\forall x)\mathcal{A})[\vartheta] = &\quad inf\{\mathcal{I}(\mathcal{A})[\vartheta'] \mid \vartheta' \;\; x\text{–equivalente a } \;\vartheta\},
\end{aligned}$$

donde A y A_i son, como es costumbre, fórmulas atómicas, p es un símbolo de predicado, $@$ un símbolo de conectiva, $\mathcal{B}$ una fórmula que hace de cuerpo, $\mathcal{A}$ cualquier fórmula, y por $\dot{@}$ denotamos la función de verdad de la conectiva $@$. Omitiremos ϑ en la interpretación de una fórmula cuando su valoración no sea relevante.

Empleamos teorías difusas como medio de describir aquellos problemas reales de los que sólo dispongamos de un conocimiento vago o incierto.

Definición 3.1.2. *Una* teoría difusa *es una función parcial* T *que asocia a cada fórmula un elemento (grado de verdad) del retículo* L*.*

Un programa multi-adjunto*,* $\mathcal{P}$*, es una teoría difusa tal que el dominio,* $\mathrm{dom}(\mathcal{P})$*, es un conjunto finito de reglas y* L *es un retículo multi-adjunto equipado con varios pares adjuntos.*

[2]Véase la Sección 3.3 posterior para más detalle.

Un programa lógico multi-adjunto, por tanto, puede verse como un conjunto de pares $\langle \mathcal{R}; r \rangle$, donde $\mathcal{R}$ es una regla y $r = \mathcal{P}(\mathcal{R}) \in L$ es su *grado de verdad*, que expresa la confianza que el usuario del sistema tiene en la regla $\mathcal{R}$. A modo de mnemotécnico y para simplificar la sintaxis, escribimos "$\mathcal{R}$ with r" en lugar de $\langle \mathcal{R}; r \rangle$. Los grados de verdad deben ser asignados axiomáticamente por un experto.

Una interpretación de Herbrand $\mathcal{I}$ es un *modelo de Herbrand* de una teoría difusa $\mathcal{P}$ si para cada regla $\mathcal{R} \in \mathrm{dom}(\mathcal{P})$, $\mathcal{I}(\mathcal{R}) \geq \mathcal{P}(\mathcal{R})$. Decimos entonces que la interpretación $\mathcal{I}$ *satisface* al programa $\mathcal{P}$.

Los pares adjuntos $(\&_i, \leftarrow_i)$ cumplen la propiedad adjunta (véase la Definición 2.4.1). Por ello satisfacen la regla de inferencia del *modus ponens (generalizado)*:

$$\frac{\langle A \leftarrow_i \mathcal{B}; x \rangle \qquad \langle \mathcal{B}; y \rangle}{\langle A; x \&_i y \rangle}$$

Esto es, si una interpretación $\mathcal{I}$ es un modelo de $A \leftarrow_i B$ (es decir, $\mathcal{I}(A \leftarrow_i B) \succeq x$) y de $\mathcal{B}$ (es decir, $\mathcal{I}(\mathcal{B}) \succeq y$) entonces $\mathcal{I}$ es un modelo de A (esto es, $\mathcal{I}(A) \succeq x \&_i y$). Demostramos la corrección del mecanismo operacional como sigue: $x \leq \mathcal{I}(A \leftarrow_i B) = \mathcal{I}(A) \dot{\leftarrow}_i \mathcal{I}(B) \leq \mathcal{I}(A) \dot{\leftarrow}_i y$, ya que "$\leftarrow_i$" es no-creciente en su segundo argumento y $\mathcal{I}(B) \succeq y$; entonces, por la propiedad adjunta, $\mathcal{I}(A) \succeq x \&_i y$.

La semántica declarativa de los programas lógicos multi-adjuntos está definida en [MOV04] en términos del operador de punto fijo. En la Sección 3.3 de esta tesis abordamos ésta y otras semánticas del lenguaje multi-adjunto, junto a las relaciones entre ellas.

3.2. Semántica operacional basada en un sistema de transción de estados

Al contrario que en la programación lógica clásica, donde el mecanismo operacional se basa en refutación, en [MOV01d, MOV04] se ha definido la semántica operacional del lenguaje lógico multi-adjunto como un sistema de transición de estados dividido en dos fases. En la primera de ellas, denominada *fase admisible*, se aplican pasos admisibles en los que se explotan los átomos del objetivo. La siguiente fase, denominada *fase interpretativa*, consiste en la explotación de los agregadores mediante pasos interpretativos. Parte del trabajo realizado en esta tesis se centra en la me-

jora y reformulación de la fase interpretativa de este lenguaje, como explicamos en capítulos posteriores.

3.2.1. Pasos admisibles

Detallamos aquí los conceptos de paso de computación admisible difuso, derivación admisible difusa y respuesta computada difusa, de forma parecida a como se hace en [MOV04].

El mecanismo operacional que definimos a continuación es un proceso de razonamiento que implementa una generalización del *modus ponens* y proporciona una cota inferior del grado de verdad del objetivo bajo cualquier modelo del programa.

A partir de un objetivo, en cuyo contexto se ha seleccionado un átomo A, si existe una regla $\langle A' \leftarrow_i \mathcal{B}; r \rangle$ del programa y una sustitución $\theta = mgu(\{A = A'\})$[3], damos un *paso admisible* al reemplazar A por la expresión $(r \&_i \mathcal{B})\theta$ en el contexto del objetivo de partida.

La repetición de este proceso toma el nombre de *derivación admisible*, y termina cuando se obtiene una fórmula que no contiene ningún átomo. La fórmula resultante, que contiene grados de verdad y agregadores, se interpreta en el retículo asociado obteniendo un único grado de verdad. Se garantiza que dicho grado de verdad es, efectivamente, una cota inferior del grado de verdad del objetivo gracias a la propiedad adjunta del par $(\leftarrow_i, \&_i)$.

El mencanismo procedimental, descrito a continuación, busca obtener una cota de la respuesta computada difusa óptima para la pregunta planteada [MO02]. En este contexto, una computación es un proceso dividido en dos fases procedimentales. La primera de tipo operacional y la segunda de tipo interpretativo. Esta separación, si bien no es necesaria (existen otros lenguajes, como los contemplados en [VP96, Voj01], donde se combinan ambas fases), permite simplificar las definiciones subyacentes.

Para la formalización del concepto de computación admisible difusa, denotamos por $\mathcal{C}[A]$ o, más generalmente, $\mathcal{C}[A_1, \ldots, A_n]$, una $\mathcal{L}$-fórmula donde A, o $A_1, \ldots, A_n$ son subexpresiones (normalmente átomos) que aparecen en el contexto –posiblemente vacío– $\mathcal{C}[\,]$. Además, usamos la expresión $\mathcal{C}[A/A']$ (y su extensión respectiva) para significar el reemplazamiento de A por A' en el contexto $\mathcal{C}[\,]$.

[3] Por $mgu(E)$ denotamos el *unificador más general* de un conjunto de expresiones E.

Definición 3.2.1. *Sea $\mathcal{Q}$ un objetivo y σ una sustitución, un* estado *es un par $\langle \mathcal{Q}; \sigma \rangle$. Sea $\mathcal{E}$ el conjunto de estados. Dado ahora un programa multi-adjunto $\mathcal{P}$, definimos una* computación admisible difusa *como un sistema de transición de estados, cuya relación de transición $\rightarrow_{AS} \subset (\mathcal{E} \times \mathcal{E})$ es la menor relación que satisface las siguientes* reglas admisibles*:*

Regla 1.

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \,\&_i\, \mathcal{B}])\theta; \sigma\theta \rangle \text{ si } \begin{cases} (1) & A \text{ es el átomo seleccionado en } \mathcal{Q} \\ (2) & \theta = mgu(\{A' = A\}) \\ (3) & \langle A' \leftarrow_i \mathcal{B}; v \rangle \text{ en } \mathcal{P} \text{ y } \mathcal{B} \text{ no es vacío.} \end{cases}$$

Regla 2.

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle \text{ si } \begin{cases} (1) & A \text{ es el átomo seleccionado en } \mathcal{Q} \\ (2) & \theta = mgu(\{A' = A\}) \\ (3) & \langle A' \leftarrow; v \rangle \text{ en } \mathcal{P}. \end{cases}$$

Regla 3.

$$\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle \text{ si } \begin{cases} (1) & A \text{ es el átomo seleccionado en } \mathcal{Q} \\ (2) & \text{no existe regla en } \mathcal{P} \text{ cuya cabeza unifique con } A. \end{cases}$$

Los conceptos comunes de la programación lógica difusa, como el de derivación o paso de derivación, se adaptan de manera muy natural al marco multi-adjunto. Por tanto, se ha obviado hasta ahora que las variables en las fórmulas son renombradas al aplicarse los pasos de computación.

Obsérvese el efecto que tiene la Regla 3, que permite contemplar posibles pasos de fallo o, dicho de otro modo, permite continuar una derivación aún cuando falle un subobjetivo. Esto distingue especialmente el marco MALP sobre la programación lógica pura, en la que los subobjetivos fallidos hacen fallar todo el objetivo principal. La justificación de este comportamiento radica en que en el caso de MALP un subobjetivo F puede fallar (su grado de verdad puede ser el ínfimo) y no así el objetivo en que se inserta, pues F puede estar conectado al resto de la fórmula mediante conectivas distintas de la conjunción (al contrario que en programación lógica clásica).

En adelante, mediante la notación $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ y $\rightarrow_{AS3}$ nos referirnos explícitamente a la aplicación de cada una de las reglas admisibles arriba formalizadas. Además, anotaremos si procede qué regla del programa se ha empleado en cada paso de computación mediante un superíndice sobre estos símbolos. Como es usual, además, un número n sobre estos símbolos (como en $\rightarrow^n_{AS}$) denota la aplicación sucesiva de n de estos pasos, y con $\rightarrow^*_{AS}$ nos referimos a una secuencia arbitaria de cero o más pasos.

Definición 3.2.2. *Sea $\mathcal{P}$ un programa y $\mathcal{Q}$ un objetivo. Una secuencia $\mathcal{E}_0 \rightarrow_{AS} \mathcal{E}_1 \rightarrow^*_{AS} \mathcal{E}_n$ es una derivación admisible* completa *o de* éxito *si:*

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

Obsérvese que una $\mathcal{L}$-fórmula sin átomos, a la que se ha llegado a partir de un objetivo tras una secuencia de pasos admisibles, puede ser interpretada directamente en el retículo multi-adjunto L. Atendiendo a este resultado, extendemos la noción de respuesta computada de acuerdo a la siguiente definición, donde $\mathcal{V}ar(s)$ denota al conjunto de variables distintas que aparecen en s, y $\theta[\mathcal{V}ar(s)]$, la sustitución obtenida a partir de θ sobre $\mathcal{V}ar(s)$.

Definición 3.2.3. *Sea $\mathcal{P}$ un programa lógico multi-adjunto y $\mathcal{Q}$ un objetivo. Una* derivación admisible *es una secuencia $\langle \mathcal{Q}; id \rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \theta \rangle$. Cuando $\mathcal{Q}'$ es una fórmula que no contiene átomos, el par $\langle \mathcal{Q}'; \sigma \rangle$, donde $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, se denomina una* respuesta computada admisible *(a.c.a.,* admissible computed answer*) para esta derivación.*

Definición 3.2.4. *Sea $\mathcal{P}$ un programa multi-adjunto y $\mathcal{Q}$ un objetivo. Dada una derivación admisible $\langle \mathcal{Q}; id \rangle \rightarrow^*_{AS} \langle @(r_1, \ldots, r_n); \theta \rangle$, con $r_i \in L$ para todo $i \in \{1, \ldots, n\}$, el par $\langle \hat{@}(r_1, \ldots, r_n); \sigma \rangle$, donde $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, es una* respuesta computada difusa *(f.c.a., del inglés* fuzzy computed answer*) para esta derivación.*

Ilustramos las últimas definiciones por medio del siguiente ejemplo.

Ejemplo 3.2.5. *Sea $\mathcal{P}$ el programa lógico multi-adjunto que sigue,*

$$\begin{array}{lll}
\mathcal{R}_1: & p(X) \leftarrow_{\texttt{prod}} q(X,Y) \wedge_{\texttt{G}} r(Y) & \textit{with } 0.8 \\
\mathcal{R}_2: & q(a,Y) \leftarrow_{\texttt{prod}} s(Y) & \textit{with } 0.7 \\
\mathcal{R}_3: & q(Y,a) \leftarrow_{\texttt{luka}} r(Y) & \textit{with } 0.8 \\
\mathcal{R}_4: & r(Y) \leftarrow & \textit{with } 0.7 \\
\mathcal{R}_5: & s(b) \leftarrow & \textit{with } 0.9
\end{array}$$

para el que las etiquetas `prod`*,* `G` *y* `luka` *significan* lógica del producto*,* lógica de Gödel *y* lógica de Łukasiewicz *respectivamente. Esto es,* $\dot{\&}_{\texttt{prod}}(x,y) = x \cdot y$*,* $\dot{\wedge}_{\texttt{G}}(x,y) = min\{x,y\}$*, y* $\dot{\&}_{\texttt{luka}}(x,y) = max\{0, x+y-1\}$*.*

En la siguiente derivación admisible para el objetivo $\leftarrow p(X) \wedge_{\texttt{G}} r(a)$ *y el programa* $\mathcal{P}$*, subrayamos la expresión seleccionada en cada paso admisible:*

$$\begin{array}{rl}
\langle \underline{p(X)} \wedge_{\texttt{G}} r(a); id \rangle \rightarrow_{AS1}{}^{\mathcal{R}_1} & \langle (0.8 \&_{\texttt{prod}} (\underline{q(X_1,Y_1)} \wedge_{\texttt{G}} r(Y_1))) \wedge_{\texttt{G}} r(a); \sigma_1 \rangle \\
\rightarrow_{AS1}{}^{\mathcal{R}_2} & \langle (0.8 \&_{\texttt{prod}} ((0.7 \&_{\texttt{prod}} \underline{s(Y_2)}) \wedge_{\texttt{G}} r(Y_2))) \wedge_{\texttt{G}} r(a); \sigma_2 \rangle \\
\rightarrow_{AS2}{}^{\mathcal{R}_5} & \langle (0.8 \&_{\texttt{prod}} ((0.7 \&_{\texttt{prod}} 0.9) \wedge_{\texttt{G}} \underline{r(b)})) \wedge_{\texttt{G}} r(a); \sigma_3 \rangle \\
\rightarrow_{AS2}{}^{\mathcal{R}_4} & \langle (0.8 \&_{\texttt{prod}} ((0.7 \&_{\texttt{prod}} 0.9) \wedge_{\texttt{G}} 0.7)) \wedge_{\texttt{G}} \underline{r(a)}; \sigma_4 \rangle \\
\rightarrow_{AS2}{}^{\mathcal{R}_4} & \langle (0.8 \&_{\texttt{prod}} ((0.7 \&_{\texttt{prod}} 0.9) \wedge_{\texttt{G}} 0.7)) \wedge_{\texttt{G}} 0.7; \sigma_5 \rangle
\end{array}$$

donde

$$\begin{array}{l}
\sigma_1 = \{X/X_1\} \\
\sigma_2 = \{X/a, X_1/a, Y_1/Y_2\} \\
\sigma_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\} \\
\sigma_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\} \\
\sigma_5 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\}
\end{array}$$

Entonces, la respuesta computada admisible para esta derivación admisible es el par $\langle (0.8 \&_{\texttt{prod}} ((0.7 \&_{\texttt{prod}} 0.9) \wedge_{\texttt{G}} 0.7)) \wedge_{\texttt{G}} 0.7; \sigma_5 \rangle$*. Además, la respuesta computada difusa*

es el par $\langle 0.504, \{X/a\}\rangle$*, ya que:*

$$\begin{gathered}\mathcal{I}((0.8\&_{\mathtt{prod}}((0.7\&_{\mathtt{prod}}0.9)\wedge_{\mathtt{G}}0.7))\wedge_{\mathtt{G}}0.7) = \\ \vdots \\ (0.8\dot{\&}_{\mathtt{prod}}((0.7\dot{\&}_{\mathtt{prod}}0.9)\dot{\wedge}_{\mathtt{G}}0.7))\dot{\wedge}_{\mathtt{G}}0.7 = \\ \vdots \\ 0.504\dot{\wedge}_{\mathtt{G}}0.7 = \\ 0.504\end{gathered}$$

y $\sigma_5[\mathcal{V}ar(\mathcal{Q})] = \{X/a\}$*.*

3.2.2. Pasos interpretativos

Como se ha indicado anteriormente, durante la fase operacional, la aplicación de pasos admisibles provoca que eventualmente se exploten todos los átomos de un objetivo. Llegamos entonces a una fórmula que no contiene átomos sino grados de verdad unidos por agregadores que puede ser interpretada en el retículo multi-adjunto asociado L.

Esta fase interpretativa fue diseñada en [JMP06c] por nuestro grupo. Tiene forma de sistema de transición de estados, al igual que la fase operacional. De este modo nos desprendemos de elementos "ruidosos" como la función de selección (para esta fase interpretativa) y los correspondientes resultados de independencia.

Abordamos ahora las nociones de paso de computación interpretativo y derivación interpretativa, y repasamos en estos términos la de respuesta computada difusa.

Definición 3.2.6 (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* $\rightarrow_{IS} \subset (\mathcal{E}\times\mathcal{E})$ *se define como*

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

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

Definición 3.2.7. *Sea* $\mathcal{P}$ *un programa y* $\langle \mathcal{Q};\sigma\rangle$ *una respuesta computada admisible (a.c.a.), es decir,* $\mathcal{Q}$ *es un objetivo que no contiene átomos. Una* derivación interpretativa *es una secuencia* $\langle \mathcal{Q};\sigma\rangle \rightarrow_{IS}^{*} \langle \mathcal{Q}';\sigma\rangle$*.*

En adelante, por *derivación completa* nos referimos a la secuencia de pasos admisibles/interpretativos de la forma $\langle \mathcal{Q}; id\rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \sigma\rangle \rightarrow^*_{IS} \langle r; \sigma\rangle$, donde los pares $\langle \mathcal{Q}'; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ y $\langle r; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ son, respectivamente, la a.c.a. y la f.c.a. para la derivación. Cuando convenga la denotaremos por $\langle \mathcal{Q}; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\rangle$ y $\langle r; \sigma\rangle$ será la respuesta computada difusa de la derivación correspondiente.

Ejemplo 3.2.8. *Completamos ahora la derivación previa del Ejemplo 3.2.5, ejecutando los pasos interpretativos necesarios para obtener la respuesta computada difusa (f.c.a.) con respecto al retículo* $([0,1], \leq)$.

$$\langle (0.8 \&_{\texttt{prod}} (\underline{0.7 \&_{\texttt{prod}} 0.9}) \&_{\texttt{G}} 0.7)) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$

$$\langle (0.8 \&_{\texttt{prod}} (\underline{0.63 \&_{\texttt{G}} 0.7})) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$

$$\langle (\underline{0.8 \&_{\texttt{prod}} 0.63}) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$

$$\langle \underline{0.504 \&_{\texttt{G}} 0.7}; \{X/a\}\rangle \rightarrow_{IS}$$

$$\langle 0.504; \{X/a\}\rangle$$

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

3.3. Semánticas declarativas por teoría de modelos y punto fijo

Junto a la semántica operacional descrita en la sección anterior, se han desarrollado otras semánticas para la programación lógica multi-adjunta, entre las que destacan las semánticas declarativas por punto fijo (formalizada por los propios creadores del lenguaje MALP en [MOV04]) y por teoría de modelos propuesta por miembros de nuestro grupo en el trabajo [JMP09c]. Al contrario que la semántica operacional, que da las pautas del funcionamiento de un programa, las semánticas declarativas buscan dar significado a dicho programa.

En esta sección, partiendo del trabajo realizado en [Pen10], repasamos a las nociones básicas y características principales de ambas semánticas, así como exploraremos las equivalencias existentes entre ellas y con la semántica operacional.

Introducimos a continuación las definiciones formales, desde un punto de vista semántico, de interpretación, de modelo y de operador $T_{\mathcal{P}}$ de punto fijo para un

programa $\mathcal{P}$ lógico multi-adjunto, de forma similar a como se hace en [MOV04]. Presentaremos también ejemplos donde concretaremos los puntos fijos de este operador.

Definición 3.3.1. *Una interpretación de Herbrand difusa*[4] *es una aplicación* $\mathcal{I} : B_{\mathcal{P}} \longrightarrow L$*, donde* $B_{\mathcal{P}}$ *es la base de Herbrand del programa* $\mathcal{P}$ *y* $(L, \leq)$ *es el retículo multi-adjunto asociado a dicho programa.*

Una interpretación de Herbrand, $\mathcal{I}$, se extiende de manera natural al conjunto de fórmulas básicas del lenguaje. Para interpretar una fórmula A no básica (cerrada, y universalmente cuantificada en el caso del lenguaje multi-adjunto), basta tomar

$$\mathcal{I}(A) = inf\{\mathcal{I}(A\xi) \mid A\xi \text{ es una instancia básica de A}\}$$

Llamamos $\mathcal{H}$ al conjunto de interpretaciones de Herbrand. El orden de $\mathcal{H}$ está inducido por el orden del conjunto L:

$$\mathcal{I}_j \leq \mathcal{I}_k \Longleftrightarrow \mathcal{I}_j(F) \leq \mathcal{I}_k(F), \forall F \in B_{\mathcal{P}}$$

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

Definición 3.3.2. *Una interpretación* $\mathcal{I}$ *satisface una regla* $\langle A \leftarrow_i \mathcal{B}; v\rangle$ *si, y sólo si,* $v \leq \mathcal{I}(A \leftarrow_i \mathcal{B})$*. Una interpretación* $\mathcal{I}$ *es un modelo de Herbrand*[5] *de* $\mathcal{P}$ *si, y sólo si,* $\mathcal{I}$ *satisface todas las reglas de* $\mathcal{P}$*.*

Por otra parte, en un retículo completo $(L, \leq)$, se dice que una función $f : L \longrightarrow L$ es monótona si, y sólo si, $\forall x, y \in L, x \leq y \Longrightarrow f(x) \leq f(y)$. Un punto fijo de una función f se define como un elemento $x \in L$ tal que $f(x) = x$. Recogemos a continuación el siguiente teorema de Knaster-Tarski (véase [Tar55]) para el estudio de puntos fijos[6].

Teorema 3.3.3. *Sea* f *una función monótona sobre un retículo completo* $(L, \leq)$*. Entonces,* f *tiene un punto fijo.*

[4] En ocasiones, diremos sólo interpretación por abreviar y pese a que sólo consideramos interpretaciones de Herbrand. Es decir, basaremos la semántica declarativa por teoría de modelos en las interpretaciones de Herbrand.

[5] En ocasiones, diremos sólo modelo.

[6] Otros teoremas de punto fijo pueden verse en [KM97, Sto04].

Se tiene que el conjunto de puntos fijos de f es un retículo completo, por lo que existe el menor punto fijo de f, y se puede obtener iterando f sobre el elemento $\bot \in L$. Esto es, el menor punto fijo de f es el supremo de la sucesión no decreciente $x_0, \ldots, x_i, x_{i+1}, \ldots, x_\lambda, \ldots$, donde $x_0 = \bot$ y se da que para cualquier $i \geq 0$,, $x_{i+1} = f(x_i)$, y para un índice determinado λ, $y_\lambda = sup\{y_i : f(y_i) = y_i, i > \lambda\}$.

En la siguiente definición recogemos la extensión realizada en [MOV01d, MOV04] del operador de punto fijo $T_\mathcal{P}$ definido por [EK76] para el lenguaje multi-adjunto. Con este operador los autores aportan la semántica de un programa multi-adjunto $\mathcal{P}$ que toman como el menor punto fijo de $T_\mathcal{P}$. Esta construcción es equivalente a la noción de modelo mínimo de Herbrand difuso, como recogemos en el Teorema 3.3.13.

Definición 3.3.4. *Sea $\mathcal{P}$ un programa lógico multi-adjunto, $\mathcal{I}$ una interpretación de Herbrand y A una fórmula básica. Definimos el operador $T_\mathcal{P}$ como una aplicación en el conjunto de interpretaciones de Herbrand tal que para cada átomo básico A*

$$T_\mathcal{P}(\mathcal{I})(A) = sup\{v\dot{\&}_i\mathcal{I}(\mathcal{B}\theta) \mid \langle H \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}, A = H\theta\}$$

En [MOV04] se demuestra también el resultado por el que $T_\mathcal{P}$ permite caracterizar las interpretaciones que son modelo de $\mathcal{P}$, que mostramos a continuación.

Teorema 3.3.5. *Una interpretación $\mathcal{I}$ de Herbrand es un modelo de un programa multi-adjunto $\mathcal{P}$ si, y sólo si, $T_\mathcal{P}(\mathcal{I}) \leq \mathcal{I}$.*

Entonces, tenemos que para todo modelo de Herbrand $\mathcal{I}$ de $\mathcal{P}$,

$$T_\mathcal{P}(\mathcal{I})(A) = sup\{v\dot{\&}_i\mathcal{I}(\mathcal{B}\theta) \mid \langle H \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}, A = H\theta\} \leq \mathcal{I}(A)$$

En general, la igualdad $T_\mathcal{P}(\mathcal{I})(A) = \mathcal{I}(A)$ no tiene por qué alcanzarse. Del mismo modo sucede en el caso de la programación lógica pura, en la que el grado de verdad de un átomo decrece conforme más se evalúa el operador $T_\mathcal{P}$ sobre un átomo.

Además, puede existir más de un único punto fijo tanto en la programación lógica clásica como en la multi-adjunta, como ilustramos con el siguiente ejemplo (para el caso clásico):

Ejemplo 3.3.6. *Sea $\mathcal{P}$ el programa lógico formado por la cláusula $p(a) \leftarrow p(a)$. La base de Herbrand de $\mathcal{P}$ es el conjunto $\mathcal{B}_H = \{p(a)\}$ y los subconjuntos de la base de Herbrand $\mathcal{I}_1 = \varnothing, \mathcal{I}_2 = \{p(a)\}$ son los únicos modelos de Herbrand de $\mathcal{P}$. Ambos son puntos fijos del operador (de punto fijo) definido en [Llo87].*

El mismo ejemplo puede diseñarse para el caso multi-adjunto de forma sencilla, sin más que entender que todo programa lógico definido es ya multi-adjunto. En el siguiente ejemplo aportamos un caso más específico:

Ejemplo 3.3.7. *Considérese $\mathcal{P}$ el programa lógico multi-adjunto formado por la única regla $\langle p(a) \leftarrow p(a); 0.5\rangle$ y con retículo asociado el intervalo $([0,1], \leq)$. Sea $(\&_{\mathtt{G}}, \leftarrow_{\mathtt{G}})$ el par adjunto en $([0,1], \leq)$ de la lógica de* Gödel, *con funciones de verdad de $\&_{\mathtt{G}}$ y $\leftarrow_{\mathtt{G}}$ dadas por:*

$$\dot{\&}_{\mathtt{G}}(x,y) = inf\{x,y\} \quad y \quad x \dot{\rightarrow}_{\mathtt{G}}\, y = \begin{cases} \top, & si\ x \leq y \\ y, & en\ otro\ caso \end{cases}$$

Entonces, las interpretaciones $\mathcal{I}_1, \mathcal{I}_2$ definidas por $\mathcal{I}_1(p(a)) = 0, \mathcal{I}_2(p(a)) = 0.5$ verifican:

$$T_{\mathcal{P}}(\mathcal{I}_1)(p(a)) = sup\{0.5 \dot{\&} \mathcal{I}_1(p(a))\} = sup\{0.5 \dot{\&} 0\} = 0 = \mathcal{I}_1(p(a))$$

$$T_{\mathcal{P}}(\mathcal{I}_2)(p(a)) = sup\{0.5 \dot{\&} \mathcal{I}_2(p(a))\} = sup\{0.5 \dot{\&} 0.5\} = 0.5 = \mathcal{I}_2(p(a))$$

y, por tanto, ambos son puntos fijos del operador $T_{\mathcal{P}}$, resultando $\mathcal{I}_1$ el menor punto fijo.

Una vez introducida la semántica de punto fijo, abordamos el otro modelo semántico para la programación lógica multi-adjunta especialmente relevante para nosotros, que es la semántica por teoría de modelos. Esta semántica declarativa se expresa en términos del modelo mínimo difuso (tal como se recoge en el trabajo [JMP09c]). Esta semántica reproduce en el contexto difuso la construcción clásica de modelo mínimo de Herbrand (véase [Llo87]), aceptada generalmente como la semántica declarativa de este tipo de programas.

En la última década se han realizado numerosas investigaciones de esta semántica usando teoría de modelos ([VP96, Ses02, SOD09]) en el área de la programación lógica difusa, aunque no concretamente para la programación lógica multi-adjunta. Dicha "laguna" para este marco fue cubierta por la semántica declarativa de [JMP09c] basada en lo que los autores llaman modelo mínimo de Herbrand difuso y cuyas nociones básicas exponemos a continuación.

Definición 3.3.8. *Sea $\mathcal{P}$ un programa lógico multi-adjunto con retículo asociado $(L, \leq)$. La interpretación $\mathcal{I}_{\mathcal{P}} = inf\{\mathcal{I}_j : \mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}\}$ se dice el modelo mínimo de Herbrand difuso*[7] *de $\mathcal{P}$.*

[7] En ocasiones diremos sólo modelo mínimo difuso o modelo mínimo.

El teorema siguiente justifica la definición de $\mathcal{I}_\mathcal{P}$ como el modelo mínimo de Herbrand difuso.

Teorema 3.3.9. *Sea $\mathcal{P}$ un programa lógico multi-adjunto con retículo asociado L. La aplicación $\mathcal{I}_\mathcal{P} = inf\{\mathcal{I}_j : \mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}\}$ es el menor modelo de $\mathcal{P}$.*

En dicho trabajo los autores demuestran este resultado, con la condición esencial de que el retículo asociado al programa sea un retículo multi-adjunto. Mostramos la necesidad de esta hipótesis para el Teorema 3.3.9 en el ejemplo siguiente.

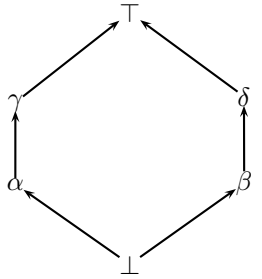
Ejemplo 3.3.10. *Si $(\&_\mathtt{G}, \leftarrow_\mathtt{G})$ es el par de conectivas cuyas funciones de verdad están definidas por*

$$\dot{\&}_\mathtt{G}(x,y) = inf\{x,y\} \quad y \quad x \dot{\rightarrow}_\mathtt{G} y = \begin{cases} \top, & si\ x \leq y \\ y, & en\ otro\ caso \end{cases}$$

no forman un par adjunto en el retículo que se muestra en la figura posterior puesto que, verificándose $\alpha \dot{\&} \delta \leq \beta$, no se cumple $\alpha \leq \beta \dot{\leftarrow} \delta$.

$\mathcal{R}_1 : \quad \langle p \quad \leftarrow_\mathtt{G} q; \quad \alpha \rangle$

$\mathcal{R}_2 : \quad \langle q \quad \leftarrow_\mathtt{G} ; \quad \beta \rangle$



	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$	$\mathcal{I}_4$	$\mathcal{I}_5$	$\mathcal{I}_6$	$\mathcal{I}_7$	$\mathcal{I}_8$	$\mathcal{I}_9$	$\mathcal{I}_{10}$	$\mathcal{I}_{11}$	$\mathcal{I}_{12}$
p	β	δ	$\top$	α	γ	δ	$\top$	α	γ	$\top$	α	γ
q	β	β	β	β	β	δ	δ	δ	δ	$\top$	$\top$	$\top$

Para el programa $\mathcal{P}$ dado, la tabla anterior muestra el conjunto $\mathcal{K} = \{\mathcal{I}_1, \ldots, \mathcal{I}_{12}\}$ de todos los modelos de Herbrand de $\mathcal{P}$.

En efecto, por definición de modelo, $\mathcal{I}_j$ es modelo de Herbrand de la regla $\mathcal{R}_2$ si, y sólo si, $\beta \leq \mathcal{I}_j(q)$, por lo que $\mathcal{I}_j(q)$ puede tomar los valores $\beta, \delta, \top \in L$. Además, $\mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{R}_1$ si, y sólo si, $\alpha \leq \mathcal{I}_j(p \leftarrow_{\mathtt{G}} q) = \mathcal{I}_j(p) \dot{\leftarrow}_{\mathtt{G}} \mathcal{I}_j(q)$. Entonces, si fijamos como $\mathcal{I}_j(q)$ uno de los valores anteriores, y añadimos la condición $\alpha \leq \mathcal{I}_j(p) \dot{\leftarrow}_{\mathtt{G}} \mathcal{I}_j(q)$, quedan determinados los posibles valores de $\mathcal{I}_j(p)$ y, en consecuencia, los posibles modelos $\mathcal{I}_j \in \mathcal{K}$.

Finalmente, es sencillo comprobar que el ínfimo de $\mathcal{K}$ es la interpretación $\mathcal{I}_\mathcal{P}$ tal que $\mathcal{I}_\mathcal{P}(p) = \bot$ y $\mathcal{I}_\mathcal{P}(q) = \beta$, pero $\mathcal{I}_\mathcal{P} \notin \mathcal{K}$, es decir, $\mathcal{I}_\mathcal{P}$ no es un modelo de Herbrand $\mathcal{P}$.

Los conceptos de interpretación y de modelo se pueden definir en términos conjuntistas. Así, una interpretación de $\mathcal{P}$, en lugar de entenderla como una aplicación, se define como la correspondiente relación binaria. En la siguiente proposición justificamos este hecho:

Proposición 3.3.11. *Sea $\mathcal{P}$ un programa lógico multi-adjunto con retículo asociado $(L, \leq)$. Sea $\mathcal{I}$ una interpretación de Herbrand de $\mathcal{P}$, es decir, una aplicación $\mathcal{I} : B_\mathcal{P} \longrightarrow L$. Entonces $\mathcal{I}$ determina una única relación binaria $R_\mathcal{I} \subset B_\mathcal{P} \times L$.*

Atendiendo a este resultado, es posible considerar una interpretación de Herbrand $\mathcal{I}$ como un conjunto $R_\mathcal{I}$ (que, igualmente, denotaremos por $\mathcal{I}$ cuando sea posible por mor de la simplicidad).

Dado que cabe considerar cada modelo como un conjunto (en particular, un subconjunto de $B_\mathcal{P} \times L$), entonces el modelo mínimo corresponde a la intersección de todos los modelos, como expresa el siguiente teorema.

Teorema 3.3.12. *El modelo mínimo difuso de $\mathcal{P}$ es la intersección de todos los modelos de Herbrand $\mathcal{P}$, es decir, $\mathcal{I} = \cap \mathcal{I}_j$, donde $\mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}$, para todo j.*

Una vez expuestos los conceptos sobre los que se construyen la semánticas de punto fijo y de modelo mínimo de Herbrand difuso para la programación lógica multi-adjunta, pasamos a investigar las relaciones entre ambas semánticas y con la semántica operacional descrita en la sección anterior.

En primer lugar, corroboramos la equivalencia entre la semántica declarativa por modelo mínimo difuso y la semántica de punto fijo construida en [MOV04] para la programación MALP.

Teorema 3.3.13. *Dado el programa lógico multi-adjunto $\mathcal{P}$ con retículo (completo) asociado $(L, \leq)$, $\mathcal{I}_\mathcal{P}$ es modelo mínimo de Herbrand difuso si, y sólo si, $\mathcal{I}_\mathcal{P}$ es el menor punto fijo de $T_\mathcal{P}$.*

Según este resultado, la semántica declarativa de un programa lógico multi-adjunto $\mathcal{P}$ se puede obtener mediante la iteración del operador $T_\mathcal{P}$ sobre el ínfimo de interpretaciones, esto es, la interpretación que asigna $\bot$ a todos los elementos de la base de Herbrand. Obsérvese que $T_\mathcal{P}$ puede no ser continuo. En ese caso, y a diferencia de la programación lógica pura, puede ser necesario un número infinito no numerable de iteraciones para alcanzar el punto fijo [MOV01d].

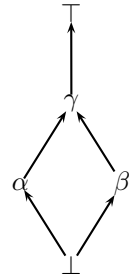
En el ejemplo presentado a continuación ilustramos esta equivalencia para un programa multi-adjunto.

Ejemplo 3.3.14. *Consideremos el siguiente programa lógico multi-adjunto $\mathcal{P}$ formado por hechos, reglas en la que podemos tomar cuerpo vacío o cuerpo con el elemento $\top$ del retículo asociado $(L, \leq)$ determinado por el diagrama de Hasse de la figura:*

$\mathcal{R}_1 : \quad \langle p \quad \leftarrow_{\mathtt{G}} ; \quad \alpha \rangle$

$\mathcal{R}_2 : \quad \langle p \quad \leftarrow_{\mathtt{G}} ; \quad \beta \rangle$

$\mathcal{R}_3 : \quad \langle q \quad \leftarrow_{\mathtt{G}} ; \quad \beta \rangle$



	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$	$\mathcal{I}_4$	$\mathcal{I}_5$	$\mathcal{I}_6$
p	γ	γ	γ	$\top$	$\top$	$\top$
q	β	γ	$\top$	β	γ	$\top$

Las funciones de verdad de las conectivas $(\&_{\mathtt{G}}, \leftarrow_{\mathtt{G}})$ (que siguen la lógica de Gödel*), forman un par adjunto en el retículo $(L, \leq)$ escogido.*

Para este programa existen seis modelos distintos (las interpretaciones de Herbrand $\mathcal{I}_1, \ldots, \mathcal{I}_6$ de la tabla anterior) siendo $\mathcal{I}_\mathcal{P} = \mathcal{I}_1$ el modelo mínimo de Herbrand difuso. En efecto, se puede comprobar fácilmente que $\mathcal{I}_\mathcal{P}$ es el ínfimo del conjunto de $\{\mathcal{I}_1, \ldots, \mathcal{I}_6\}$.

Además, resulta ser un punto fijo, puesto que

$$T_\mathcal{P}(\mathcal{I}_\mathcal{P})(p) = sup\{\alpha \dot{\&}_{\mathtt{G}} \top, \beta \dot{\&}_{\mathtt{G}} \top\} = sup\{\alpha, \beta\} = \gamma = \mathcal{I}_\mathcal{P}(p)$$

$$T_\mathcal{P}(\mathcal{I}_\mathcal{P})(q) = sup\{\beta \dot{\&}_{\mathtt{G}} \top\} = \beta = \mathcal{I}_\mathcal{P}(q)$$

y su unicidad es también sencilla de comprobar. Tengamos en cuenta que, como era de esperar, $\mathcal{I}_\mathcal{P}$ satisface los Teoremas 3.3.9 y 3.3.12.

Observamos, por otra parte, que la semántica operacional (cuya equivalencia con la de punto fijo se trata en [MOV01d]) no goza en general (como ocurre en este caso) de garantías de completitud: α, β son respuestas correctas para el programa $\mathcal{P}$ y el objetivo p y, por tanto, γ también es una respuesta correcta; sin embargo, γ no es una respuesta computada.

Esta equivalencia entre ambas semánticas declarativas –las que se obtienen por punto fijo y por modelo mínimo–, lleva consigo otras implicaciones. A través de la equivalencia contemplada en [MOV04] entre la semántica por punto fijo y la operacional, resulta que la semántica por modelo mínimo es también equivalente a la operacional en los términos precisados en dicho trabajo. Es decir, la corrección fuerte está siempre garantizada, pero sólo se alcanza la completitud aproximada incorporando a los programas multi-adjuntos la noción de reductante.

Pasamos ahora a caracterizar la noción de respuesta correcta haciendo uso del modelo mínimo difuso. Además, dado que en programación lógica pura el modelo mínimo de Herbrand no permite caracterizar completamente las respuestas correctas, los resultados obtenidos en programación lógica multi-adjunta mejoran los clásicos de la programación lógica[8]. Esta caracterización permite en la práctica la obtención del conjunto de respuestas correctas de un programa a partir del modelo mínimo difuso, tal como ilustra el Ejemplo 3.3.17 de esta sección.

Definición 3.3.15 ([MOV04])**.** *Sea $\mathcal{P}$ un programa multi-adjunto y G un objetivo. El par $\langle \lambda; \theta \rangle$ es una respuesta correcta para $\mathcal{P}$ y G si, y sólo si, $\lambda \leq \mathcal{I}_j(G\theta)$, para todo modelo de Herbrand $\mathcal{I}_j$ de $\mathcal{P}$.*

El siguiente teorema caracteriza la noción de respuesta correcta $\langle \lambda; \theta \rangle$ en términos del modelo mínimo de Herbrand difuso, como anticipábamos, a la vez que muestra que $\mathcal{I}_\mathcal{P}(G\theta)$ es la mejor respuesta correcta que puede obtenerse para el objetivo G en el programa $\mathcal{P}$.

Teorema 3.3.16. *Sea $\mathcal{P}$ un programa multi-adjunto y G un objetivo. El par $\langle \lambda; \theta \rangle$ es una respuesta correcta para $\mathcal{P}$ y G si, y sólo si, $\lambda \leq \mathcal{I}_\mathcal{P}(G\theta)$, donde $\mathcal{I}_\mathcal{P}$ es el modelo mínimo difuso de $\mathcal{P}$.*

Para adaptar la notación conjuntista usual de la programación lógica al contexto multi-adjunto, escribimos $(G\theta, \alpha) \in \mathcal{I}_\mathcal{P}$ si $G\theta$ es un objetivo básico $\mathcal{I}_\mathcal{P}(G\theta) = \alpha$ y

[8] En ello influirá decisivamente el carácter no refutacional del lenguaje multi-adjunto.

el modelo mínimo difuso se concibe como un conjunto. La diferencia más relevante entre esta definición de respuesta correcta con la dada para la programación lógica clásica radica en el caracter refutacional de la resolución-SLD, tal y como se indica en [Llo87], mientras que la del marco multi-adjunto no lo es. En el siguiente ejemplo se ilustra cómo extraer las respuestas correctas dado el modelo mínimo difuso.

Ejemplo 3.3.17. *Para el programa $\mathcal{P}$ siguiente cuyo retículo asociado $(L, \leq)$ es el considerado en el Ejemplo 3.3.14,*

$$\begin{array}{lllll} \mathcal{R}_1: & \langle p(a) & \leftarrow_{\mathtt{G}} ; & & \alpha\rangle \\ \mathcal{R}_2: & \langle p(b) & \leftarrow_{\mathtt{G}} ; & & \beta\rangle \\ \mathcal{R}_3: & \langle q(a) & \leftarrow_{\mathtt{G}} p(a); & & \gamma\rangle \end{array}$$

es sencillo comprobar que el modelo mínimo de Herbrand difuso $\mathcal{I}_\mathcal{P}$ está definido por $\mathcal{I}_\mathcal{P}(p(a)) = \alpha$, $\mathcal{I}_\mathcal{P}(p(b)) = \beta$, $\mathcal{I}_\mathcal{P}(q(a)) = \alpha$, $\mathcal{I}_\mathcal{P}(q(b)) = \bot$. Entonces:

i) Para el objetivo $p(a)$ se obtiene el conjunto de respuestas correctas

$$\{\langle \lambda; id\rangle : \lambda \in L, \lambda \leq \alpha\}$$

ii) Para el objetivo $p(b)$ se obtiene el conjunto de respuestas correctas

$$\{\langle \lambda; id\rangle : \lambda \in L, \lambda \leq \beta\}$$

iii) Para el objetivo $q(a)$ se obtiene el conjunto de respuestas correctas

$$\{\langle \lambda; id\rangle : \lambda \in L, \lambda \leq \alpha\}$$

iv) Para el objetivo $p(x)$, el conjunto de respuestas correctas es

$$\{\langle \lambda; \theta\rangle : \lambda \in L, \lambda \leq \mathcal{I}_\mathcal{P}(p(x)\theta)\} = \{\langle \bot; \{X/a\}\rangle, \langle \alpha; \{X/a\}\rangle, \langle \bot; \{X/b\}\rangle, \langle \beta, \{X/b\}\rangle\}$$

Recuérdese que el resultado de aplicar el modelo mínimo $\mathcal{I}_\mathcal{P}$ sobre $p(x)$ nos da $\mathcal{I}_\mathcal{P}(p(x)) = inf\{\mathcal{I}_\mathcal{P}(p(x)\sigma) : p(x)\sigma$ es básica $\} =^9 inf\{\mathcal{I}_\mathcal{P}(p(a)), \mathcal{I}_\mathcal{P}(p(b))\}$ $= inf\{\alpha, \beta\} = \bot$.

[9] Las sustituciones contemplarán sólo términos del universo de Herbrand del programa en lugar de variables.

v) Para el objetivo $q(x)$, el conjunto de respuestas correctas es

$$\{\langle\lambda;\theta\rangle : \lambda \in L, \lambda \leq \mathcal{I}_{\mathcal{P}}(q(x)\theta)\} = \{\langle\alpha;\{X/a\}\rangle, \langle\bot;\{X/b\}\rangle\}$$

De manera análoga al caso anterior, se tiene ahora que la interpretación $\mathcal{I}_{\mathcal{P}}(q(x)) = inf\{\mathcal{I}_{\mathcal{P}}(q(a)), \mathcal{I}_{\mathcal{P}}(q(b))\} = inf\{\alpha, \bot\} = \bot$ (resulta sencillo comprobar que el modelo mínimo ha de estar definido de este modo a partir de las fórmulas $q(a), q(b)$).

A continuación aportamos la corrección de la semántica procedimental mediante un razonamiento similar al realizado por [Llo87] para la lógica pura, con la diferencia del caracter difuso y no refutacional, ya mencionado, de nuestro lenguaje.

Teorema 3.3.18 (Corrección parcial)**.** *Sea $\mathcal{P}$ un programa lógico multi-adjunto, A un objetivo atómico y $\langle\lambda;\theta\rangle$ una respuesta computada difusa para A en $\mathcal{P}$. Entonces, $\langle\lambda;\theta\rangle$ es una respuesta correcta para A en $\mathcal{P}$.*

Este resultado es demostrado formalmente en [Pen10]. La completitud de la semántica operacional, por su parte, es demostrada en el siguiente siguiente, recogido en [MOV04]Cabe precisar que su justificación emplea la noción de reductante (véase [MOV04, JMP06b, JMP07a, JMP09a]).

Teorema 3.3.19 (Completitud aproximada)**.** *Sea $\mathcal{P}$ un programa, A un átomo básico y $\langle\lambda;id\rangle$ una respuesta correcta para A en $\mathcal{P}$. Entonces, existe una secuencia de respuestas computadas $\langle\lambda_n;id\rangle$ para A en $\mathcal{P}$ tal que $\lambda \leq sup\{\lambda_n : n \in \mathbb{N}\}$.*

Como se recoge en [MO02], en el caso proposicional se puede obtener la mayor respuesta correcta (esto es, el supremo de las respuestas correctas) para un objetivo A y un programa $\mathcal{P}$ mediante el operador $T_{\mathcal{P}}$. Además, suponiendo terminación y continuidad, también se alcanza la mayor respuesta computada.

Con respecto a la noción de *consecuencia lógica*, en [MPV12a] identificamos por primera vez este concepto para el lenguaje MALP, y lo ubicamos en su semántica declarativa en relación al de *respuesta correcta*. Damos a continuación su definición en términos puramente multi-adjuntos.

Definición 3.3.20. *Sea $\mathcal{P}$ un programa lógico multi-adjunto, y $\mathcal{A} = \langle A;\alpha\rangle$ una fórmula multi-adjunta. $\mathcal{A}$ es una* consecuencia lógica *para $\mathcal{P}$ si, y sólo si, todo modelo de $\mathcal{P}$ es modelo de $\mathcal{A}$.*

El siguiente teorema, en el que se considera el modelo mínimo de Herbrand en lugar de todos los modelos, se demuestra fácilmente sin más que considerar la definición de *modelo mínimo de Herbrand difuso*.

Teorema 3.3.21. *Sea $\mathcal{P}$ un programa multi-adjunto, y $\mathcal{A}$ una fórmula multi-adjunta. $\mathcal{A} = \langle A; \alpha \rangle$ es una consecuencia lógica de $\mathcal{P}$ si y sólo si $\mathcal{I}_\mathcal{P}$ es modelo de $\mathcal{A}$.*

Los siguientes teoremas de [MPV12a] relacionan la noción de *consecuencia lógica* con la de *respuesta correcta* en términos multi-adjuntos:

Teorema 3.3.22. *Sea $\mathcal{P}$ un programa multi-adjunto, y $\mathcal{G}$ un objetivo. Si $\langle \lambda; \theta \rangle$ es una respuesta correcta para $\mathcal{P}$ y $\mathcal{G}$, entonces $\langle \mathcal{G}\theta; \lambda \rangle$ es una consecuencia lógica de $\mathcal{P}$.*

Demostración. Sea $\mathcal{I}_\mathcal{P}$ es modelo mínimo de $\mathcal{P}$. Vemos que, por definición de respuesta correcta se requiere que $\lambda \leq \mathcal{I}_\mathcal{P}(\mathcal{G}\theta)$. □

Teorema 3.3.23. *Sea $\mathcal{P}$ un programa multi-adjunto, y $\mathcal{A} = \langle A; \alpha \rangle$ una fórmula tal que A es un objetivo. Si $\mathcal{A}$ es una consecuencia lógica de $\mathcal{P}$ entonces el par $\langle \alpha; id \rangle$ es una respuesta correcta para $\mathcal{P}$ y A.*

Demostración. Por el Teorema 3.3.21, $\mathcal{I}_\mathcal{P}$ es modelo de $\mathcal{A}$, por lo que $\alpha \leq \mathcal{I}_\mathcal{P}(A)$ y, por tanto, $\langle \alpha; id \rangle$ es una respuesta correcta para $\mathcal{P}$ y $\mathcal{A}$. □

El siguiente teorema es la extensión del dado por [Llo87], que caracteriza el modelo mínimo de Herbrand con el conjunto de fórmulas de la base de Herbrand que son consecuencias lógicas de un programa:

Teorema 3.3.24. *Sea $\mathcal{I}_\mathcal{P}$ el modelo mínimo de Herbrand de un programa multi-adjunto $\mathcal{P}$ con retículo asociado L. Entonces, $\mathcal{I}_\mathcal{P} = \{\mathcal{A} = \langle A; \alpha \rangle \in \mathcal{B}_\mathcal{P} : \mathcal{A}$ es una consecuencia lógica de $\mathcal{P}\}$.*

Demostración. Si $\mathcal{A} \in \mathcal{I}_\mathcal{P} \subset \mathcal{B}_\mathcal{P} \times L$, entonces $\alpha \leq \mathcal{I}_\mathcal{P}(A)$, de modo que $\mathcal{A}$ es una consecuencia lógica de $\mathcal{P}$ y esto muestra que $\mathcal{I}_\mathcal{P} \subseteq \{\mathcal{A} \in \mathcal{B}_\mathcal{P} : \mathcal{A}$ es una consecuencia lógica de $\mathcal{P}\}$. La inclusión inversa es análoga. □

Terminamos esta sección indicando que en el trabajo [MPV14c], hemos realizado una reformulación de la semántica declarativa del lenguaje MALP empleando conjuntos difusos. En particular, y siguiendo el estilo conjuntista de [Llo87] el concepto de *interpretación* se ha tomado como un conjunto (en este caso, difuso) sobre la

base de Herbrand difusa. En este caso, la relación de orden entre interpretaciones, $\mathcal{I}_1 \leq \mathcal{I}_2$, se desprende naturalmente de la relación de inclusión de conjuntos difusos $\mathcal{I}_1 \subseteq \mathcal{I}_2$. Con objeto de demostrar la corrección de esta aproximación, reformulamos todos los teoremas necesarios vistos en esta sección. En particular, probamos que el modelo mínimo de Herbrand difuso de un programa es, de hecho, la intersección de los modelos de dicho programa. Demostramos también que una respuesta correcta (o una consecuencia lógica) para la versión sin conjuntos difusos de la semántica declarativa de MALP se corresponde a la respuesta correcta (o la consecuencia lógica) según la nueva versión con conjuntos difusos, así como la corrección de la semántica operacional con respecto a esta última semántica declarativa, o la equivalencia con la semántica por punto fijo.

3.4. Coste computacional de las derivaciones

En esta sección nos centramos en medir de una manera óptima el coste del mecanismo operacional de MALP. El cálculo del coste computacional necesario para ejecutar un objetivo en programación declarativa consiste normalmente en contar el número de pasos aplicados hasta alcanzar sus soluciones. Este método, que es suficientemente preciso para la fase operacional, se vuelve inadecuado en la interpretativa. Para ello, recogemos de [Mor13] una nueva medida de coste (interpretativo) más refinada y realista para calcular el esfuerzo computacional requerido para resolver un objetivo. Este método (presentado por vez primera en [MM09c]) mejora otras medidas de coste más simples, propuestas también por nuestro grupo de investigación, ya que tiene en cuenta no sólo el número de pasos interpretativos dados durante las derivaciones, sino también su complejidad (entendida como el número de llamadas intermedias a otras conectivas y a operadores primitivos).

La importancia que damos en este trabajo a la exactitud de la medida de coste se justifica porque la correcta valoración del coste computacional de la ejecución de programas puede jugar un papel muy importante para dotar de propiedades de eficiencia a otras técnicas de transformación de programas como plegado/desplegado, evaluación parcial, etc. Dichas técnicas, que están siendo desarrolladas en nuestro grupo, ya están obteniendo interesantes resultados gracias, en parte, a que con la medida de coste que proponemos aquí se muestra de una manera totalmente exacta la pérdida o ganancia de eficiencia en los programas transformados.

A continuación detallamos la medida del coste computacional de la ejecución de

programas lógicos (y, por extensión, en programas lógico difusos) y explicamos con detalle la medida más realista mencionada anteriormente.

Como ya se ha anticipado, una forma clásica y sencilla de estimar el coste computacional asociado a una derivación consiste en contar el número de pasos computacionales llevados a cabo en ella. Así, dada la derivación D, definimos su:

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

Sirva el ejemplo siguiente para ilustrar en la práctica las medidas de coste $\mathcal{O}_c(D)$ y $\mathcal{I}_c(D)$ sobre la derivación obtenida al ejecutar un objetivo sobre un programa lógico multi-adjunto.

Ejemplo 3.4.1. *Sea $\mathcal{P}$ el siguiente programa lógico multi-adjunto:*

$$\begin{array}{lllll} \mathcal{R}_1: & p(X) & \leftarrow_{\mathtt{P}} & \&_{\mathtt{G}}(\vee_{\mathtt{L}}(q(X), 0.6), r(X)) & \textit{with}\ \ 0.9 \\ \mathcal{R}_2: & q(a) & \leftarrow & & \textit{with}\ \ 0.8 \\ \mathcal{R}_3: & r(X) & \leftarrow & & \textit{with}\ \ 0.7 \end{array}$$

donde las etiquetas `L`*,* `G` *y* `P` *aluden, respectivamente, a la* lógica de Lukasiewicz*, la* lógica de Gödel *y a la* lógica del producto*. Ahora, podemos generar la siguiente derivación admisible (subrayamos el átomo seleccionado en cada paso para una mayor claridad):*

$$\begin{array}{ll} \langle \underline{p(X)};\ id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(\underline{q(X1)}, 0.6), r(X1)));\{X/X_1\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), \underline{r(a)}));\{X/a, X_1/a\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), 0.7));\{X/a, X_1/a, X_2/a\}\rangle & \end{array}$$

Si completamos la derivación previa aplicando 3 pasos interpretativos obtenemos la respuesta computada difusa final $\langle 0.63; \{X/a\}\rangle$, generando la siguiente derivación interpretativa D_1:

$$\begin{array}{lll} D_1: & [\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{\vee_{\mathtt{L}}(0.8, 0.6)}, 0.7)); \theta\rangle & \rightarrow_{IS} \\ & \langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(1, 0.7)}); \theta\rangle & \rightarrow_{IS} \\ & \langle \underline{\&_{\mathtt{P}}(0.9, 0, 7)}; \theta\rangle & \rightarrow_{IS} \\ & \langle 0.63; \theta\rangle] & \end{array}$$

Como resultado, en el Ejemplo 3.4.1 los costes operacional e interpretativo para la derivación D_1 son es $\mathcal{O}_c(D_1) = 3$ y $\mathcal{I}_c(D_1) = 3$. Es de esperar que $\mathcal{I}_c$ informe sobre el número de conectivas evaluadas en una derivación, del mismo modo que $\mathcal{O}_c$ informa del número de átomos evaluados. No obstante, esto no se da pues, por su definición, $\mathcal{I}_c$ informa únicamente de aquellas conectivas indicadas explícitamente en los cuerpos de las reglas. Esta característica puede provocar resultados aberrantes, como el ilustrado en el siguiente ejemplo:

Ejemplo 3.4.2. *Una versión simplificada de la regla $\mathcal{R}_1$, cuyo cuerpo sólo contiene un símbolo agregador es:*

$$\mathcal{R}_1^* : p(X) \leftarrow_{\mathtt{P}} @_1(q(X), r(X)) \quad with \quad 0.9$$

donde $@(x_1, x_2) \triangleq \&_{\mathtt{G}}(\vee_{\mathtt{L}}(x_1, 0.6), x_2)$.

Se observa que $\mathcal{R}_1^*$ tiene exactamente el mismo significado (la misma interpretación) que $\mathcal{R}_1$, aunque difieren en su sintaxis. Ahora, usamos la regla $\mathcal{R}_1^*$ en lugar de $\mathcal{R}_1$ para generar la siguiente derivación D_1^* la cual retorna exactamente la misma respuesta computada difusa que D_1:

$$\begin{array}{lll}
D_1^* : & [\langle \underline{p(X)};\ id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1^*} \\
& \langle \&_{\mathtt{P}}(0.9, @_1(\underline{q(X1)}, r(X1)); \{X/X_1\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\
& \langle \&_{\mathtt{P}}(0.9, @_1(0.8, \underline{r(a)})); \{X/a, X_1/a\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\
& \langle \&_{\mathtt{P}}(0.9, \underline{@_1(0.8, 0.7)}); \{X/a, X_1/a, X_2/a\}\rangle & \rightarrow_{IS} \\
& \langle \underline{\&_{\mathtt{P}}(0.9, 0.7)}; \{X/a, X_1/a, X_2/a\}\rangle & \rightarrow_{IS} \\
& \langle 0.63; \{X/a, X_1/a, X_2/a\}\rangle] &
\end{array}$$

Se puede apreciar que en ambas derivaciones obtenemos el mismo coste operacional $\mathcal{O}_c(D_1) = \mathcal{O}_c(D_1^*) = 3$. Sin embargo, aunque las conectivas $\&_G$ y $\vee_L$ han sido evaluadas en ambas derivaciones, se da que $\mathcal{I}_c(D_1^*) = 2 \neq 3 = \mathcal{I}_c(D_1)$. Esto es, su coste, medido según se indica, es menor pese a que implica la evaluación de las mismas conectivas $\vee_L$ y $\&_G$, además de la de $@_1$.

Como solución a este efecto, se puede considerar una nueva medida de costeque considere la definición de pesos asociados a las conectivas.

Definimos la noción de peso de una conectiva siguiendo el ejemplo de [Mor13].

Definición 3.4.3 (Peso de una conectiva)**.** *Sea $(L, \preceq)$ un retículo multi-adjunto que contiene la definición de conectiva $@(x_1, \ldots, x_n) \triangleq E$, donde la secuencia (posiblemente vacía) de operadores primitivos y conectivas que aparecen en E son respectivamente $op_1, \ldots, op_r$ y $@_1, \ldots, @_s$. El peso de una conectiva $@$, denotado por $\mathcal{W}(@)$,*

es un par de números naturales $\langle m, n\rangle$ *obtenidos por las funciones* $Calls$ *y* $Opers$ *(es decir,* $Calls(@) = m$ *y* $Opers(@) = n$*), como sigue:*

$$Calls(@) = 1 + Calls(@_1) + \cdots + Calls(@_s)$$

$$Opers(@) = r + Opers(@_1) + \cdots + Opers(@_s)$$

Finalmente, respecto a estas definiciones para la medida de coste concluimos lo siguiente. En cuanto a la función $Calls$, ésta incluye el coste de realizar la llamada correspondiente a la propia conectiva, además de las llamadas indirectas a otras conectivas, deshaciendo así el efecto pernicioso de las medidas anteriores de [JMP07b].Respecto a $Opers$, esta función reproduce la noción más débil de peso de una conectiva recogida en [JMP07b], donde se cuenta la evaluación directa e indirecta de los operadores primitivos involucrados en su definición. Esta medida mejorada de *coste interpretativo* de una derivación D, llamada $\mathcal{I}_c^+(D)$, consiste en un par obtenido por la acumulación de pesos de las conectivas evaluadas en todos los pasos interpretativos llevados a cabo en la derivación D.

Ejemplo 3.4.4. *Consideremos de nuevo el conjunto de conectivas que hemos usado en los ejemplos previos, por los cuales tenemos que:*

$$\begin{array}{rcl}
\mathcal{W}(\vee_{\mathtt{L}}) & = & \langle Calls(\vee_{\mathtt{L}}), Opers(\vee_{\mathtt{L}})\rangle \\
& = & \langle 1, 2\rangle \\
\mathcal{W}(\&_{\mathtt{G}}) & = & \langle Calls(\&_{\mathtt{G}}), Opers(\&_{\mathtt{G}})\rangle \\
& = & \langle 1, 1\rangle \\
\mathcal{W}(\&_{\mathtt{P}}) & = & \langle Calls(\&_{\mathtt{P}}), Opers(\&_{\mathtt{P}})\rangle \\
& = & \langle 1, 1\rangle \\
\mathcal{W}(@_1) & = & \langle Calls(@_1), Opers(@_1)\rangle \\
& = & \langle 1 + Calls(\vee_{\mathtt{L}}) + Calls(\&_{\mathtt{G}}), 0 + Opers(\vee_{\mathtt{L}}) + Opers(\&_{\mathtt{G}})\rangle \\
& = & \langle 1 + 1 + 1, 0 + 1 + 2\rangle \\
& = & \langle 3, 3\rangle.
\end{array}$$

Así, para la derivación D_1 *en el Ejemplo 3.4.1 tenemos:*

$$\begin{array}{rcl}
\mathcal{I}_c^+(D_1) & = & \mathcal{W}(\vee_{\mathtt{L}}) + \mathcal{W}(\&_{\mathtt{G}}) + \mathcal{W}(\&_{\mathtt{P}}) \\
& = & \langle 1, 2\rangle + \langle 1, 1\rangle + \langle 1, 1\rangle \\
& = & \langle 3, 4\rangle
\end{array}$$

mientras que para la derivación D_1^* (la cual realiza 2 pasos interpretativos) en el Ejemplo 3.4.2 la situación es:

$$\begin{aligned}\mathcal{I}_c^+(D_1^*) &= \mathcal{W}(@_1) + \mathcal{W}(\&_{\mathtt{P}}) \\ &= \langle 3,3\rangle + \langle 1,1\rangle \\ &= \langle 4,4\rangle\end{aligned}$$

Con respecto a las medidas de coste sobre las derivaciones D_1 y D_1^*, se observa que el coste operacional $\mathcal{O}_c$ de ambas coincide, lo cual es natural. No ocurre así con el coste interpretativo $\mathcal{I}_c$, que considera menos costosa D_1^*:

$$\mathcal{I}_c(D_1) = 3 > 2 = \mathcal{I}_c(D_1^*)$$

$$\mathcal{I}_c^*(D_1) = \mathcal{I}_c^*(D_1^*) = 4$$

si bien la nueva técnica de medición aporta un resultado diferente que revela que D_1^* es, de hecho, más costosa (pues en última instancia evalúa los mismos operadores que D_1, y hace una llamada extra a la definición de la conectica $@_1$:

$$\mathcal{I}_c^+(D_1) = \langle 3,4\rangle < \mathcal{I}_c^+(D_1^*) = \langle 4,4\rangle.$$

A lo largo de los ejemplos de esta sección hemos mostrado que las medidas de coste que sólo cuentan el número de conectivas invocadas directamente ($\mathcal{I}_c$) provocan efectos irreales cuando aparecen conectivas complejas (que llaman a un gran número de operadores primitivos). La medida de coste $\mathcal{I}_c^+$ tiene en cuenta el peso de cada conectiva, así como el coste de obtener su definición. Empleando medidas de coste realistas, como esta última, es posible determinar que la regla $\mathcal{R}_1$ es más eficiente que $\mathcal{R}_1^*$ en tanto que evita el coste de resolver el cuerpo de una conectiva más.

En un caso extremo, pero ni mucho menos irreal o forzado, como el que exponemos a continuación observamos la relevancia del uso de una medida de coste apropiada. Este ejemplo puede ser producido efectivamente tras la aplicación de técnicas de transformación como las usadas en nuestro grupo. Asumimos la siguiente secuencia de definiciones de conectivas:

$$\begin{aligned}@_{100}(x_1,x_2) &\triangleq @_{99}(x_1,x_2)\\ @_{99}(x_1,x_2) &\triangleq @_{98}(x_1,x_2)\\ @_{98}(x_1,x_2) &\triangleq @_{97}(x_1,x_2)\\ @_{97}(x_1,x_2) &\triangleq @_{96}(x_1,x_2)\\ &\vdots\\ @_1(x_1,x_2) &\triangleq x_1 * x_2\end{aligned}$$

Para la evaluación de los objetivos (tan distintos entre sí) de la forma $@_{100}(0.9, 0.8)$ y $@_1(0.9, 0.8)$, las medidas de coste $\mathcal{I}_c$ y $\mathcal{I}_c^*$ (basadas en el número de pasos interpretativos [JMP06c] y en pesos de pasos interpretativos sin tener en cuenta el coste de las llamadas a las conectivas [JMP07b]) asignan a ambas computaciones el mismo coste interpretativo, 1.

En cambio, ambos objetivos exhiben costes distintos empleando la medida $\mathcal{I}_c^+$ presentada en [Mor13]. En particular, la función $Calls$ en devuelve 100 para el primer objetivo y 1 para el segundo. La medida de coste $\mathcal{I}_c^+$ se presenta, por tanto, como la más razonable y precisa para guiar procesos como el de la transformación automática de programas en el sentido de lograr programas más eficientes, rápidos y, en última instancia, útiles.

3.5. Pasos interpretativos cortos

Otro método para medir de forma precisa el coste computacional de la fase imperativa, desarrollado en nuestro grupo, fue presentado por primera vez en [MM09a], y referida en [MM09b] y [MMPV10b]. Este método mantiene intacta la medida de coste y, en cambio, modifica la propia definición de paso interpretativo mediante la introducción de los llamados pasos interpretativos cortos. La forma que tienen los pasos cortos de evaluar una conectiva es sustituirla, en primer lugar, por su definición, que puede ser en términos de otras conectivas o de operadores primitivos (decimos entonces que expande la definicón de la conectiva). De este modo, el número de pasos cortos realizados se corresponde a la medida proporcionada por $\mathcal{I}_c^+$ pero, además, este enfoque puede incluirse de forma natural en el entorno de programación $\mathcal{FLOPER}$.

A continuación presentamos la definición de esta nueva técnica.

Definición 3.5.1 (Paso interpretativo corto)**.** *Sea $\mathcal{P}$ un programa MALP, $\mathcal{Q}$ 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 una conectiva definido en el retículo multi-adjunto $(L, \leq)$ asociado a $\mathcal{P}$, y $r_1, \ldots, r_n$ son elementos de L. Formalizamos la noción de* paso interpretativo corto *como un sistema de transición de estados, cuya relación de transición $\rightarrow_{SIS} \subseteq (\mathcal{E} \times \mathcal{E})$ es la menor relación binaria $\mathcal{E} \times \mathcal{E}$ 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) $\langle Q[\Omega(r_1,\ldots,r_n)];\sigma\rangle \rightarrow_{SIS1} \langle Q[\Omega(r_1,\ldots,r_n)/E'];\sigma\rangle$, *si* Ω *es una conectiva definida 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) $\langle Q[\Omega(r_1,\ldots,r_n)];\sigma\rangle \rightarrow_{SIS2} \langle Q[\Omega(r_1,\ldots,r_n)/r];\sigma\rangle$, *si* Ω *es un operador primitivo que, una vez evaluado con sus parámetros* $r_1,\ldots,r_n$, *produce el resultado* $r \in L$.

Empleamos los símbolos $\rightarrow_{SIS1}$ y $\rightarrow_{SIS2}$ para distinguir qué regla de computación *interpretativa corta* se ha empleado en cada caso. En adelante, a una secuencia de *pasos interpretativos cortos* la llamaremos *derivación interpretativa*, que ya no se referirá a una secuencia de *pasos interpretativos*.

Esto implica una revisión de la Definición 3.2.7, aunque esta revisión no afecta a la esencia de "respuesta computada difusa", ya que sólo afecta al número de pasos y la longitud de la derivación, y ambos tipos de derivación llegan a los mismos estados finales (a las mismas respuestas computadas difusas finales).

Ejemplo 3.5.2. *Recordando de nuevo la a.c.a. obtenida en el Ejemplo 3.4.1 se puede llegar a la respuesta computada difusa* $\langle 0.63; \{X/a\}\rangle$ *(conseguida en ese caso por medio de pasos interpretativos). Ahora generaremos la misma f.c.a a través de la siguiente derivación interpretativa* D_2, *la cual está basada en pasos interpretativos cortos:*

$$
\begin{array}{lll}
D_2: & [\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{\vee_{\mathtt{L}}(0.8, 0.6)}, 0.7)); \{X/a\}\rangle & \rightarrow_{SIS1} \\
& \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(min(1, \underline{0.8 + 0.6}), 0.7)); \{X/a\}\rangle & \rightarrow_{SIS2} \\
& \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{min(1, 1.4)}, 0.7)); \{X/a\}\rangle & \rightarrow_{SIS2} \\
& \langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(1, 0.7)}); \{X/a\}\rangle & \rightarrow_{SIS1} \\
& \langle \&_{\mathtt{P}}(0.9, \underline{min(1, 0.7)}); \{X/a\}\rangle & \rightarrow_{SIS2} \\
& \langle \underline{\&_{\mathtt{P}}(0.9, 0.7)}; \{X/a\}\rangle & \rightarrow_{SIS1} \\
& \langle \underline{0.9 * 0.7}; \{X/a\}\rangle & \rightarrow_{SIS2} \\
& \langle 0.63; \{X/a\}\rangle] &
\end{array}
$$

De la misma manera, regresando al Ejemplo 3.4.2 en el cual cambiábamos la regla $\mathcal{R}_1$ *por otra semánticamente equivalente pero con distinta sintaxis, podemos reconstruir 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* $\rightarrow_{SIS1}$ *en la L-expresión* $\&_{\mathtt{P}}(0.9, \underline{@_1(0.8, 0.7)})$, *ésta se transforma en el subobjetivo* $\&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), 0.7))$, *y los estados posteriores de la derivación interpretativa coinciden con los de la derivación* D_2 *previa.*

Del mismo modo que en la Sección 3.4, analizamos aquí las medidas de coste correspondientes a las derivaciones D_1, D_1^*, D_2 y D_2^*, sólo que a diferencia de en esa sección, en lugar de alternar entre las medidas $\mathcal{I}_c^+$ e $\mathcal{I}_c$, usamos aquí siempre la segunda (que cuenta el número de pasos interpretativos sin considerar la complejidad de cada conectiva) y variamos entre usar pasos interpretativos y pasos interpretativos cortos (que recogen en su propia definición la complejidad de las conectivas).

Obsérvese, en primer lugar, que el coste operacional de las cuatro derivaciones es el mismo, como era de esperar, ya que nada en la fase operacional ha sido alterado. En cambio, mientras que $\mathcal{I}_c(D_1) = 3 > 2 = \mathcal{I}_c(D_1^*)$, ahora resulta que $\mathcal{I}_c(D_2) = 7 < 8 = \mathcal{I}_c(D_2^*)$. Es el uso de los pasos interpretativos cortos, según la Definición 3.5.1, en las derivaciones D_2 y D_2^* lo que produce estas diferencias en los resultados, y premite decidir cuál de ellas tiene mejor comportamiento computacional.

Como ha quedado suficientemente justificado en el apartado anterior, el uso de la medida $\mathcal{I}_c$ en derivaciones basadas en pasos interpretativos produce ressultados incorrectos, pues todas las conectivas se ponderan con el mismo coste computacional, de forma independiente a su complejidad real. Resuelve este problema la noción de paso interpretativo corto al hacer visible en la propia derivación todas las conectivas y operadores primitivos involucrados en la definición de una conectiva encontrada en cualquier estado de la derivación. Nótese cómo, en la derivación D_2, las definiciones de tres conectivos ($@_1, \&_{\mathtt{G}}$ y $\&_{\mathtt{P}}$) se han expandido en tres pasos interpretativos cortos de tipo 1, y en cuatro pasos interpretativos cortos de tipo 2 se han resuelto los operadores primitivos $+$, min (dos veces) y $*$. En D_2^* se ha realizado el mismo esfuerzo computacional, más un paso interpretativo corto de tipo 1 extra para expandir la definición de $@_1$, lo cual justifica que $\mathcal{I}_c(D_2) = 7 < 8 = \mathcal{I}_c(D_2^*)$ y contradice las medidas imprecisas del Ejemplo 3.4.1, en el sentido de que el esfuerzo computacional correspondiente a las derivaciones $D-1$ y D_2 (usando ambas la regla $\mathcal{R}_1$) es ligeramente inferior al que corresponde a las derivaciones D_1^* y D_2^* (que usan la regla $\mathcal{R}_1^*$ con el conectivo añadido $@_1$).

Esta técnica alcanza una precisión equivalente a la obtenida por la medida de coste optimizada $\mathcal{I}_c^+$, y permite una comparación adecuada de la complejidad computacional de programas obtenidos mediante técnicas de transformación como el marco plegado/desplegado descrito en [JMP05a, GM08b]. Del mismo modo que en el ejemplo de la sección anterior –que, recordamos, puede darse realmente en secuencias de transformación de programas–, partimos de la siguiente definición altamente anidada 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, $@_1(x_1, x_2) \triangleq x_1 * x_2$.

Para dos expresiones de la forma $@_{100}(0.9, 0.8)$ y $@_1(0.9, 0.8)$, las medidas de coste consistentes en contar el número de pasos interpretativos y en sus pesos correspondientes conforme a [JMP07b] asignan el mismo coste interpretativo (la unidad) a ambas derivaciones. En cambio, tanto la medida de coste $\mathcal{I}_c^+$, detallada en el apartado anterior, como la nueva aproximación basada en pasos interpretativos cortos descrita aquí distinguen ambos casos con claridad, ya que el número de pasos cortos de tipo 1 realizados en cada uno es totalmente distinto (100 y 1, respectivamente).

Este nuevo método aporta una gran exactitud en la medición de costes sin necesidad de alterar la medida de coste, así como la posibilidad de generar trazas muy detalladas de las derivaciones en nuestro entorno de programación $\mathcal{FLOPER}$, lo que resulta muy interesante de cara al usuario, como veremos en la Sección 3.6.

3.6. Respuestas computadas con trazas declarativas

Hemos abordado en la sección 3.5 la reformulación de la fase interpretativa mediante la noción de paso interpretativo corto, el cual considera explícitamente la expansión de la definición de una conectiva difusa o la evaluación de un operador primitivo. Este método permite una medición precisa del coste computacional de la fase interpretativa. Este mecanismo operacional puede ser simulado mediante retículos diseñados específicamente con este propósito, con la ventaja añadida de que permiten visualizar en forma de traza declarativa toda la información asociada a la ejecución de estos programas, como detallamos en [MMPV11c, MMPV11b, MMPV11a].

La ventaja de esta aproximación consiste en que al ejecutar un objetivo se obtiene, además de la (posible) respuesta computada difusa, información ligada a los pasos admisibles e interpretativos llevados a cabo para alcanzarla. Esta información incluye etiquetas con las reglas que se han usado (en el orden adecuado) para dar los pasos admisibles, además del nombre de las conectivas y operadores primitivos evaluados (en su orden). Esta información se recoge en la propia respuesta computada difusa como un parámetro más.

La clave para lograr este propósito reside en el uso de retículos diseñados específicamente para ello, que son capaces de almacenar la información relativa a la ejecución de un objetivo. Estos retículos cumplen también las propiedades exigidas por el carácter multi-adjunto, tal y como demostramos en el trabajo [MMPV12b], y recogemos ampliamente en la Sección 2.4 del capítulo inicial de esta memoria. Pasa-

mos ahora a detallar los retículos que hemos diseñado e ilustrar con unos ejemplos cómo se obtiene la información de la ejecición en la respuesta computada difusa.

Se pueden diseñar numerosos modelos para lograr este objetivo, de los que los más básicos se limitan a documentar el número de pasos admisibles realizados, y los más complejos informan de las reglas, conectivas y operadores primitivos evaluados en cada computación.

Considérese el siguiente programa lógico multi-adjunto $\mathcal{P}$, que sirve de referencia a lo largo de esta sección:

Ejemplo 3.6.1. *Sea $\mathcal{P}$ el siguiente programa lógico multi-adjunto:*

$$\begin{array}{lll} \mathcal{R}_1 : p(X) & \leftarrow_{\mathtt{P}} \quad \&_{\mathtt{G}}(q(X), @_{\mathtt{aver}}(r(X), s(X))) & \textit{with} \quad 0.9 \\ \mathcal{R}_2 : q(a) & \leftarrow & \textit{with} \quad 0.8 \\ \mathcal{R}_3 : r(X) & \leftarrow & \textit{with} \quad 0.7 \\ \mathcal{R}_4 : s(X) & \leftarrow & \textit{with} \quad 0.5 \end{array}$$

si ejecutamos el objetivo `p(X)` *contra este programa, interpretado en el retículo multi-adjunto "num.lat", obtenemos la siguiente derivación, dividida en la fase de resolución difusa y la fase interpretativa, que nos conduce a la respuesta computada difusa* $\langle 0.54; \{X/a\}\rangle$*:*

Fase de resolución difusa:

$$\begin{array}{ll} \langle \underline{p(X)};\ id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver}}(r(X1), s(X1)))); \{X/X_1\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_4} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\}\rangle & \end{array}$$

Fase interpretativa:

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

Utilizaremos este ejemplo a lo largo de esta sección para comparar los distintos métodos y las trazas que generan. Las diferencias entre un modelo y otro se limitan al retículo que asociamos al programa en cada caso.

Recordamos que el significado de un programa depende del retículo asociado. Dicho de otro modo, el resultado de interpretar un objetivo en un programa dado puede ser distinto según el retíclo asociado a dicho programa, como ilustramos con el siguiente ejemplo. Sea $\mathcal{L}^*$ un retículo idéntico a $\mathcal{L} = \langle [0,1], \leq \rangle$, y que consta de las mismas conectivas (esto es, las conjunciones y disyunciones de Łukasiewicz, Gödel y del Producto, junto a la media aritmética), más una definición alternativa de la media aritmética, definida como la media entre los valores obtenidos por las disyunciones de Gödel y Łukasiewicz sobre los argumentos, es decir, $@_{\texttt{aver2}}(x_1, x_2) \triangleq (\vee_{\texttt{G}}(x_1, x_2) + \vee_{\texttt{L}}(x_1, x_2)) * 0.5$.

Ejecutamos ahora el mismo objetivo sobre el programa del Ejemplo 3.6.1. La "f.c.a" –o respuesta computada difusa– resultante es $[\texttt{Truth\_degree} = \texttt{0.72}, \texttt{X} = \texttt{a}]$, distinta a la obtenida al interpretar el programa con el retículo $\mathcal{L}$.

Queda claro que el retículo asociado a un programa dado afecta a las respuestas computadas difusas de los objetivos evaluados, así como la posibilidad de cambiar dicho retículo. Este hecho permite introducir los diferentes retículos que proponemos en esta sección, producidos por sucesivos refinamientos de la técnica que presentamos a continuación.

La posibilidad de recoger la información asociada a la ejecución de un objetivo, en forma de trazas declarativas, nos fue inspirada por el denominado *dominio de cualificación* $(\mathcal{W}, \leq)$ $\mathcal{W}$ empleado en el marco QLP (*Qualified Logic Programming*) de [RD08]. Este modelo pretende representar costes de pruebas, medidas como la profundidad ponderada de árboles de prueba. Como ya detallamos en la Sección 2.4, aunque está próximo a MALP, el esquema QLP admite un repertorio menor de conectivos en el cuerpo de las reglas del programa. Para desarrollar esta técnica partimos de un retículo multi-adjunto que emula esta capacidad de los dominios de cualificación. Tomamos $\mathcal{W} = \mathbb{N} \cup \{\infty\}$, con orden $\leq$ invertido con respecto al usual (de forma que $\infty = inf(\mathcal{W})$ y $0 = sup(\mathcal{W})$). Como demostramos en el capítulo inicial, el retículo $\mathcal{W}$ es también multi-adjunto.

Al asociar el nuevo retículo $\mathcal{W}$ al programa del Ejemplo 3.6.1 cambiamos los pesos de cada regla del programa a 1. Esto puede leerse como que asociamos a cada regla el peso correspondiente a su coste en la derivación, concretamente, 1, pues su ejecución conlleva la aplicación de un único paso admisible. Más aún, ya que en este retículo la operación aritmética "+" define una conjunción (*t-norma*), usamos esta definición para todas las conjunciones que aparecen en el programa (es decir, $\&_{\texttt{P}}(x, y) \triangleq x + y$, $\&_{\texttt{G}}(x, y) \triangleq x + y$ y $@_{\texttt{aver}}(x, y) \triangleq x + y$).

El objetivo $p(X)$ produce la misma derivación en la fase admisible que en el Ejemplo 3.6.1, pero terminando ahora en el estado $\langle \&_{\mathtt{P}}(1, \&_{\mathtt{G}}(1, @_{\mathtt{aver}}(1,1))); \{X/a\}\rangle$. Por tanto, dado que $\&_{\mathtt{P}}(1, \&_{\mathtt{G}}(1, @_{\mathtt{aver}}(1,1))) = +(1, +(1, +(1,1))) = 4$, la respuesta computada difusa, o f.c.a., es $\langle 4; \{X/a\}\rangle$, lo cual indica que el objetivo "$p(X)$" se satisface cuando X es a, tras aplicar 4 pasos admisibles. Este retículo supone ya un primer avance en lo que respecta a la información recogida sobre la traza de ejecución, pues da cuenta del número de pasos admisibles que realizados. Como desventaja inicial de este método, obsérvese que se pierde la noción original de grado de verdad en favor de la medida de coste.

Para solucionar este problema, desarrollamos un retículo más expresivo con forma de *producto cartesiano*, cuya validez como retículo mutli-adjunto fue demostrada en el trabajo [MMPV12a], y cuyos resultados teóricos repasamos ampliamente en la Sección 2.4 de esta memoria. Con esta clase de retículos, el peso asociado a cada regla de programa es un par que consta de un grado de verdad y una medida de coste. Definimos el producto cartesiano de dos retículos multi-adjuntos $\mathcal{L}$ y $\mathcal{W}$, $\mathcal{L} \times \mathcal{W}$, del siguiente modo:

$\langle r, w\rangle \in \mathcal{L} \times \mathcal{W} \iff r \in \mathcal{L} \wedge w \in \mathcal{W}$
$\langle r_1, w_1\rangle \leq \langle r_2, w_2\rangle \iff r_1 \leq r_2 \wedge w_1 \geq w_2$
$\langle 1, 0\rangle = \mathcal{S}up\{\mathcal{L} \times \mathcal{W}\}$
$\langle 0, \infty\rangle = \mathcal{I}nf\{\mathcal{L} \times \mathcal{W}\}$
$\langle r_1, w_2\rangle \&_i \langle r_2, w_2\rangle \triangleq \langle r_1 \&_i r_2, w_1 + w_2\rangle$

Para cargarlo en nuestra herramienta de programación $\mathcal{FLOPER}$, debemos definir en PROLOG el nuevo retículo (como hacíamos con los anteriores "num.lat", "bool.lat", etc.), cuyos elementos pueden ser expresados, por ejemplo, como términos de la forma:

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

Mostramos en la Figura 3.1 el código necesario para manipular estos grados de verdad.

Para proseguir el ejemplo, asumimos que asignamos los pesos siguientes a las reglas del programa de ejemplo: "`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 $\mathcal{R}_4$. Con estos valores, la respuesta computada difusa para el objetivo `p(X)` es $\langle \mathtt{info}(0.54, 4); \{X/a\}\rangle$, y significa que el objetivo se satisface con un grado de verdad 0.54 cuando X cambia por a, y que han sido necesarios 4 pasos admisibles para alcanzar esta solución.

Hasta este punto hemos visto cómo incluir nueva información en las respuestas computadas, en particular, el número de pasos admisibles llevados a cabo. Un ob-

```
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).
```

Figura 3.1: Retículo que informa del número de pasos admisibles dados

jetivo más ambicioso consiste en documentar toda la traza de ejecución. De nuevo, el retículo necesario para llevar esto a cabo define grados de verdad con dos componentes, uno para el grado de verdad y otro con una etiqueta, o cadena de caracteres, que recoge información sobre las reglas del programa empleadas y los conectivos y operadores evaluados. Como en la ocasión anterior, en la que fue necesario justificar el caracter multi-adjunto del retículo de los pesos $\mathcal{W}$, demostramos en la Sección 2.4 (que recoge la prueba original de [MMPV12b]) la validez del retículo basado en etiquetas, o *strings*, $\mathcal{S}$.

Este resultado tiene su raíz en el trabajo [MMPV11c], donde diseñamos un retículo para informar sobre los pasos admisibles e interpretativos de una derivación. En ese caso, informábamos sobre los pasos interpretativos "clásicos", y no los operadores primitivos ni conectivas intermedias. Concretamente, el retículo empleado en ese trabajo es el siguiente.

```
bot(info(0,_)).                                    top(info(1,_)).
member(info(X,_)):-number(X),0=<X,X=<1.
leq(info(X1,_),info(X2,_)) :- X1 =< X2.
and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-pri_prod(X1,Y1,Z1),
                  pri_app(X2,Y2,Dat1), pri_app(Dat1,'&PROD.',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).
```

Figura 3.2: Retículo que permite mostrar los pasos interpretativos clásicos

Aquí definimos los miembros del retículo con la forma "`info(Fuzzy_Truth_Degree, Label)`", de modo que incorporan tanto un grado de verdad como una "etiqueta".

Además, para llevar la cuenta de las conectivas utilizadas, cada una de ellas incorpora una etiqueta sobre sí misma en su parámetro de salida (como hace, por ejemplo, `and_prod` incluyendo `&PROD.` en la salida). Para introducir estas etiquetas se hace uso de un operador primitivo de concatenación de caracteres, `pri_app`. Además de estas consideraciones en el retículo, es necesario modificar los pesos de las reglas del programa para que sean miembros del retículo, esto es, para que, además de llevar un grado de verdad, aporten una etiqueta que represente a esa regla.

```
p(X) <prod &(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.').
```

Para ilustrar mejor este efecto, hemos empleado la versión `@aver2`, que hace llamadas intermedias a otras conectivas. Para este programa y retículo, la evaluación del objetivo `p(X)` resulta en la respuesta computada difusa con el grado de verdad esperado (0.72) junto a una traza declarativa:

```
>> run.
[Truth_degree=info(0.72,  RULE1.RULE2.RULE3.RULE4.
                          @AVER2.&GODEL.&PROD),      X=a]
```

El significado de esta respuesta computada es que el objetivo `p(X)` es cierto con un grado de verdad de 0.72 cuando X cambia por a, y que, además, para demostrar esto se han explotado, en este orden, las reglas de programa 1, 2, 3 y 4, seguido de la evaluación de la conectiva `@aver2`, `&Godel` y `&Prod`. Estas dos últimas conectivas han sido llamadas por encontrarse en la definición de la primera.

Como hemos dicho anteriormente, este retículo se limita a informar sobre las conectivas evaluadas, pero no sobre los operadores primitivos. Como ya se ha justificado en la Sección 3.4, puede ser necesario tener información al respecto de los operadores primitivos empleados, y no sólo de las conectivas. Para responder a esta necesidad, proponemos un nuevo retículo para tener en cuenta la evaluación de los operadores primitivos.

Dicho retículo, ilustrado en la Figura 3.3 y presentado en [MMPV11a], se construye análogamente al anterior. La diferencia radica en que no sólo las conectivas añaden etiquetas a su salida; también lo hacen los operadores primitivos. Obsérvese cómo éstos se expanden con un parámetro más que informa sobre ellos. Por ejemplo,

```
member(info(X,_)):-number(X),0=<X,X=<1.              bot(info(0,_)).
leq(info(X1,_),info(X2,_)):- X1 =< X2.               top(info(1,_)).

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 3.3: Código del retículo con etiquetas capaz de mostrar las trazas de ejecución

`pri_add` incorpora el parámetro de salida `#ADD`. Dichas etiquetas se unen entre sí y con las del conectivo invocador, obteniendo así una traza completa que informa sobre los pasos interpretativos "cortos".

Para probar el nuevo retículo empleamos el programa de la versión anterior (que empleaba el retículo de [MMPV11c]). Dado que esta modificación en el retículo no afecta a la definición de sus elementos, el programa puede permanecer inalterado. En este caso, la evaluación del objetivo `p(X)` resulta en la siguiente 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]
```

Obsérvese que el grado de verdad (0.72), así como las etiquetas que informan sobre las reglas de programa y las conectivas explotadas que aparecen directamente en el programa, conservando el orden relativo que tenían en el caso anterior. La diferencia aquí radica en la aparición, en la traza, de nuevas conectivas y operadores primitivos. En particular, aparecen la disyunción de Gödel seguida de la operación primitiva *max*, la disyunción de Łukasiewicz seguida de dos operadores primitivos, y el agregador de la media aritmética acompañado de sus respectivos operadores primitivos. Todo ello aparece inmediatamente después de la etiqueta `@AVER2`, lo que indica que se han realizado las operaciones referidas a continuación. Las otras conectivas que ya aparecían previamente también están acompañadas aquí por los operadores primitivos que los definen.

Como conclusión, cabe reseñar que los efectos conseguidos por el uso de retículos especiales, para obtener información sobre la ejecución de los objetivos, son sólo un ejemplo, y que otros retículos pueden ser más o menos finos en su representación de las trazas declarativas, u obtener otra información. Ha sido gracias a la demostración del caracter multi-adjunto de estos retículos, visto en la Sección 2.4, y del producto cartesiano de retículos multi-adjuntos, que se ha logrado obtener este efecto.

3.7. Igualdad estricta basada en similaridad para MALP

Entre las familias de lenguajes de programación lógico difusos existen algunos basados en sofisticadas relaciones de similaridad/proximidad [Ses02], como LIKELOG SQLP y Bousi~Prolog, presentados en la introducción de esta memoria. También es cierto que otros lenguajes declarativos, como los funcionales, emplean nociones de igualdad muy distintas a la clásica de Prolog, de las que los lenguajes lógicos pueden obtener gran beneficio. Si bien los estilos de programación lógico y funcional ya han sido integrados satisfactoriamente en el pasado, no ocurre así con el estilo funcional y el lógico difuso.

En los trabajos [MPV12b, MPV14b, JMV15] hemos abordado este tópico, con el ánimo de incorporar a la programación MALP la noción de igualdad estricta, combinada con la de igualdad, para dotar al lenguaje del más amplio abanico de opciones para representar la igualdad.

A continuación detallamos con más profundidad tres nociones de igualdad presentes en la programación declarativa.

- **Igualdad sintáctica.** Constituye el modelo más simple de igualdad. Se emplea en la programación lógica pura y algunos casos de la lógica difusa (como MALP, por ejemplo). En este ámbito, dos elementos se consideran *iguales* si tienen la misma sintaxis. Así, $f(a)$ es igual a $f(a)$, pero nunca a $g(b)$.

- **Igualdad estricta.** Al tomar en consideración los lenguajes basados en la evaluación perezosa, se introduce la noción de igualdad estricta, para la que dos elementos (típicamente, expresiones) se consideran *iguales* si el resultado de su evaluación es el mismo dato. La característica de este tipo de igualdad es que la expresión tiene que poder evaluarse en tiempo finito. Así, si la evaluación de $f(a)$ no termina, no puede decirse que sea igual a sí misma, estrictamente hablando. Por el contrario, dos expresiones de diferente sintaxis, como $g(b)$ y $h(c)$ pueden ser estrictamente iguales si producen el mismo valor tras ser evaluadas mediante reescritura (o mediante *narrowing*).

- **Igualdad basada en similaridad.** Este modelo emerge como consecuencia de difuminar la programación lógica pura. En este contexto, una relación construida a partir de *ecuaciones de similaridad/proximidad* permite evaluar como

iguales "en cierto grado" dos expresiones sintácticamente distintas. Por ejemplo, dadas las ecuaciones $eq(a, b) = 0.5$ y $eq(f, g) = 0.3$, se puede probar que $f(a)$ y $g(b)$ son similares con un determinado grado de verdad, que depende de cómo se agreguen los grados de las ecuaciones.

El objetivo principal de esta sección consiste en adaptar a MALP las otras dos nociones de igualdad que no le son nativas, esto es, igualdad estricta y basada en similaridad. El resultado de esta investigación es la herramienta SSE, que se puede acceder libremente desde la dirección `http://dectau.uclm.es/sse/`.

3.7.1. Relaciones de similaridad en la programación lógica difusa

Una relación de similaridad es una noción matemática que permite relacionar pares de entidades que se consideran similares con un grado de verdad. Estas relaciones son muy parecidas a las de equivalencia (y, por tanto, a los operadores de clausura).

Formalmente, una relación de similaridad $\Re$ sobre un dominio $\mathcal{U}$ es un subconjunto difuso $\Re : \mathcal{U} \times \mathcal{U} \to [0, 1]$ de $\mathcal{U} \times \mathcal{U}$ tal que $\forall x, y, z \in \mathcal{U}$, se cumplen las propiedades reflexiva, $\Re(x, x) = 1$, simétrica $\Re(x, y) = \Re(y, x)$ y transitiva $\Re(x, z) \geq \Re(x, y) \wedge \Re(y, z)$.

Un modo sencillo para introducir relaciones de similaridad en la programación lógica consiste en su descripción mediante las así llamadas *ecuaciones de similaridad*. Estas ecuaciones tienen la forma $eq(s_1, s_2) = \alpha$, lo que relaciona las entidades s_1 y s_2 con el grado de verdad α, perteneciente a un retículo de modo que si $\alpha = \top$ se desprende que s_1 es idéntico a s_2 y conforme menor sea α, menor es el parecido entre las entidades (hasta llegar a $\bot$, valor para el cual las dos entidades no guardan ninguna semejanza). En particular, en MALP, dicho retículo es el asociado al programa.

Esta es la aproximación elegida por lenguajes como LIKELOG y Bousi~Prolog [JRG09a], en los que al conjunto de reglas de programa se le acompaña de un conjunto de ecuaciones de similaridad empleadas en tiempo de unificación. Estos lenguajes emplean, en lugar de la unificación sintáctica de Prolog, la llamada *unificación débil* [JRG09a]. De las ecuaciones de similaridad se considera que relacionan siempre símbolos de la misma aridad y naturaleza (nunca relacionan una función y un predicado, por ejemplo), y que cumplen las propiedades reflexiva, simétrica y transitiva.

$\texttt{sse}(c, d)$ $\texttt{with}\ \Re(c, d)$

$\texttt{sse}(c(x_1, .., x_n), d(y_1, .., y_n)) \leftarrow_G \texttt{sse}(x_1, y_1)\ \&_G \ldots \&_G \texttt{sse}\ (x_n, y_n)$ $\texttt{with}\ \Re(c, d)$

Figura 3.4: Reglas MALP que definen la "Igualdad estricta basada en similaridades"

Ejemplo 3.7.1. *Considérese el término* `autor(Wells)` *y uno que se pretende que sea similar,* `escritor('H. G. Wells')`*. Siguiendo la noción de igualdad sintáctica, ambos términos son totalmente distintos. Sin embargo, es razonable considerar que, en realidad, son bastante similares. Para detallar en qué grado son similares, es necesario introducir las ecuaciones* $eq(autor, escritor) = 0.9$ *y* $eq(Wells, `H.G.Wells') = 0.8$ *(usamos* 0.8*, pues* $Wells$ *a secas podría referirse al cineasta). Al introducir estas ecuaciones en un intérprete* LIKELOG *o* BOUSI~PROLOG*, ambos términos sí podrían ser considerados iguales con un grado de verdad de* $80\,\%$ *(o, siendo más precisos,* 0.8*, si empleamos la conjunción del mínimo para agregar los valores), como se espera.*

3.7.2. Reglas MALP para modelar SSE

En este apartado detallamos el tipo de reglas que es necesario incluir en un programa MALP para simular la igualdad estricta basada en similaridades, de acuerdo al trabajo que realizamos en [MPV12b].

En la Figura 3.4 se muestran reglas MALP, donde se asume que c y d son constantes (esto es, símbolos de función de aridad 0) en la primera regla y funciones de aridad n en la segunda, $\Re(c, d)$ representa el grado de similaridad entre ambos símbolos.

Ejemplo 3.7.2. *Utilizamos ahora los términos del ejemplo 3.7.1 para componer las reglas* MALP *correspondientes.*

```
sse(wells,         wells)         with 1.
sse(wells,         'H. G. Wells') with 0.8.
sse('H. G. Wells', wells)         with 0.8.
sse('H. G. Wells', 'H. G. Wells') with 1.

sse(autor(X),    autor(Y))    <prod sse(X,Y) with 1.
sse(autor(X),    escritor(Y)) <prod sse(X,Y) with 0.9.
sse(escritor(X), autor(Y))    <prod sse(X,Y) with 0.9.
sse(escritor(X), escritor(Y)) <prod sse(X,Y) with 1.
```

$$
\begin{array}{ll}
\langle sse(c(t_1,\ldots,t_n),x)); id\rangle & \rightarrow_{AS}{}^{\mathcal{R}} \\
\langle s_0 \& sse(t_1,x'_1)\ \&\ldots\&\ sse(t_n,x'_n); \{x/c'(x'_1,\ldots,x'_n), x_1/t_1,\ldots,x_n/t_n\}\rangle & \rightarrow_{AS} \\
\ldots\ \rightarrow_{AS}\ \ldots & \rightarrow_{AS} \\
\langle s_0\ \&\ s_1\ \&\ldots\&\ sse(t_n,x'_n); \{x/c'(t'_1,\ldots,x'_n), x_1/t_1,\ldots,x_n/t_n, x'_1/t'_1\}\rangle & \rightarrow_{AS} \\
\ldots\ \rightarrow_{AS}\ \ldots & \rightarrow_{AS} \\
\langle s_0\ \&\ s_1\ \&\ldots\&\ s_n); \{x/c'(t'_1,\ldots,t'_n), x_1/t_1,\ldots,x_n/t_n, x'_1/t'_1,\ldots,x'_n/t'_n\}\rangle &
\end{array}
$$

Figura 3.5: Demostración del Teorema 3.7.5

En este caso hemos empleado la conjunción del producto para agregar los grados de verdad, como puede verse por el uso de su implicación adjunta en las cuatro últimas reglas.

El uso de estas ocho reglas en un programa MALP *permiten interpretar que los términos* `autor(wells)` *y* `escritor('H. G. Wells')` *se consideren similares (mediante la evaluación de* `sse(autor(wells),escritor('H. G. Wells'))`*) con el grado de verdad* $0.9 \cdot 0.8 = 0.72$*, de modo parecido a como lo harían* LIKELOG *o* BOUSI∼PROLOG*. Sin embargo, estos lenguajes sólo reportan una solución para el objetivo* `sse(autor(wells),X)`*, en particular,* `X=autor(wells)`*. Dado que con dicho objetivo se trata de averiguar términos similares a* `autor(wells)`*, nuestro método mejora los lenguajes anteriores, pues aporta las cuatro respuestas:*

```
< 1, {Y/autor(wells)} >
< 1, {Y/autor('H. G. Wells')} >
< 1, {Y/escritor(wells)} >
< 1, {Y/escritor('H. G. Wells')} >
```

Con objeto de probar formalmente las propiedades de este procedimiento, introducimos ahora la siguiente definición auxiliar:

Definición 3.7.3 (Términos similares)**.** *Sean t y t' dos términos básicos, $\Re$ una relación de similaridad y $\mathcal{L} = \langle L, \preceq, \leftarrow, \&\rangle$ un retículo multi-adjunto. Decimos que t y t' son similares de acuerdo a $\Re$ y $\&$, con grado de similaridad $s \in L$, si la evaluación de la función $\mathcal{M}_\top(t,t')$ devuelve $s \neq \bot$, donde $\mathcal{M}_\top$ se define recursivamente como sigue:*

$$
\mathcal{M}_\top(t,t') = \begin{cases}
\Re(t,t'), & \text{si } t \text{ y } t' \text{ son constantes} \\
\Re(c,c')\&\mathcal{M}_\top(t_1,t'_1)\& & \text{si } t = c(t_1,\ldots,t_n) \text{ y} \\
\ldots\&\mathcal{M}_\top(t_n,t'_n) & t' = c'(t'_1,\ldots,t'_n)
\end{cases}
$$

El siguiente resultado relaciona la noción teórica de términos similares con el procedimiento ilustrado por la Figura 3.4.

Teorema 3.7.4. *Sean t y t' dos términos básicos, $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ un retículo multi-adjunto, $\Re$ una relación de similaridad y $\mathcal{P}_{sse}^{\Re}$ el conjunto de reglas* MALP *que definen el predicado sse respecto de $\Re$. Entonces, t y t' son términos similares de acuerdo a $\Re$ y $\&$, con grado de similaridad $s \in L$, si y sólo si $\langle s, id \rangle$ es una respuesta computada difusa para el objetivo $sse(t, t')$ en $\mathcal{P}_{sse}^{\Re}$.*

Demostración. Demostramos la afirmación previa por inducción estructural sobre t y t'.

• Caso base. Asumimos que t y t' son constantes y, entonces, $\Re(t, t') = s \neq \bot$, por lo que existe una regla $[\mathcal{R} : sse(t, t') \; with \; s]$ en $\mathcal{P}_{sse}^{\Re}$. Es fácil ver que $\mathcal{M}_{\top}(t, t') = \Re(t, t') = s$ sin más que realizar con la regla $\mathcal{R}$ el siguiente paso admisible: $\langle sse(t, t'), id \rangle \rightarrow_{AS}{}^{\mathcal{R}} \langle s, id \rangle$.

• Paso de inducción. Tenemos que $t = c(t_1, \ldots, t_n)$ y $t' = c'(t'_1, \ldots, t'_n)$. Asumiendo que $\Re(c, c') = s_0 \neq \bot$ y $\mathcal{M}_{\top}(t_i, t'_i) = s_i \neq \bot$, $1 \leq i \leq n$, entonces $\mathcal{M}_{\top}(t, t') = s_0 \& s_1 \& \ldots \& s_n \neq \bot$. Más aún, dado que nuestra técnica genera la regla (en $\mathcal{P}_{sse}^{\Re}$):

$\mathcal{R} : sse(c(x_1, \ldots, x_n), c(x'_1, \ldots, x'_n)) \leftarrow sse(x_1, x'_1) \& \ldots \& sse(x_n, x'_n) \;\; with \; s_0$

y que por hipótesis de inducción, podemos asumir que $\langle s_i, id \rangle$ es una respuesta computada para el objetivo $sse(t_i, t'_i)$, $1 \leq i \leq n$, se puede generar la siguiente secuencia de pasos admisibles (en los que, por motivos de legibilidad, omitimos parte de la componente de sustitución):

$$\begin{array}{ll} \langle sse(c(t_1, \ldots, t_n), c'(t'_1, \ldots, t'_n)); id \rangle & \rightarrow_{AS}{}^{\mathcal{R}} \\ \langle s_0 \;\&\; sse(t_1, t'_1) \;\& \ldots \&\; sse(t_n, t'_n); id \rangle & \rightarrow_{AS} \ldots \rightarrow_{AS} \\ \langle s_0 \;\&\; s_1 \;\& \ldots \&\; sse(t_n, t'_n); id \rangle & \rightarrow_{AS} \ldots \rightarrow_{AS} \\ \langle s_0 \;\&\; s_1 \;\& \ldots \&\; s_n; id \rangle & \end{array}$$

lo que concluye esta demostración. □

El siguiente teorema refueza el anterior estableciendo la capacidad de nuestro método para generar pares de términos básicos similares, y no sólo compararlos.

Teorema 3.7.5. *Sean t y t' dos átomos básicos, x una variable, $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ un retículo multiadjunto, $\Re$ una relación de similaridad y $\mathcal{P}_{sse}^{\Re}$ el conjunto de reglas* MALP *que definen el predicado sse respecto de $\Re$. Entonces, t y t' son términos*

similares de acuerdo a $\Re$ *y* $\&$ *con grado de similaridad* $s \in L$*, si y sólo si* $\langle s, \{x/t'\}\rangle$ *es una respuesta computada difusa para el objetivo* $sse(t, x)$ *en* $\mathcal{P}^{\Re}_{sse}$.

Demostración. Utilizamos de nuevo inducción estructural sobre t, en una demostración muy parecida a la del Teorema 3.7.4.

• Caso base. Asumimos aquí que t y t' son constantes similares y que $\Re(t, t') = s \neq \bot$, por lo que la regla $[\mathcal{R} : sse(t, t')\ with\ s]$ pertenece a $\mathcal{P}^{\Re}_{sse}$. Entonces, obviamente, $\phi(t, t') = \Re(t, t') = s$, de modo que también es posible realizar, con la regla $\mathcal{R}$ el siguiente paso admisible: $\langle sse(t, x), id\rangle \rightarrow_{AS}{}^{\mathcal{R}} \langle s, \{x/t'\}\rangle$.

• Paso de inducción. Ahora tenemos que $t = c(t_1, \ldots, t_n)$ y $t' = c'(t'_1, \ldots, t'_n)$. Asumiendo que $\Re(c, c') = s_0 \neq \bot$ y $\mathcal{M}_\top(t_i, t'_i) = s_i \neq \bot$, $1 \leq i \leq n$, entonces $\mathcal{M}_\top(t, t') = s_0 \& s_1 \& \ldots \& s_n \neq \bot$. Más aún, dado que nuestra técnica genera la regla (que pertenece a $\mathcal{P}^{\Re}_{sse}$):

$\mathcal{R} : sse(c(x_1, \ldots, x_n), c(x'_1, \ldots, x'_n)) \leftarrow sse(x_1, x'_1) \& \ldots \& sse(x_n, x'_n)\ \ with\ s_0$

y por la hipótesis de inducción podemos asumir que $\langle s_i, \{x'_i/t'_i\}\rangle$ es una respuesta computada difusa para el objetivo $sse(t_i, x'_i)$, $1 \leq i \leq n$, entonces, es posible generar la derivación de la Figura 3.5 (para simplificar, de nuevo, hemos omitido en cada estado parte del componente de sustitución), lo que concluye la prueba. □

La aplicación repetida del teorema anterior implica el siguiente resultado que, en esencia, confirma el poder de nuestro método para producir pares de términos similares.

Corolario 3.7.6. *Sean* t *y* t' *dos términos básicos,* x *y* x' *dos variables,* $\mathcal{L} = \langle L, \preceq, \leftarrow, \&\rangle$ *un retículo multi-adjunto,* $\Re$ *una relación de similaridad y* $\mathcal{P}^{\Re}_{sse}$ *el conjunto de reglas* MALP *que definen el predicado sse respecto de* $\Re$*. Entonces,* t *y* t' *son términos similares de acuerdo a* $\Re$ *y* $\&$*, con grado de similaridad* $s \in L$*, si y sólo si* $\langle s, \{x/t, x'/t'\}\rangle$ *es una respuesta computada difusa para* $sse(x, x')$ *en* $\mathcal{P}^{\Re}_{sse}$.

3.7.3. Aspectos de la implementación

Una vez definida una relación de similaridad, queda obtener de ella el código para emplearla en un programa MALP. Para este propósito hemos diseñado la herramienta SSE, accesible en `http://dectau.uclm.es/sse/`, que toma unas ecuaciones de similaridad y devuelve el código equivalente.

Para introducir en el sistema una relación de similaridad, empleamos un fichero en texto plano con extensión ".sim", con la misma sintaxis que se describirá más adelante

en la Sección 6.5. Esta sintaxis se basa en la que emplean los lenguajes LIKELOG y BOUSI~PROLOG, y consiste en una lista de expresiones de la forma `símbolo1~ símbolo2 = grado de verdad`, si bien en nuestro método hemos querido dar más potencia a esta sintaxis añadiendo junto a cada símbolo su aridad (salvo en caso de que la aridad sea 0). Así, la relación del Ejemplo 3.7.1 se modela mediante el siguiente texto:

```
autor/1 ~ escritor/1 = 0.9.
wells ~ 'H. G. Wells' = 0.8.
```

Téngase en cuenta que, dado que el lenguaje MALP acepta retículos más amplios que el intervalo $[0, 1]$, las ecuaciones (y, por tanto, relaciones) de similaridad también puedem emplear dichos retículos.

Una vez introducidas estas ecuaciones de similaridad en el sistema, éste procede a realizar las clausuras reflexiva, simétrica y transitiva, de acuerdo al Algoritmo 1 (inspirado en [JRG09a], pero generalizado a retículos más allá de $[0, 1]$, en particular, cualquier retículo completo), para obtener explícitamente todos los pares de la relación de similaridad.

Como resultado de la aplicación del Algoritmo 1, el sistema contempla la relación completa introducida parcialmente por el usuario mediante ecuaciones de similaridad.

Ejemplo 3.7.7. *Considerando de nuevo el ejemplo 3.7.1, la aplicación del Algoritmo 1 sobre dichas ecuaciones de similaridad produce la siguiente matriz:*

	wells	*'H. G. Wells'*	*autor*	*escritor*
wells	*1*	*0.8*	*0*	*0*
'H. G. Wells'	*0.8*	*1*	*0*	*0*
autor	*0*	*0*	*1*	*0.9*
escritor	*0*	*0*	*0.9*	*1*

El último paso consiste en traducir la relación de similaridad (dada en forma de matriz de adyacencias) en un conjunto de reglas MALP que definan el predicad `sse`.

El resultado de aplicar los Algoritmos 1 y 2 sobre las ecuaciones de similaridad dadas en el Ejemplo 3.7.1 es el conjunto de reglas MALP del Ejemplo 3.7.2, como queríamos.

Dado que $R = \{R_i, i \in \{0, \ldots, N\}\}$ es un conjunto de reglas MALP, también es un programa válido, se puede cargar de forma natural en la herramienta $\mathcal{FLOPER}$,

Algorithm 1

Require: Una matriz de adyacencias $M = [m_{ij}]$, cada una de cuyas filas y columnas representa un símbolo que aparece en las ecuaciones de similaridad, y donde cada celda m_{ij} tiene el valor dado por la ecuación que relacione los símbolos i y j. Todos los elementos de la matriz triangular superior deben estar inicializados a $\bot$.

Ensure: Una matriz de adyacencias $M^{\equiv}$ correspondiente a la clausura reflexiva, simétrica y transitiva de R.

for all $\langle i, i\rangle$ de M **do** {Clausura reflexiva}
  $m_{ii} := \top$;
end for
for all $\langle i, j\rangle$ de M, tal que $m_{ij} \neq \bot$ **do** {Clausura simétrica}
  $m_{ji} := m_{ij}$;
end for
for all columns k y celda $\langle i, j\rangle$ de M **do** {Clausura transitiva}
  $m_{ij} := m_{ij} \vee (m_{ik} \wedge m_{kj})$; donde $\vee$ y $\wedge$ son, respectivamente, los operadores del supremo y el ínfimo;
end for
$M^{\equiv} := M$

Algorithm 2

Require: Una matriz de adyacencias M correspondiente a la relación de similaridad R.

Ensure: Un conjunto $R = \{R_i, i \in \{0, \ldots, N\}\}$ de reglas MALP.

for all $M_{ij} = V$ en M, donde $i = A/n$ y $j = B/n$ **do**
  $body :=$ "$with'' + V$;
  for all $k \in \{n, \ldots, 1\}$ **do**
    $body :=$ "$, sse(X_k, Y_k)'' + body$;
  end for
  if $n > 0$ **then**
    $body :=$ " $< -\ sse(X_1, Y_1)'' + body$;
  end if
  $R_i :=$ "$sse(A(X_1, \ldots, X_n), B(Y_1, \ldots, Y_n))''$;
  $R_i := R_i + body$;
end for

descrita en el Capítulo 4, y realizar consultas sobre la similaridad de diferentes términos.

3.8. Programación lógica difusa desde la teoría de categorías

En esta sección recogemos el trabajo desarrollado en [EGH$^+$13a, EGH$^+$13b], y que tiene por objeto proveer a la programación lógica difusa de un sustento teórico desde la teoría de categorías. Empezamos investigando en proceso de difuminación por el que se pasa de una lógica bi-valuada a una multi-valuada. Mostraremos que la incorporación de la incertidumbre en este marco está lejos de ser una tarea trivial.En el marco clásico, términos y sentencias son conjuntos, si bien en nuestra aproximación ambos son objetos categóricos. Téngase en cuenta, además, que en teoría de categorías no se distingue entre "término" y "átomo" como se hace en programación lógica, siendo ambos objetos de la misma categoría. Las propuestas basadas en teoría de conjuntos, al tratar con "conjuntos de sentencias" y "conjuntos de átomos básicos", pueden llevar fácilmente a confusión y a construcciones no deseadas si las generalizaciones se llevan a cabo únicamente sustituyendo el uso de "conjuntos" por "conjuntos difusos". Presentamos algunos resultados básicos sobre cómo la adaptación a un marco estrictamente categórico permite una mayor precisión al distinguir entre términos y sentencias, donde los símbolos de predicado pasan a formar parte de una signatura que se mantiene aparte de la de los términos.

3.8.1. Sentencias como pares de términos

De forma intuitiva, los términos se producen a través de signaturas de tal forma que las variables y constantes son términos, y si $t_1, \ldots, t_n$ son términos, también lo es $\omega(t_1, \ldots, t_n)$, donde ω pertenece a la signatura dada. Desde el punto de vista de la teoría de categorías, estos términos se producen por functores que son extensibles a mónadas (véase, por ejemplo [EGHK14, Hel13]). Las sentencias a su vez, se definen como pares $(P(x), Q(x))$, donde $P(x)$ y $Q(x)$ son términos, P y Q son símbolos de predicado pertenecientes a la signatura, y x e y son variables. El par $(P(x), Q(x))$ correspondería a "$P(x)$ se infiere de $Q(x)$". Esto distingue a las sentencias, dadas por el functor "par de términos", de los términos en sí, que son objetos monádicos.

En la programación lógica basada en lógica de primer orden la signatura subya-

cente es de gran relevancia. Las descripciones informales de la lógica de primer orden no son siempre claras sobre si los predicados son símbolos operadores, relaciones, o funciones. Incluso en [Llo84] se considera que el alfabeto contiene indistintamente símbolos de función y símbolos de predicado. Esto lleva a que la distinción entre términos y las fórmulas no es clara.

En nuestra aproximación categórica, llamamos "alfabeto" a la signatura subyacente de tipos y operadores, con la característica de que contempla el uso de muchos tipos distintos, mientras que [Llo84] contempla un solo tipo. Consideraremos los predicados como operadores, de manera que los átomos serán "términos" (en un sentido categórico), pero las cláusulas de programas serán sentencias. Básicamente, esto lleva a que la conjunción de predicados seguirán siendo términos, pero no así las cláusulas que contienen la implicación, pues la implicación no se incluye como operador en la signatura subyacente.

El marco categórico de las construcciones monádicas que emplearemos a continuación aparece en [EGHK14], de donde hemos tomado también la notación categórica para esta sección. Estas mónadas hacen un uso amplio de los conceptos del *álgebra categórica*, que data de los estudios en equivalencias naturales de [EM45]. Las categorías monoidales cerradas surgen en [EK66], si bien adquieren su formulación más simple y limpia en [ML71].

Con respecto a las extensiones tradicionales de la programación lógica bi-valuada a la multi-valorada, en la primera, una vez se da la negación, se definen la implicación y la conjunción mutuamente. En la tradición intuicionista se evita la negación como elemento básico, de modo que para relacionar la implicación y la conjunción se emplea la propiedad de la residuación, que es, categóricamente, una adjunción dada como una conexión de Galois. Esto permite definir la negación más adelante. Los retículos residuados han sido estudiados extensamente, pues son unas estructuras algebráicas atractivas para definir la semántica de los lenguajes.

Esta convención se ha adoptado en numerosas ocasiones en las lógicas multi-valoradas. Por ejemplo, la noción multi-adjunta aparece en la programación lógica referida al uso de retículos residuados que proveen la semántica deseada para relacionar implicación y conjunción.

Nótese que en este paso de la lógica bi-valuada a la multi-valorada, si bien se ha alterado la noción de verdad, el lenguaje fundamental al respecto de los términos y las sentencias se ha conservado intacto. Se sigue la tradición de la extensión del cálculo proposicional al de predicados conservando el significado de la operación de

implicación.

Como se ha mencionado anteriormente, nuestro ámbito en esta sección es la lógica como un objeto categórico, que se construye funcionalmente y monádicamente con morfismos entre subestructuras en lógica. Así, no nos proponemos construir una "lógica universal", por lo que los programas lógicos pueden tener diferentes reglas de inferencia, y transformaciones entre programas lógicos carecen de sentido a menos que se puedan transformar sus lógicas subyacentes.

Por lo común, las aproximaciones a la programación lógica difusa, como las vistas en [BMP95, Ebr01, IK85, Lee72, LL90, MCS11b, RD08, Voj01], difieren esencialmente en la teoría subyacente de incertidumbre y vaguedad (teoría de probabilidad, lógica posibilista, lógica difusa y lógica multi-valorada) y en cómo se manejan los valores de incertidumbre/vaguedad asociados a las reglas.

La teoría de categorías entra en escena para definir la programación lógica en [RB85], donde se muestra una equivalencia entre las nociones de co-ecualizadores y el unificador más general. Sin embargo, cabe precisar que esta equivalencia no es suficiente al trasladarnos a un ámbito multi-valorado.

3.8.2. Signaturas, términos y sentencias

Introducimos en esta sección las nociones de teoría de categorías empleadas en adelante. Repasamos más adelante los conceptos de signatura y término monádico e introduciremos las sentencias en el contexto de la programaciónl lógica. Por último, consideraremos la semántica declarativa por punto fijo.

Conceptos y nociones categóricos

Sea $\mathtt{C}$ una categoría, y S un conjunto de tipos. Entonces, $\mathtt{C}_S$ es una categoría con los objetos $X_S = (X_\mathtt{s})_{\mathtt{s}\in S}$, tales que cada $X_\mathtt{s} \in \mathrm{Ob}(\mathtt{C})$. Los morfismos entre X_S y Y_S son $f_S \colon X_S \to Y_S$, donde $f_S = (f_\mathtt{s})_{\mathtt{s}\in S}$ y cada $f_\mathtt{s} \in \hom_\mathtt{C}(X_\mathtt{s}, Y_\mathtt{s})$. La composición de morfismos se define de acuerdo a los tipos: $f_S \circ g_S = (f_\mathtt{s} \circ g_\mathtt{s})_{\mathtt{s}\in S}$.

A veces necesitaremos referirnos a un objeto $X_\mathtt{s}$ de $\mathrm{Ob}(\mathtt{C})$ cuando X_S se dé en una forma u otra. A tal fin definimos un functor $\mathsf{arg}^\mathtt{s} \colon \mathtt{C}_S \to \mathtt{C}$ tal que $\mathsf{arg}^\mathtt{s} X_S = X_\mathtt{s}$ y $\mathsf{arg}^\mathtt{s} f_S = f_\mathtt{s}$. Especialmente, al trabajar en $\mathtt{Set}_S$, $\mathrm{card}(S) > 1$, conviene definir los dos functores: $\phi^{S\backslash\mathtt{s}} : \mathtt{Set}_S \to \mathtt{Set}_S$ tales que $\phi^{S\backslash\mathtt{s}} X_S = X'_S$, donde, para todo $\mathtt{t} \in S\backslash\{\mathtt{s}\}$ tenemos $X'_\mathtt{t} = X_\mathtt{t}$, y $X'_\mathtt{s} = \emptyset$. De forma similar, definimos el functor $\phi^\mathtt{s} : \mathtt{Set}_S \to \mathtt{Set}_S$ como $\phi^\mathtt{s} X_S = X'_S$, donde, para todo $\mathtt{t} \in S\backslash\{\mathtt{s}\}$ tenemos $X'_\mathtt{t} = \emptyset$, y $X'_\mathtt{s} = X_\mathtt{s}$. Las acciones sobre los morfismos se definen del modo usual.

Claramente, un functor $\mathsf{F}\colon \mathtt{C} \to \mathtt{D}$ se puede extender a otro $\mathsf{F}_S = (\mathsf{F})_{\mathtt{s}\in S}\colon \mathtt{C}_S \to \mathtt{D}_S$ (para todo $\mathtt{s} \in S$, el functor permanece inalterado). Por ejemplo, tanto el functor potencia $\mathsf{P}\colon \mathtt{Set} \to \mathtt{Set}$ como el functor potencia multi-valorado $\mathsf{L}\colon \mathtt{Set} \to \mathtt{Set}$ permiten determinar functores en $\mathtt{Set}_S$, y escribimos $\mathsf{P}_S = (\mathsf{P})_{\mathtt{s}\in S}$ y $\mathsf{L}_S = (\mathsf{L})_{\mathtt{s}\in S}$. Igualmente son de interés los functores $\mathsf{G}_\mathtt{s}\colon \mathtt{C}_S \to \mathtt{D}$, $\mathtt{s} \in S$, pues podemos determinar functores $\mathsf{G}\colon \mathtt{C}_S \to \mathtt{D}_S$ tales que para todo $X_S \in \mathrm{Ob}(\mathtt{C}_S)$ tenemos $\mathsf{G}X_S = (\mathsf{G}_\mathtt{s}X_S)_{\mathtt{s}\in S}$. Obsérvese que ahora tenemos $\mathsf{G}_\mathtt{s} = \mathsf{arg}^\mathtt{s} \circ \mathsf{G}$.

Ahora, considérense cualesquiera dos functores $\mathsf{F}, \mathsf{G}\colon \mathtt{C} \to \mathtt{D}$. Una transformación natural τ entre F y G, denotada por $\tau\colon \mathsf{F} \to \mathsf{G}$, asigna a cada $\mathtt{C}$-objecto X un $\mathtt{D}$-morfismo $\tau_X : \mathsf{F}X \to \mathsf{G}X$ que satisface $\mathsf{G}f \circ \tau_X = \tau_Y \circ \mathsf{F}f$ para todo $f \in \hom_\mathtt{C}(X, Y)$. Nótese que $\mathtt{C}_S$ también es una categoría, por lo que podemos tener transformaciones naturales entre functores en $\mathtt{C}_S$, por ejemplo.

Por último, recuperamos la noción de mónada $\mathbf{F}$ sobre una categoría $\mathtt{C}$, que es un triple (F, η, μ), donde $\mathsf{F}\colon \mathtt{C} \to \mathtt{C}$ es un functor (covariante), y $\eta\colon \mathrm{id} \to \mathsf{F}$ y $\mu\colon \mathsf{F} \circ \mathsf{F} \to \mathsf{F}$ son transformaciones naturales para las que se cumple $\mu \circ \mathsf{F}\mu = \mu \circ \mu\mathsf{F}$ y $\mu \circ \mathsf{F}\eta = \mu \circ \eta\mathsf{F} = \mathrm{id}_\mathsf{F}$.

Signaturas y construcción de términos monoidales

Una signatura multi-tipada $\Sigma = (S, \Omega)$ consiste en un conjunto S de tipos (*sorts* en inglés), y un conjunto Ω de operadores. Aquí S es un conjunto de índices, mientras que Ω puede ser un objeto en $\mathtt{Set}_S$. Los operadores en $\Omega_\mathtt{s}$ es denotan como $\omega : \mathtt{s}_1 \times \cdots \times \mathtt{s}_n \to \mathtt{s}$.

Es conveniente usar la notación $\Omega^{\mathtt{s}_1\times\cdots\times\mathtt{s}_n\to\mathtt{s}}$ para el conjunto, como un objeto en $\mathtt{Set}$, de operadores $\omega : \mathtt{s}_1 \times \cdots \times \mathtt{s}_n \to \mathtt{s} \in \Omega_\mathtt{s}$ con n dado, y $\Omega^{\to\mathtt{s}}$ para el conjunto de constantes $\omega :\to \mathtt{s}$. Con estas notaciones, mantenemos un seguimiento explícito de los tipos de los operadores así como de sus aridades, y consideramos

$$\Omega_\mathtt{s} = \coprod_{\substack{\mathtt{s}_1,\ldots,\mathtt{s}_n \\ n\leq k}} \Omega^{\mathtt{s}_1\times\cdots\times\mathtt{s}_n\to\mathtt{s}}.$$

En estructuras algebráicas para valores de verdad, preferiremos usar cuantales dado que juegan un rol importante al hacer uso de categorías monoidales cerradas en la construcción formal de signaturas. Los cuantales cumplen las propiedades de los retículos residuados, y los retículos residuados completos son cuantales. Nos restringimos, además, a cuantales $\mathfrak{Q}$ que sean conmutativos y unitarios, pues esto hace que la categoría de Goguen[10] $\mathtt{Set}(\mathfrak{Q})$ sea una categoría cerrada monoidal simétri-

[10]Los objetos en una categoría de Goguen son pares (A, α), donde $\alpha : A \to Q$ es un mapeado.

ca y, por tanto, bi-cerrada. Esta categoría de Goguen incorpora toda la estructura necesaria para modelar la incertidumbre mediante las categorías subyacentes para los términos difusos sobre signaturas apropiadas, construidos por sus términos monoidales [EGHK14]. Obsérvese, de hecho, que la signatura, como objeto categórico, incorpora también la incertidumbre. Recuérdese que $(A,\alpha)\otimes(B,\beta)=(A\times B,\alpha\odot\beta)$ provee la operación monoidal sobre objetos de la categoría de Goguen. Si $\odot$ es el operador de encuentro, entonces $\otimes$ es el producto categórico.

Una signatura $(S,(\Omega,\alpha))$ sobre $\mathtt{Set}(\mathfrak{Q})$ tiene, típicamente, S como un conjunto "crisp", y así $\alpha:\Omega\to Q$ asigna valores de incertidumbre a los operadores. Para la construcción del término monádico necesitamos objetos $(\Omega^{\mathtt{s}_1\times\cdots\times\mathtt{s}_n\to\mathtt{s}},\alpha^{\mathtt{s}_1\times\cdots\times\mathtt{s}_n\to\mathtt{s}})$ para los operadores $\omega:\mathtt{s}_1\times\cdots\times\mathtt{s}_n\to\mathtt{s}$ con n dado, y $(\Omega^{\to\mathtt{s}},\alpha^{\to\mathtt{s}})$ para las constantes $\omega:\to\mathtt{s}$. Estos objetos son proporcionados por (Ω,α).

En nuestra construcción general del functor de términos tenemos

$$\Psi_{\mathtt{m},\mathtt{s}}((X_\mathtt{t})_{\mathtt{t}\in\mathtt{S}})=\Omega^{\mathtt{s}_1\times\ldots\times\mathtt{s}_n\to\mathtt{s}}\otimes\bigotimes_{i=1,\ldots,n}X_{\mathtt{s}_i},$$

lo que se especializa, en el caso de la categoría de Goguen, a

$$\begin{aligned}\Psi_{\mathtt{m},\mathtt{s}}(((X_\mathtt{t},\delta_\mathtt{t}))_{\mathtt{t}\in\mathtt{S}})&=(\Omega^{\mathtt{s}_1\times\ldots\times\mathtt{s}_n\to\mathtt{s}},\ \alpha^{\mathtt{s}_1\times\ldots\times\mathtt{s}_n\to\mathtt{s}})\otimes\bigotimes_{i=1,\ldots,n}(X_{\mathtt{s}_i},\delta_{\mathtt{s}_i})\\&=(\Omega^{\mathtt{s}_1\times\ldots\times\mathtt{s}_n\to\mathtt{s}}\times\prod_{i=1,\ldots,n}X_{\mathtt{s}_i},\ \alpha^{\mathtt{s}_1\times\ldots\times\mathtt{s}_n\to\mathtt{s}}\odot\bigodot_{i=1,\ldots,n}\delta_{\mathtt{s}_i}).\end{aligned}$$

Los pasos inductivos empiezan con $\mathsf{T}^1_{\Sigma,\mathtt{s}}=\coprod_{\mathtt{m}\in\hat{\mathtt{S}}}\Psi_{\mathtt{m},\mathtt{s}}$, y, para $\iota>1$, se procede con $\mathsf{T}^\iota_{\Sigma,\mathtt{s}}X_\mathtt{S}=\coprod_{\mathtt{m}\in\hat{\mathtt{S}}}\Psi_{\mathtt{m},\mathtt{s}}(\mathsf{T}^{\iota-1}_{\Sigma,\mathtt{t}}X_\mathtt{S}\sqcup X_\mathtt{t})_{\mathtt{t}\in\mathtt{S}})$, y $\mathsf{T}^\iota_{\Sigma,\mathtt{s}}f_\mathtt{S}=\coprod_{\mathtt{m}\in\hat{\mathtt{S}}}\Psi_{\mathtt{m},\mathtt{s}}(\mathsf{T}^{\iota-1}_{\Sigma,\mathtt{t}}f_\mathtt{S}\sqcup f_\mathtt{t})_{\mathtt{t}\in\mathtt{S}})$.

Esto nos permite definir los functores T^ι_Σ mediante $\mathsf{T}^\iota_\Sigma X_\mathtt{S}=(\mathsf{T}^\iota_{\Sigma,\mathtt{s}}X_\mathtt{S})_{\mathtt{s}\in\mathtt{S}}$, y $\mathsf{T}^\iota_\Sigma f_\mathtt{S}=(\mathsf{T}^\iota_{\Sigma,\mathtt{s}}f_\mathtt{S})_{\mathtt{s}\in\mathtt{S}}$. Hay una transformación natural $\Xi^{\iota+1}_\iota:\mathsf{T}^\iota_\Sigma\to\mathsf{T}^{\iota+1}_\Sigma$ tal que $(\mathsf{T}^\iota_\Sigma)_{\iota>0}$ es un sistema inductivo de endofunctores con $\Xi^{\iota+1}_\iota$ como su mapeado. Existe el límite inductivo $\mathsf{F}=\mathrm{ind}\varinjlim\mathsf{T}^\iota_\Sigma$ y el functor de términos final T_Σ es $\mathsf{T}_\Sigma=\mathsf{F}\sqcup\mathsf{id}_{\mathtt{Set}_\mathtt{S}}$. También tenemos $\mathsf{T}_\Sigma X_\mathtt{S}=(\mathsf{T}_{\Sigma,\mathtt{s}}X_\mathtt{S})_{\mathtt{s}\in\mathtt{S}}$, y T_Σ no es estrictamente idempotente, si bien existe un isomorfismo natural entre $\mathsf{T}_\Sigma\mathsf{T}_\Sigma$ y T_Σ.

Para más detalles concernientes a la construcción de este functor, véase [EGHK14].

Sentencias en un contexto de programación lógica

Sea $\Sigma_0=(S_0,\Omega_0)$ una signatura sobre $\mathtt{Set}$, y $\mathbf{T}_{\Sigma_0}$ la mónada de términos sobre $\mathtt{Set}_{S_0}$. Para las variables en X_{S_0}, el conjunto de términos $\mathsf{T}_{\Sigma_0}X_{S_0}$, como objeto de

$\mathtt{Set}_{S_0}$, corresponden a los "términos" y, de forma similar $\mathsf{T}_{\Sigma_0}\emptyset_{S_0}$ será el conjunto de los llamados "términos básicos" en el sentido de [Llo84].

Con objeto de introducir predicados como operadores de una signatura separada, de modo que el functor resultante pueda componerse con el de términos, asumimos que Σ contiene un tipo `bool`, que no aparece relacionado con ningún operador de Ω, esto es, establecemos $S = S_0 \cup \{\mathtt{bool}\}$, $\mathtt{bool} \notin S_0$, y $\Omega = \Omega_0$. Esto significa que $\mathsf{T}_{\Sigma,\mathtt{bool}}X_S = X_{\mathtt{bool}}$, y, para cualquier sustitución $\sigma_S : X_S \to \mathsf{T}_\Sigma X_S$, tenemos $\sigma_{\mathtt{bool}}(x) = x$ para todo $x \in X_{\mathtt{bool}}$. Se espera que la composición del functor de "predicados" con T_Σ corresponda al functor de "predicados como términos".

Podemos ahora separar la lógica proposicional de la lógica de predicados, y decidir también si incluir o no la negación. El efecto de ello es que la implicación pasa a ser un dispositivo sintáctico donde la conjunción (y, en caso de introducirse, la negación), produce términos de términos.

Para conseguir este objetivo, sea $\Sigma_{PL} = (S_{PL}, \Omega_{PL})$ la signatura de la *lógica proposicional bi-valuada* subyacente, donde $S_{PL} = S$, y $\Omega_{PL} = \{\mathtt{F}, \mathtt{T} :\to \mathtt{bool}, \& : \mathtt{bool} \times \mathtt{bool} \to \mathtt{bool}, \neg : \mathtt{bool} \to \mathtt{bool}\} \cup \{\mathtt{P}_i : \mathtt{s}_{i_1} \times \cdots \times \mathtt{s}_{i_n} \to \mathtt{bool} \mid i \in I, \mathtt{s}_{i_j} \in S\}$. Así como `bool` no produce términos adicionales, salvo por variables adicionales que sean términos al usar Σ, los tipos en S_{PL}, distintos de `bool`, no construirán términos adicionales salvo variables. A primera vista añadir predicados como operadores, incluso si no producen términos, parece superfluo, pero está justificado por la composición de estos functores de términos con T_Σ.

En el caso multi-valorado, tendríamos un tipo `lat`, tal que $\mathfrak{A}(\mathtt{lat}) = L$, fuera el conjunto correspondiente a un retículo completo $\mathfrak{L}$. De hecho $\mathfrak{L}$ puede ser, más específicamente, un retículo residuado, cuando se desee que la conjunción esté residuada con el operador de implicación (en el retículo) o un cuantal, lo que justifica el uso de subcategorías monoides cerradas. La elección de un retículo o un cuantal se justifica según el contexto de la aplicación a construir.

Obsérvese que es posible incluir los dos tipos `bool` y `lat` en la misma signatura, si uno necesita distinguir el caso bi-valuado del multi-valorado también a nivel sintáctico.

En la signatura *lógica proposicional multi-valorada* $\Sigma^{mv}_{PL} = (S^{mv}_{PL}, \Omega^{mv}_{PL})$ las constantes se mapean algebráicamente a valores de incertidumbre. En lo que sigue, no distinguiremos explícitamente entre Σ^{mv} y Σ, así que cuando escribamos Σ, el retículo subyacente puede ser bi-valuado o multi-valorado. Introducimos ahora la notación $\Sigma_{PL\setminus\neg}$ para la signatura en la que se prescinde del operador $\neg$ y $\Sigma_{PL\setminus\neg,\&}$ para la

signatura en la se prescinde tanto de $\neg$ como de &.

El conjunto (en términos de ZFC) de "términos" sobre Σ se da por

$$\bigcup_{\mathtt{s}\in S}(\mathsf{T}_{\Sigma,\mathtt{s}} \circ \phi^{S\setminus\mathtt{bool}})X_S,$$

y el conjunto de fórmulas lógicas proposicionales es

$$\bigcup_{\mathtt{s}\in S}(\mathsf{arg}^{\mathtt{s}} \circ \mathsf{T}_{\Sigma_{PL}} \circ \phi^{\mathtt{bool}})X_S = (\mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL}} \circ \phi^{\mathtt{bool}})X_S.$$

Usamos la expresión $\mathsf{arg}^{\mathtt{s}} \circ \mathsf{T}_{\Sigma_{PL}}$ en lugar de $\mathsf{T}_{\Sigma_{PL},\mathtt{s}}$ por conveniencia. Nótese cómo

$$(\mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \phi^{\mathtt{bool}})X_S = \{\mathtt{F},\mathtt{T}\}.$$

Las sentencias, esto es, fórmulas en lógica proposicional, se dan mediante el functor

$$\mathsf{Sen}_{PL} = \mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL}} \circ \phi^{\mathtt{bool}},$$

y las sentencias en "lógica de cláusulas de Horn" se dan por el functor

$$\begin{aligned}\mathsf{Sen}_{HCL} &= (\mathsf{arg}^{\mathtt{bool}})^2 \circ (((\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{T}_{\Sigma}) \times (\mathsf{T}_{\Sigma_{PL\setminus\neg}} \circ \mathsf{T}_{\Sigma})) \circ \phi^{S\setminus\mathtt{bool}}) \\ &= (\mathsf{arg}^{\mathtt{bool}})^2 \circ ((\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \times \mathsf{T}_{\Sigma_{PL\setminus\neg}}) \circ \mathsf{T}_{\Sigma} \circ \phi^{S\setminus\mathtt{bool}})\end{aligned}$$

Nótese también que $\mathsf{Sen}_{HCL}X_S$ es un objeto en $\mathtt{Set}$, y, por tanto, el par $(h,b) \in \mathsf{Sen}_{HCL}X_S$, como sentencia que representa una "cláusula de Horn", significa que h es un "átomo" y b es una conjunción de "átomos". Además, $(h,\mathtt{T})$ es un "hecho", $(\mathtt{F},b)$ es un "objetivo", y $(\mathtt{F},\mathtt{T})$ es un "fallo".

Estos desarrollos permiten aproximaciones similares para otras lógicas. Intuitivamente, el functor identidad es el functor de sentencias para lambda-términos (usados como "sentencias") en λ-cálculo, y id^2 es el functor de sentencias para ecuaciones (usadas como "sentencias") en lógica ecuacional.

Estamos ahora en posición de introducir *sustitución de variables*. Efectivamente, dado que tenemos la mónada $\mathbf{T}_\Sigma = (\mathsf{T}_\Sigma, \eta, \mu)$, podemos realizar la sustitución de variables $\sigma_S \colon \phi^{S\setminus\mathtt{bool}}X_S \to \mathsf{T}_\Sigma\phi^{S\setminus\mathtt{bool}}Y_S$, esto es, las variables $\phi^{S\setminus\mathtt{bool}}X_S$ son sustituidas por los términos $\mathsf{T}_\Sigma\phi^{S\setminus\mathtt{bool}}Y_S$. La sustitución es sensible al tipo: $\sigma_S = (\sigma_{\mathtt{s}})_{\mathtt{s}\in S}$ tal que $\sigma_{\mathtt{s}} \colon \mathsf{arg}^{\mathtt{s}}(\phi^{S\setminus\mathtt{bool}}X_S) \to \mathsf{T}_{\Sigma,\mathtt{s}}\phi^{S\setminus\mathtt{bool}}Y_S$. Tenemos que:

$$\mu \circ \mathsf{T}_\Sigma\sigma_S : \mathsf{T}_\Sigma\phi^{S\setminus\mathtt{bool}}X_S \to \mathsf{T}_\Sigma\phi^{S\setminus\mathtt{bool}}Y_S$$

$$\sigma_S^{head} = \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}}(\mu \circ \mathsf{T}_\Sigma \sigma_S) : (\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} X_S \to (\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} Y_S$$

$$\sigma_S^{body} = \mathsf{T}_{\Sigma_{PL\setminus\neg}}(\mu \circ \mathsf{T}_\Sigma \sigma_S) : (\mathsf{T}_{\Sigma_{PL\setminus\neg}} \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} X_S \to (\mathsf{T}_{\Sigma_{PL\setminus\neg}} \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} Y_S$$

$$(\sigma_S^{head}, \sigma_S^{body}) = (\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \times \mathsf{T}_{\Sigma_{PL\setminus\neg}})(\mu \circ \mathsf{T}_\Sigma \sigma_S) : ((\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \times \mathsf{T}_{\Sigma_{PL\setminus\neg}}) \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} X_S \to ((\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \times \mathsf{T}_{\Sigma_{PL\setminus\neg}}) \circ \mathsf{T}_\Sigma)\phi^{S\setminus\mathtt{bool}} Y_S$$

Finalmente,

$$\sigma^{HC} = (\sigma_{\mathtt{bool}}^{head}, \sigma_{\mathtt{bool}}^{body}) : \mathsf{Sen}_{HCL} X_S \to \mathsf{Sen}_{HCL} Y_S$$

Obsérvese que σ_S^{head}, σ_S^{body} y $(\sigma_S^{head}, \sigma_S^{body})$ son morfismos en $\mathtt{Set}_S$ pero σ^{HC} es un morfismo en $\mathtt{Set}$.

Es claro que la categoría de Goguen $\mathtt{Set}(\mathfrak{Q})$ es un candidato para ser la categoría subyacente. La sustitución de T_Σ por el functor compuesto $\mathsf{Q} \circ \mathsf{T}_\Sigma$, proporciona otro estilo de extensión difusa.

Álgebras, modelos y punto fijo

En el caso bi-valuado, $\mathfrak{A}(\mathtt{bool})$ siempre es $\{false, true\}$, de modo que $\mathfrak{A}(\mathtt{F}) = false$ y $\mathfrak{A}(\mathtt{T}) = true$. Además, se espera que $\mathfrak{A}(\&) : \mathfrak{A}(\mathtt{bool}) \times \mathfrak{A}(\mathtt{bool}) \to \mathfrak{A}(\mathtt{bool})$, se defina por la tradicional "tabla de verdad". Generalmente, $\mathfrak{A}(\mathtt{s}_0)$ se denota por D de modo que la semántica para un operador n-ario (sintáctico) $\omega : \mathtt{s}_0 \times \cdots \times \mathtt{s}_0 \to \mathtt{s}_0$ es una operación n-aria (una función) $\mathfrak{A}(\omega) : D^n \to D$. Un álgebra multi-tipada no es un álgebra tradicional –ni tan siquiera una tupla de álgebras tradicionales– , dado que un operador ω puede estar definido en función de muchos tipos y, por ello, la semántica de ω no es una función n-aria sobre ningún conjunto. Por ejemplo, si para la signatura $\Sigma_{PL} = (S_{PL}, \Omega_{PL})$ asignamos un par, el álgebra "multi-tipada", $(\mathsf{T}_{\Sigma_{PL}} X_S, (\mathfrak{A}(\omega))_{\omega \in \Omega_{PL}})$, donde $X_{\mathtt{s}} = \varnothing$ si $\mathtt{s} \neq \mathtt{bool}$. Por tanto, $(\bigcup_{\mathtt{s} \in S}(\mathsf{arg}^{\mathtt{s}} \circ \mathsf{T}_{\Sigma_{PL}}) X_S, (\mathtt{F}, \mathtt{T}, \&, \neg))$ sirve como un álgebra booleana tradicional bajo determinadas leyes ecuacionales.

Para un programa de un número finito de cláusulas $\Gamma = \{(h_1, b_1), \ldots, (h_n, b_n)\} \subseteq \mathsf{Sen}_{HCL} X_S$, basadas en Σ y Σ_{PL}, asignamos un objeto $\mathtt{Set}_S$

$$(U_\Gamma)_S = \mathsf{T}_\Sigma \emptyset_S = (\mathsf{T}_{\Sigma,\mathtt{s}} \emptyset_S)_{\mathtt{s} \in S}$$

donde $\mathsf{T}_{\Sigma,\mathtt{s}}\emptyset_S$ es el conjunto de todos los términos básicos de tipo $\mathtt{s}$, y, efectivamente, $\mathsf{T}_{\Sigma,\mathtt{bool}}\emptyset_S = \emptyset$. Nótese cómo $\bigcup_{\mathtt{s}\in S}(U_\Gamma)_\mathtt{s}$ corresponde a la visión tradicional y no tipada de *universo de Herbrand* como un conjunto en el sentido de ZFC.

También nos interesa el objeto `Set`

$$B_\Gamma = (\mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{T}_\Sigma)\ \emptyset_S$$

que corresponde a la *base de Herbrand* en el sentido tradicional [Llo84].

Las interpretaciones de Herbrand de un programa Γ son subconjuntos $\mathcal{I} \subseteq B_\Gamma$, esto es, $\mathcal{I} \in \mathsf{P}B_\Gamma$.

Por conveniencia, cuando tratemos con el operador de consecuencias inmediatas para las consideraciones del punto fijo, haremos uso de la *base de expresiones de Herbrand*

$$B_\Gamma^\& = (\mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL\setminus\neg}} \circ \mathsf{T}_\Sigma)\ \emptyset_S.$$

Obsérvese que una interpretación de Herbrand $\mathcal{I}$ se extiende canónicamente a una *interpretación de expresión de Herbrand* $\mathcal{I}^\& \subseteq B_\Gamma^\&$. De igual modo, si $\mathcal{I} \in \mathsf{L}B_\Gamma$, se puede extender $\mathcal{I}$ a una *interpretación de expresión difusa de Herbrand* $\mathcal{I}^\&$ (semánticamente) como sigue: para cada elemento $b \in B_\Gamma^\&$ de la forma $b = b_1 \& \cdots \& b_n$ tenemos $\mathcal{I}^\&(b) = \bigwedge\{\mathcal{I}(b_1), \ldots, \mathcal{I}(b_n)\}$ y para un elemento atómico $b \in B_\Gamma^\&$, $\mathcal{I}^\&(b) = \mathcal{I}(b)$. Sin embargo, es cuestionable denominar 'ìnterpretación" a $\mathcal{I} \in \mathsf{L}B_\Gamma$.

La interpretación de Herbrand es el "álgebra de términos básicos" [Llo84] en el sentido del álgebra universal. Se trata de una T_Σ-álgebra, en lugar de una Σ-álgebra, que corresponde a "interpretación", hablando en todo caso en el ámbito de los términos básicos (en el sentido de que los conjuntos de variables son vacíos).

La extensión al caso multi-valorado se divide en dos posibilidades. En una se realiza la composición del functor del conjunto potencia multi-valorado L con el término de functores. El resultado de esto es un estilo de "lógica con borrosidad". La otra posibilidad consiste en definir el functor de términos sobre la categoría de Goguen, lo que produce un estilo de "lógica difusa". Por ello, esta extensión no debe verse simplemente como la sustitución del functor P a L, siendo $\mathfrak{L}$ el retículo completo subyacente, y extendiendo las interpretaciones de Herbrand a *interpretaciones de Herbrand difusas* de un programa Γ por $\mathcal{I} \in \mathsf{L}B_\Gamma$. Como hemos dicho, existen dos vías de acción. En una, anotamos los grados de verdad "desde fuera", como se ha mencionado, produciendo sentencias mediante el functor $\mathsf{L} \circ \mathsf{Sen}_{HCL}$, e interpretaciones como conjuntos difusos de predicados

$$\mathsf{L}B_\Gamma = (\mathsf{L} \circ \mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{T}_\Sigma)\ \emptyset_S.$$

Una interpretación difusa en este caso es tan sólo un mapeado $\mathcal{I} : B_\Gamma \to L$, y la incertidumbre surge de los términos, mientras que las sustituciones permanecen inalteradas.

Por otro lado, podemos introducir los grados de verdad "en el interior" para producir una *lógica de cláusulas de Horn difusas de sustitución* con el functor de sentencias

$$\mathsf{Sen}_{SFHCL} = (\mathsf{arg}^{\mathtt{bool}})^2 \circ ((\mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \times \mathsf{T}_{\Sigma_{PL\setminus\neg}}) \circ \mathsf{L}_S \circ \mathsf{T}_\Sigma \circ \phi^{S\setminus\mathtt{bool}})$$

de modo que el conjunto de predicados básicos sobre conjuntos difusos de términos es

$$B_\Gamma^{\mathsf{L}} = (\mathsf{arg}^{\mathtt{bool}} \circ \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{L}_S \circ \mathsf{T}_\Sigma)\ \emptyset_S$$

definiendo la correspondiente extensión $B_\Gamma^{\mathsf{L},\&}$ del modo usual. Los conjuntos difusos de predicados básicos resultantes se consiguen al considerar la transformación natural llamada "swapper", intercambiador

$$\varsigma : \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}} \circ \mathsf{L}_S \to \mathsf{L}_S \circ \mathsf{T}_{\Sigma_{PL\setminus\neg,\&}}$$

dada en [EGHK11] para el caso multi-tipado, y en [EGOV00] para el de un solo tipo. De hecho, se puede usar $\mathsf{arg}^{\mathtt{bool}}{}_{\varsigma_{\mathsf{T}_\Sigma\emptyset_S}} : B_\Gamma^{\mathsf{L}} \to \mathsf{L}B_\Gamma$. Nótese cómo $\mathsf{L}B_\Gamma^{\mathsf{L}}$ puede corresponder a una base de Herbrand como conjunto con consideraciones de incertidumbre tanto para conjuntos de cláusulas como para conjuntos de términos.

Al respecto de los puntos fijos, consideramos primero la sustitución "crisp" de términos básicos, esto es, un $\mathtt{Set}_S$-morfismo $\sigma_S \colon X_S \to \mathsf{T}_\Sigma \varnothing_S$. Por la discusión precedente, esto induce un morfismo $\sigma^{HC} \colon \mathsf{Sen}_{HCL} X_S \to \mathsf{Sen}_{HCL} \varnothing_S$. Se puede definir ahora un mapeado $\varpi : \mathsf{L}B_\Gamma \to \mathsf{L}B_\Gamma$, donde el retículo subyacente $\mathfrak{L}$ para el functor multi-valorado del conjunto potencia L es un retículo completo, por

$$\varpi(\mathcal{I})(\sigma^{head}_{\mathtt{bool}}(h)) = \bigvee_{(h,b)\in\Gamma} \mathcal{I}^{\&}(\sigma^{body}_{\mathtt{bool}}(b)).$$

Cuando $(h, b) \in B_\Gamma \times B_\Gamma$ es tal que $(h, b) \notin R_{\sigma^{HC}}$ (el rango de σ^{HC}), entonces $\varpi(\mathcal{I})(h) = \mathcal{I}(h)$ y $\varpi(\mathcal{I})(b) = \mathcal{I}(b)$.
Claramente, ϖ es monótono, y es bien sabido que ϖ tiene un punto fijo mínimo y otro máximo.

Sin embargo, esto es una aproximación simplista a los modelos difusos, dado que las sustituciones no son difusas. Se pueden proporcionar los mapeados $\sigma_S^{\mathsf{L},head}$

y $\sigma_S^{\mathsf{L},body}$ con L_S "en el interior" para las sustituciones de términos básicos difusos, esto es, un $\mathtt{Set}_S$-morfismo de la forma $\sigma_S^{\mathsf{L}} \colon X_S \to \mathsf{L}_S \mathsf{T}_\Sigma \varnothing_S$.

Considerando el efecto sobre sustituciones con conjuntos difusos de términos, se puede considerar un mapeado $\varpi^{\mathsf{L}} : \mathsf{L}B_\Gamma^{\mathsf{L}} \to \mathsf{L}B_\Gamma^{\mathsf{L}}$, usando $\mathsf{arg}^{\mathtt{bool}} \varsigma_{\mathsf{T}_\Sigma \emptyset_S} : B_\Gamma^{\mathsf{L}} \to \mathsf{L}B_\Gamma$, de varios modos, por ejemplo, como

$$\varpi^{\mathsf{L}}(\mathcal{I})(\sigma_{\mathtt{bool}}^{\mathsf{L},head}(h)) = (\bigvee_{t \in B_\Gamma} (\mathsf{arg}^{\mathtt{bool}} \varsigma_{\mathsf{T}_\Sigma \emptyset_S}(h))(t)) \wedge \mathcal{I}^{\mathsf{L},\&}(\sigma_{\mathtt{bool}}^{\mathsf{L},body}(b)).$$

En este caso, ϖ^{L} también es monótono.

3.9. Conclusiones

En este capítulo hemos repasado los principales conceptos del lenguaje MALP, desde su sintaxis y sus diferentes semánticas hasta el cálculo de su coste computacional. A continuación concretamos los diferentes propósitos de este capítulo al respecto de este lenguaje.

- Hemos justificado la elección de la programación lógica multi-adjunta como marco de nuestras investigaciones en el campo de la programación lógica difusa. Su alto nivel de generalidad y expresividad, así como su semántica operacional claramente definida, hacen de él (en nuestra opinión), un marco idóneo para nuestros trabajos en esta creciente área de conocimiento.
- Hemos descrito su semántica operacional, que se concibe en base a un sistema de transición de estados y dos fases claramente diferenciadas, en las que se pueden aplicar pasos admisibles y pasos interpretativos (fase de resolución difusa y fase interpretativa, respectivamente). La reformulación y optimización de esta última fase es uno de los avances más significativos de esta tesis, y será explicada en profundidad en el siguiente capítulo.
- También hemos descrito su semántica declarativa, reformulada en términos de conjuntos difusos [MPV14c].
- A fin de ubicar dicha semántica operacional, la hemos revisado y comparado con otras ya descritas en la literatura para MALP, como la semántica de punto fijo o la semántica declarativa basada en el modelo mínimo de Herbrand difuso, estableciendo las equivalencias existentes entre ellas, en particular, su completitud [MMPV12a, MMPV11b].

- Para ello hemos estimado el coste computacional asociado a la ejecución de programas lógico multi-adjuntos. Este punto es de vital importancia para estimar la ganancia en eficiencia en los programas transformados por técnicas ampliamente estudiadas por nuestro grupo de investigación, como el plegado/desplegado. El método clásico de contar el número de pasos de ejecución dados no era una medida realista, ya que en la fase interpretativa no se tenía en cuenta las diferentes complejidades de las conectivas evaluadas.

- Como solución, proponemos la nueva medida de coste $\mathcal{I}_c^+$, la cual, gracias a la noción de "peso de una conectiva" explicada en la Sección 3.4, y a diferencia de otras medidas estudiadas en nuestro grupo en el pasado (como $\mathcal{I}_c$ o $\mathcal{I}_c^*$), sí tiene en cuenta todos los operadores que se evalúan dentro de una conectiva así como posibles llamadas anidadas a otras conectivas.

- A pesar de que con la medida de coste mejorada ya podemos estimar con exactitud el coste asociado a cualquier derivación, y comprobar así si los programas transformados tienen "mejor" o "peor" comportamiento computacional, en este trabajo quisimos ir un paso más allá para poder mostrar todo lo que está ocurriendo explícitamente durante la fase interpretativa. Para ello, redefinimos dicha fase creando los llamados pasos interpretativos cortos como detallamos en la Sección 3.5.

- Con ellos, pretendemos mostrar explícitamente en la derivación tanto si se está "expandiendo" una conectiva (hecho capturado por los pasos interpretativos cortos de tipo 1) como si se está evaluando un operador primitivo (hecho reflejado por los pasos interpretativos cortos de tipo 2). Esta técnica, además de hacer mucho más sencilla la medición del coste computacional asociado a la ejecución de programas, nos brinda la posibilidad de generar información muy detallada de las trazas de ejecución.

- Esta idea de "documentar" la ejecución de programas la recogemos en la Sección 3.6, en la que explicamos cómo redefinir los retículos multi-adjuntos sobre los que se interpretan cualquier programa y objetivo dado, para que sean capaces de incorporar la información más relevante producida durante la fase de ejecución. Dicha traza de ejecución se presenta al usuario junto con la respuesta computada difusa, y en ella, además del grado de verdad con el que se alcanza la solución (si la hubiere), gracias a nuestras investigaciones somos

capaces de mostrar qué reglas se han utilizado para realizar los pasos admisibles, y qué operadores y conectivas se han ido evaluando en cada paso durante la fase interpretativa (para lo cual es fundamental la utilización de los pasos interpretativos cortos).

- Nuestra aportación en este campo consiste tanto en la redefinición de la fase interpretativa como en la generación de trazas de ejecución. Para manejar estas nuevas técnicas, el usuario dispone de tres nuevas opciones como "ismode", "lat" o "show", tal y como mostramos en la Sección 4.3. La primera permite mostrar los pasos interpretativos cortos, y las dos últimas gestionan los retículos multiadjuntos que nos permitirán mostrar las trazas asociadas a la ejecución de programas.

- También hemos introducido un método para convertir relaciones de similaridad en reglas MALP, de modo que podamos disfrutar de una noción de igualdad más rica que la igualdad sintáctica de este lenguaje, concretamente, una forma de igualdad estricta basada en similaridades. Además, este método sólo requiere del usuario que presente un conjunto mínimo de ecuaciones de similaridad (en lugar de dar explicitamente todos los pares de la relación), que es expandido más adelante mediante las clausuras reflexiva, simétrica y transitiva. El resultado de este método es una herramienta libremente accesible desde `http://dectau.uclm.es/sse/`.

En la última sección de este capítulo hemos abordado una temática menos relacionada con el lenguaje MALP en sí, y más con la fundamentación de la programación lógica difusa desde el punto de vista de la teoría de categorías, intentando en todo momento describir un lenguaje lo más próximo posible a MALP, a la vez que aprovechando la riqueza de este marco matemático para introducir nociones novedosas como la signatura multi-tipada.

A este respecto, procedemos contemplando la lógica como una estructura que contiene signaturas, términos, sentencias, conjuntos estructurados de sentencias, implicaciones, álgebras, satisfacciones, axiomas, teorías y cálculo de demostraciones. Las signaturas contienen tipos y operadores, y su significado está proporcionado por las álgebras. Los términos se construyen a partir de operadores en la signatura, y las sentencias se construyen empleando los términos como bloques de construcción. La implicación es una relación entre conocimiento dado y conocimiento a demostrar. La satisfacción es la contraparte semántica de la implicación, y provee la noción de

conclusión lógica. Los axiomas describen conocimiento dado, y actuan a modo de programa lógico. Las reglas de inferencia determinan cómo llegar a las conclusiones en una cadena de implicaciones.

Se ha identificado la diferencia entre la lógica no tipada y la multi-tipada, y, del mismo modo, entre la lógica clásica ("crisp") y las lógicas difusas. También distinguimos entre "programación lógica difusa", que enriquece las consideraciones de categorías subyacentes, y "programación lógica con borrosidad", que compone términos monádicos usando `Set` como categoría subyacente.

Las consideraciones difusas en la lógica se relacionan, por tanto, a estructuras que contienen signaturas difusas, términos difusos, sentencias difusas, conjuntos difusos estructurados de sentencias, implicaciones difusas, álgebras difusas, satisfacciones difusas, axiomas difusos, teorías difusas y cálculos de demostraciones difusos.

Capítulo 4

Programación lógica difusa en el entorno $\mathcal{FLOPER}$

En el capítulo anterior hemos detallado las características y algunas de las propiedades del lenguaje multi-adjunto, poniendo especial atención a las definiciones de respuesta correcta y respuesta computada. También hemos explicado algunas técnicas relacionadas con el cálculo de coste computacional de la evaluación de un objetivo MALP y para la documentación de las respuestas computadas difusas.

Con objeto de poner a prueba este marco teórico con problemas del mundo real, en nuestro grupo de investigación hemos desarrollado desde 2007 una herramienta que permite la ejecución y depuración de programas MALP, que hemos bautizado $\mathcal{FLOPER}$ (por *Fuzzy LOgic Programming Environment for Research*), en cuyo desarrollo he participado activamente desde el año 2011 ([MV14, MMPV10b, MMPV10a, MMPV10c, JMPV14, IMPV15]).

En la Sección 4.1 detallamos las nociones básicas de la herramienta $\mathcal{FLOPER}$ atendiendo a su interfaz textual, mientras que su interfaz visual se recoge en la Sección 4.2. A continuación se detalla la implementación de la fase interpretativa, que tratamos en [MMPV10b], trabajo que no habría sido posible antes de implementar la noción de *retículo multi-adjunto* en $\mathcal{FLOPER}$, como se hizo en [MMPV10a, MMPV10c].

En la Sección 4.4 recogemos los avances de [MMPV11a, MMPV11c], donde implementamos la noción de traza declarativa, que es una cadena de caracteres que acompaña a las respuestas computadas dadas por el sistema para documentar el

proceso de evaluación de un objetivo. Este trabajo no habría sido posible sin antes demostrar el carácter multi-adjunto del retículo de cadenas de caracteres, o *strings* [MMPV12c, MMPV12b] (detallados en el Capítulo 2) y del producto cartesiano de retículos multi-adjuntos [MMPV12a] (detallado en el Capítulo 3). Hasta este punto, todos estos desarrollos están recogidos en el trabajo [MV14].

Concluímos el capítulo tras la Sección 4.5, donde se detalla el desarrollo de la herramienta $\mathcal{FLOPER}$-online, que facilita el uso de la herramienta desde un navegador, teniendo conexión a Internet.

4.1. Un entorno de programación lógica difusa

Uno de los principales desafíos que deben afrontar las tecnologías *software* es dar respuesta a problemas en entornos cambiantes e imprecisos, en forma de nuevos métodos, técnicas, plataformas y herramientas. Respecto a los lenguajes, hemos justificado que el paradigma declarativo disfruta de una serie de ventajas basadas en su fundamentación en algún tipo de formalismo subyacente, como la correspondencia entre su sintaxis (programas) y su semántica (su significado) o la posibilidad de analizar (manual o automáticamente) un programa difuso para extraer conclusiones o incluso transformarlo. Hemos descrito en el capítulo anterior un lenguaje lógico difuso, MALP, orientado a este tipo de problemas. Más allá de eso, se impone la necesidad de crear un entorno de programación que dé soporte sistemático y racional al desarrollo del *software*.

Viene a responder a esta necesidad el prototipo $\mathcal{FLOPER}$, diseñado en nuestro grupo de investigación y en cuyo desarrollo he tenido una contribución significativa, como queda reflejado en los artículos [MMPV10a, MMPV10b, MMPV10c, MV14, JMPV14, IMPV15]. $\mathcal{FLOPER}$ cuenta, además, con una página web propia, `http://dectau.uclm.es/floper/`, que informa de sus capacidades y permite probarla online.

4.1.1. Características generales de $\mathcal{FLOPER}$

Nuestra aproximación a una herramienta de desarrollo de aplicaciones lógico difusas está parcialmente inspirada en [GMV04], donde se presenta un intérprete para un lenguaje lógico difuso próximo a MALP. En dicho trabajo se utiliza programación lógica con restricciones sobre números reales ($CLP(\mathcal{R})$) para lograr una implementación eficiente. En lo que respecta a $\mathcal{FLOPER}$, se ha optado por el uso del lenguaje

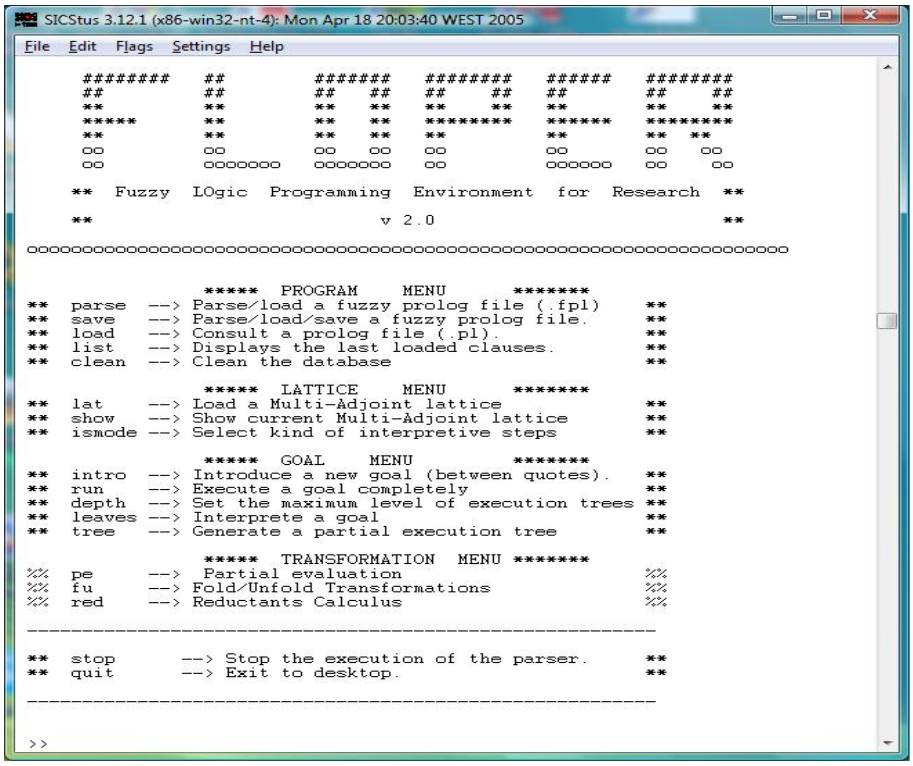


Figura 4.1: Interfaz de texto de $\mathcal{FLOPER}$

de programación Prolog sin considerar una extensión $CLP(\mathcal{R})$, ya que éste es suficiente para implementar la herramienta, además de que la simplicidad tanto de la técnica como del lenguaje objeto resulta accesible a una audiencia más amplia.

Inicialmente, nuestro sistema estaba restringido a programas cuyo retículo asociado fuera el intervalo cerrado $[0, 1]$ con la relación de orden usual, y equipado con conectivas procedentes de alguna lógica difusa convencional (como la lógica del producto, lógica de Łukasiewicz y lógica de Gödel). Actualmente, la herramienta puede tratar programas asociados a cualquier tipo de retículo multi-adjunto como los descritos en la Sección 2.4. Añadimos a este respecto que, gracias a esta mejora en la expresividad de la herramienta, es posible ejemplificar en la práctica desarrollos teóricos como las trazas declarativas (entre otros) anotadas en las respuestas del sistema.

Nuestra herramienta ofrece una interfaz avanzada similar a la de muchos otros

entornos de desarrollo y ofrece recursos familiares para los programadores acostumbrados a los lenguajes declarativos. A bajo nivel, $\mathcal{FLOPER}$ ha sido implementado en Sicstus PROLOG v.3.12.5, aunque está diseñado para ser ejecutado en cualquier intérprete de PROLOG.

El núcleo de la herramienta se basa en un analizador sintáctico y un emulador de la semántica operacional descrita en la Sección 3.2. El analizador sintáctico ha sido desarrollado a partir de *Gramáticas de Cláusulas Definidas* (Definite Clause Grammars, DCG's), un recurso del lenguaje PROLOG que aporta una notación adecuada para enunciar las reglas de producción de una gramática de forma ágil y precisa. El proceso de análisis permite introducir programas y objetivos difusos al sistema, y el emulador de la semántica operacional devuelve las respuestas computadas difusas (esto es, pares que incluyen un grado de verdad y una sustitución) aún cuando todos los cómputos intermedios se han ejecutado en un entorno de programación lógica pura (no difusa). Una vez se carga este núcleo en un intérprete PROLOG, se muestra mediante una interfaz de texto las opciones disponibles (Figura 4.1), que detallamos a continuación.

El primer submenú, el **menú de programa**, ofrece (entre otras) opciones para:

• `Cargar` un archivo PROLOG con extensión `'.pl'`. Esta acción permite leer un programa PROLOG a partir de un archivo. Dicho programa puede consistir en una serie de definiciones clásicas (no difusas), aunque su uso más adecuado es el de definir variables lingüísticas.

• `Listar` el conjunto de cláusulas PROLOG cargadas desde un archivo `'.pl'` así como las que se han obtenido después de compilar un archivo `'.fpl'`. Este comando muestra también el programa difuso original contenido en este último archivo, como se observa en la Figura 4.2.

• `Analizar` un programa difuso contenido en un archivo con extensión `'.fpl'`. Este es el comando principal que se encarga de introducir en el sistema el código difuso. Para ello produce dos clases de código. El primero, que llamamos *de alto nivel*, consiste en una traducción regla por regla de código difuso a código PROLOG, que puede ejecutarse en cualquier entorno PROLOG a costa de perder determinadas propiedades de la semántica de MALP. El segundo, que llamamos *de bajo nivel*, sólo puede ser ejecutado en $\mathcal{FLOPER}$, si bien conserva completamente la semántica operacional descrita en la Sección 3.2.

•`Guardar` el código PROLOG resultante del comando anterior en un archivo.

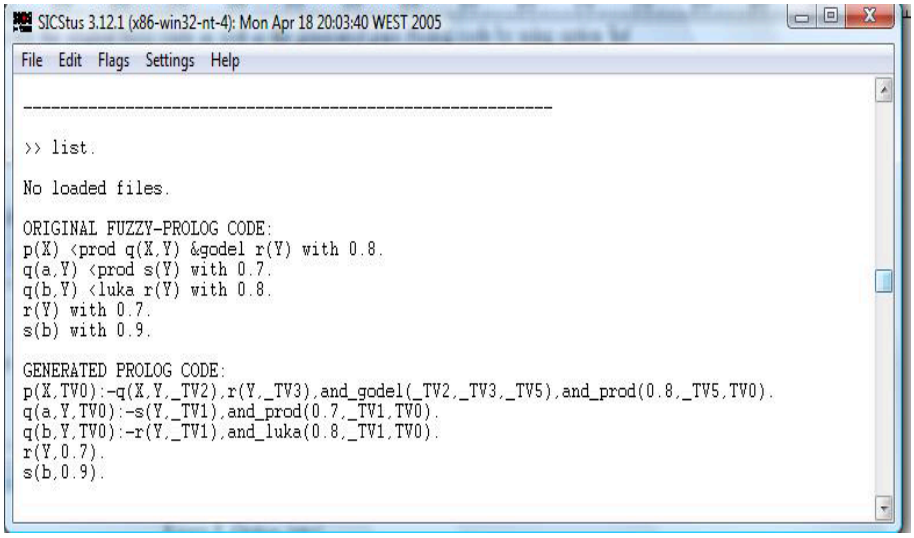


Figura 4.2: Ejecución de la opción "List"

El segundo submenú está dedicado a la gestión de los **retículos**, y ofrece comandos para:

• `Cambiar` el retículo cargado en ese momento en el sistema (opción "lat") indicando al sistema el fichero (con extensión `.lat`) que describe el nuevo retículo. Por defecto todo programa MALP está asociado inicialmente al retículo $([0,1], \leq)$, definido en el fichero `num.lat`. La implementación y gestión interna de dichos retículos por parte de $\mathcal{FLOPER}$ se detalla ampliamente en la sección posterior.

• `Mostrar` el retículo cargado.

• `Elegir` mediante el comando "ismode" el tipo de pasos interpretativos a usar en las derivaciones. Se permite elegir entre los pasos interpretativos originales (etiqueta `medium`), pasos interpretativos cortos descritos en la Sección 3.5 (etiqueta `small`) y la total omisión de la fase interpretativa en la presentación del árbol de derivación, dejando únicamente la respuesta computada difusa, si la hubiera (etiqueta `large`). Detallamos con más profundidad este comando en la Sección 4.3.1.

El tercer submenú recoge los comandos dedicados a la evaluación de un objetivo. En concreto:

• `Introducir` un objetivo en el sistema. El objetivo debe escribirse entre comillas simples, como se aprecia en la Figura 4.3. Inicialmente no hay ningún objetivo cargado en el sistema, por lo que siempre será necesario invocar esta opción antes de lanzar cualquier ejecución o generación de árboles de desplegado.

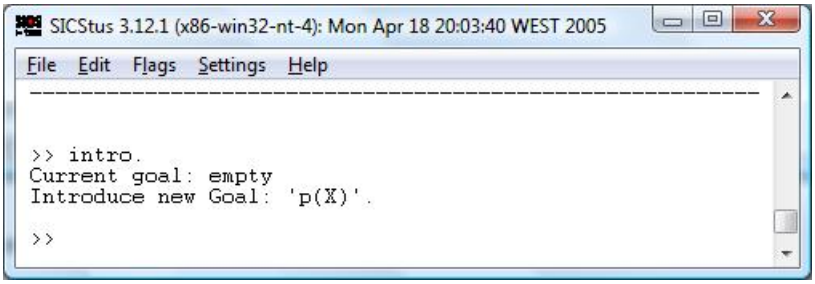


Figura 4.3: Opción "Intro"

• `Ejecutar` un objetivo difuso (previamente introducido por teclado mediante el comando **intro**). La implementación de este comando se basa en la traducción a código PROLOG del programa y el objetivo difuso introducidos. Recordamos que en esta implementación, todos los cálculos internos (incluyendo compilación y ejecución) son derivaciones PROLOG puras mientras que las entradas (programas y objetivos difusos) y salidas (respuestas computadas difusas) tienen siempre apariencia difusa. Esta técnica se detalla ampliamente en la Sección 4.1.3.

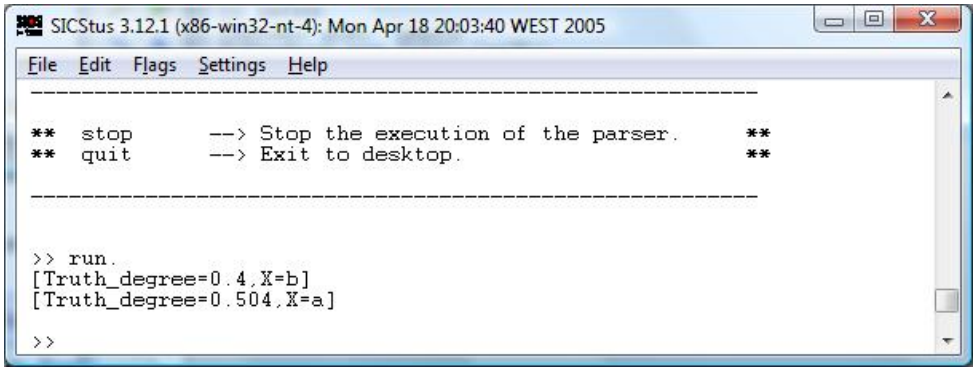


Figura 4.4: Opción "Run"

La Figura 4.4 muestra la ejecución del objetivo `p(X)` previamente introducido contra el programa de la Figura 4.2, con la respuesta computada difusa esperada.

• `Generar` árboles de ejecución parcial y otras tareas relacionadas, como cambiar la `profundidad` de los árboles o visualizar sus hojas (comandos "`tree`", "`depth`" y "`leaves`", respectivamente). Todas estas funcionalidades de $\mathcal{FLOPER}$ se basan en el código *de bajo nivel* producido en el análisis del programa y del objetivo. Detallamos esta técnica en la Sección 4.1.4. Mostramos un ejemplo de árbol de ejecución mediante el comando **tree** (utilizando de nuevo el ejemplo de la Figura 4.2 y el objetivo `p(X)`) en la Figura 4.3.

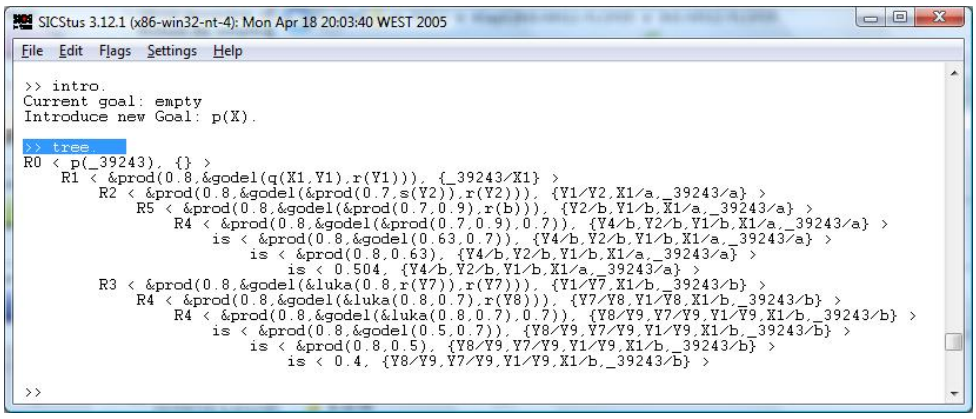


Figura 4.5: Opción "tree"

Finalmente, el **menú de transformación** recoge comandos relacionados con las técnicas de transformación automática de programas estudiadas en nuestro grupo, no implementadas todavía en la herramienta. En concreto uno de los objetivos prioritarios en el desarrollo actual del sistema consiste en la implementación de tres de ellas: las técnicas de evaluación parcial de objetivos, transformaciones de programas basadas en plegado/desplegado y el cálculo de reductantes.

4.1.2. Gestión de retículos y modificadores lingüísticos en $\mathcal{FLOPER}$

Como se recoge en la Sección 2.4, un retículo multi-adjunto es un retículo completo equipado con un número arbitrario de pares $(\&_i, \leftarrow_i)$, donde $\leftarrow_i$ es una implicación y

$\&_i$ es su conjunción adjunta. Una de las capacidades más notorias de la herramienta $\mathcal{FLOPER}$ es el uso de diferentes retículos para interpretar diferentes programas difusos, sobresaliendo así con respecto a la mayoría de sistemas similares, que sólo aceptan el retículo usual $[0, 1]$. De hecho, nuestra herramienta ofrece un submenú dedicado a la gestión del retículo, y que incluye los comandos **show** y **lat** para mostrar y cambiar el retículo cargado, respectivamente.

Esta característica es posible gracias a un método para introducir un retículo en la herramienta, consistente en la descripción del mismo en un fichero con formato PROLOG y extensión ".lat". El ejemplo más ilustrativo de este tipo de ficheros es "bool.lat", que describe el retículo multi-adjunto binario $\{0, 1\}$, y que usamos para ejemplificar los predicados que definen estos ficheros.

- `member/1`. Este predicado se usa para definir el conjunto base del retículo. Se satisface sobre cada grado de verdad válido para el retículo. En el caso de retículos finitos es recomendable definir también `members/1`, cuyo parámetro es una lista de todos los grados de verdad. En el caso booleano, ambos predicados pueden ser modelados con: `member(0). member(1).` y `members([0,1]).`

- `bot/1` y `top/1`. Se satisfacen sobre el ínfimo y el supremo del retículo, respectivamente. Para el retículo booleano se implementan como `bot(0).` y `top(1).`

- `leq/2`. Modela la relación de orden del retículo. Sólo se satisface cuando se evalúa sobre dos elementos comparables del retículo, cuando el primero es menor o igual que el segundo. En un retículo con orden total este predicado se satisface para todo par de elementos (evaluados en un orden determinado). Para el retículo booleano, este predicado se implementa con los hechos `leq(0,X).` y `let(X,1).`

- Finalmente, se define un conjunto de conectivos de la forma $\&_{label_1}$ (conjunción), $\vee_{label_2}$ (disyunción), o $@_{label_3}$ (agregador), con aridades n_1, n_2 y n_3, respectivamente, mediante cláusulas de la siguiente forma:

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

  donde el último parámetro de cada predicado corresponde a una variable que unifica con el resultado de la evaluación del conectivo. Para el caso booleano la conjunción clásica se puede modelar mediante estos dos hechos:

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

Obsérvese que los programas MALP cuyas reglas son de la forma:

"$A \leftarrow_{bool} \&_{bool}(B_1, \ldots, B_n)$ *with* 1"

donde A y B_i son átomos, engloban completamente la estructura y el funcionamiento de los programas PROLOG clásicos, con cláusulas de la forma "$A : -B_1, \ldots, B_n$".

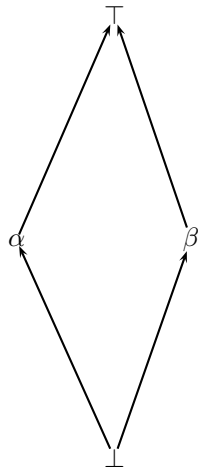
```
member(X) :- number(X),0=<X,X=<1.  %% no members/1 (infinite lattice)
bot(0).                top(1).              leq(X,Y) :- X=<Y.

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 4.6: Fichero "num.lat", que modela el retículo multi-adjunto $([0,1], \leq)$

El fichero "num.lat", mostrado en la Figura 4.25, sigue el estilo PROLOG descrito previamente. En él se ha modelado un retículo más flexible que el correspondiente a "bool.lat", dado que describe los grados de verdad del dominio infinito (nótese que esta condición impide la implementación completa del predicado "`members/1`") [0,1], permitiendo así utilizar los operadores de conjunción y disyunción dados por las tres lógicas difusas típicas descritas anteriormente (las lógicas de *Łukasiewicz*, *Gödel* y del *producto*), así como el agregador *average*.

También se han definido 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, de forma que las cláusulas que definen los conectivos hacen uso de estos predicados y logran un mayor nivel de expresividad (como es el caso de las conectivas difusas que estamos considerando: conjunciones, disyunciones y agregadores).

Hasta este punto hemos detallado los ficheros "num.lat" y "bool.lat", que definen dos retículos totalmente ordenados. Para cubrir completamente la capacidad de $\mathcal{FLOPER}$ de trabajar con distintos retículos multi-adjuntos, ejemplificamos un retículo dotado de un orden parcial con el retículo que aparece en la Figura 4.7 junto al fichero "four.lat", que lo define. Para este retículo la conjunción y la implicación corresponden a las de la lógica de *Gödel*, donde la conjunción de *Gödel* queda expresada como $\&_{\mathtt{G}}(x, y) \triangleq inf(x, y)$, debiendo observarse la sustitución de "min" por "inf".




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

leq(bottom,X). leq(alpha,alpha).
leq(beta,beta). leq(beta,top).
leq(alpha,top). leq(X,top).

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).
```

Figura 4.7: Retículo "four.lat"

El retículo de la Figura 4.7 ejemplifica la flexibilidad del sistema $\mathcal{FLOPER}$ para utilizar estas estructuras. En su implementación, obsérvese en el código PROLOG que le acompaña que se han introducido cinco cláusulas para definir el operador primitivo "`pri_inf/3`", que devuelve el ínfimo de dos elementos. Considérense los siguientes aspectos:

- Puesto que los grados de verdad α y β (o sus correspondientes representaciones como términos PROLOG, "`alpha`" y "`beta`") no son comparables, los objetivos

"`?- leq(alpha,beta).`" y "`?- leq(beta,alpha).`" siempre producen fallo.

- En cambio, un objetivo de la forma "`?- pri_inf(alpha,beta,X).`", o de la forma inversa "`?- pri_inf(beta,alpha,X).`", en lugar de fallar, devuelve el resultado correspondiente "`X=bottom`".

- La implementación del predicado "`pri_inf/3`" es necesaria en la definición dada de "`and_godel/3`".

Como ejemplo final representamos la llamada álgebra de Borel, ya que aúna las caractarerísticas más destacadas de los retículos vistos, correspondientes a los ficheros "bool.lat", "num.lat" y "four.lat". El álgebra de Borel se basa en la unión de intervalos (donde $\mathcal{B}([0,1])$ es la unión de subintervalos de $[0,1]$, véase [VGM02, MPS11]), y se describe del siguiente modo:

- Un elemento de este retículo es una lista de pares que representan intervalos.

- El supremo es el número real 1 (exactamente, el intervalo $[1,1]$), y el ínfimo es el número real 0 (el intervalo $[0,0]$).

- Una unión de intervalos, U, es menor o igual que otra unión de intervalos U' si para cada $I \in U$, existe otro intervalo $I' \in U'$, tal que $I \subseteq I'$.

El código PROLOG que implementa este retículo es el siguiente:

```
member([i(X,Y)])          :-    number(X),number(Y),X=<Y.
member([i(X,Y),i(Z,T)|U]) :-    number(X),number(Y),number(Z),
                                 X=<Y, Y < Z,member([i(Z,T)|U]).
bot([i(0,0)]).                  top([i(1,1)]).

leq([X1,X2|X],Y) :-             existsGreater(X1,Y),leq([X2|X],Y).
leq([X1],Y)      :-             existsGreater(X1,Y).

existsGreater(i(Xb,Xt),[i(Yb,Yt)|Y]) :-  Yb=<Xb, Xt=<Yt,!.
existsGreater(X,[_|Y])               :-   existsGreater(X,Y).
```

Figura 4.8: Retículo 'borel.lat'

Modificadores y variables lingüísticas en $\mathcal{FLOPER}$

Otra de las características más sobresalientes del sistema $\mathcal{FLOPER}$ consiste en su capacidad para manejar modificadores lingüísticos y variables lingüísticas. Un modificador lingüístico, o cerca semántica, equivale a un adverbio del lenguaje ordinario que matiza el uso del predicado difuso. Está asociado a una función de verdad, y se puede ver como un agregador unario tal que, aplicado sobre una expresión difusa, altera su grado de verdad final. Algunos ejemplos de modificadores bien conocidos son *muy* y *apenas*, donde el primero tiende a decrementar el grado de verdad, y el segundo, a incrementarlo. Dado que, según nuestra forma de introducir los modificadores lingüísticos, éstos no se diferencian de los agregadores, el lugar para su definición es el fichero PROLOG que contiene el propio retículo sobre el que se interpretan. Por ejemplo, en el retículo $([0,1],\leq)$ usado en ejemplos previos, definimos algunos modificadores a través de las siguientes expresiones de la función de verdad:

modificador	funcin	implementacin
extremo	$extre(x) = x^4$	agr_extre(X, TV0) :− TV0 is X ∗ X ∗ X ∗ X.
muy	$muy(x) = x^2$	agr_muy(X, TV0) :− TV0 is X ∗ X.
masmenos	$masme(x) = x^{1/2}$	agr_masme(X, TV0) :− TV0 is sqrt(X).
apenas	$apenas(x) = x^{1/4}$	agr_apenas(X, TV0) :− TV0 is sqrt(sqrt(X)).

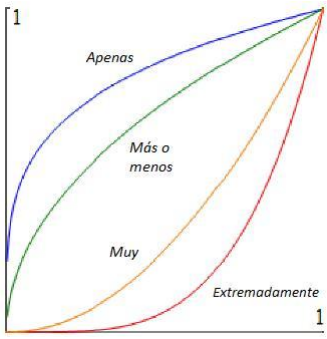


Figura 4.9: Modificadores lingüísticos

Los modificadores lingüísticos incrementan la expresividad del lenguaje difuso y, como dijimos anteriormente, deben ser definidos como parte del retículo multi-adjunto asociado al programa MALP en cuestión.

Por otro lado, las variables lingüísticas representan símbolos lingüísticos definidos por medio de conjuntos difusos. Una variable lingüística es caracterizada por una

tupla $\langle x, T, U, G, M\rangle$, donde x es el nombre de la variable, T el conjunto de símbolos lingüísticos de términos de x, U el universo donde se define x, G representa una regla sintáctica para generar términos lingüísticos, y M es una regla semántica para asignar a cada símbolo lingüístico t su conjunto difuso $M(t)$. Por ejemplo, cabe definir la variable lingüística *distancia*, con términos $\{cerca, lejos\}$, definida sobre el universo $[0, +\infty)$ en kilómetros, como recoge la Figura 4.10.

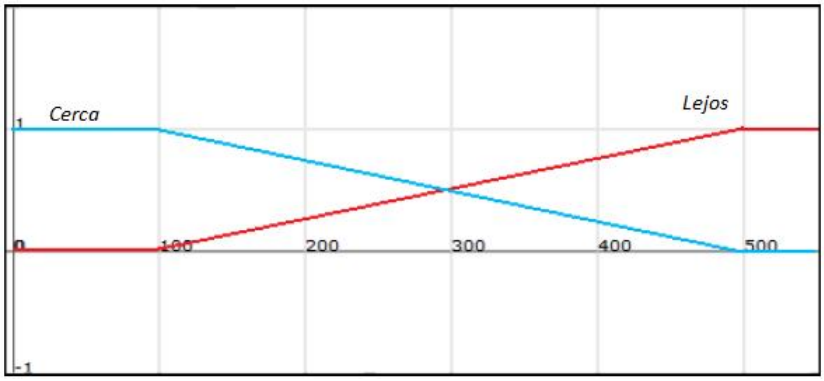


Figura 4.10: Variable lingüística "distancia" con los términos "cerca" y "lejos"

Se define esta variable en un fichero PROLOG donde definimos un predicado difuso por cada término. Aquí, los argumentos son una variable de entrada (`D`) para la distancia, y una variable de salida (`TV`) para el grado de pertenencia de la variable de entrada al conjunto difuso. En tanto que se cargue este programa PROLOG en $\mathcal{FLOPER}$, dichas definiciones pasarán a estar disponibles para el correspondiente programa MALP.

```
cerca(D,1) :- D=<100.
cerca(D,TV):- 100<D, D=<500, TV is (500-D)/400.
cerca(D,0) :- 500<D.

lejos(D,0) :- D=<100.
lejos(D,TV):- 100<D, D=<500, TV is (D-100)/400.
lejos(D,1) :- 500<D.
```

A continuación ofrecemos un ejemplo más complejo donde diseñamos una aplicación turística que computa el mejor destino de vacaciones. La base de datos incluye

algunas ciudades e información relevante relacionada con ellas (tipo de clima, buenas vistas, distancia desde nuestra ciudad, etc.). La elección de un lenguaje difuso para implementar esta aplicación se justifica por las características claramente difusas de parte de la información que maneja. El programa consta de doce hechos y una sola regla que introduce tanto variables como modificadores lingüísticos.

Utilizando variables lingüísticas "difuminamos" la distancia como sigue.

```
buen_tiempo(madrid) with 0.8.
buen_tiempo(estambul) with 0.7.
buen_tiempo(moscú) with 0.2.
buen_tiempo(sydney) with 0.5.

buenas_vistas(madrid) with 0.6.
buenas_vistas(estambul) with 0.7.
buenas_vistas(moscú) with 0.2.
buenas_vistas(sydney) with 0.6.

distancia_origen(madrid, 250).
distancia_origen(estambul, 3700).
distancia_origen(moscú, 4200).
distancia_origen(sydney, 18000).

buen_destino(X) <- @apenas(distancia_origen(X,D) & cerca(D))
                       @aver
                       @muy(buen_tiempo(X) & buenas_vistas(X)).
```

El significado asociado al uso que damos de los modificadores lingüísticos en la última regla MALP consiste en que le damos poca importancia a la distancia y, en cambio, estamos más interesados en el tiempo y las vistas.

La evaluación del objetivo `buen_destino(X)` en $\mathcal{FLOPER}$ para el programa previo produce los siguientes resultados (que indican que el mejor destino, de acuerdo a nuestra especificación, es Madrid, seguido de lejos por Estambul y Sidney, mientras Moscú tiene asociado el grado de verdad 0).

```
[Truth_degree=0.0,X=moscú]
[Truth_degree=0.005000000000000009,X=sydney]
[Truth_degree=0.07999999999999996,X=estambul]
```

```
[Truth_degree=0.5245698525097306,X=madrid]
```

Nótese que, dada la definición actual de los predicados difusos "X está cerca" y "X está lejos", si la distancia es superior a 500 kilómetros siempre tendrá grado de verdad 0 en cuanto a "cerca", y si está por debajo de 100 kilómetros, "lejos" tendrá grado de verdad 0. Ahora bien, estas nociones se vuelven poco útiles para discriminar destinos fuera de ese rango, produciendo el mismo grado de verdad para París que para Canberra. Resolvemos este problema mediante reglas no lineales para describir los términos "cerca" y "lejos" como las siguientes, que muestran la flexibilidad de $\mathcal{FLOPER}$ para tratar predicados definidos mediante expresiones aritméticas variadas.

```
cerca(D,TV) :- TV is 250/(D+250).
lejos(D,TV)  :- TV is 1 - 250/(D+250).
```

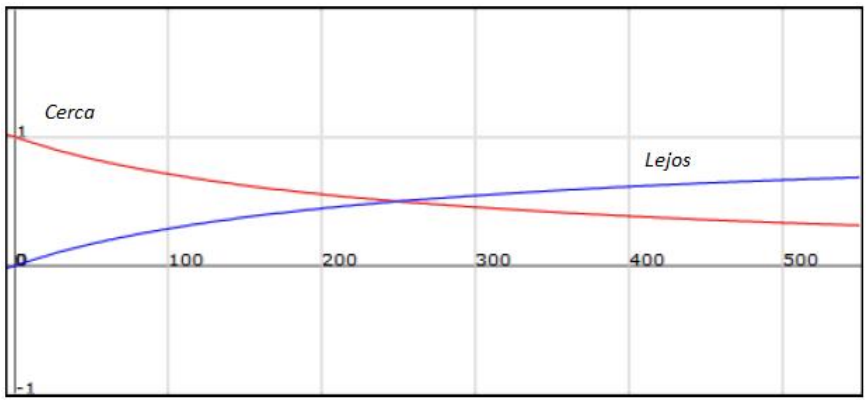


Figura 4.11: Variable lingüística 'distancia' con versiones flexibles de 'cerca' y 'lejos'

A modo de resumen, destacamos que la gestión e implementación de retículos multiadjuntos, modificadores y variables lingüísticas por parte de $\mathcal{FLOPER}$ (mediante ficheros ".lat") aporta una flexibilidad y potencia expresiva muy interesante a nuestra herramienta.

4.1.3. Traduccion del código difuso a código PROLOG

Dedicamos esta sección a detallar las dos estrategias empleadas en $\mathcal{FLOPER}$ para generar código difuso, cada una de ellas basadas en un modo de traducir dicho código

a código PROLOG puro. Estas técnicas producen resultados que difieren tanto en su modo de uso como en su significado. La técnica desarrollada en primer lugar (véase [JMP06c]) genera a partir de un programa MALP otro programa PROLOG independiente de la herramienta $\mathcal{FLOPER}$ (esto es, ejecutable en cualquier intérprete de PROLOG) cuya semántica es muy cercana a la del programa original. Esta técnica también exige la traducción del objetivo a evaluar. El resultado de este proceso es la evaluación aparente de un objetivo difuso sobre un programa difuso, obteniendo un conjunto de respuestas computadas difusas (pares formados por un grado de verdad y una sustitución) si bien los cálculos intermedios se realizan en un contexto lógico puro (no difuso). Esta técnica fue implementada en $\mathcal{FLOPER}$ desde su primera versión, como recogen los trabajos [AMM07a, AMM07b].

Concretamos a continuación los convenios sintácticos asumidos en esta primera forma de traducción.

- Cada **átomo** en el contexto de una regla difusa se traduce como un átomo PROLOG que comparte el mismo símbolo de predicado, si bien su aridad se incrementa en una unidad. Los parámetros del átomo resultante son los mismos que los del original, y en el mismo orden, más un argumento extra consistente en una variable (llamada "variable grado de verdad"), etiquetada como `_TVi`, y cuya función es unificar con el grado de verdad que le corresponde al átomo durante su evaluación.

- Cada **agregador** $@_i$ ($\&_i$ y $|_i$, respectivamente, para cada conjunción y disyunción) se traduce como un átomo cuyo símbolo de predicado, al que llamamos *predicado de agregación*, tiene la forma `agr_`i (`and_`i y `or_`i), y sus parámetros son las variables grado de verdad de los átomos que dicho agregador conecta, más una variable que unifica con el resultado de evaluar la correspondiente función de verdad.

- Con objeto de implementar el punto anterior, la **función de verdad** $\dot{@}_i$ ($\dot{\&}_i$ o $\dot{|}_i$) asociada a cada agregador $@_i$ (conjunción $\&_i$ o disyunción $|_i$), se traduce como una cláusula PROLOG de forma que el predicado de su cabeza coincide con el del punto anterior, y cuyo cuerpo es tal que la evaluación de `agr_i(`X_1`,...,`X_n`,TV)` (`and_i(`X_1`,...,`X_n`,TV)` o `or_i(`X_1`,...,`X_n`,TV)`) unifique la variable `TV` con x, si $@_i(X_1, \ldots, X_n) = x$ ($\&_i(X_1, \ldots, X_n) = x$ o $|_i(X_1, \ldots, X_n) = x$). A continuación ilustramos este procedimiento para las conjunciones del *Producto*, de *Gödel* y de *Łukasiewicz*.

```
and_prod(X,Y,Z)   :-  Z  is  X * Y.
and_godel(X,Y,Z)  :-  (X=<Y,Z=X;X>Y,Z=Y).
and_luka(X,Y,Z)   :-  H  is  X+Y-1,  (H=<0,Z=0;H>0,Z=H).
```

- Los **hechos del programa difuso** $A \leftarrow \nu$ (esto es, las reglas sin cuerpo) se traducen del mismo modo que los átomos, con la salvedad de que como último parámetro, en lugar de una variable, se incluye el grado de verdad ν, como se observa en el Ejemplo 4.1.1 que muestra un programa asociado al retículo $([0,1], \leq)$.

- Las **reglas de programa difuso** se traducen como cláusulas Prolog, sustituyendo cada átomo y cada agregador por su traducción, y uniendo los átomos resultantes por comas. Los átomos mantienen, una vez traducidos, el mismo orden que en la regla original, y los agregadores se ubican al final de la cláusula conservando también el orden entre ellos, a fin de que todas las variables grado de verdad se encuentren instanciadas en el momento de evaluarlos. Concretamente, se ubica en último lugar el predicado de agregación que modela la conjunción adjunta al operador implicación de la regla, haciendo uso también del grado de verdad de ésta. Por ejemplo, las tres primeras reglas del Ejemplo 4.1.1 pueden representarse por las cláusulas Prolog:

  $\texttt{p}(\texttt{X}, \texttt{\_TV0}) : -\texttt{q}(\texttt{X}, \texttt{Y}, \texttt{\_TV1}), \texttt{r}(\texttt{Y}, \texttt{\_TV2}),\ \texttt{and\_godel}(\texttt{\_TV1}, \texttt{\_TV2}, \texttt{\_TV3}),$
  $\texttt{and\_prod}(0.8, \texttt{\_TV3}, \texttt{\_TV0}).$

  $\texttt{q}(\texttt{a}, \texttt{Y}, \texttt{\_TV0}) : -\texttt{s}(\texttt{Y}, \texttt{\_TV1}),\ \texttt{and\_prod}(0.7, \texttt{\_TV1}, \texttt{\_TV0}).$

  $\texttt{q}(\texttt{b}, \texttt{Y}, \texttt{\_TV0}) : -\texttt{r}(\texttt{Y}, \texttt{\_TV1}),\ \texttt{and\_luka}(0.8, \texttt{\_TV1}, \texttt{\_TV0}).$

- Un **objetivo difuso** se traduce como un objetivo Prolog del mismo modo que el cuerpo de una regla, esto es, reemplazando cada átomo por su traducción, ubicándolos en la fórmula en su orden original de aparición, y reemplazando cada agregador por un predicado de agregación con los parámetros correspondientes. Por ejemplo, el objetivo `p(X) &godel r(a)` puede representarse por el siguiente objetivo Prolog:

  $? - \texttt{p}(\texttt{X}, \texttt{\_TV1}), \texttt{r}(\texttt{a}, \texttt{\_TV2}), \texttt{and\_godel}(\texttt{\_TV1}, \texttt{\_TV2}, \texttt{\_TV3}).$

Mediante este procedimiento, el sistema $\mathcal{FLOPER}$ traduce el programa lógico multiadjunto original y su objetivo a código Prolog estándar, que se puede ejecutar en cualquier intérprete de Prolog.

La segunda técnica referida, que detallamos en la siguiente sección, emulada la semántica operacional de MALP.Esta segunda técnica, si bien no es independiente de $\mathcal{FLOPER}$, conserva completamente la semántica del lenguaje original.

4.1.4. Ejecución de programas y árboles de desplegado

La técnica de traducción que acabamos de describir es útil para la resolución de objetivos y la ejecución de programas de forma transparente al usuario, pero resulta insuficiente para realizar acciones más sofisticadas. En concreto, obsérvese que, si bien este método permite simular derivaciones difusas completas (a través de las derivaciones PROLOG correspondientes basadas en resolución-SLD), no permite generar derivaciones parciales, así como tampoco aplicar un único paso admisible/interpretativo sobre una expresión difusa.

Este tipo de manipulaciones de bajo nivel son necesarias para generar derivaciones de un número fijo de pasos, modificar el cuerpo de una regla de programa, aplicar sustituciones en su cabeza, etc., con objeto de implementar determinadas técnicas de transformación para este tipo de programas. Un ejemplo de estos mecanismos lo constituye la transformación de desplegado difuso desarrollada en nuestro grupo [GM08a, JMP04, MM09d], que se define como la sustitución de una regla de programa $\mathcal{R} : \langle A \leftarrow_i \mathcal{B}; \alpha\rangle$ por el conjunto de reglas $\{\langle A\sigma \leftarrow_i \mathcal{B}'; \alpha\rangle \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS/IS} \langle \mathcal{B}'; \sigma\rangle\}$. Este mecanismo exige la generación de derivaciones de un solo paso, la modificación del cuerpo de una regla y la aplicación de sustituciones en su cabeza. Para lograrlo, seguimos la estrategia de emular a través de PROLOG el mecanismo operacional de MALP, para lo cual hemos desarrollado en nuestro grupo un método de representación del código difuso que la herramienta $\mathcal{FLOPER}$ puede manipular. Esta técnica fue presentada por primera vez en [MM08d] y se fue refinando en los trabajos [MM08b, MM08a, MM08c]. En el siguiente ejemplo, con objeto de ilustrar las diferencias entre el código difuso y las representaciones propuestas en la sección anterior y la presente, mostramos una cláusula MALP junto a sus dos versiones traducidas

Ejemplo 4.1.1. ***Código*** MALP ***original***

```
p(X) < prod q(X,Y)  &godel r(Y)  with 0.8.
```

Traducción a código PROLOG ***ejecutable, según la Sección 4.1.3 anterior***

```
p(X, _TV0) :- q(X, Y, _TV1), r(Y, _TV2), and_godel(_TV1, _TV2, _TV3),
                                          and_prod(0.8, _TV3, _TV0).
```

***Traducción a código* PROLOG *manipulable por* $\mathcal{FLOPER}$**

```
rule(number(1),
  head(atom(pred(p,1),var('X')])),
  impl(prod),
  body(and(godel,2,
          [atom(pred(q,2),
             [var('X'),var('Y')]),
          atom(pred(r,1),
             [var('Y')])])),
  td(0.8)).
```

Como se aprecia en este ejemplo, este procedimiento de traducción se basa en la creación de una estructura en forma de árbol, similar a un árbol semántico, donde queda representado cada elemento de la regla difusa.

Por cada regla MALP *se genera un hecho* PROLOG *que especifica uno a uno los elementos de dicha regla. Esta representación del código facilita su modificación, ya sea para realizar las operaciones más específicas como la aplicación de una sustitución, la aplicación de un paso admisible, etc., hasta las complejas, como la generación/visualización de un árbol de desplegado para un programa y objetivo dados, con una profundidad determinada por el usuario, como ilustra la Figura 4.12.*

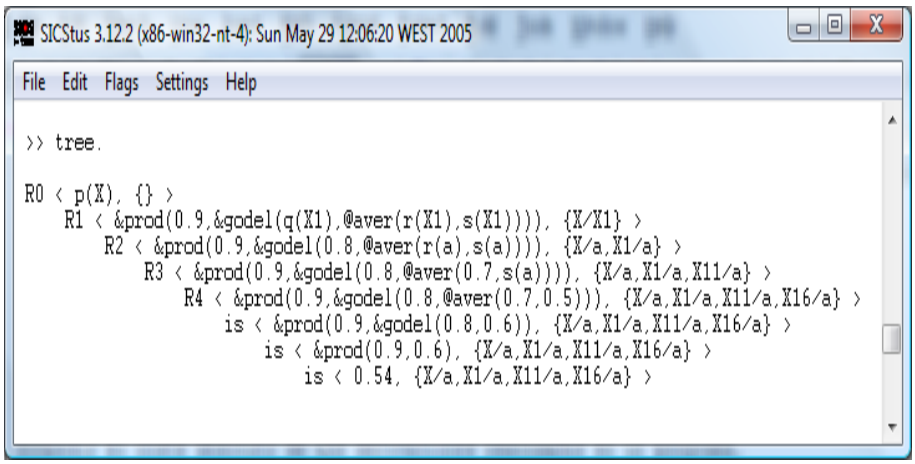


Figura 4.12: Generación de árbol de desplegado en $\mathcal{FLOPER}$

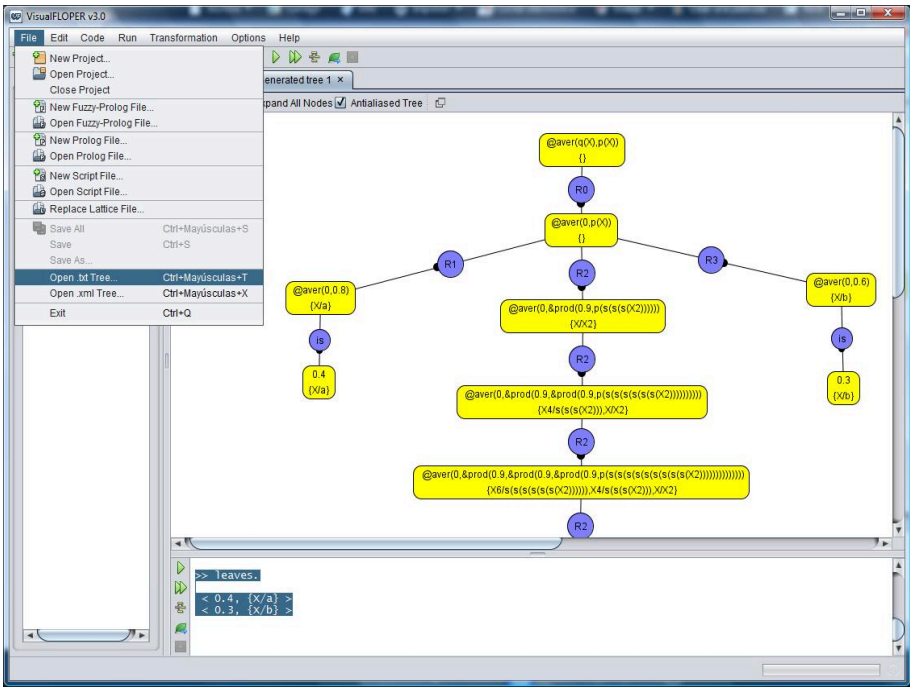


Figura 4.13: Árbol con una rama infinita y un paso $\rightarrow_{AS3}$

Ponemos fin a este bloque con un sencillo ejemplo que ilustra la mayor pertinencia de esta técnica, en lo que respecta a la semántica, que la de la presentada en la Sección anterior 4.1.3. Recordamos que esta última se basa en emular el código difuso mediante un programa PROLOG puro, que puede caer en un bucle o fallar directamente al buscar átomos no definidos, y que por ello existen situaciones en las que el programa PROLOG generado por la técnica de la sección anterior no encuentra todas las soluciones que sí localiza la presente técnica. Considérese el programa:

```
p(a)                          with 0.8.
p(X) <prod p(s(s(s(X))))      with 0.9.
p(b)                          with 0.6.
```

Se aprecia que el objetivo "`p(X)`" admite dos soluciones ($\langle 0.8, \{X/a\}\rangle$ y $\langle 0.6, \{X/b\}\rangle$, producidas al dar un paso admisible con la primera y la tercera regla, respectivamente). No obstante, al traducir el programa y el objetivo por la técnica presentada en la sección anterior, la segunda regla induce una rama infinita entre las hojas aso-

ciadas a las respuestas computadas difusas en el correspondiente árbol de ejecución. Otro problema emerge en la evaluación de un objetivo más complejo como "`q(X) @aver p(X)`", consistente en el fallo del programa al no estar definido el predicado `q/1`, cuando, en un contexto MALP, esto conlleva la sustitución del átomo `q(X)` por 0 a través de un paso $\rightarrow_{AS3}$ de acuerdo con la Definición 3.2.1 y, finalmente, la interpretación de la fórmula $\mathcal{I}(p(X))\dot{@}_a ver0$, produciéndose dos soluciones. Mostramos a continuación el código del árbol de ejecución tal y como se muestra desde la interfaz de texto de $\mathcal{FLOPER}$, junto a una imagen del mismo árbol desde la interfaz gráfica[1] del programa. En ambas representaciones del árbol se aprecian las dos respuestas correctas 0.3 y 0.4 cuando X cambia por a y cuando X cambia por b, respectivamente.

```
R0 < @aver(q(X),p(X)), {} >
    R0 < @aver(0,p(X)), {} >
        R1 < @aver(0,0.8), {X/a} >
            is < 0.4, {X/a} >
        R2 < @aver(0,&prod(0.9,p(s(s(s(X2))))))), {X/X2} >
            R2 < @aver(0,&prod(0.9,&prod(0.9,p(s(s(s(..... >
                R2 < @aver(0,&prod(0.9,&prod (0.9,&prod(0.9,p(s...>
        R3 < @aver(0,0.6), {X/b} >
            is < 0.3, {X/b}} >
```

Como ilustra el ejemplo, la capacidad de $\mathcal{FLOPER}$ de generar trazas basadas en árboles de ejecución provee a la herramienta de capacidad de depuración. Esta capacidad permite al usuario descubrir soluciones para consultas incluso cuando el proceso de compilación y ejecución en PROLOG puro explicado en la sección previa resulta insuficiente.

4.2. Interfaz gráfica, proyectos y "scripts" en Visual-$\mathcal{FLOPER}$.

Una interfaz gráfica que permita la interacción con el usuario de una forma sencilla e intuitiva es una característica muy recomendable y casi obligada para cualquier aplicación informática en la actualidad. Hemos puesto muchos esfuerzos en nuestro

[1] Explicamos con mayor detenimiento la interfaz gráfica de $\mathcal{FLOPER}$ en la Sección 4.2.

grupo en los últimos años por dotar a $\mathcal{FLOPER}$ de una interfaz de esta clase que facilite su uso a una comunidad más amplia de usuarios en diferentes entornos (académico, empresarial, etc.).

El resultado de estos esfuerzos es Visual$\mathcal{FLOPER}$, que dota a $\mathcal{FLOPER}$ de una interfaz gráfica que permite llevar a cabo, de forma intuitiva, las mismas operaciones, y añade nuevas características y funcionalidades muy útiles de cara al usuario, y que detallamos a lo largo de esta sección.

4.2.1. Características generales de Visual$\mathcal{FLOPER}$

La interfaz gráfica ha sido implementada utilizando el lenguaje JAVA, e integrada perfectamente con el núcleo de la herramienta, que sigue siendo código PROLOG puro. Los procesos descritos en la Sección 4.1, entre los que se cuentan cargar un programa, analizarlo y ejecutarlo, siguen realizándose internamente igual que en la herramienta de programación original $\mathcal{FLOPER}$, ya que Visual$\mathcal{FLOPER}$ constituye un programa separado que, además, introduce nuevas características.

La Figura 4.14 ilustra la interfaz Visual$\mathcal{FLOPER}$ tal y como aparece al invocar la aplicación, esto es, antes de cargar ningún proyecto. Hemos marcado en rojo las diferentes partes de la interfaz para facilitar su descripción. En el recuadro denotado como **1** se ubican todas las acciones que se pueden realizar desde el programa, ordenadas por menús que detallamos a continuación, acompañadas de una barra en la que se disponen numerosos atajos visuales a las acciones de uso más frecuente. En el área **2** destacamos el árbol de ficheros cargados en el programa, englobados todos ellos en sus correspondientes proyectos (detallamos el concepto de proyecto más adelante).

Esta interfaz gráfica de $\mathcal{FLOPER}$ maneja explícitamente la noción de proyecto, que puede definirse como todos aquellos ficheros empleados por $\mathcal{FLOPER}$ en una misma sesión de trabajo. Este es un concepto propio de Visual$\mathcal{FLOPER}$ pues, como sucede con los scripts, no tiene correspondencia con la interfaz primitiva del entorno. Cada proyecto contiene un directorio para los programas lógicos difusos (con extensión ".fpl") y otro para los programas lógicos PROLOG (con extensión ".pl"), además de un directorio para almacenar los ficheros de script y, al mismo nivel de la raíz, un único fichero para el retículo multi-adjunto (con extensión ".lat") asociado al programa.

En el área **3** se visualizan los programas, retículos y árboles de desplegado en el primer caso; y la salida de $\mathcal{FLOPER}$ (que incluye las operaciones que se ejecutan y

sus resultados), en el área **4**.

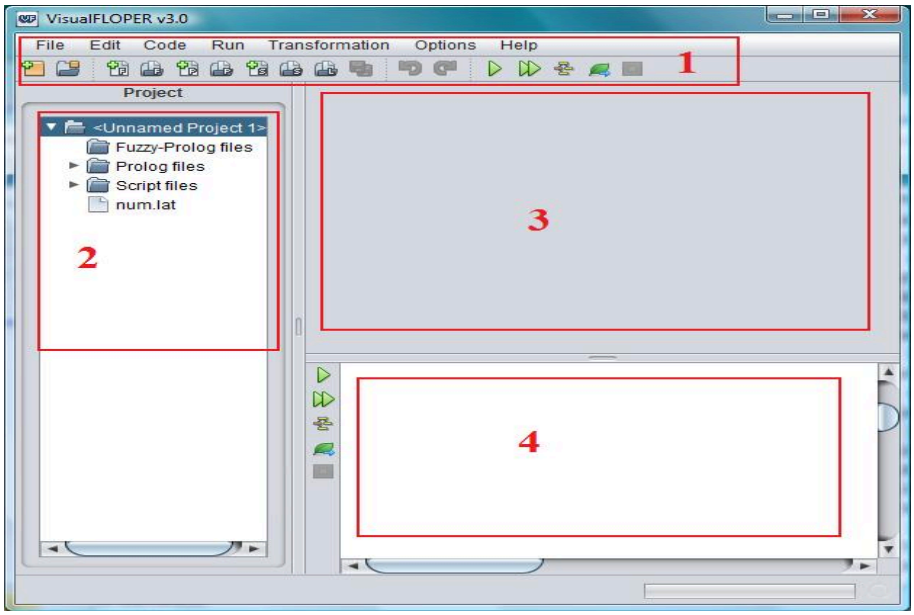


Figura 4.14: Interfaz gráfica de $\mathcal{FLOPER}$: Visual$\mathcal{FLOPER}$

Repasamos a continuación los distintos menús que ofrece Visual$\mathcal{FLOPER}$, e indicamos en cada uno las funcionalidades que ofrecen y sus equivalencias con las opciones disponibles en $\mathcal{FLOPER}$.

- **Menú "File"**: incorpora las principales acciones dedicadas a la gestión de proyectos. Como explicamos más detenidamente en una sección posterior, tanto el programa multi-adjunto usado como el retículo donde se interpreta deben estar contenidos en el mismo proyecto. Este menú permite crear un nuevo proyecto o abrir uno existente (con extensión ".vfp"), así como crear o añadir en su contexto un fichero lógico difuso (".fpl"), un fichero Prolog (".pl") para añadir código Prolog puro, o un script (".vps"), así como reemplazar el fichero que contiene el retículo donde se ejecutan los programas (por defecto se carga el retículo "num.lat", expuesto en la Figura 4.25). En la siguiente imagen mostramos el menú "File" donde se listan, entre otras, las opciones que acabamos de explicar.

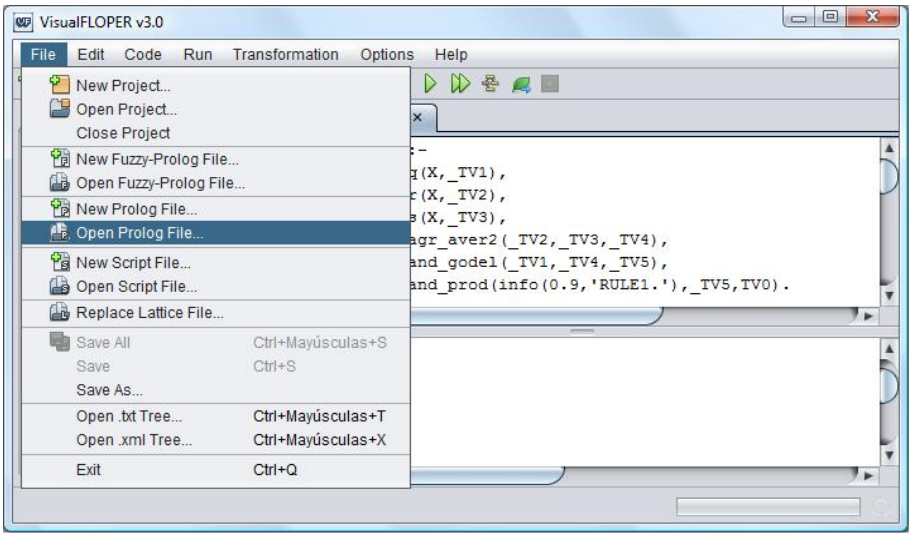


Figura 4.15: Menú "File" en Visual$\mathcal{FLOPER}$

- **Menú "Code"**: este menú incluye opciones para listar y analizar programas difusos (opciones "List" y "Parse" del menú de $\mathcal{FLOPER}$). A través de "List High Level Prolog Code" el programa muestra las cláusulas de los ficheros Prolog ".pl" incluidos en el proyecto, junto con las cláusulas Prolog generadas en el análisis de un fichero ".fpl" según la Sección 4.1.3. Otras opciones de "Code" permiten guardar dichas cláusulas en otros ficheros (como hace "Save" en $\mathcal{FLOPER}$) y mostrarlos en el área de visualización de programas de la interfaz. Por último, la opción "Show lattice" muestra el código del retículo multi-adjunto cargado, del mismo modo que "Show" en el menú de retículos de $\mathcal{FLOPER}$.

- **Menú "Run"**: incluye opciones relativas a la ejecución de objetivos difusos. La opción "Execute goal" realiza conjuntamente las funciones "Intro" y "Run" de $\mathcal{FLOPER}$, esto es, pide al usuario la introducción de un objetivo y lo ejecuta con respecto al programa previamente cargado. En caso de que el programa difuso no hubiera sido analizado con anterioridad, esta opción invoca al analizador de manera transparente al usuario.

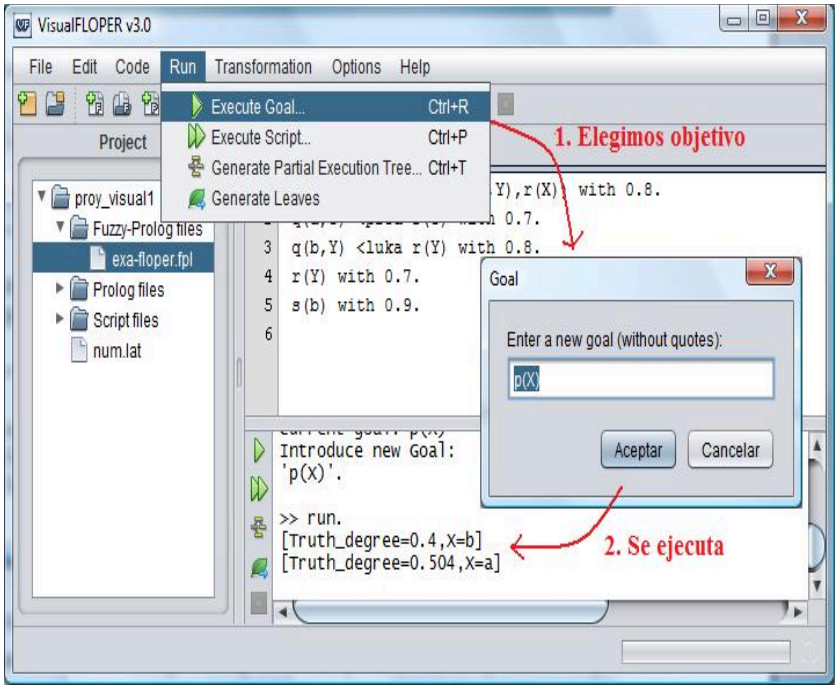


Figura 4.16: Menú "Run" en Visual$\mathcal{FLOPER}$

La opción "*Generate Partial Execution Tree*" genera un árbol de trazas de ejecución. Estos árbols son una de las características gráficas más llamativas del entorno por su alto nivel de expresividad (la Figura 4.13 ilustra un ejemplo de árbol de desplegado en este entorno gráfico). Estos árboles son la contraparte gráfica de los generados en $\mathcal{FLOPER}$ con la opción "tree", que tienen un aspecto menos atractivo. Explicamos la opción "Execute scripts" con más detenimiento en el apartado siguiente, ya que es una novedad sin contrapartida en la versión clásica de $\mathcal{FLOPER}$. La Figura 4.16 muestra cómo se ejecuta un objetivo en este entorno.

- **Menú "Options"**: este menú aglutina opciones más heterogéneas de la herramienta. La opción "*Depth*" permite al usuario determinar la profundidad máxima de los árboles de derivación (obtenidos por la opción "*Tree*") que calcula el sistema, mientras que con "*Ismode*" se determina el tipo de pasos interpretativos a realizar para generar dicha derivación, como hacen los comandos

homónimos de $\mathcal{FLOPER}$. Permitimos elegir entre tres tipos distintos de pasos: los llamados "pasos medios", que corresponden al concepto de paso interpretativo visto en la Sección 3.2.2; los "pasos largos", que en una sola etapa resuelven toda una fórmula; y los "pasos cortos", que se explicarán minuciosamente en la Sección 3.5 de esta memoria.

- El menú **"Transformation"** recoge las mismas opciones que su homónimo de la versión de texto de $\mathcal{FLOPER}$. Las opciones relativas a la evaluación parcial de programas, transformaciones basadas en plegado y desplegado así como cálculo de reductantes, se encuentran aún en proceso de implementación y todavía no están disponibles en nuestra herramienta.

- Para finalizar, los menús **"Edit"** y **"Help"** son los clásicos de cualquier aplicación con interfaz gráfica en el sentido de que ofrecen opciones textuales, como copiar y pegar en el primer caso, e información sobre la versión de la aplicación y un manual de usuario (en fase de construcción) en el segundo.

Obsérvese que se conservan todas las opciones de la versión clásica de $\mathcal{FLOPER}$, si bien se presentan con un aspecto más intuitivo y fácil de manejar para el usuario. Destacamos entre ellas la nueva representación de los árboles de desplegado, que resulta mucho más visual. Además de las funcionalidades originales, la interfaz gráfica incluye nuevas características, como la noción de proyecto y de script, así como una simplificación de la sintaxis, que detallamos a continuación.

Respecto a la sintaxis, con objeto de facilitar la escritura de programas lógico difusos, la herramienta ofrece las siguientes facilidades:

- El peso asociado a una regla puede ser omitido si coincide con el elemento $\top$ del retículo correspondiente. En estos casos, además, deja de ser necesario etiquetar la implicación, puesto que no tiene efecto a nivel operacional en tanto que toda conjunción adjunta & cumple que &(1,x)=x.

- Si no se etiqueta alguna implicación o conectiva, el sistema selecciona por defecto una definición arbitraria de dicha implicación o conectiva (remitimos de nuevo a la Figura 4.25, donde se definen los distintos operadores).

- El sistema permite incluir cláusulas PROLOG entre los símbolos "`$$`", como ocurre con "`$p(X):-!,var(X).$`". También permite insertar código PROLOG puro entre símbolos "`{}`" dentro del cuerpo de una regla lógico difusa MALP, como ilustra la regla "`p([H|T]) <- {Y is H+1} q(Y) & p(T).`".

La noción de proyecto es central en la versión gráfica de $\mathcal{FLOPER}$. En este contexto denominamos *proyecto* al conjunto formado por los ficheros que describen un programa MALP, cláusulas PROLOG, scripts y un único retículo multi-adjunto. Agrupamos varios ficheros difusos (o bien PROLOG, o de script) en un mismo proyecto, lo que resulta en un código más ordenado. Describimos a continuación el propósito de cada tipo de fichero dentro de un proyecto:

- el tipo asociado a la extensión ".fpl" representa ficheros que contienen reglas de programa. Si bien en la versión de $\mathcal{FLOPER}$ basada en texto un programa estaba definido exclusivamente por un único fichero ".fpl", en Visual$\mathcal{FLOPER}$ todos los ficheros con esta extensión se toman como partes del mismo programa difuso, lo que permite un código más modular,
- un fichero con extensión ".lat", que en realidad contiene un programa PROLOG que modela el (único) retículo (en principio, multi-adjunto, aunque no necesariamente) asociado al proyecto,
- varios ficheros adicionales ".pl", que incluyen código PROLOG útil para representar modificadores, variables lingüísticas y predicados auxiliares,
- los ficheros con extensión ".sim" representan ecuaciones de similaridad, utilizadas por el sistema para recrear, mediante los cierres reflexivo, simétrico y transitivo, una relación de similaridad completa (esto se verá más detenidamente en la Sección 6.5), y
- los ficheros "script", con extensión ".vfs", que permiten ejecutar varios comandos en $\mathcal{FLOPER}$ seguidos con una sola llamada.

Un nuevo proyecto en blanco se crea a partir de la opción "*New Project*" del menú "File". Sobre un proyecto siempre es posible añadir (y quitar) los ficheros necesarios. En el caso del retículo multi-adjunto, sólo un fichero, y no varios, puede formar parte del proyecto. Al guardar dicho proyecto se genera un fichero con extensión ".vfp" que describe la ubicación de todos sus ficheros, guardados, a su vez, conservando la estructura del proyecto en tres directorios distintos (uno para los ficheros difusos, otro para los programas PROLOG y otro para los scripts, permaneciendo el retículo en la raíz del proyecto junto al fichero ".vfp". La Figura 4.17 ilustra, en primer plano, un diálogo para elegir el proyecto a abrir, y en segundo, la estructura del proyecto con sus distintos ficheros una vez cargado en el sistema.

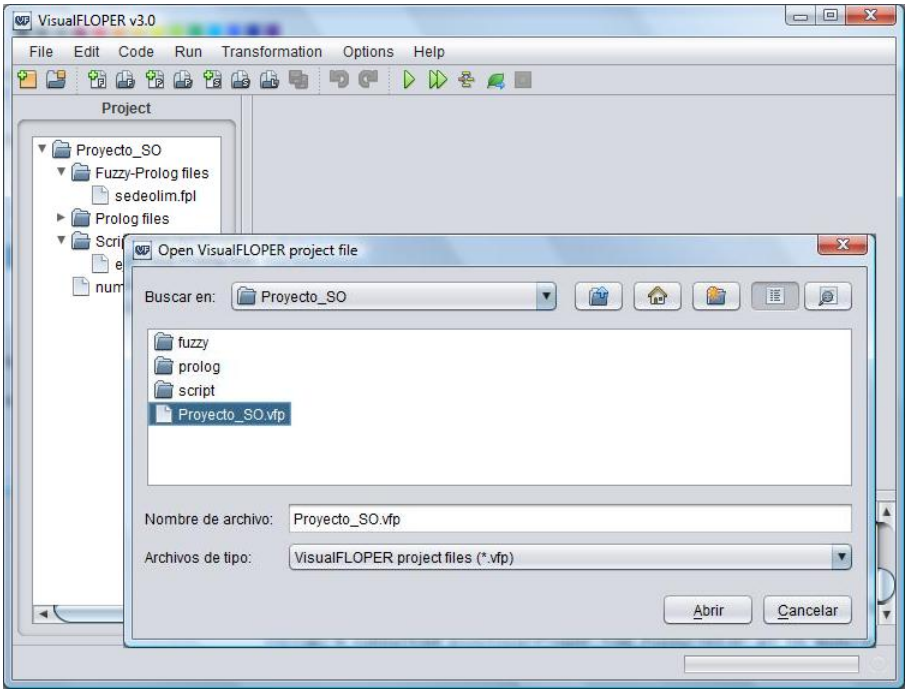


Figura 4.17: Abriendo un proyecto mediante la opción 'Open Project'

La segunda novedad que aporta Visual$\mathcal{FLOPER}$ con respecto a la herramienta original es la posibilidad de ejecutar scripts. Un script es un fichero de texto plano con extensión ".vfs") que contiene una serie de comandos $\mathcal{FLOPER}$ en el orden que se desea que se ejecuten una vez invocado. Cada línea del fichero de script puede ser una orden (en cuyo caso empieza por `ord`) o un argumento para la orden anterior (en ese caso empieza por `arg`). Sirva de ejemplo la siguiente secuencia de órdenes:

```
ord: intro.
arg: p(X).
ord: depht.
arg: 4.
ord: tree.
ord: run.
```

Al ejecutar este script, la aplicación introduce el objetivo `p(X)` en el sistema, establece en 4 la profundidad máxima de los árboles de ejecución y crea uno para dicho

objetivo y el programa cargado para, finalmente, ejecutarlo mediante la opción run, por este orden.

4.2.2. Un ejemplo completo usando Visual$\mathcal{FLOPER}$

Cerramos esta sección explicando en detalle el procedimiento para ejecutar un objetivo con respecto a un programa lógico multi-adjunto en nuestra herramienta, Visual$\mathcal{FLOPER}$, y explorar otras características que ofrece este entorno gráfico de programación.

Para ello nos valemos de un ejemplo que, además, tiene el propósito de acercar la programación lógica difusa a problemas del mundo real, en tanto que pretende deducir cuál de las ciudades descritas reúne las mejores condiciones para ser sede olímpica. Este programa lógico multi-adjunto, guardado en el fichero ``sedeolim.fpl'', consiste en las siguientes reglas:

```
so(X) <prod s(X) &luka (i(X) |godel t(X)) with 1.

s(mad)   with 0.9.
i(mad)   with 0.9.
t(mad)   with 0.7.

s(est)   with 0.6.
i(est)   with 0.6.
t(est)   with 0.5.

s(tok)   with 0.9.
i(tok)   with 0.6.
t(tok)   with 0.8.
```

El significado de los predicados difusos es el siguiente: `s` indica el nivel de seguridad que ofrece la ciudad en cuestión, `i` la calidad de las infraestructuras deportivas, `t` la calidad de los transportes y `so` indica la idoneidad de la ciudad como sede olímpica. Las ciudades candidatas propuestas son Madrid, Estambul y Tokio, y el comité de expertos ha valorado los diferentes parámetros de cada ciudad con un valor entre 0 y 1, como se muestran los nueve hechos del programa.

El primer paso consiste en crear un nuevo proyecto vacío llamado (por ejemplo) "Proyecto_SO", y añadir un fichero multi-adjunto. Empleamos el retículo por defec-

to "num.lat", que modela el intervalo $[0, 1]$, cuya implementación detallamos en la Sección 4.1.2, y su código está expuesto en la Figura 4.25. Recordemos que dicho retículo incluye las siguientes conectivas basadas en diferentes lógicas difusas:

```
&prod(x,y) = x * y
&godel(x,y) = min{x,y}
&luka(x,y) = max{(x+y)-1,0}

|prod(x,y)  = (x+y) - (x*y)
|godel(x,y) = max{x,y}
|luka(x,y)  = min{x+y,1}
```

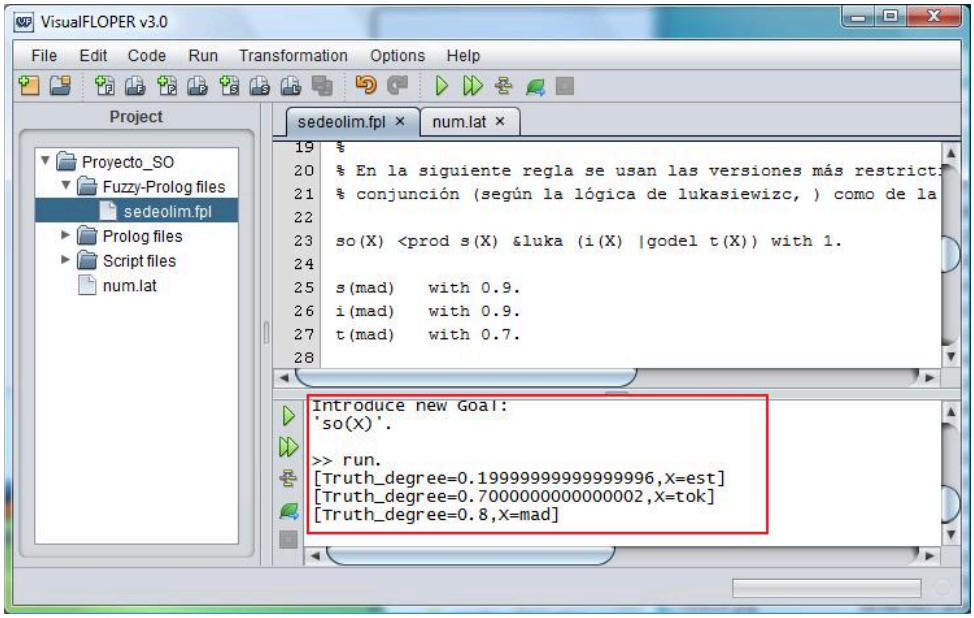


Figura 4.18: Ejecución del objetivo `so(X)` en Visual$\mathcal{FLOPER}$

Atendiendo a los resultados que darían estas conectivas ante diferentes valores de X e Y, podemos afirmar que la lógica del producto es realista en ambos casos (para la conjunción y para la disyunción); la lógica de Gödel es optimista para la conjunción y pesimista para la disyunción, y justo el caso contrario ocurre para la lógica de Łukasiewicz. Cuando hablamos de lógicas "optimistas", "realistas" o "pesimistas" nos

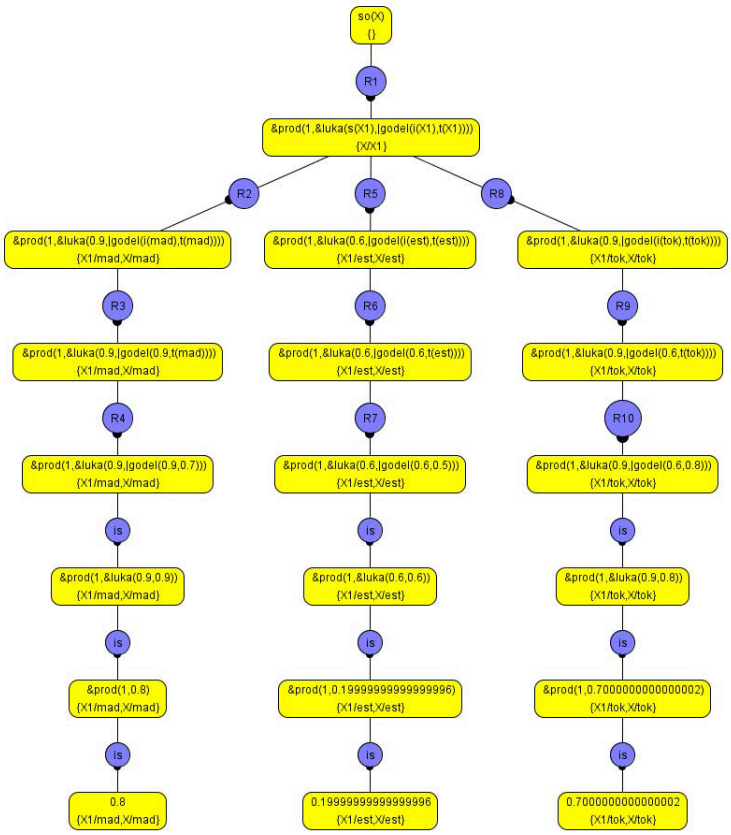


Figura 4.19: Archivo `.png` que contiene el árbol de ejecución para el objetivo `so(X)`

referimos a que devuelven valores mayores, medios y menores (respectivamente) para los mismos parámetros.

$$\&_{luka}(x,y) \leq \&_{prod}(x,y) \leq \&_{godel}(x,y) \leq |_{godel}(x,y) \leq |_{prod}(x,y) \leq |_{luka}(x,y)$$

Esta es una muestra más de la flexibilidad y potencia del lenguaje MALP para ofrecer al usuario la posibilidad de elegir entre distintas opciones para modelar de la manera más precisa posible determinados problemas del mundo real. En la definición del predicado `so` (sede olímpica) en el programa de ejemplo hemos usado las versiones más restrictivas (o pesimistas) para el operador conjunción y el operador disyunción (la conjunción de Łukasiewicz y la disyunción de Gödel, respectivamente).

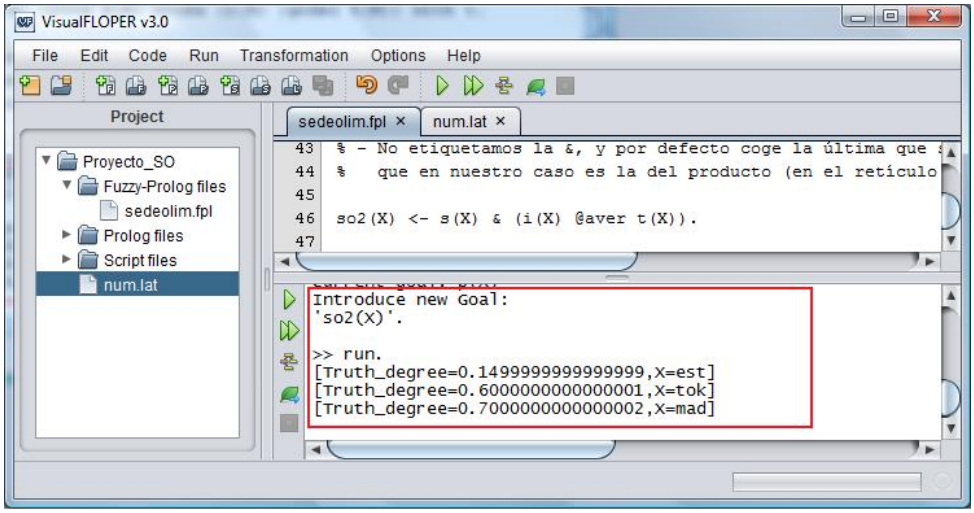


Figura 4.20: Ejecución del objetivo `so2(X)` en Visual$\mathcal{FLOPER}$

Mediante la opción "run" introducimos el objetivo '**so(X)**', y el sistema devuelve los resultados que se muestran en la Figura 4.18. En ellos se observa que, dadas las reglas y datos establecidos, Madrid es la ciudad que ofrece mejores prestaciones, con un grado de verdad 0.8 (esto es, un grado de confianza del 80 %). La segunda mejor ciudad, Tokio, es valorada con un 70 %. Muy lejos de sus competidores aparece Estambul, con un 19 %.

La opción "tree" muestra el árbol de derivación generado por Visual$\mathcal{FLOPER}$. Estos árboles se pueden manipular con facilidad, ya sea arrastrando los nodos y arcos para reorganizar su visualización, o eligiendo mostrar sólo el esqueleto (la forma) del árbol y ocultar el contenido de los nodos. La herramienta permite exportar el árbol en tres formatos distintos: como imagen (extensión `.png`), texto plano (extensión `.txt`) o fichero xml (extensión `.xml`). En el primer caso la imagen guardada corresponde a la mostrada por la herramienta, como la reflejada en la Figura 4.19.

Añadamos ahora una nueva regla, `so2`, para ilustrar las consecuencias de usar diferentes conectivas en el cuerpo de las reglas.

```
so2(X) <- s(X) & (i(X) @aver t(X)).
```

La evaluación del objetivo `so2(X)` sobre el mismo programa, produce los resultados reflejados en la Figura 4.20. Madrid se muestra, nuevamente, como la mejor candi-

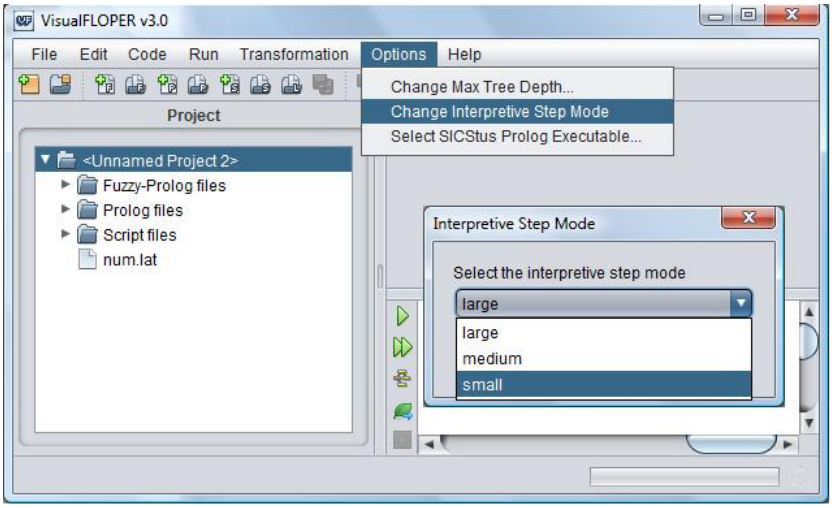


Figura 4.21: Opción **ismode** en Visual$\mathcal{FLOPER}$

data, si bien con menor nivel de confianza y ligeramente más cerca de las otras dos posibles sedes. En concreto, Madrid exhibe un grado de verdad de 0.7, seguido por el grado de verdad 0.6 de Tokio y, a mayor distancia, Estambul con un 0.14 de nivel de confianza.

Pese a su sencillez, este programa ilustra claramente las posibilidades que ofrece el lenguaje multi-adjunto para modelar problemas más complejos. Nuestra herramienta de programación, $\mathcal{FLOPER}$, mejorada con la interfaz gráfica Visual$\mathcal{FLOPER}$ descrita en esta sección, pretende convertirse en un excelente y flexible entorno para ejecutar, depurar y tratar este tipo de programas.

4.3. Implementación en Visual$\mathcal{FLOPER}$ de la fase interpretativa

En esta sección detallamos la implementación de algunas de las técnicas descritas en el Capítulo 3 en el entorno de programación $\mathcal{FLOPER}$. Estas ampliaciones y mejoras de nuestra herramienta de programación fueron presentadas en los trabajos [MMPV10a, MMPV10c]. Tratamos, en primer lugar, la implementación del mecanismo operacional basado en los pasos interpretativos cortos que, como se recoge en

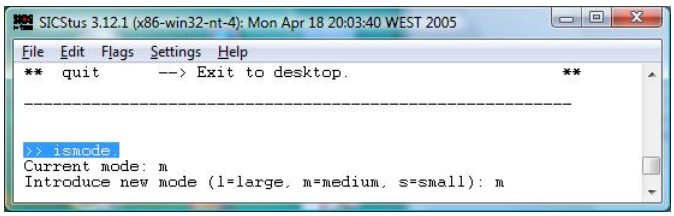


Figura 4.22: Opción IsMode en $\mathcal{FLOPER}$

la Sección 3.5, es una reformulación de la fase interpretativa que evalúa explícitamente todas las operaciones relacionadas con las conectivas y operadores primitivos durante una ejecución. La segunda técnica cuya implementación detallamos en este capítulo, que está íntimamente relacionada con la primera, es la expuesta en la Sección 3.6, y consiste en la incorporación de información relativa a la evaluación de un objetivo en las respuestas computadas correspondientes a dicha evaluación mediante el uso de retículos apropiados.

Para dar cabida a estas nuevas características se ha expandido la funcionalidad de $\mathcal{FLOPER}$ con tres nuevas opciones que detallamos a continuación, y que ya han sido mencionadas en la Sección 4.1.1.

4.3.1. Opción Ismode

La opción **ismode** ofrece tres alternativas para realizar la fase interpretativa, cada una de ellas produciendo un efecto diferente en el árbol de ejecución que se muestra con **tree**. Mediante esta opción es posible visualizar los pasos interpretativos cortos definidos teóricamente en la Sección 3.5 en las derivaciones. Esta alternativa es la que con mayor precisión refleja las operaciones realizadas durante la fase interpretativa. Las otras dos posibilidades abstraen cada paso interpretativo –a través del uso de pasos interpretativos "clásicos"– o, incluso, la fase interpretativa al completo. Ilustramos estas características en las Figuras 4.22 y 4.21, tanto en la versión de texto de $\mathcal{FLOPER}$ como a través de su interfaz gráfica, respectivamente.

A continuación proponemos un árbol de desplegado, tanto en la interfaz de texto como en la gráfica, para ejemplificar las diferentes posibilidades que proporciona la opción **ismode**. Dicho árbol corresponde a la ejecución del programa del Ejemplo 4.1.1, tomando `p(X)` como objetivo.

Modo 'large'. La elección de este modo de ejecución produce la omisión de toda

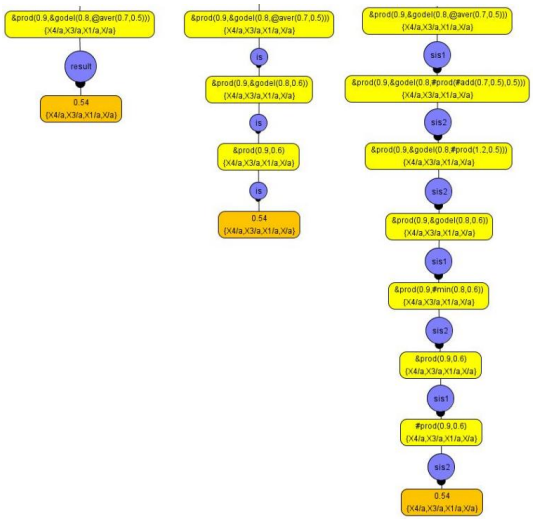


Figura 4.23: Opciones **large**, **medium** y **small** en la interfaz gráfica de $\mathcal{FLOPER}$

la fase interpretativa en el árbol, a excepción de las respuestas computadas difusas que pueda haber en el mismo. Como se aprecia en traza siguiente, tras cuatro pasos admisibles –con las reglas 1, 2, 3 y 4– se presenta directamente la f.c.a con la etiqueta `result` sin mostrar la fase interpretativa.

```
>> 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,X3/a} >
        R4 < &prod(0.9,&godel(0.8,@aver(0.7,0.5))), {X/a,...,X4/a} >
          result < 0.54, {X/a,X1/a,X3/a,X4/a} >
```

Modo 'medium'. Este modo produce la generación de derivaciones interpretativas contemplando la definición "clásica" de los pasos interpretativos dada en la Definición 3.2.6.

```
>> 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,...,X3/a} >
        R4 < &prod(0.9,&godel(0.8,@aver(0.7,0.5))), {X/a,...,X4/a}>
          is < &prod(0.9,&godel(0.8,0.6)), {X/a,X1/a,X3/a,X4/a} >
            is < &prod(0.9,0.6), {X/a,X1/a,X3/a,X4/a} >
              is < 0.54, {X/a,X1/a,X3/a,X4/a} >
```

Modo 'small'. Este modo produce, de manera equivalente al anterior, la generación de derivaciones interpretativas contemplando la definición (en este caso) novedosa los denominados "pasos interpretativos cortos" dados por la Definición 3.5.1.

```
>> tree.
R0 < p(X), {} >
 ....
   sis1 < &prod(0.9,&godel(0.8,#prod(#add(0.7,0.5),0.5))),{X/a,...}>
     sis2 < &prod(0.9,&godel(0.8,#prod(1.2,0.5))), {X/a,...,X4/a} >
       sis2 < &prod(0.9,&godel(0.8,0.6)), {X/a,X1/a,X3/a,X4/a} >
         sis1 < &prod(0.9,#min(0.8,0.6)), {X/a,X1/a,X3/a,X4/a} >
           sis2 < &prod(0.9,0.6), {X/a,X1/a,X3/a,X4/a} >
             sis1 < #prod(0.9,0.6), {X/a,X1/a,X3/a,X4/a} >
               sis2 < 0.54, {X/a,X1/a,X3/a,X4/a} >
```

Considérese la Figura 4.23 donde se ilustra la derivación interpretativa proporcionada por la interfaz gráfica de $\mathcal{FLOPER}$ en cada uno de los tres modos ofrecidos por la opción **ismode** (esto es, omitiendo la fase interpretativa, realizando pasos interpretativos "clásicos", o pasos interpretativos cortos).

4.3.2. Opciones Lat y Show

Las opciones **lat** y **show** están estrechamente relacionadas con la gestión de los retículos multi-adjuntos. La primera de ellas permite elegir el retículo en el que interpretar un programa y un objetivo dados, lo cual tiene un efecto decisivo en la semántica del programa.

Recuérdese que, como comentamos de manera sucinta en la Sección 4.1.1 de esta memoria, el sistema $\mathcal{FLOPER}$ está diseñado para considerar un único retículo cada vez. Dicho retículo es introducido por el usuario mediante la opción **lat**, salvo el retículo usual $([0,1], \leq)$ implementado en el fichero `num.lat`, que es el retículo por defecto. Elegir un nuevo retículo desasigna el retículo anterior, de manera que siempre hay un único retículo cargado.

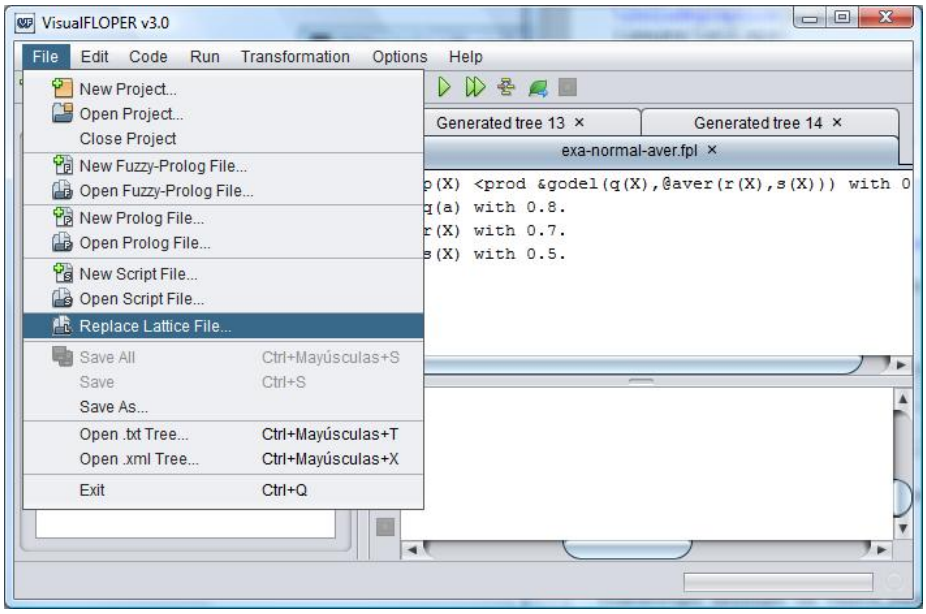


Figura 4.24: Opción **lat** en Visual$\mathcal{FLOPER}$

Como se adelanta en la Sección 3.6, definidos los retículos multi-adjuntos mediante ficheros PROLOG. La opción 'lat' pide al usuario la ubicación del fichero donde está definido el retículo correspondiente:

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

En la interfaz gráfica de Visual$\mathcal{FLOPER}$ la opción **lat** se encuentra en el menú **file**, como ilustra la Figura 4.24.
Una vez indicada la ubicación del fichero, $\mathcal{FLOPER}$ almacena el retículo como un conjunto de cláusulas PROLOG alojadas en el módulo **lat**. De este modo, un cambio de retículo conlleva la eliminación del retículo anterior mediante el borrado sistemático del módulo **lat**, antes de cargar en éste el programa PROLOG correspondiente al nuevo retículo. Durante la carga de un retículo, el sistema comprueba la presencia de los predicados obligatorios (`member`, `bot`, `top` y `leq`), así como la de `members`, que indica que el retículo es finito. El resultado de esta tarea es una salida que indica la presencia o ausencia de dichos predicados, como puede ser la siguiente:

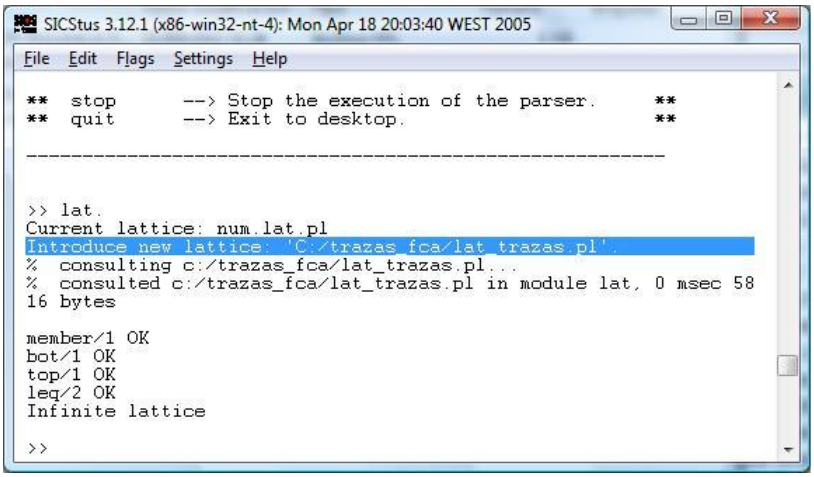


Figura 4.25: Ejemplo de la ejecución de la opción **lat** en $\mathcal{FLOPER}$

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

En esta salida, $\mathcal{FLOPER}$ indica que los predicados `member`, `bot` y `leq` están presentes en el retículo, pero que el predicado `top` no existe. Igualmente, tampoco ha encontrado el predicado `members`, de modo que supone que el retículo es infinito. En caso de encontrar dicho predicado, el sistema muestra `Finite lattice` en lugar de `Infinite lattice`. La Figura 4.25 muestra la ejecución de esta opción desde el interfaz de texto de $\mathcal{FLOPER}$.

Por otra parte, el comando **show**, del submenú de retículos, muestra por pantalla todas las cláusulas que definen el retículo cargado. Ilustramos con la Figura 4.26 el modo de uso de esta opción en la interfaz de texto de $\mathcal{FLOPER}$, donde se muestra el retículo cargado, concretamente, el retículo $([0,1],\leq)$ implementado en el fichero "num.lat".

4.4. Trazas declarativas en $\mathcal{FLOPER}$

Ahora que ya han sido explicadas las características de selección de tipo de paso y la introducción de retículos en el sistema, cabe detallar uno de los trabajos

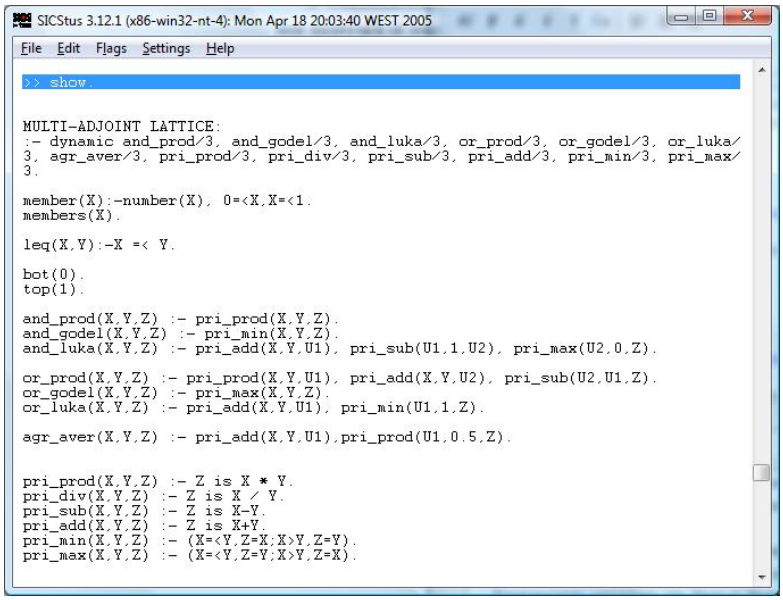


Figura 4.26: Ejemplo de la ejecución de la opción **show** en $\mathcal{FLOPER}$

desarrollados a lo largo de esta tesis, consistente en el diseño de retículos orientados a la obtención de información de ejecución [MMPV11c, MMPV11a]. Esta técnica puede verse como un método de depuración de programas difusos.

En primer lugar es necesario cargar, mediante la opción **lat** el retículo explicado en la Sección 3.6, cuyo efecto es el de incluir una traza de ejecución junto con la respuesta computada. La Figura 4.27 ilustra este retículo, tal como lo muestra la opción **show** en $\mathcal{FLOPER}$. En dicha imagen se presenta tanto el código del retículo, en la parte inferior de la herramienta, como el programa MALP de ejemplo, en la parte superior.

Al evaluar el objetivo `p(X)` en el contexto de este programa, el sistema usará el retículo previamente descrito para interpretar la solución, y el resultado será el que muestra la Figura 4.28. La f.c.a indica que existe una solución para el objetivo `p(X)` cuando `X` cambiar por `a`, con grado de verdad 0.72; además, la propia f.c.a nos permite seguir los pasos que se han utilizado para alcanzar la solución. En particular, se han dado cuatro pasos admisibles empleando las reglas 1,2,3 y 4, respectivamente, y trece pasos interpretativos cortos, cada uno interpretando las diferentes conectivas y operadores primitivos.

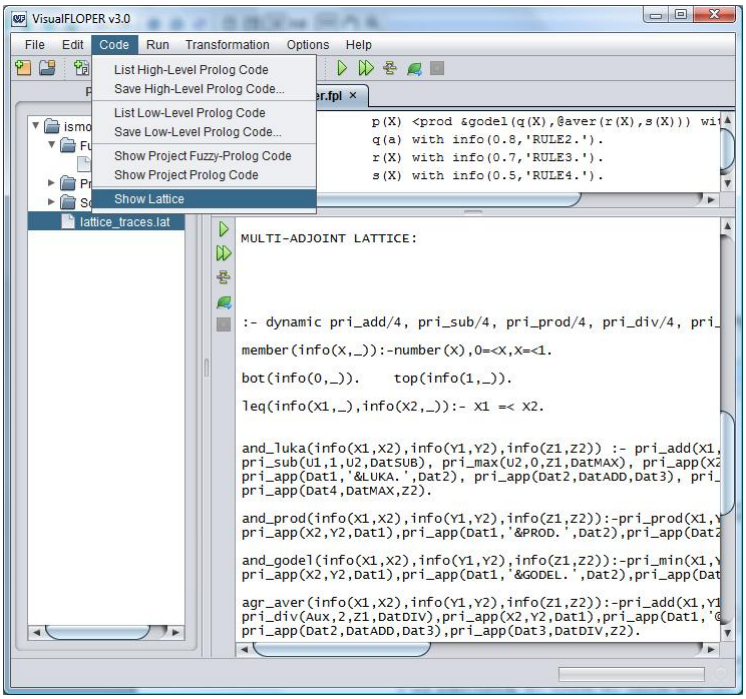


Figura 4.27: Opción 'Show': retículo capaz de generar trazas de ejecución

4.5. $\mathcal{FLOPER}$ online

Además de las interfaces de texto y gráfica (véanse las Secciones 4.1 y 4.2 del Capítulo 4), hemos desarrollado recientemente una página web con el ánimo de permitir el uso de la herramienta $\mathcal{FLOPER}$ a través de Internet, sin requerir al usuario la instalación de ningún software. La interacción online con el sistema se realiza mediante la URL `http://dectau.uclm.es/floper/?q=sim/test`. Bajo el título *FLOPER Online* se muestra una interfaz dividida en tres grandes áreas de texto. El área de entrada o *Input*, mostrada en la Figura 4.29, se ubica en la parte superior de la ventana y, bajo ella, se haya el área de salida u *Output*, ilustrada por la Figura 4.30.

El área de entrada muestra tres cajas de texto. En la primera, etiquetada como "FASILL program", es donde el usuario puede introducir un conjunto de reglas FASILL, esto es, un programa difuso[2]. La segunda, etiquetada como "Lattice", con-

[2]FASILL es un lenguaje que extiende a MALP y que detallamos en el Capítulo 6.

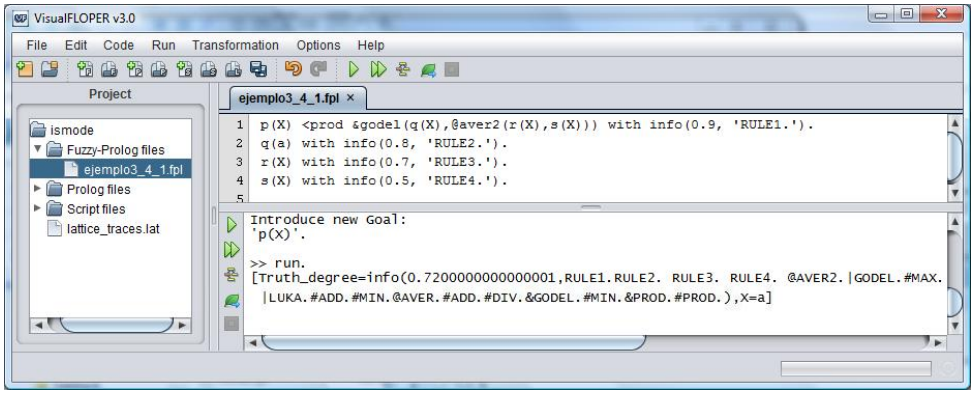


Figura 4.28: Ejemplo de la traza de ejecución en la respuesta computada difusa

tiene el retículo asociado al programa previamente introducido. Por defecto dicha caja contiene el retículo $([0,1],\leq)$, expresado como un programa como hemos visto previamente en la Subsección 4.3.2 del Capítulo 4. Aquí el usuario es libre de modificar el retículo por defecto o de implementar otro, siempre que observe las restricciones sintácticas indicadas en la Sección 4.1.1. En la tercera caja, "Similarity equations", el usuario es libre de escribir un conjunto de ecuaciones de similaridad utilizando la signatura del programa de la primera caja. Bajo estas cajas, el usuario puede introducir el objetivo a evaluar en una caja de texto (en la Figura 4.29 el objetivo es `good_hotel(X)`). Finalmente, al pulsar el botón *Submit*, la página envía al servidor el programa, su retículo asociado y las ecuaciones de similaridad para ejecutarlas.

El resultado es devuelto a la página y mostrado en el área de salida de dos formas. En primer lugar, bajo la etiqueta de "F.c.a's for goal . . . " (donde se indica el objetivo introducido), el sistema muestra las respuestas computadas difusas para el objetivo y el programa. En la Figura 4.30 esto corresponde a `< 0.4, {X/Ritz} >` y `< 0.38, {X/hydropolis} >`, como se esperaba. Siguiendo esta lista de respuestas computadas, en la caja inferior se muestra finalmente el árbol de derivación mostrado de la forma usual en modo texto. Esta herramienta online se ha implementado como una página PHP dentro de la web de FASILL. El documento PHP envía el contenido de las cajas (el programa FASILL, el retículo, las ecuaciones de similaridad y el objetivo) a sí mismo mediante el método "post" y se vuelve a cargar. Cuando la página se carga con parámetros "post" crea los ficheros pertinentes en el servidor

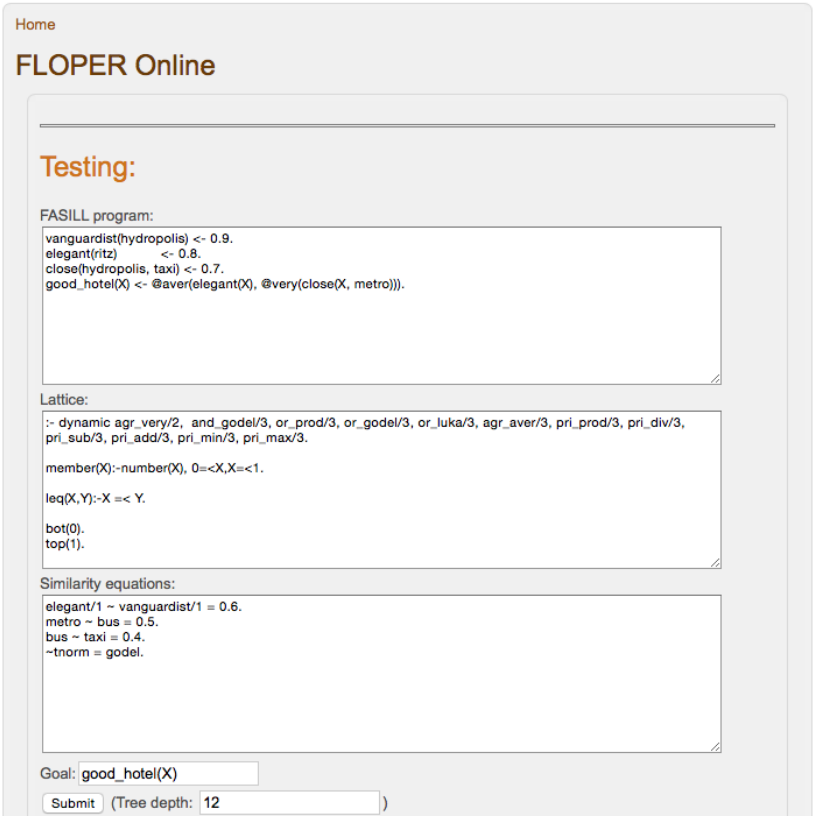


Figura 4.29: Captura de pantalla del área de entrada de $\mathcal{FLOPER}$ online

(para albergar el programa FASILL, el retículo y las ecuaciones de similaridad) y llama al intérprete de Prolog. Entonces, consulta el entorno FLOPER, residente en el servidor, donde carga los ficheros y consulta el objetivo. La salida de esta tarea (esto es, las f.c.a's y el árbol correspondientes) se muestran finalmente en la página.

4.6. Conclusiones

En este amplio capítulo hemos repasado todo lo concerniente a la programación lógica multi-adjunta, antes de adentrarnos en la descripción de la herramienta desarrollada por nuestro grupo de investigación, $\mathcal{FLOPER}$, para trabajar con este lenguaje. El usuario objetivo de esta herramienta es el programador Prolog, motivo por el que todas las decisiones de diseño se han centrado en hacer la experiencia de

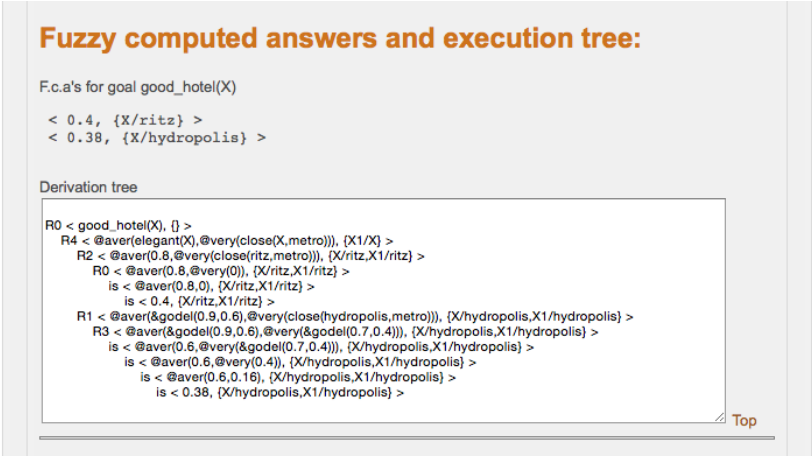


Figura 4.30: Captura de pantalla del área de salida de $\mathcal{FLOPER}$ online

uso de $\mathcal{FLOPER}$ lo más próxima posible al uso de Prolog. De hecho, puede ser instalado fácilmente en la mayoría de plataformas Prolog.

A modo de síntesis, expondremos a continuación las ideas y contribuciones más relevantes que hemos aportado en el capítulo.

- Hemos expuesto las características más relevantes de la herramienta de programación lógica difusa $\mathcal{FLOPER}$ [MV14], desarrollada durante los últimos años en nuestro grupo de investigación y en parte por el autor de esta tesis, como se muestra en los trabajos [MMPV10a, MMPV10c], donde se implementa el manejo de grados de verdad en la herramienta; y [MMPV10b], donde se modela la fase interpretativa de la semántica de MALP.

- Se han implementado conceptos teóricos como los de [MMPV11a, MMPV11c], donde se investiga la inclusión de trazas declarativas en las respuestas computadas del sistema. La implementación de más alcance, sin embargo, corresponde a la de [JMPV14, IMPV15], donde implementamos en $\mathcal{FLOPER}$ el lenguaje de nuevo cuño FASILL.

- Las técnicas más interesantes de esta aplicación, como la ejecución de programas multi-adjuntos basada en una traducción a código Prolog del código difuso; la generación de árboles de desplegado y trazas de ejecución a través de una representación de bajo nivel del código difuso; y la gestión e imple-

mentación de retículos multi-adjuntos, modificadores y variables lingüísticas en $\mathcal{FLOPER}$ también han sido explicadas en profundidad.

- Por último, hemos mostrado la reciente versión gráfica de $\mathcal{FLOPER}$ desarrollada en el grupo, que dota al sistema de una interfaz amigable en la que poder llevar a cabo todas las funcionalidades de la herramienta de manera más visual; y donde la noción de proyecto aparece como pieza clave para la gestión de programas lógicos multi-adjuntos.

- Todavía se está realizando un gran trabajo en el grupo DEC-Tau con respecto a esta herramienta. En concreto, el trabajo de fin de grado titulado "Diseño gráfico de retículos multi-adjuntos sobre el entorno FLOPER", desarrollado por María del Señor Martínez Ruiz, del que he sido codirector, presenta una nueva interfaz gráfica para el modelado de retículos multi-adjuntos.

Nuestra intención última es que $\mathcal{FLOPER}$ se constituya en la principal plataforma sobre la que implantar nuestros resultados fundamentales, permitiendo (además de programar y desarrollar aplicaciones escritas en lenguajes lógicos difusos) la incorporación de potentes técnicas declarativas de manipulación de programas. Para avanzar en esta dirección, como trabajo futuro en el grupo DEC-Tau ya estamos abordando las siguientes mejoras de esta herramienta:

- Plantear la generación de código para un máquina virtual basada en alguna extensión del diseño de la WAM, que es un estándar para la implementación eficiente de lenguajes de programación lógica.

- Seguir trasladando los resultados e implementaciones presentes en $\mathcal{FLOPER}$ a su nueva y mejorada versión gráfica Visual$\mathcal{FLOPER}$.

Capítulo 5

Aplicaciones

Introducidos en capítulos anteriores el lenguaje MALP y el entorno $\mathcal{FLOPER}$, este capítulo presenta tres aplicaciones que hemos desarrollado con estas herramientas. En la Sección 5.1 recogemos un interesante trabajo que permite la retroalimentación entre el lenguaje FuzzyXPath y el entorno $\mathcal{FLOPER}$ [ALMV13, ALMV14, ABL$^+$15]. FuzzyXPath es una extensión difusa, desarrollada en $\mathcal{FLOPER}$, del conocido lenguaje XPath, diseñado para la búsqueda de información en documentos XML. Aquí empleamos este lenguaje para buscar información dentro de los árboles de ejecución dados por $\mathcal{FLOPER}$ en formato XML y obtener así conocimiento acerca de la ejecución de un objetivo.

A continuación, la Sección 5.2 estudia la satisfactibilidad de teorías difusas de un modo paralelo y complementario al de las conocidas técnicas SMT (Satisfiability Modulo Theories). Aquí proponemos $\mathcal{FLOPER}$ como un demostrador automático de teoremas difusos [BMVV15, BMVV13], cuya principal ventaja radica en su capacidad innata de tratar grados de verdad distintos de los usuales (el intervalo $[0, 1]$).

En las Secciones 5.3 y 5.4, referimos la aportación de nuestro grupo –a través de estas herramientas difusas– al área de computación en la nube (*cloud*), [VMTT15, VTMT13]. En particular, en la primera de estas secciones aportamos un algoritmo difuso para la gestión del riesgo en un entorno de overbooking en la nube, mientras que en la segunda proporcionamos un método para resolver el problema de la colocación (o ubicación) de máquinas virtuales en una infraestructura cloud.

5.1. Depuración de computaciones con FuzzyXPath

En esta sección nos centramos en la capacidad del lenguaje FuzzyXPath, desarrollado en nuestro grupo de investigación, para explorar árboles de derivación generados por $\mathcal{FLOPER}$ y exportados como documentos en formato XML [ALMV13, ALMV14, ABL+15]. Esto puede servir como depuración de programas lógicos difusos para descubrir, entre otras cosas, el conjunto de reglas de programa empleados en una derivación, el conjunto de respuestas computadas difusas para un objetivo dado y realizar recorridos en profundidad o anchura en el árbol.

Resumimos ahora brevemente las principales características del lenguaje FuzzyXPath descrito en [ALM11b, ALM11a] (que dispone de una herramienta que se puede descargar libremente a través de `http://dectau.uclm.es/fuzzyXPath/`). En este dialecto flexible de XPath se han incorporado dos restricciones estructurales denominadas `down` y `deep`, a las que se puede asociar un grado concreto de relevancia. Así, mientras `down` provee un conjunto de respuestas favoreciendo las que se encuentran antes en un sentido "vertical" (de arriba abajo) en el documento XML, `deep` ofrece el conjunto de respuestas favoreciendo las que se encuentran antes en un sentido horizontal (de izquierda a derecha o, dicho de otra forma, de menos a más anidamiento) en el documento. Ambas restricciones se pueden usar juntas, asignando un grado de importancia respecto a la distancia del elemento raíz del XML.

Además, FuzzyXPath incorpora variantes difusas de las conectivas *and* y *or* de XPath. Los operadores clásicos *and* y *or* se usan en el XPath estándar sobre condiciones booleanas, y permiten imponer requisitos booleanos a las respuestas. Las condiciones booleanas de Xpath se pueden referir al valor de los atributos y el contenido de los nodos en la forma de igualdad, rangos de literales, etc. Sin embargo, los operadores *and* y *or*, aplicados sobre dos condiciones booleanas, pueden no ser lo bastante precisos cuando el programador no da la misma importancia a las dos condiciones. Algunas respuestas se pueden descartar aún siendo de interés para el programador, mientras que otras de poco interés estarían siendo seleccionadas. Los programadores también podrían necesitar saber si una solución es mejor que otra. Cuando varias condiciones booleanas se imponen en un objetivo, cada una contribuye a satisfacer las preferencias del programador de modos distintos y, quizá, la satisfacción que asignase el programador a cada respuesta sería diferente.

El arsenal de operaciones de XPath ha sido enriquecido con variantes difusas de *and* y *or*. En particular, se han considerado tres versiones de *and*: *and+*, *and* y *and-* (y equivalentemente para *or*: *or+*, *or* y *or-*), lo que permite mayor flexibilidad

en la composición de condiciones difusas. Estas tres versiones de cada operador se obtienen sin coste alguno al adaptar el lenguaje XPath al paradigma difuso. Uno de los elementos mejor conocidos de la lógica difusa es la introducción de diversas versiones difusas de los operadores booleanos. Las lógicas del *Producto*, *Łukasiewicz* y *Gödel* están consideradas como las más prominentes en el ámbito difuso y proporcionan una variada semántica a los operadores difusos. En nuestro trabajo, las versiones difusas aportan un mecanismo para fortalecer (y debilitar) las condiciones (que no son booleanas, sino difusas) en el sentido de las preferencias del programador, usando para ello condiciones difusas más fuertes (o débiles). La combinación de operadores difusos en los objetivos permite satisfacer el conjunto ordenado de condiciones difusas de acuerdo a los requisitos del programador.

Además, se ha equipado a FuzzyXPath con un operador adicional que es también tradicional en la lógica difusa: el operador de la media aritmética *avg*. Este operador ofrece la posibilidad de asignar un peso específico a cada condición difusa. Esto permite al programador cuantificar la importancia de cada condición.

Finalmente, se ha dotado al lenguaje base de FuzzyXPath de un mecanismo de umbralización para filtrar las preferencias del programador, de modo que éste devuelva únicamente las soluciones que se satisfacen por encima de un porcentaje determinado.

Describimos a continuación la semántica del lenguaje FuzzyXPath:

xpath :=	[‘**[**’deep-down‘**]**’]path
path :=	literal | text() | node | **@**att | node/path | node//path
node :=	QName | QName[cond]
cond :=	xpath op xpath | xpath num-op *número*
deep :=	DEEP=*número*
down :=	DOWN=*número*
deep-down :=	deep | down | deep ‘**;**’ down
num-op :=	$> \mid = \mid < \mid <>$
fuzzy-op :=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{*número*,*número*}
op :=	num-op | fuzzy-op

Básicamente, este lenguaje extiende XPath del siguiente modo:

– **Restricciones estructurales**. Una expresión XPath dada se puede enriquecer con «[DEEP $= r_1$; DOWN $= r_2$]», lo que significa que la profundidad (*depth*) en el documento de un elemento se penaliza con r_1, y que la posición vertical de los elementos se penaliza con r_2 proporcionalmente a la distancia (esto es, la longitud de la rama y el peso del árbol, respectivamente). En particular,

$\&_{\mathrm{P}}(x,y)$	=	$x * y$	$\|_{\mathrm{P}}(x,y)$	=	$x + y - x * y$	*Producto: and/or*
$\&_{\mathrm{G}}(x,y)$	=	$\min(x,y)$	$\|_{\mathrm{G}}(x,y)$	=	$\max(x,y)$	*Gödel: and+/or-*
$\&_{\mathrm{L}}(x,y)$	=	$\max(x+y-1,0)$	$\|_{\mathrm{L}}(x,y)$	=	$\min(x+y,1)$	*Łuka.: and-/or+*

Figura 5.1: Operadores lógicos difusos

«[DEEP = 1; DOWN = r_2]» puede usarse para penalizar únicamente respecto al orden (vertical) en el documento. DEEP reconoce //, es decir, la profundidad en el árbol XML sólo se computa cuando se exploran los nodos descendientes, mientras que DOWN reconoce tanto / como //. Nótese que ambos, DEEP y DOWN, se pueden usar varias veces en la ruta de la expresión y/o en cualquier otra sub-ruta incluida en las condiciones.

– **Operadores flexibles en las condiciones**. Se consideran tres versiones difusas de la conjunción y la disyunción clásicas (t-normas y t-conormas, respectivamente [SS83, KMP00]), llamadas también conectivas o agregadores. Estas tres versiones describen escenarios *pesimista*, *realista* y *optimista*. En las expresiones XPath, las versiones difusas de las conectivas debilitan las condiciones booleanas. Además, asumiendo dos *RSV*'s (*Retrieval Status Values*, valores numéricos que indican en qué grado una respuesta satisface una consulta FuzzyXPath) distintos, r_1 y r_2, el operador *avg* está definido de forma obvia, con un sabor difuso, como $(r_1 + r_2)/2$, mientras que su variante basada en pesos (por ejemplo, el que asigna el peso p_1 a la primera condición y p_2 a la segunda, $avg\{p_1, p_2\}$) se define como $(p_1 * r_1 + p_2 * r_2)/(p_1 + p_2)$.

En general, una expresión FuzzyXPath define, con respecto a un documento XML, una secuencia de subárboles del documento donde cada subárbol tiene un RSV asociado. Las condiciones XPath, definidas como operadores difusos aplicados a condiciones XPath, computan un RSV nuevo a partir de los RSV's implicados en las expresiones XPath, aportando a un tiempo un RSV nuevo al nodo. Para ilustrar estas explicaciones, veamos algunos ejemplos de las expresiones de la versión difusa de XPath, de acuerdo al documento XML de la Figura 5.2.

Ejemplo 5.1.1. *Consideremos el objetivo* FuzzyXPath *de la Figura 5.3, que pide el título (title) penalizando las apariciones en el documento en proporción de* 0.8 *y* 0.9 *por anidamiento, respectivamente, y para el que obtenemos el fichero mostrado en*

```
<bib>
    <name>Classic Literature</name>
    <book year="2001" price="45.95">
        <title>Don Quijote de la Mancha</title>
        <author>Miguel de Cervantes Saavedra</author>
        <references>
            <novel year="1997" price="35.99">
                <name>La Galatea</name>
                <author>Miguel de Cervantes Saavedra</author>
                <references>
                    <book year="1994" price="25.99">
                        <title>
                                Los trabajos de Persiles y Sigismunda
                            </title>
                        <author>Miguel de Cervantes Saavedra</author>
                    </book>
                </references>
            </novel>
        </references>
    </book>
    <novel year="1999" price="25.65">
        <title>La Celestina</title>
        <author>Fernando de Rojas</author>
    </novel>
</bib>
```

Figura 5.2: Documento XML de entrada de nuestros ejemplos

Document	**RSV computation**
`<result>`	
`<title rsv="0.8000">`	$0.8000 = 0.8$
`Don Quijote de la Mancha</title>`	
`<title rsv="0.7200">`	$0.7200 = 0.8 * 0.9$
`La Celestina</title>`	
`<title rsv="0.2949">`	$0.2949 = 0.8^5 * 0.9$
`Los trabajos de Persiles...</title>`	
`</result>`	

Figura 5.3: Ejecución del objetivo «//book[@year<2000 avg{3,1} @price<50]/title»

la Figura 5.3. En dicho documento hemos incluido, como atributo de cada subárbol, su RSV correspondiente. Los RSV's más altos corresponden a los libros principales del documento, y los más bajos representan los libros que aparecen en posiciones anidadas (aquellos anotados como referencias).

Ejemplo 5.1.2. *La Figura 5.4 muestra la respuesta asociada a cada búsqueda de libros, posiblemente referenciado* directa o indirectamente *desde otros libros cuyos años de publicacion son relevantes, en un grado tres veces mayor que sus precios.*

Document	**RSV computation**
<result> <title **rsv**="1.00"> Los trabajos de Persiles... </title> <title **rsv**="0.25"> Don Quijote de la Mancha </title> </result>	$1.00 = (3*1+1*1)/(3+1)$ $0.25 = (3*0+1*1)/(3+1)$

Figura 5.4: Ejecución del objetivo «/bib[DEEP=0.5]//book[@year<2000 avg{3,1} @price<50]/title»

Document	**RSV computation**
<result> <title **rsv**="0.25"> Don Quijote de la Mancha </title> <title **rsv**="0.0625"> Los trabajos de Persiles... </title> </result>	$0.25 = (3*0+1*1)/(3+1)$ $0.0625 = 0.5^4 * (3*1+1*1)/(3+1)$

Figura 5.5: Ejecución del objetivo «/bib[DEEP=0.8;DOWN=0.9]//title»

Finalmente, en la Figura 5.5 hemos combinado ambos tipos de operadores estructurales/condicionales y la lista priorizada de soluciones está al revés.

Finalmente, se usa el comando «[FILTER = r]» al principio de un objetivo para filtrar el final de su conjunto de soluciones en el sentido de que sólo aquellas con un RSV no inferior a r se incluyan en la salida.

5.1.1. Árboles de depuración XML en $\mathcal{FLOPER}$

Presentamos ahora un programa MALP de ejemplo en el que nos basaremos para realizar consultas FuzzyXPath e ilustrar así las aportaciones realizadas en este ámbito.

El programa $\mathcal{P}$ que sigue modela, a través del predicado `oc`/1, las posibilidades de una ciudad para ser elegida como "ciudad olímpica" (esto es, para llevar a cabo los juegos olímpicos). El predicado que indica esa posibilidad, `oc`/1, se define en la regla $\mathcal{R}_1$, cuyo cuerpo recoge la información de otros tres predicados, `s`/1, `f`/1 y `w`/1, cuyo significado es, respectivamente, el nivel de *s*eguridad, las *i*nfraestructuras y el *c*lima de cada ciudad (otros criterios como la rotación continental, aspectos

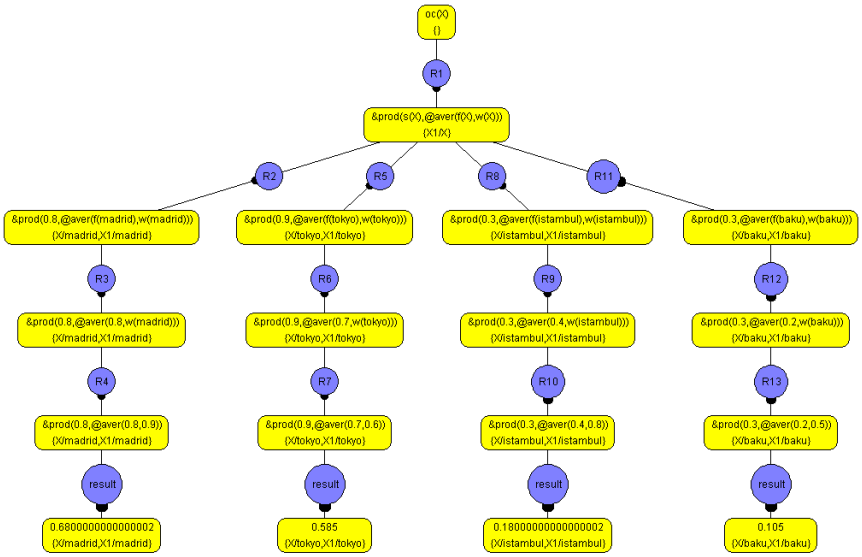


Figura 5.6: Árbol de ejecución del programa $\mathcal{P}$ y objetivo `oc(X)`

político/econónimos, etc., se pueden agregar fácilmente a la definición de oc/1). Estos predicados se definen en las reglas desde la $\mathcal{R}_2$ hasta la $\mathcal{R}_{13}$ para cuatro ciudades (*Madrid*, *Istambul*, *Tokyo* and *Baku*), de modo que para cada ciudad la característica modelada por cada predicado es mejor cuanto mayor sea el grado de verdad de la regla.

$\mathcal{R}_1$:	$oc(X)$	<-	$s(X)$ &$prod$	$(f(X)$	$@aver\ w(X))$	$with$ 1.
$\mathcal{R}_2$:	$s(madrid)$		$with$ 0.8.	$\mathcal{R}_5$:	$s(tokyo)$	$with$ 0.9.
$\mathcal{R}_3$:	$f(madrid)$		$with$ 0.8.	$\mathcal{R}_6$:	$f(tokyo)$	$with$ 0.7.
$\mathcal{R}_4$:	$w(madrid)$		$with$ 0.9.	$\mathcal{R}_7$:	$w(tokyo)$	$with$ 0.6.
$\mathcal{R}_8$:	$s(istambul)$		$with$ 0.3.	$\mathcal{R}_{11}$:	$s(baku)$	$with$ 0.3.
$\mathcal{R}_9$:	$f(istambul)$		$with$ 0.4.	$\mathcal{R}_{12}$:	$f(baku)$	$with$ 0.2.
$\mathcal{R}_{10}$:	$w(istambul)$		$with$ 0.8.	$\mathcal{R}_{13}$:	$w(baku)$	$with$ 0.5.

Como se ha visto en el Capítulo 4, $\mathcal{FLOPER}$ provee dos métodos para evaluar un objetivo, dados un programa MALP y su retículo correspondiente. La opción "`run`" traduce el programa difuso completo a un programa Prolog y evalúa el objetivo

(traducido del mismo modo), obteniendo el conjunto de respuestas computadas difusas. Por otro lado, la opción "`tree`" muestra el árbol de ejecución (o derivación) para el objetivo dado[1]. Nos centramos aquí en la segunda opción, la que obtiene un árbol (detallando al completo el comportamiento del programa) para analizarlo seguidamente con FuzzyXPath.

Consideremos el programa $\mathcal{P}$ descrito previamente, y el objetivo "`oc(X)`", cuyo grado de verdad se corresponde con la elegibilidad de cada una de las cuatro ciudades en $\mathcal{P}$ como "Ciudad Olímpica". Utilizamos la opción "`tree`" para obtener el árbol de ejecución, que $\mathcal{FLOPER}$ genera en tres formatos distintos. En primer lugar, el árbol se muestra en modo gráfico como un fichero PNG, tal y como aparece en la Figura 5.6. Recordamos que los estados están representados en los nodos amarillos, mientras que los nodos azules informan de la regla del programa empleada para pasar de un estado a otro.

$\mathcal{FLOPER}$ también puede generar el árbol de ejecución en dos formatos textuales. El primero contiene una descripción en texto plano del árbol, mientras que el segundo provee una estructura XML de dicha descripción, lo cual lo convierte en el foco de interés de esta sección. En este formato XML, definimos la etiqueta "`node`" para contener toda la información relativa a un nodo, como la regla utilizada para alcanzar dicho estado (por ejemplo "`R0`" en el caso del primer estado), la fórmula del estado, la sustitución acumulada y los nodos hijos, dados por las etiquetas "`rule`", "`goal`", "`substitution`" y "`children`", respectivamente. El contenido de las etiquetas 'rule", "`goal`" y "`substitution`" es una cadena de caracteres, mientras que el contenido de la etiqueta "`children`" es un conjunto de etiquetas "`node`", como se ve en las siguientes líneas, correspondientes al archivo XML asociado al árbol mostrado en la Figura 5.6.

```
<node>
  <rule>R0</rule>
  <goal>oc(X)</goal>
  <substitution>{}</substitution>
  <children>
    <node>
      <rule>R1</rule>
      <goal>and_prod(s(X),agr_aver(f(X),w(X)))</goal>
      <substitution>{X1/X}</substitution>
      <children>
        <node>
          <rule>R2</rule>
```

[1]Los usuarios pueden seleccionar el nivel máximo de progundidad para el árbol, lo cual es necesario para el caso de los árboles infinitos.

```
<goal>and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
<substitution>{X/madrid,X1/madrid}</substitution>
<children>
  <node>
    <rule>R3</rule>
    <goal>and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
    <substitution>{X/madrid,X1/madrid}</substitution>
    <children>
      <node>
        <rule>R4</rule>
        <goal>and_prod(0.8,agr_aver(0.8,0.9))</goal>
        <substitution>{X/madrid,X1/madrid}</subtitution>
        <children>
          <node>
            <rule>result</rule>
            <goal>0.6800000000000002</goal>
            <substitution>{X/madrid,X1/madrid}
            </substitution>
            <children>
            </children>
          </node>
        </children>
      </node>
    </children>
  </node>
  ...
              <node>
                <rule>result</rule>
                <goal>0.585</goal>
                <substitution>{X/tokyo,X1/tokyo}
                </substitution>
                <children>
                </children>
              </node>
    ...
  </node>
 </children>
</node>
```

5.1.2. Exploración de Árboles de Derivación con FuzzyXPath

En esta subsección presentamos un potente método para explorar automáticamente el comportamiento de un programa MALP usando la herramienta FuzzyXPath descrita en la Sección 5.1. Pretendemos usar FuzzyXPath sobre el árbol de ejecución generado por $\mathcal{FLOPER}$ para un programa y objetivo determinados. Dicho árbol se obtiene a través de la opción "`tree`" usando el árbol con formato XML explicado anteriormente en la Subsección 5.1.1. Por ejemplo, un objetivo sencillo pero

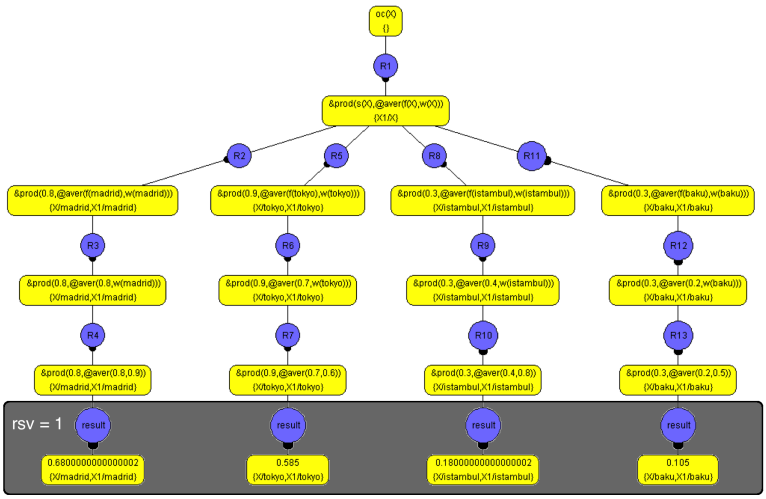
interesante es "`//node/rule`", que muestra todas las reglas explotadas a lo largo de la ejecución del objetivo (en el caso del árbol mostrado en la Figura 5.6, obtendríamos el conjunto completo de las reglas definidas en el programa $\mathcal{P}$ del ejemplo).
Procedemos ahora con la intención de obtener el conjunto completo de respuestas computadas difusas (*fuzzy computed answers*) de un programa y objetivo determinados. Esta información, recolectada siempre en las hojas de los árboles de ejecución (incluso cuando existe la posibilidad de encontrar hojas que no contengan respuestas computadas difusas, como veremos más adelante) como ilustra la Figura 5.7, se puede obtener mediante el objetivo FuzzyXPath "`//node[/rule/text()=result]`", que devuelve cada nodo tal que el contenido de su etiqueta *tag* es "result". El texto XML mostrado debajo de la Figura 5.7 representa la salida del intérprete FuzzyXPath para ese objetivo, donde los nodos seleccionados se han resaltado dentro de un recuadro oscuro dentro del árbol. Nótese que el fichero XML resultante contiene cuatro soluciones (una para cada ciudad), donde el atributo "`rsv`" indica lo bien que cada ciudad encaja en el objetivo original (en este ejemplo, este valor es el mismo en todos los casos, esto es, el máximo 1).

Fuertemente relacionado con el experimento previo, pero no orientado a respuestas computadas difusas, el objetivo "`//node[children[not(text())]]`" devuelve las hojas del árbol. Nótese que, en el caso del programa $\mathcal{P}$ y objetivo "`oc(X)`", la salida correspondiente es, una vez más, el mismo mostrado anteriormente en la Figura 5.7, pero, como se dijo en el párrafo anterior, esta coincidencia no se da necesariamente.

De hecho, lo contrario se puede formular mediante "`//node[children[not(text())] and rule/text()<>''result'']/goal`", que informa de si el árbol tiene una hoja parcialmente evaluada (esto es, una hoja que no contiene una respuesta computada difusa) ya que devuelve nodos al final de cada rama que no contienen "`result`" en la etiqueta *rule*. El significado de este objetivo reside en su capacidad de encontrar posibles fuentes de bucles infinitos.

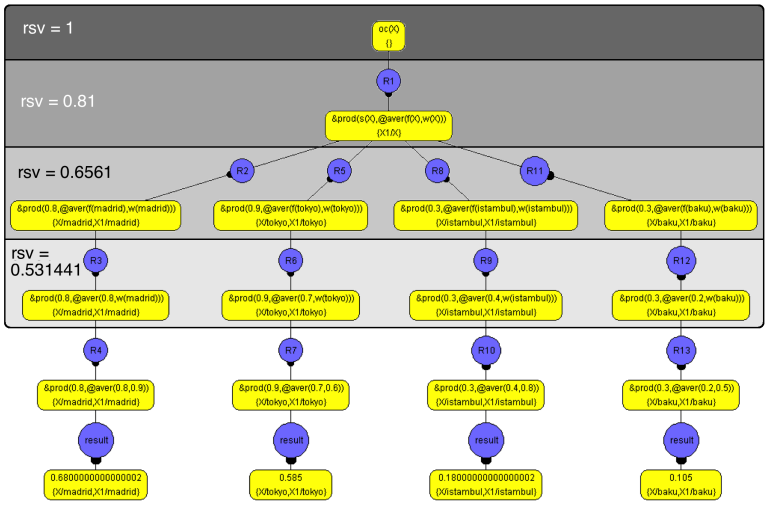
Así, consideremos el ejemplo de la Figura 5.9, donde se observa claramente una rama infinita (en el centro) entre dos ramas que contienen respuestas computadas difusas. Esta figura corresponde al árbol de ejecución para el objetivo "`q(X) @aver p(X)`" con respecto al programa

```
p(a) with 0.8.
p(X) <prod p(s(s(s(X)))) with 0.9.
p(b) with 0.6.
```




```
<result>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.6800000000000002</goal>
    <substitution>{X/madrid, X1/madrid}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.585</goal>
    <substitution>{X/tokyo, X1/tokyo}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.18000000000000002</goal>
    <substitution>{X/istambul, X1/istambul}</substitution>
    <children></children>
  </node>
  <node rsv="1.0">
    <rule>result</rule>
    <goal>0.105</goal>
    <substitution>{X/baku, X1/baku}</substitution>
    <children></children>
  </node>
</result>
```

Figura 5.7: Ejecución de los objetivos «//node[/rule/text()=result]» y «//node[children[not(text())]]»

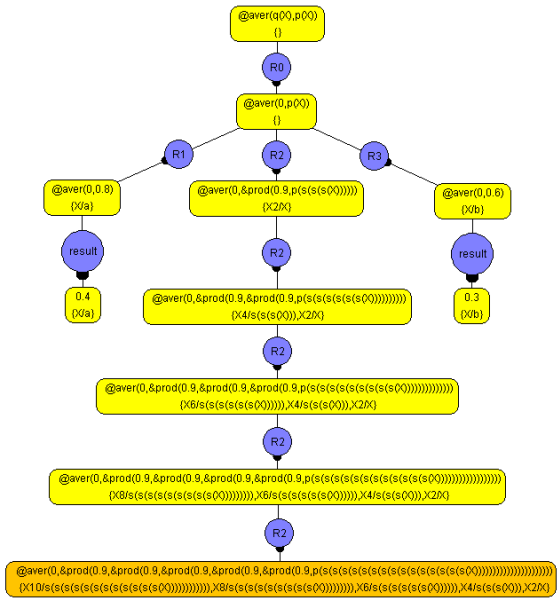



```
<result>
  <goal rsv="1.0">oc(X)</goal>
  <goal rsv="0.81">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="0.6561">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="0.6561">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.6561">and_prod(0.3,agr_aver(f(istambul),w(istambul)))</goal>
  <goal rsv="0.6561">and_prod(0.3,agr_aver(f(baku),w(baku)))</goal>
  <goal rsv="0.531441">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="0.531441">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.531441">and_prod(0.3,agr_aver(0.4,w(istambul)))</goal>
  <goal rsv="0.531441">and_prod(0.3,agr_aver(0.2,w(baku)))</goal>
</result>
```

Figura 5.8: Ejecución del objetivo «[FILTER=0.5][DEEP=0.9]//node/goal»

Nótese que, en esta figura, la búsqueda de nodos con un campo "`children`" vacío mediante el uso de la consulta "`//node[children[not(text())]]/goal`" devuelve tres soluciones, que son las tres hojas del árbol, sin distinguir cuáles de ellas corresponden a objetivos total o parcialmente evaluados. Para obtener sólo las respuestas correctas difusas del árbol usamos "`//node[rule/text()=''result'']/goal`", como se ve en la Figura 5.9. Finalmente, para obtener el último nodo de la rama central (infinita), utilizamos la segunda consulta mostrada en la Figura 5.9, es decir, "`//node[children[not(text())] and rule/text()<>''result'']/goal`".

Con objeto de aprovechar de las mejoras introducidas en el lenguaje FuzzyXPath, la siguiente consulta hace uso de los comandos "`DEEP`" y "`FILTER`" para realizar



Ejecución del objetivo «//node[children[not(text())]]/goal»

```
<result>
  <goal>0.4</goal>
  <goal>agr_aver(0,and_prod(0.9,...p(s(...(s(X))))))</goal>
  <goal>0.3</goal>
</result>
```

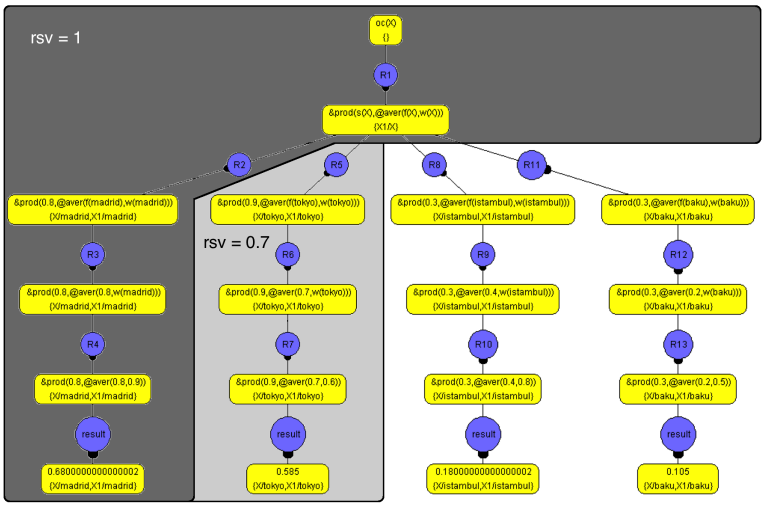
Ejecución de «//node[children[not(text())] and rule/text()<>“result”]/goal»

```
<result>
  <goal>agr_aver(0,and_prod(0.9,...p(s(...(s(X))))))</goal>
</result>
```

Ejecución del objetivo «//node[rule/text()=“result”]/goal»

```
<result>
  <goal>0.4</goal>
  <goal>0.3</goal>
</result>
```

Figura 5.9: Ejemplo de un árbol con una rama infinita

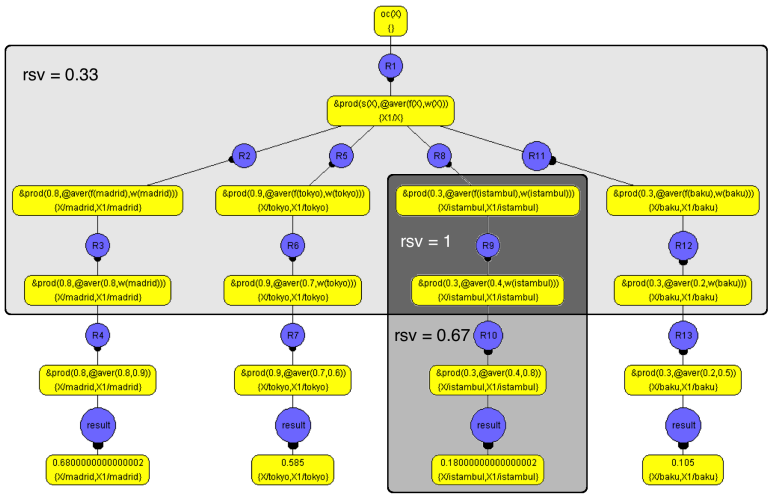



```
<result>
  <goal rsv="1.0">oc(X)</goal>
  <goal rsv="1.0">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="1.0">and_prod(0.8,agr_aver(0.8,0.9))</goal>
  <goal rsv="1.0">0.6800000000000002</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.7">and_prod(0.9,agr_aver(0.7,0.6))</goal>
  <goal rsv="0.7">0.585</goal>
</result>
```

Figura 5.10: Ejecución del objetivo «[FILTER=0.5][DOWN=0.7]//node/goal»

un recorrido en anchura en árboles de ejecución como se muestra en la Figura 5.8. En la salida XML resultante se han seleccionado diez nodos del árbol de ejecución con diferentes valores RSV, desde 1 en el caso del objetivo original (que no ha sido penalizado) hasta 0.531441 para los nodos de la cuarta fila, que representan nodos cuya progundidad ("DEEP-level") está en el límite. Obsérvese que el uso de la directiva "`DEEP`" segrega los nodos del árbol de arriba abajo, pues los nodos más bajos en el árbol se representan con mayor profundidad en el fichero XML de entrada.

Análogamente, en la Figura 5.10 usamos "`DOWN`" en lugar de "`DEEP`" para producir un recorrido en profundidad en árboles de ejecución. En este caso, la consulta segrega los nodos de izquierda a derecha en columnas pues, cuanto más a la izquierda aparece




```
<result>
  <goal rsv="1.0">and_prod(0.3,agr_aver(f(istambul),w(istambul)))</goal>
  <goal rsv="1.0">and_prod(0.3,agr_aver(0.4,w(istambul)))</goal>
  <goal rsv="0.6667">and_prod(0.3,agr_aver(0.4,0.8))</goal>
  <goal rsv="0.6667">0.18000000000000002</goal>
  <goal rsv="0.3333">and_prod(s(X),agr_aver(f(X),w(X)))</goal>
  <goal rsv="0.3333">and_prod(0.8,agr_aver(f(madrid),w(madrid)))</goal>
  <goal rsv="0.3333">and_prod(0.9,agr_aver(f(tokyo),w(tokyo)))</goal>
  <goal rsv="0.3333">and_prod(0.3,agr_aver(f(baku),w(baku)))</goal>
  <goal rsv="0.3333">and_prod(0.8,agr_aver(0.8,w(madrid)))</goal>
  <goal rsv="0.3333">and_prod(0.9,agr_aver(0.7,w(tokyo)))</goal>
  <goal rsv="0.3333">and_prod(0.3,agr_aver(0.2,w(baku)))</goal>
</result>
```

Figura 5.11: Ejecución del objetivo «node[/goal[contains(text(),“w(”)] aver{1,2} substitution[contains(text(),“istambul”)]]//goal»

un nodo en el árbol, más arriba aparece en el XML y, por ello, menos lo penaliza la directiva “`DOWN`”. Como anteriormente, se han seleccionado diez nodos con RSV entre 1 –los nodos superiores en el XML– hasta 0.7, como se muestra en la segunda columna.

Finalmente, para ilustrar el alto poder expresivo de FuzzyXPath, modelamos en la Figura 5.11 diversas consultas uniendo diversos conceptos (entre ellos, los tópicos “weather” e “Istambul” modelados en $\mathcal{P}$ como el predicado “w” y la constante “$istambul$”, respectivamente). Inicialmente preguntamos por nodos que informen so-

bre el tiempo ("weather"), esto es, centrados en la cuarta fila del árbol de ejecución (por tanto, tienen presente la subcadena "`w(`" en la etiqueta "*goal*"). La segunda preferencia pregunta por nodos en la rama que contiene la palabra "`istambul`" en la etiqueta "*substitution*". Para unir estas dos restricciones, en lugar de usar un "*and*" o un "*or*" difuso (o incluso diferentes variantes difusas de ambos conectivos, ya implementadas en FuzzyXPath), ilustramos el uso de una media aritmética asignando dos veces más importancia al segundo requisito que al primero. La expresión FuzzyXPath de esta consulta se halla en el título de la Figura 5.11, donde mostramos gráficamente el conjunto de soluciones así como la salida en el fichero XML resultante.

5.2. MALP como un demostrador SMT

La investigación en SAT (Boolean Satisfiability) y SMT (Satisfiability Modulo Theories) representa una larga tradicción de éxito en el desarrollo de resolvedores de teoremas automáticos altamente eficientes para la lógica clásica. Más recientemente, se han llevado a cabo intentos de cubrir la lógica difusa, como sucede en los avances presentados en [ABMV12, VBG12]. Además, si la resolución automática de teoremas supone un punto de partida para los fundamentos de la programación lógica así como uno de sus campos de aplicación [Llo87, Sti88, Fit96], en esta sección mostramos algunas líneas preliminares sobre cómo la programación lógica difusa puede enfrentar la demostración automática de teoremas difusos, recogidos de los trabajos [BMVV15, BMVV13].

5.2.1. Motivación

Comenzamos nuestro discurso con un sencillo ejemplo motivador. Asumimos que contamos con un sencillo chip digital que cuenta sólo con un puerto de entrada y uno de salida, de manera que invierte en Out la señal recibida de In. El comportamiento de dicho chip se puede representar con la siguiente fórmula propopsicional

$F : (In' \wedge Out) \vee (In \wedge Out')$.

Dependiendo de cómo se interprete cada símbolo propopsicional, obtenemos el siguiente conjunto de interpretaciones para la fórmula:

$$I1: \ \{In = 0, Out = 0\} \ \Rightarrow \ F = 0 \qquad I2: \ \{In = 0, Out = 1\} \ \Rightarrow \ F = 1$$
$$I3: \ \{In = 1, Out = 0\} \ \Rightarrow \ F = 1 \qquad I4: \ \{In = 1, Out = 1\} \ \Rightarrow \ F = 0$$

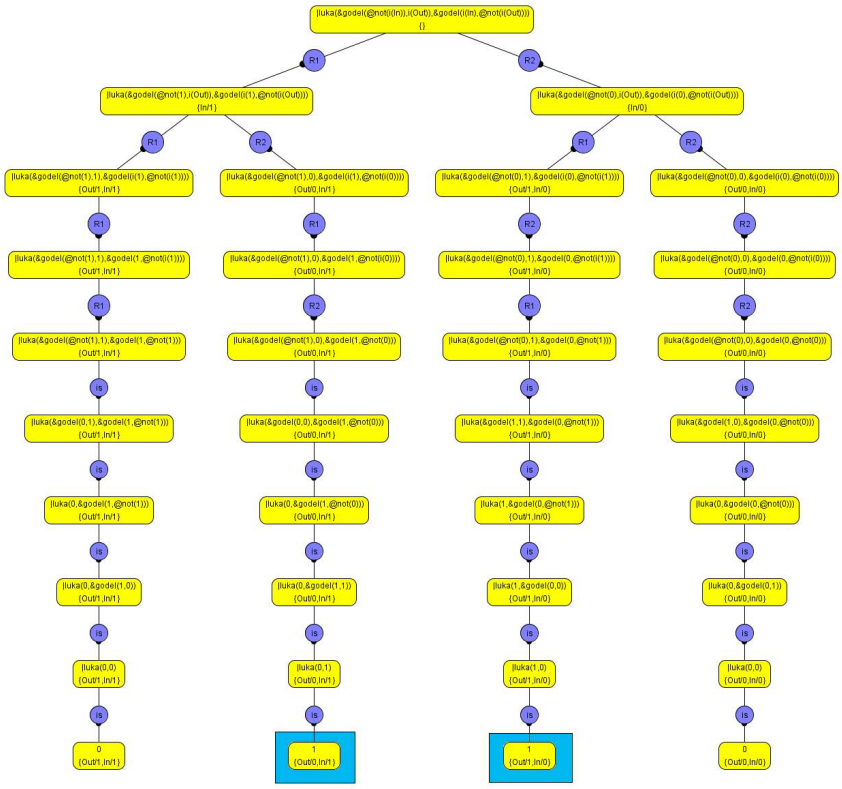


Figura 5.12: Interpretación de una fórmula con dos valores para las señales/proposiciones In y Out.

Un resolvedor SAT puede probar fácilmente que F es satisfacible ya que presenta dos modelos (esto es, interpretaciones de las variables proposicionales In y Out para las que la fórmula se interpreta con grado de verdad 1) representados por $I2$ e $I3$. Un modo alternativo para obtener explícitamente estas interpretaciones consiste en usar el entorno de lógica difusa $\mathcal{FLOPER}$ descrito en el capítulo 4. Como explicamos en el resto de la sección, cuando $\mathcal{FLOPER}$ ejecuta el siguiente objetivo (que representa la fórmula F) "`(@not(i(In)) & i(Out)) | (i(In) & @not(i(Out)))`" con respecto a programa lógico difuso compuesto por dos reglas: "`i(1) with 1`" y "`i(0) with 0`", dibuja el árbol mostrado en la Figura 5.12, donde los modelos $I2$ e $I2$ aparecen en dos hojas centrales del árbol dentro de la caja azul. Cada rama en el árbol comienza

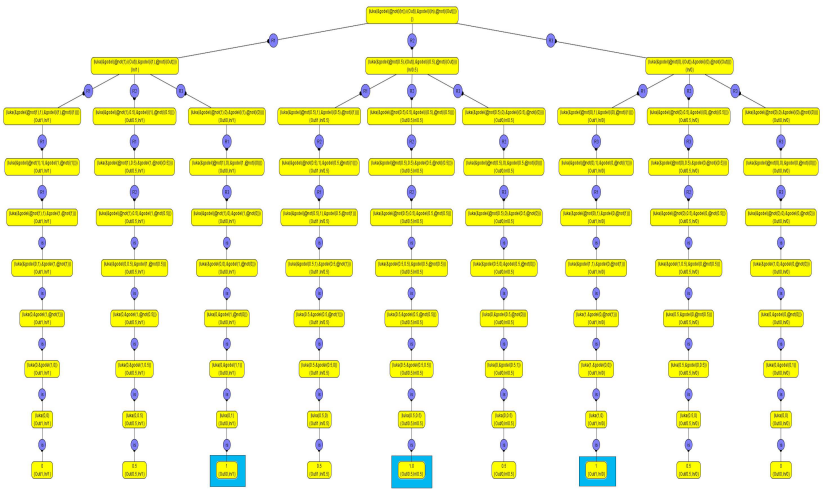


Figura 5.13: Interpretación de una fórmula con tres grados de verdad para las señales/proposiciones In y Out.

interpretando las variables In y Out y continua con la evaluación de los operadores que aparecen en F.

Adviértase que mientras la fórmula F describe el comportamiento de nuestro chip de una manera "implícita", el conjunto de los modelos $I2$ y $I3$ describe "explícitamente" el funcionamiento correcto del chip (cualquier otra interpretación no es un modelo y representa un comportamiento anormal del chip), de ahí la importancia de encontrar el conjunto completo de modelos para una fórmula dada.

Modelamos ahora una versión "analógica" del chip, donde las señales de entrada y salida pueden variar en un rango infinito de valores entre 0 y 1, de modo que Out debe representar el "complementario" de In. El nuevo comportamiento del chip puede expresarse de nuevo por medio de la fórmula anterior, pero teniendo en cuenta que las conectivas involucradas en F deben definirse de un modo difuso como sigue:

x'	$=$	$1 - x$	Negación de la lógica del Producto
$x \wedge y$	$=$	$min(x, y)$	Conjunción de la lógica de Gödel
$x \vee y$	$=$	$min(x + y, 1)$	Disyunción de la lógica de Łukasiewicz

En este caso podemos usar un resolvedor SMT para demostrar que F es satisfacible. Siguiendo la aproximación del trabajo en [ABMV12], se puede comprobar fácilmente

que F es satisfacible[2] con el siguiente script SMT-LIB que codifica las conectivas previas en *SAT modulo linear real arithmetic*:

```
; Set logic: Quantifier Free Linear Real Arithmetic
(set-logic QF_LRA)

; min(x,y)
(define-fun min ((x Real) (y Real)) Real
  (ite (> x y) y x))

; x' = 1 - x
(define-fun agr_not ((x Real)) Real
  (- 1 x))

; &G(x,y) = min{x,y}
(define-fun and_godel ((x Real) (y Real)) Real
  (min x y))

; |L(x,y) = min{x+y,1}
(define-fun or_luka ((x Real) (y Real)) Real
  (min (+ x y) 1))

; Declaration of variables
(declare-fun x () Real)
(declare-fun y () Real)

; Ordering relation
(assert (>= x 0))
(assert (<= x 1))
(assert (>= y 0))
(assert (<= y 1))

; Formula to check
(assert (= (or_luka (and_godel (agr_not x) y)
                    (and_godel x (agr_not y))) 1))

; Check for satisfiability
(check-sat)
```

[2]La fórmula tiene infinitos modelos de la forma $\{In = x, Out = y\}$ tales que $x + y = 1$.

El código SMT-LIB se puede comprender con facilidad. Recordamos que la expresión `ite` corresponde a la construcción *if-then-else*. Nótese que todas las conectivas necesarias han sido codificadas como se hace en [ABMV12]. Merece la pena advertir que, aparte de demostrar la satisfacibilidad de una fórmula, los resolvedores SMT se pueden usar para demostrar que una fórmula es un teorema mediante el chequeo de la insatisfacibilidad de su negada.

Por otra parte, la Figura 5.13 muestra también tres modelos en el árbol mostrado por $\mathcal{FLOPER}$ al considerar únicamente tres tipos de valores (concretamente, 0, 0.5 y 1) para interpretar In y Out. Dichos modelos incluyen, aparte de $I2$ e $I3$ (vistos anteriormente), la interpretación $\{IN = 0.5, OUT = 0.5\}$ ya que, como se puede ver detallado en las computaciones realizadas a lo largo de la rama central del árbol, tenemos:

$$\begin{array}{llll}
(0.5' \wedge 0.5) \vee (0.5 \wedge 0.5') & = & ((1-0.5) \wedge 0.5) \vee (0.5 \wedge 0.5') & = (0.5 \wedge 0.5) \vee (0.5 \wedge 0.5') \\
= min(0.5, 0.5) \vee (0.5 \wedge 0.5') & = & 0.5 \vee (0.5 \wedge 0.5') & = 0.5 \vee (0.5 \wedge (1-0.5)) \\
= 0.5 \vee (0.5 \wedge 0.5) & = & 0.5 \vee min(0.5, 0.5) & = 0.5 \vee 0.5 \\
= min(0.5 + 0.5, 1) & = & min(1, 1) & = 1
\end{array}$$

Análogamente, se puede comprobar en la segunda rama del árbol que $\{IN = 1, Out = 0.5\}$ no es un modelo (de hecho, el chip no puede devolver una señal 0.5 cuando su entrada es 1) ya que:

$$\begin{array}{llllllll}
(1' \wedge 0.5) \vee (1 \wedge 0.5') & = & ((1-1) \wedge 0.5) \vee (1 \wedge 0.5') & = & (0 \wedge 0.5) \vee (1 \wedge 0.5') & = \\
min(0, 0.5) \vee (1 \wedge 0.5') & = & 0 \vee (1 \wedge 0.5') & = & 0 \vee (1 \wedge (1-0.5)) & = \\
0 \vee (1 \wedge 0.5) & = & 0 \vee min(1, 0.5) & = & 0 \vee 0.5 & = \\
min(0 + 0.5, 1) & = & min(0.5, 1) & = & 0.5 &
\end{array}$$

5.2.2. MALP, $\mathcal{FLOPER}$ y la Demostración Automática de Teoremas

En lo que sigue describimos un sencillo *subconjunto del lenguaje* MALP[3] (ver el Capítulo 3 para una formulación completa de este marco), en el que conservamos el uso de distintas concectivas: conectivas de implicación (denotadas por $\leftarrow_1, \leftarrow_2, \ldots$); conectivas de conjunción ($\wedge_1, \wedge_2, \ldots$); conectivas de disyunción ($\vee_1, \vee_2, \ldots$); y operadores híbridos (generalmente denotados por $@_1, @_2, \ldots$), todos los cuales, como vimos en el Capítulo 3, se agrupan bajo el nombre de "agregadores" y cuya función de verdad $\dot{@} : L^n \to L$, recordamos, debe ser monótona. Podemos usar de nuevo las ya conocidas lógicas de *Łukasiewicz*, *Gödel* y *producto*, considerados, respectivamente, versiones pesimista, optimista y realista de la conjunción. También conservamos

[3]Multi-Adjoint Logic Programming.

en este subconjunto de MALP la posibilidad de usar cualquier retículo multi-adjunto $(L, \leq)$, sobre el que se interpretan las conectivas y que, además, aporta la noción de *grado de verdad*.

Este subconjunto de MALP está diseñado para tratar fórmulas proposicionales difusas como $P \wedge Q \rightarrow P \vee Q$, donde las proposiciones P y Q se interpretan como valores de un retículo. Para ello, un *programa* se define como un conjunto de reglas (en este caso, "hechos") de la forma "H *with* v", donde H es una fórmula atómica o un átomo (generalmente llamado *cabeza*), y v es su *grado de verdad* asociado (esto es, un valor de L). Más exactamente, en nuestra aplicación las cabezas tienen siempre la forma "$i(V)$" y cada regla de programa tiene la forma "$i(v)$ *with* v". Advertimos que aún usando los mismos nombres para las constantes (que construyen los términos) y grados de verdad, el Universo de Herbrand de cada programa y su retículo asociado no deben confundirse, ya que son, de hecho, conjuntos disjuntos.

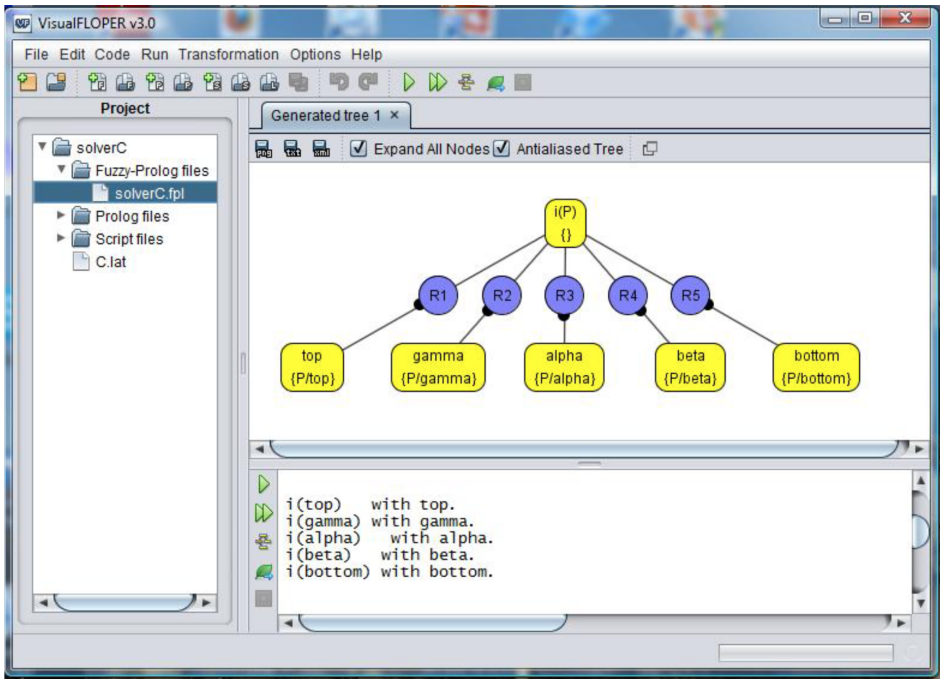


Figura 5.14: Sesión de trabajo con $\mathcal{FLOPER}$ en la que se resuelve el objetivo `i(P)`.

Un *objetivo* es, como en MALP, una fórmula construida a partir de fórmulas atómicas $B_1, \ldots, B_n$ $(n \geq 0)$, grados de verdad de L, conjunciones, disyunciones y agre-

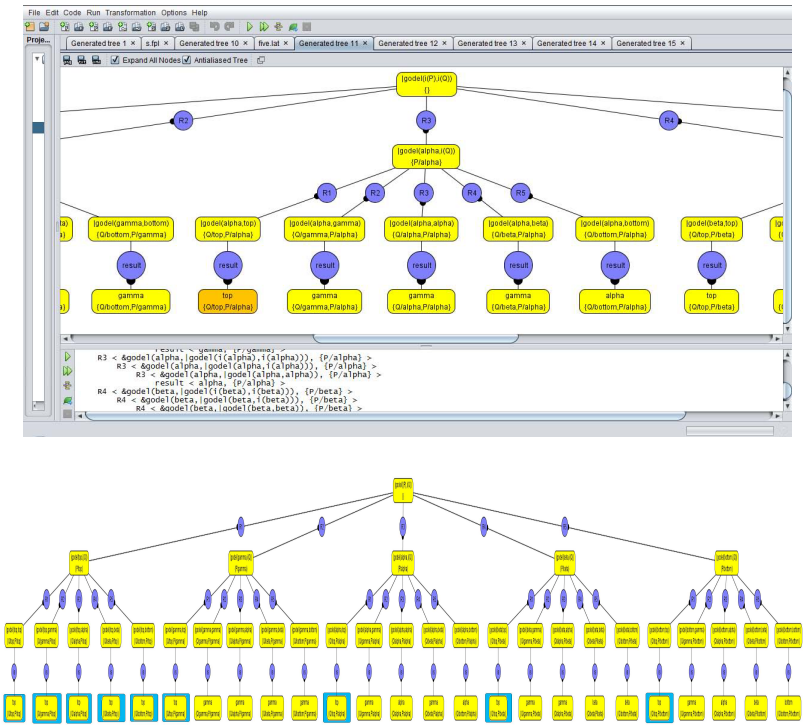
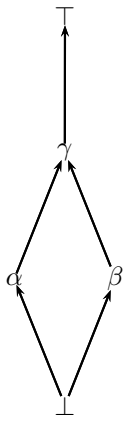


Figura 5.15: Sesión de trabajo con $\mathcal{FLOPER}$ en la que se resuelve la fórmula $P \vee Q$ (25 interpretaciones, 9 modelos).

gadores, lanzadas como una consulta al sistema. En este subconjunto de MALP, las fórmulas atómicas de un *objetivo* siempre tienen la forma "$i(P)$", siendo P un símbolo de variable. De este modo, cuando se ejecuta un objetivo simple como "$i(P)$" (como se hace en la Figura 5.14), obtenemos varias respuestas del tipo $P = v$, donde el grado de verdad resultante es "v", representando todas las interpretaciones posibles en L para la proposición P en la fórmula original.

La semántica operacional de este subconjunto del lenguage MALP coincide con la de MALP. Como herramienta para testear nuestros resultados usamos $\mathcal{FLOPER}$, que está descrito en el Capítulo 4.

A este fin, hemos introducido en el código PROLOG que acompaña al gráfico de la Figura 5.16 las cláusulas que definen los operadores primitivos "`pri_inf/3`" y "`pri_sup/3`", diseñados para devolver el *ínfimo* y *supremo* de dos elementos. Adver-




```
member(bottom).   member(alpha).
member(beta).     member(gamma).
member(top).
members([bottom,alpha,beta,gamma,top]).
leq(bottom,X).    leq(X,X).   leq(alpha,gamma).
leq(beta,gamma). leq(X,top). l(X,X) :- !.
l(X,Y):-leq(X,Y),!. l(X,Y):-leq(X,Z),l(Z,Y).
and_godel(X,Y,Z) :- pri_inf(X,Y,Z).
or_godel(X,Y,Z) :- pri_sup(X,Y,Z).
agr_impl(X,Y,Z) :- (l(Y,X),!,Z=top ; Z=X).
pri_inf(X,Y,X):- l(X,Y), !.
pri_inf(X,Y,Y):- l(Y,X), !.
pri_inf(_,_,bottom).
pri_sup(X,Y,X):- l(Y,X), !.
pri_sup(X,Y,Y):- l(X,Y), !.
pri_sup(_,_,gamma).
```

Figura 5.16: Retículo finito y parcialmente ordenado $\mathcal{F}$, modelado en PROLOG

timos de los siguientes aspectos:

- Obsérvese que, dado que los grados de verdad α y β no son comparables, cualquier objetivo de la forma "`?- leq(alpha,beta).`" o "`?- leq(beta,alpha).`" siempre va a fallar.
- Un objetivo de la forma "`?- pri_inf(alpha,beta,X).`", en vez de fallar, producirá el resultado esperado "`X=bottom`".
- Obsérvese que la implementación del predicado "`pri_inf/3`" es obligatoria para codificar la definición general de "`and_godel/3`" (se puede razonar igualmente para "`pri_sup/3`" y "`or_godel/3`").

Algunos ejemplos

Este subconjunto del lenguaje MALP es suficientemente amplio como para desarrollar un sencillo demostrador de teoremas difuso. Téngase en cuenta que nuestra herramienta se puede usar con diferentes retículos (no sólo el intervalo real [0,1]) con

un número finito de elementos –marcados con "`members`"– con un orden parcial o total entre ellos. Por tanto, podemos usar $\mathcal{FLOPER}$ para enumerar todo el conjunto de interpretaciones y modelos de las fórmulas difusas. Para ello, sólo se requiere un retículo L para construir automáticamente un programa compuesto por un conjunto de hechos de la forma "$i(\alpha)$ *with* α", para cada $\alpha \in L$. Sirva como ejemplo el programa MALP asociado al retículo $\mathcal{F}$ de la Figura 5.16, mostrado a continuación:

```
i(top)        with    top.
i(gamma)      with    gamma.
i(alpha)      with    alpha.
i(beta)       with    beta.
i(bottom)     with    bottom.
```

Una vez el retículo y el programa generado se han cargado en $\mathcal{FLOPER}$, la fórmula a evaluar se introduce como un objetivo del sistema siguiendo las siguientes convenciones:

- Si P es una variable proposicional en la fórmula original, entonces se denota como "`i(P)`" en el objetivo F.
- Si & es una conjunción de una determinada lógica "label" en la fórmula original, se denota como "&`label`" en el objetivo F.
- Para disyunciones, negaciones e implicaciones, usamos "|`label`", "@`no_label`" y "@`im_label`" en F.
- Para otros agregadores, usamos "@$_{\texttt{label}}$" en F.

En lo que sigue discutimos algunos ejemplos relacionados con el retículo mostrado en la Figura 5.16 y su programa MALP resultante.

En primer lugar, la ejecución del objetivo "`i(P)`" en $\mathcal{FLOPER}$ produce las cinco interpretaciones para `P` mostradas en la Figura 5.14. Por otra parte, consideremos ahora la fórmula proposicional $P \vee Q$, que se traduce en el objetivo MALP como "`(i(P) | i(Q))`". Este objetivo, lanzado a $\mathcal{FLOPER}$, genera un árbol[4] que representa en cada una de sus 25 hojas todo el conjunto de interpretaciones (9 de las cuales, las que están dentro de las cajas azules, son modelos) como se ve en la Figura 5.15. Véase también la Figura 5.17 asociada a la fórmula $P \wedge (P \vee P)$.

[4] Véase el capítulo 4 para una descripción más detallada de los árboles de ejecución.

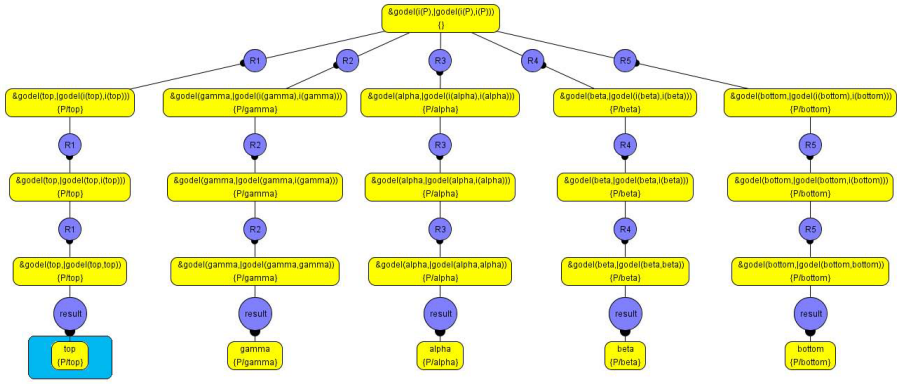


Figura 5.17: Árbol completo de la demostración de la fórmula $P \wedge (P \vee P)$ con 1 modelo entre 5 interpretaciones.

Considérese ahora la fórmula $P \wedge Q \rightarrow P \vee Q$, que se traduce en el objetivo MALP como "`(i(P) & i(Q)) @impl (i(P) | i(Q))`". Al interpretar $\mathcal{FLOPER}$ esta fórmula, el sistema devuelve la lista de respuestas mostrada en la Figura 5.18, cuyo grado máximo de verdad es "*top*", lo que demuestra que la fórmula es una tautología, como queríamos.

Indicaciones sobre las medidas de coste

Finalizamos esta subsección con algunos comentarios sobre las medidas de coste y la eficiencia del método. Dados un retículo L, una fórmula F y su árbol de demostración asociado T, definimos los siguientes valores:

- v es el número de variables distintas en F.
- v' es el número de ocurrencias (incluídas repeticiones) de variables en F.
- c es el número de conectivas en F.
- r es el número de elementos en el retículo L dados por el predicado "`members`".

Tenemos que:

- La anchura del árbol T, o el número total de interpretaciones de F, es r^v.
- El número de pasos admisibles llevados a cabo en una rama de T es v'.

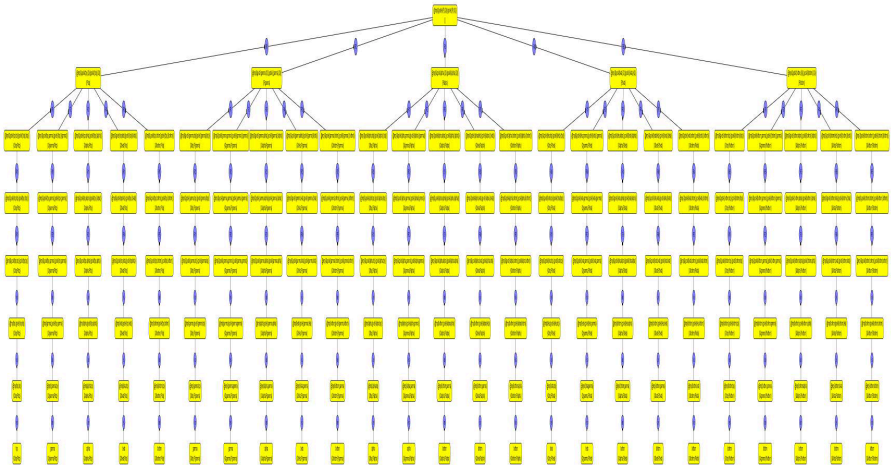


Figura 5.18: Árbol de demostración completo para la tautología $P \wedge Q \rightarrow P \vee Q$ (25 modelos).

- El número de pasos interpretativos llevados a cabo en una rama de T es c.
- La profundidad de T, o el número de pasos de computación (admisibles/interpretativas) para cada interpretación posible de F es $v' + c$.
- Una cota superior para el número de pasos admisibles en T es $|as| \leq (v' - v)r^v + \sum_{i=1}^{v} r^i$.
- Una cota superior para el número de pasos interpretativos en T es $|is| \leq cr^v$.
- Una cota superior para el número de pasos de computación (admisibles e interpretativos) es $|T| \leq (c + v' - v)r^v + \sum_{i=1}^{v} r^i$.

Volvamos ahora sobre la tautología $P \wedge Q \rightarrow P \vee Q$ para la que $\mathcal{FLOPER}$ muestra el conjunto de modelos visto en la Figura 5.18, y asumamos ahora una versión más general con la siguiente forma $P_1 \wedge \ldots \wedge P_n \rightarrow P_1 \vee \ldots \vee P_n$ para la que hemos estudiado su comportamiento en la tabla de la Figura 5.19. En el eje horizontal representamos el número n de variables proposicionales distintas que aparecen en

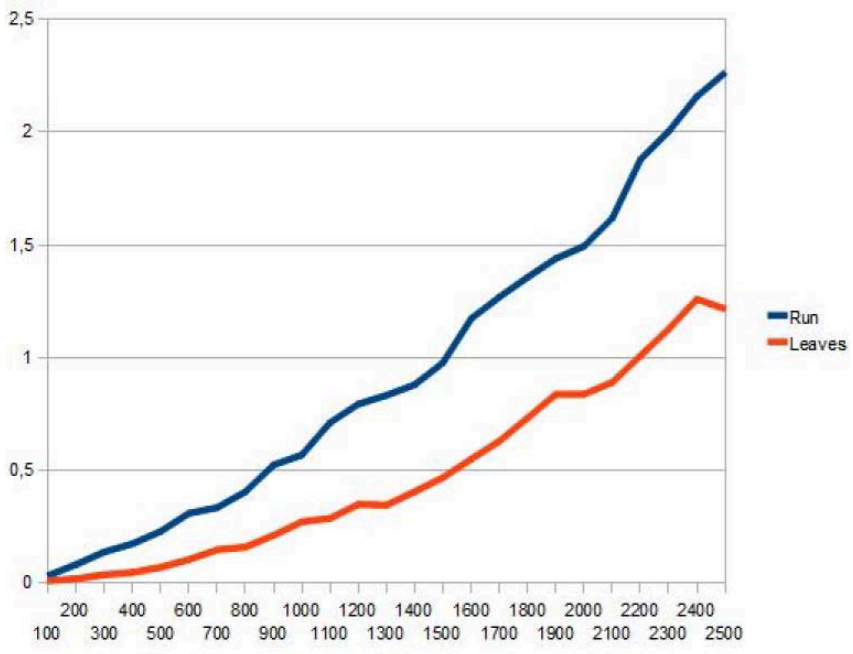


Figura 5.19: Comportamiento del método.

la fórmula, mientras que el eje vertical se refiere al número de segundos[5] necesarios para obtener el conjunto completo de interpretaciones (todas las cuales son modelos, en este caso) de la fórmula. Tanto la línea roja como la azul se refieren al método comentado a lo largo de esta sección, pero mientras que la línea roja indica que el árbol de derivación se ha producido realizando pasos admisibles e interpretativos de acuerdo a las Definiciones 3.2.1 y 3.2.6, respectivamente, la línea azul se refiere a la ejecución del código PROLOG obtenido tras compilar con $\mathcal{FLOPER}$ el programa MALP y el objetivo asociados a nuestra fórmula. Los resultados obtenidos en la figura muestran que nuestro método tiene un buen comportamiento en ambos casos, incluso para fórmulas con un gran número de variables proposicionales.

Por supuesto, el método no pretende competir con las técnicas SAT (que siempre son más rápidas y pueden tratar con fórmulas más complejas que contienen muchas más variables proposicionales), pero es importante señalar de nuevo que en nuestro

[5]Los benchmarks se han realizado en un ordenador con procesador Intel Core Duo, con 2 GB de RAM y sistema operativo Windows Vista.

caso, enfrentamos el problema de encontrar el conjunto completo de modelos para una fórmula dada en lugar de centrarnos sólo en su satisfacibilidad.

5.3. Overbooking en Cloud con MALP

La computación cloud es un reciente paradigma en el que los recursos computacionales se arrendan en Internet con un esquema de precios de pago por uso. Sus usuarios (organizaciones e individuos) pueden ajustar constantemente sus recursos cloud a sus necesidades en cada momento. Esta cualidad del paradigma cloud recibe el nombre de elasticidad [The30]. El núcleo de la infraestructura cloud son los centros de datos, que alojan cientos de miles de servidores, junto con almacén de datos y equipamiento de red, así como sistemas avanzados de refrigeración y distribución de la potencia [ZHa11]. Mediante tecnologías de virtualización, estos centros de datos (proveedores cloud) proveen de aplicaciones a múltiples usuarios en el mismo servidor físico, haciendo así eficiente el uso del hardware. En los centros de datos cloud, las aplicaciones de usuario se empaquetan como Máquinas Virtuales (VMs) [BDa03], que, en esencia, son implementaciones software de servidores compartidos en el tiempo en el hardware físico. Así, los usuarios pueden, ellos mismos o mediante software de administración automática, aumentar o disminuir el número de VMs alojados. Consecuentemente, es común entre los proveedores cloud pedir a los usuarios que especifiquen los límites superior e inferior del número de VMs que van a usar en la *solicitud de servicio* [RBa09], o tener ciertas reglas predefinidas para todos los usuarios, por ejemplo, 1-20 VMs por centro de datos para el mayor proveedor cloud, Amazon [Ama30].

Con respecto a los centros de datos (data centers), la elasticidad supone a largo plazo un problema de asignación, ya que el número exacto de máquinas virtuales asociado a cada usuario en cada momento es desconocido. Ejecutar pocas VMs repercute en un uso ineficiente del hardware de los centros de datos y disminuye los beneficios para los usuarios, mientras que ejecutar demasiadas VMs puede incidir en un bajo rendimiento y/o colisiones, mala experiencia de usuario, y consecuencias financieras negativas si se violan los Acuerdos de Nivel de Servicio (en inglés, SLA, de Service Level Agreement). Para lograr un buen equilibrio, los proveedores de cloud pueden hacer uso de mecanismos de *control de admisión* [BDE+12] para determinar cuándo una nueva petición de servicio debería admitirse o no en el centro de datos. En el trabajo [TT13b] se demostró cómo el *overbooking de recursos*, una técnica

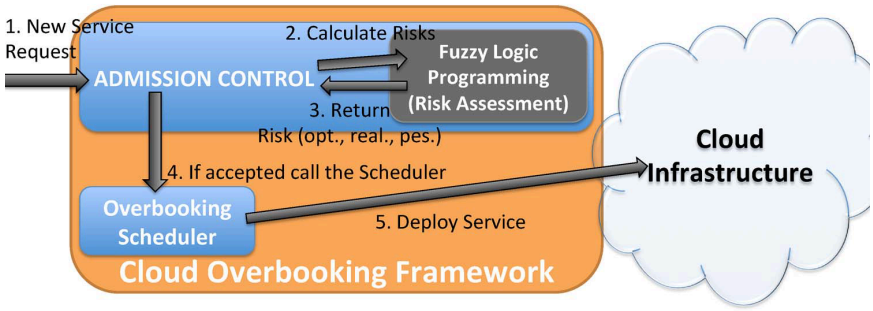


Figura 5.20: Esquema del sistema.

bien conocida del manejo de rentas de las aerolíneas y del multiplexado del ancho de banda, se puede usar para incrementar la utilización y el beneficio del proveedor, asumiendo sólo riesgos aceptables respecto a quedarse sin recursos hardware.

Un ejemplo del trabajo anterior en este área es un marco algorítmico [BDE+12] que usa la demanda efectiva de recursos cloud para estimar la capacidad física total requerida para realizar el overbooking, incluyendo la probabilidad de lanzar VMs adicionales en el futuro.

Sin embargo, evaluar el riesgo durante el control de admisión con respecto a la realización de overbooking de recursos está lejos de ser un problema trivial. El overbooking y los problemas de planificación asociados son problemas de empaquetado multi-dimensional que, normalmente, se resuelven mediante heurísticas. Tampoco está claro cómo balancear el impacto a corto y a largo plazo al decidir la aceptación o no de un trabajo. Más aún, el control de admisión está asociado a diversas incertidumbres, incluyendo el limitado conocimiento de la carga futura, los posibles efectos secundarios de asignar una VM particular, y el impacto exacto en aplicaciones de escasez potencial de recursos. Basándonos en estas propiedades del control de admisión, hemos propuesto una aproximación difusa al problema [VMTT15].

5.3.1. Gestor lógico difuso de *overbooking*

Con respecto a las formulaciones difusas del problema del control de admisión, en esta subsección presentamos un método flexible que ha sido ya implementado en MALP a través de $\mathcal{FLOPER}$. En este problema reutilizamos las definiciones estándar para las conjunciones, disyunciones e implicaciones en el retículo de los números reales

en el intervalo unidad $[0, 1]$, correspondientes a la *lógica de Łukasiewicz*, de *Gödel* y del *Producto*, cada una con diferentes capacidades para definir escenarios *pesimistas*, *optimistas* y *realistas*, respectivamente.

En esta aplicación utilizamos una versión refinada de dicho retículo, dado que identificamos la noción de grado de verdad con el de "riesgo de overbooking a lo largo de un periodo de tiempo". Esto significa que en lugar de valores puntuales, nuestro programa manipula listas de números reales como grados de verdad[6] tras analizar el comportamiento de las curvas que representan los *recursos libres*, *no solicitados* y *solicitados* (en CPU, memoria y red). Por ejemplo, si la expresión "$\&_{\mathrm{P}}(x, y) \triangleq x{*}y$" se refiere a la *lógica del Producto* para pares de valores, su versión extendida para pares de listas de valores corresponde a "$\&_{\mathrm{P}}([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \triangleq [x_1 * y_1, \ldots, x_n * y_n]$". En nuestra aplicación esta conectiva se puede definir recursivamente con el siguiente código:

```
and_prod([],[],[]).
and_prod([X|LX],[Y|LY],[Z|LZ]):- Z is X*Y, and_prod(LX,LY,LZ).
```

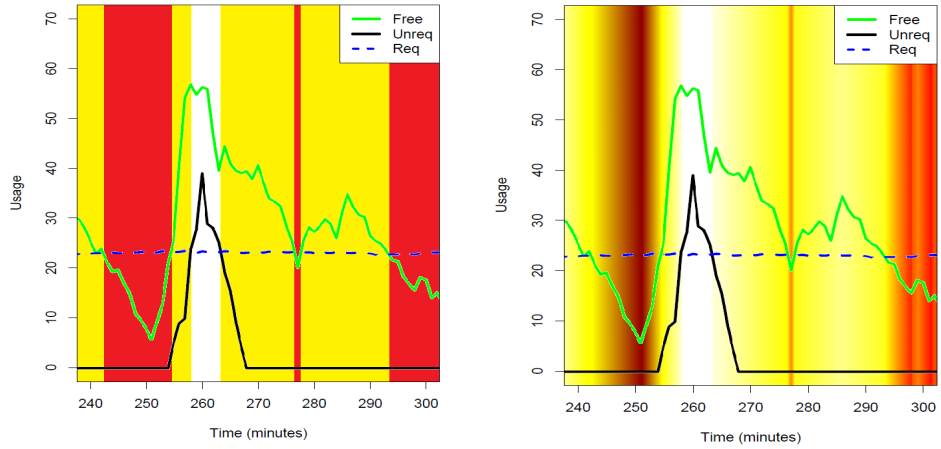


Figura 5.21: Diferentes opciones para estimar el riesgo.

En el retículo hemos implementado también versiones extendidas sobre listas para las demás conectivas vistas en el Capítulo 3, así como otras conectivas como *@append* (para concatenar dos listas de números), *@show* (que se describe más adelante),

[6] A veces acompañados de anotaciones como *max*, *avg*, *peak* por motivos de legibilidad.

@very y *@ approx* (donde recordamos que $@very(x) = x^2$ y $@approx(x) = \sqrt{x}$) conocidos como *modificadores lingüísticos*. Estos modificadores son útiles para refinar la forma más pesimista u optimista de las respuestas producidas por la aplicación bajo este escenario incierto.

Gracias al gran poder expresivo del retículo previo, es posible diseñar un programa MALP compuesto por muy pocas reglas. La primera de ellas recibe como parámetros de entrada tres listas que representan las curvas asociadas a los recursos *libres*, *no solicitados* y *solicitados*, así como un cuarto argumento que indica qué recurso, o campo (`Field`) (CPU, red o memoria) se debe considerar:

```
risk([F|Free],[U|Unreq],[R|Req],Field)<-
            @append(combine(F,U,R), risk(Free,Unreq,Req,Field))
```

Esta definición del predicado "`risk`" produce como grado de verdad una lista de números obtenidos al contrastar las curvas de entrada "`Free`", "`Unreq`" y "`Req`". Esta evaluación se realiza recursivamente llamando al predicado "`combine`" con tres valores concretos cada vez (de cada una de las curvas).

La Figura 5.21 representa gráficamente dos alternativas para realizar este contraste. En ella, el color de fondo representa el nivel de riesgo, u *output*, que cambia entre tonalidades blanca, amarilla y roja, de menor a mayor riesgo en cada instante. Una versión preliminar del predicado "`combine`" asociado a la gráfica izquierda en la Figura 5.21 se puede definir como

```
combine(Free,Unreq,Req) <- (Req>=Free & [1])|(Req>Unreq & [0.5])
```

que, en esencia, da riesgo 1 (banda roja) cuando los recursos solicitados exceden los libres, 0.5 (banda amarilla) si se encuentran entre los libres y los no solicitados, y 0 (banda blanca) en otro caso. Además de esto, hemos implementado una versión más sofisticada basada en la interpolación lineal (gráfico derecho en la Figura 5.21) de acuerdo a la siguiente fórmula:

```
inter(Req,Free,Unreq) = (Req-Unreq)/(Free-Unreq)
```

Esta fórmula devuelve riesgo 0 cuando el valor de los recursos solicitados está por debajo del de los no solicitados, un riesgo en [0,1] (las notalidades varían del blanco al rojo a través del amarillo conforme crece el riesgo) si el punto de los recursos solicitados está entre los otros dos valores, y riesgo superior a 1 (las tonalidades varían del rojo al negro conforme crece el riesgo) si la cantidad de recursos solicitados es mayor que la de los libres. En este caso decimos que hay un *pico*. Considerar los

picos nos permite mejorar la evaluación final del riesgo teniendo en cuenta el impacto de cada uno en el rendimiento del sistema.

Se invoca este programa mediante una llamada al predicado "`main`", con los parámetros apropiados:

```
main(Free,Unreq,Req,Field) <- @show(risk(Free,Unreq,Req,Field))
```

Esta regla hace uso de la conectiva "`@show`", que recibe el grado de verdad (una lista de números) producida por "`risk`" y devuelve un nuevo grado de verdad como una lista de la siguiente forma:

```
[    avg(n1), min(n2), max(n3),
over([peak(h1,l1,a1),...,peak(hi,li,ai)]),
opt(n4), real(n5), pes(n6)              ]
```

Aquí, las etiquetas "`avg`", "`min`" y "`max`" se refieren, respectivamente, a los valores medio (n_1), mínimo (n_2) y máximo (n_3), de la lista de entrada; "`over`" devuelve la lista de picos (cada uno representado por su altura máxima (h_j), longitud (l_j) y área (a_j)) y, finalmente, las etiquetas "`opt`", "`real`" y "`pes`" dan las estimaciones optimista (n_4), realista (n_5) y pesimista (n_6) –basadas en los elementos anteriores– referidas al riesgo inherente de aceptar la tarea indicada. Estas estimaciones se producen combinando la medida de la media aritmética (modulada con las conectivas @*approx* y @*very*, según nos refiramos a los casos pesimista u optimista, respectivamente) con la disyunción de todos los picos mediante distintas versiones del operador de disyunción. Este operador se modela de acuerdo a la lógica de Łukasiewicz, del Producto o de Gödel donde se da que $\forall x, y \in [0,1], x|_{\mathtt{L}} y \geq x|_{\mathtt{P}} y \geq x|_{\mathtt{G}} z$. Esto justifica una vez más el poder de la lógica difusa y los recursos expresivos de MALP para modelar escenarios pesimistas, realistas y optimistas. Por ejemplo, cuando introducimos el siguiente objetivo en $\mathcal{FLOPER}$:

```
main([50,20,40,73,99],[25,10,2,51,40],[20,23,45,60,49],cpu)
```

El sistema lo resuelve generando una lista que representa el grado de verdad final asociado a la consulta, con la siguiente forma:

```
[  avg(0.9300780175180026), min(0), max(1.3),
   over([peak(2,1.3,0.27078683857231306)]),
   opt(0.8045594490364215), real(0.9015891243354114), pes(1)]
```

En la Figura 5.22 mostramos una captura de pantalla de $\mathcal{FLOPER}$ en la que se ejecuta el objetivo anterior. En la ventana principal se observa (la parte superior

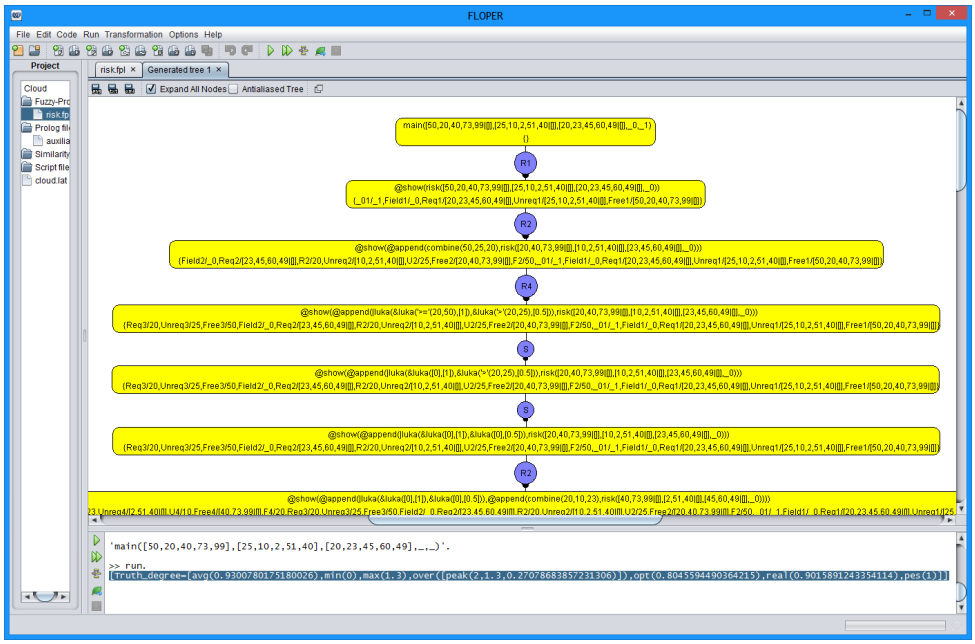


Figura 5.22: $\mathcal{FLOPER}$ ejecutando nuestra aplicación.

de) el árbol de derivación para este objetivo que, en esencia, consiste en un sistema de transición de estados donde cada estado se colorea de amarillo, y las transiciones aparecen como círculos azulos, de modo que el estado inicial se corresponde con el objetivo inicial, como raíz del árbol, y el estado final (no visible en esta figura) contiene el grado de verdad finalmente asociado a la query. En nuestro caso, esta solución corresponde al texto marcado en azul en el cuadro inferior de la captura.

5.3.2. Experimentos

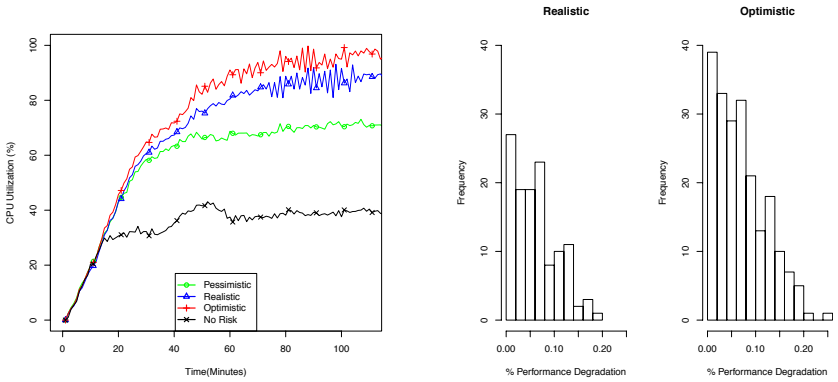
Para evaluar nuestra propuesta, la valoración difusa del riesgo se incluye en el marco de [TT13b], que recoge una técnica sencilla de control de admisión. Ahora el control de admisión emplea esta información para tomar decisiones sobre la aceptación o rechazo de servicio al realizar overbooking de recursos. Se ha reutilizado el simulador implementado en [TT13b] con objeto de simular la infraestructura cloud y la carga de trabajo.

La infraestructura cloud simulada para probar los diferentes evaluadores del ries-

go consiste en 16 nodos, cada uno de 32 núcleos. Hemos considerado cuatro tipos de máquinas virtuales según su tamaño (S, M, L y XL), de un modo parecido al usado por Amazon [Ama30], en el que cada tipo de máquina dobla la capacidad de la anterior. Partimos de 1 CPU y 1.7GB de memoria para el tipo S. Estas máquinas virtuales simulan la ejecución de una carga de trabajo dinámica compuesta de diferentes tipos de aplicaciones (algunas de las cuales presentan un comportamiento estable y otras explosivo), afinadas mediante herramientas de monitorización tras ejecutarse las aplicaciones reales. La carga de trabajo es una mezcla de aplicaciones y sigue una distribución de Poisson para las tasas de envío. Para más detalles sobre el banco de pruebas y la generación de carga de trabajo, véase [TT13b]. Con esa carga de trabajo, la evaluación del rendimiento se ha realizado generando peticiones de servicio de acuerdo a dicha distribución de Poisson. Entonces, las peticiones aceptadas (por el control de admisión) se planifican y ejecutan en los 16 nodos. Durante la ejecución hemos medido la *utilización* y la *escasez de recursos*.

Nos hemos centrado en medir el impacto del riesgo tomado por el control de admisión al realizar overbooking de recursos en los centros de datos. Los diferentes riesgos proporcionados por el motor lógico difuso se comparan entre sí y también con el caso base, en el que no se realiza overbooking y, por tanto, no se asumen riesgos. Las valoraciones se etiquetan, de menor riesgo a mayor riesgo, como *"Pessimistic"*, *"Realistic"* y *"Optimistic"* –mapeándolos a los valores respectivos calculados por el motor lógico difuso con esos nombres. El caso base se etiqueta como *"No Risk"*.

La Figura 5.23 (a) muestra la utilización de recursos alcanzada para cada toma de riesgo por parte del control de admisión. Resulta evidente que cuanto mayor riesgo se asuma, mayor es la utilización de recursos. Sin embargo, esto puede tener un impacto negativo con respecto a la escasez de recursos si se sobrepasa la capacidad total, no sólo de acuerdo a la utilización de los centros de datos sino también respecto a cada nodo individual del sistema. Debido a esto, la Figura 5.23 (b) muestra los histogramas relativos al número de veces que uno de los nodos ha sobrepasado su capacidad, y el impacto en su rendimiento (que, en un caso extremo, puede llevar a una violación de los acuerdos de nivel de servicio, SLA). El eje x representa la degradación de rendimiento experimentada cuando se sobrepasa la capacidad total en (al menos) uno de los nodos. Así, cuanto menores sean las barras, menos situaciones de riesgo existen y mejor es el rendimiento. Se desea, por tanto, que se mantengan lo más cerca posible del 0 –con menor degradación del redimiento y mayores posibilidades de resolver ésta. Como se muestra en la Figura 5.23 (a), la capacidad estructural total



(a) Utilización del data center. (b) Sobrecarga de capacidad (a nivel de nodo).

Figura 5.23: Utilización de recursos y comparación de la valoración del riesgo.

	Utilización media	**Sobrecarga de la capacidad del nodo (%)**	**Sobrecarga de la capacidad agregada (%)**
No Risk	38.9 % (1)	0	0
Pessimistic	69.1 % (1.78)	0	0
Realistic	84.6 % (2.17)	6.99	0.43
Optimistic	92.5 % (2.38)	11.88	0.84

Cuadro 5.1: Resumen del rendimiento (Figura 5.23).

no se sobrepasa. Esto significa que la *migración de VM* –reubicación de máquinas virtuales de nodos sobrecargados a otros que todavía tienen capacidad disponible– se puede usar para decrementar el riesgo. De este modo se pueden evitar algunas situaciones de sobrecarga , como se ha propuesto en Beloglazov et al. [BB13].

Finalmente, la Tabla 5.1 destaca la mejora obtenida gracias al overbooking de recursos (hasta 2.38 veces) y el coste que esto conlleva. El criterio pesimista ("*Pessimistic*") muestra la menor mejora pero, a cambio, no produce degradación del rendimiento, al contrario que las otras dos técnicas que mejoran más la utilización a costa de mayores degradaciones del rendimiento que pueden llevar a un agotamiento de recursos. Para los casos optimista ("*Optimistic*") y realista ("*Realistic*") la capacidad total de un solo nodo se ha sobrepasado alrededor del 6 % y 12 % del tiempo, respectivamente. Sin embargo, el impacto total en el rendimiento no es significativo (menos del 1 %) –calculado como el porcentaje del tiempo en que la capacidad se sobrepasa en un nodo, ponderado con la cantidad de capacidad sobrepasada.

5.4. Problema de asignación

Como se ha comentado en la sección anterior, la computación cloud provee recursos a través de Internet respondiendo generalmente a un esquema de pago por uso. La principal ventaja de este modelo radica en la posibilidad tanto por parte de los clientes como de los proveedores de ajustar la asignación de recursos en base a las necesidades presentes, característica esta que se conoce como *elasticidad* [The30]. A través de máquinas virtuales [BDa03] los centros de datos proveen diferentes aplicaciones para múltiples usuarios de manera eficiente. Estas máquinas virtuales (VMs por su acrónimo en inglés) se pueden ver como implementaciones de servidores que comparten en el tiempo el hardware físico.

Pese a las ventajas que ofrece la computacion cloud, la planificación de la capacidad de los centros de datos se ha convertido en un problema para los proveedores cloud. En la sección anterior describimos una solución difusa basada en overbooking de recursos para uno de los aspectos de este problema, conocido como el problema del control de admisión. Sin embargo, una vez se ha aceptado una máquina virtual, ésta debe ser desplegada en servidores específicos –incluso mapeada a cores específicos (pcpus) dentro de los servidores. El modo de realizar esta planificación es muy importante, ya que tiene un gran impacto en el rendimiento de las aplicaciones desplegadas, especialmente en entornos donde se ha realizado overbooking. Por ejemplo, en el estudio [TT14b] se muestra que la sola asignación o desasignación de algunas máquinas virtuales a algunos pcpus, se logró una diferenciación en la calidad de servicio (QoS por sus siglas en inglés), lo que resultó en una variación del nivel de rendimiento dependiendo del nivel empleado de QoS.

En el trabajo [VTMT13], recogido en esta sección, nuestro objetivo es mejorar el *emplazamiento intra-servidor* mediante un sistema que seleccione qué cores deben asignarse a cada VM, de modo que las interferencias entre VMs diferentes se reduzcan. Obsérvese que actualmente este es un tópico de investigación relevante, conocido como *vecino ruidoso* [VVB13]. Algunos ejemplos de esto son decidir qué VMs pueden o no pueden ser asignadas próximas [DK14], o detectar la VM antagonista [ZTH+13]. Este es un problema multi-dimensional (pues las VMs no utilizan sólo tiempo de CPU, sino también memoria y red) resuelto comúnmente mediante heurísticas. Sin embargo, debido a la complejidad del problema, y debido de todas las incertidumbres del sistema, como el estado futuro de los cores o las necesidades futuras de las aplicaciones, hemos decidido emplear una aproximación basada en programación lógica difusa, de forma similar a como hicimos para el problema de

control de admisión [VTMT13], que hemos repasado en la sección anterior. La idea principal del nuevo planificador difuso consiste en considerar tanto del estado de los cores (esto es, si el core es intensivo en cpu, memoria y/o red) como las necesidades esperadas de la VM, y así encontrar los cores con mayor afinidad a la VM entrante.

Los resultados experimentales demuestran que este sistema reduce las interferencias entre VMs. Gracias a esto, las aplicaciones pueden mantener su rendimiento todo el tiempo, lo que resulta en un rendimiento global más predecible y fiable. Esto, a su vez, permite a los operadores del centro de datos incrementar sus ingresos y hacer un uso más eficiente de sus recursos.

5.4.1. Motivación

Actualmente el número de centros de datos, así como la cantidad de energía que consumen, está creciendo a gran velocidad. Resulta significativo que, para 2020, el consumo acumulado de energía de los centros de datos consistirá en aproximadamente el 8 % de la energía mundialmente consumida [GCWK12]. Sin embargo, como hemos destacado, los recursos de los centros de datos cloud no están siendo empleados de modo eficiente. Los ratios de utilización reportados están por debajo de la mitad de la capacidad total disponible [RTG+12], lo que lleva no sólo a un importante desperdicio de energía, sino también a pérdidas de ingresos del lado de los proveedores. El problema de la utilización deriva principalmente de las dificultades en la planificación de la capacidad cuando se desconoce el número de recursos requeridos por los usuarios en el futuro. Si se sobreestima esta cantidad, el uso de recursos de los centros de datos será demasiado bajo, lo que, en consecuencia, impactará en los beneficios del proveedor cloud. Por el contrario, si se subestima esta cantidad puede llevar a la degradación del rendimiento e, incluso, colisiones, lo que implica el pago de penalizaciones y el abandono del servicio por parte de los usuarios. Por tanto, los proveedores cloud son (en su mayoría) conservadores en lo que respecta a asegurar el rendimiento de las aplicaciones.

Una vez tomada la decisión de aceptación de la VM, el siguiente paso consiste en decidir dónde ubicarla mediante el estudio de la idoneidad de cada nodo físico. Anteriores aproximaciones a este problema se basaban bien en el planificador kvm para compartir la cpu física entre las VMs [TT14a], o bien en el uso de la funcionalidad de *pinning* de kvm para proveer cierto aislamiento entre VMs, pero sólo para propósitos de diferenciación de QoS [TT14b]. A consecuencia de ello, algunas VMs podían interferir con otras, lo que tenía un impacto en el rendimiento global

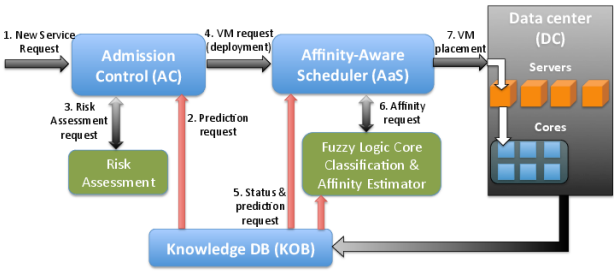


Figura 5.24: Esquema del overbooking

conocido como *problema del vecino ruidoso* [VVB13].

Ha habido muchos intentos en la literatura de atajar este problema, pero ninguno de ellos ha proporcionado una solución completa desde las peticiones de usuarios a la ubicación y co-locación de las VMs. Por ejemplo, He et al. [HGGG12] presentan un modelo probabilístico multivariable basado en reglas de (anti)afinidad entre VMs para evitar repetir una combinación que diera mal rendimiento. De forma similar, Meng et al. [MIK+10] proponen una aproximación conjunta de provisión de VM basada en estimaciones de los requisitos agregados de las VMs, que ubica y consolida VMs. Sin embargo, estos trabajos consideran únicamente el uso de CPU y asumen prediciones precisas respecto a la carga de trabajo futura. Otro ejemplo que emplea métodos de control adaptativos para detectar y resolver malas co-locaciones corresponde al trabajo de Zhang et al. [ZTH+13]. Estos autores usan la métrica de rendimiento CPI para detectar comportamientos anómalos y la aplicación que con más probabilidad los esté ocasionando, y a partir de ahí realizan migraciones para resolver el problema cuando sea posible.

En este trabajo nos centramos en evitar interferencias entre VMs co-locadas, esto es, en cómo ubicar las VMs dentro de los servidores de forma que sus rendimientos reciban el menor impacto posible. Para ello, hemos propuesto un planificador asistido por un sistema lógico difuso que tiene en cuenta las afinidades, y lo hemos incorporado al marco descrito en [TT14a], detallado conceptualmente en la Figura 5.24. En consecuencia, este marco consta actualmente de: (1) un módulo difuso de control de admisión (AC) basado en riesgo, descrito en la Sección 5.3, que evalúa el riesgo de aceptar una nueva VM en el sistema, y que ha sido presentado en [VTMT13, TT13a], y que, además, reajusta autónomamente el nivel aceptable de riesgo sobre el tiempo mediante el uso de un conjunto distribuido de controladores PID [TT14a]; (2) una

base de conocimiento (KOB) a cargo de monitorizar el estado del sistema y de las VMs; y (3) el nuevo planificador asistido por lógica difusa consciente de afinidades (AaFS) a cargo de asignar cores físicos a las VMs aceptadas. Como el sistema se ha diseñado con propósitos de overbooking, necesita encontrar buenos VM vecinos para compartir el pcpu cuando no haya suficientes pcpus disponibles. Para este fin, los cores son clasificados de manera continua dependiendo de su comportamiento reciente (con respecto al uso de cpu, memoria y red). Entonces, el módulo difuso busca (un grupo de) cores con un nivel de afinidad suficientemente alto para la VM entrante, y finalmente mapea la VM al pcpu propuesto.

Nótese que tanto el AC como el AaFS emplean la información proporcionada por KOB, ya sea para evaluar el impacto de aceptar nuevas peticiones como para clasificar y medir las afinidades entre VMs y cores. A continuación explicamos cómo se calculan estas clasificaciones y afinidades difusas.

5.4.2. Solución basada en MALP

Nuestra solución basada en MALP para el problema de la ubicación consiste en un sistema difuso que provee tres llamadas:

- *Initialize*: Mediante esta llamada el usuario establece el número de cores y la distancia entre ellos. Sólo se requiere el uso de esta llamada una vez para inicializar las principales propiedades del sistema.
- *Actualize*: Por medio de esta llamada, el usuario introduce los datos actualizados sobre el uso de recursos de cada core. Puede realizarse esta llamada tantas veces como sea necesario, y tan pronto como se reciban nuevos datos del monitor.
- *Assign*: Esta llamada es la parte central del sistema, pues responde con el core o grupo de cores más adecuados para la ubicación de una máquina virtual.

Antes de continuar, realizamos aquí algunas consideraciones necesarias. En el sistema, un core se define por un número. El número total de cores es un parámetro de la llamada *initialize*. Con objeto de producir respuestas fiables en las sucesivas llamadas a *assign*, es fundamental la noción de distancia entre cores. *Grosso modo*, este concepto indica el coste en términos de tiempo que conlleva la comunicación entre cada par de cores. La distancia es uno de los parámertos de la llamada *initialize*; en particular, el usuario proporciona un fichero (o ruta) de un programa Prolog

que define el predicado `distance/3` (esto es, `distance(`C_1`, `C_2`, `n`)`, donde C_1 y C_2 son cores y n es la distancia entre ellos). El programa PROLOG que indica estas distancias puede ser tan simple como una lista de hechos `distance`, o tan complejo como un programa que calcula distancias basándose en los números de cada core. En concreto, nuestro sistema contiene 32 cores en 4 grupos de 8 cores. La distancia entre cores del mismo grupo es de 10, mientras que la distancia entre cores de distintos grupos es 16. Otra forma de contemplar esta situación es que el coste, en términos de tiempo, de comunicar cores de diferentes grupos se toma como un 60 % superior al de comunicar cores del mismo grupo. El motivo de esta diferencia tiene que ver con detalles de la implementación física, como la disposición de los bancos de memoria o la arquitectura del sistema.

La llamada a *actualize* incluye un único parámetro, que es la ruta del fichero que incluye los nuevos datos. Este fichero indica en texto plano la utilización de recursos de todos los cores del sistema en forma de tabla. La primera línea del fichero es `CoreName NodeName CPUUsage MemUsage NetUsage TimeStamp`, de forma que cada uno de estos términos corresponde a una columna de la tabla. Cada línea del fichero indica, en este orden, un core (identificado por un número), un número de nodo, el uso de CPU, de memoria y de red, y el sello de tiempo (time stamp) correspondiente a esos usos. Una vez nuestro sistema lee esta información, pasa a actualizar su base de datos interna. Concretamente, el sistema mantiene información sobre la intensidad en el uso de un recurso (CPU, memoria y red) así como su "burstiness", término que puede traducirse libremente como "explosividad", y que se refiere a la variabilidad en el uso de un recurso en breves períodos de tiempo.

El valor para la *intensidad* de cada recurso corresponde a una media móvil (donde el número de valores tenidos en consideración se especifica como un parámetro de la llamada *initialize*), ya que esta es una medida estable para lo que se entiende que es una noción estable. De hecho, la intensidad en el uso de un recurso debería ser el valor más predecible de un core. La irregularidad en el uso de un recurso, por otro lado, está relacionada con la noción de *burstiness* mencionada anteriormente. Este valor se computa también como una media móvil, en este caso de la diferencia entre el uso efectivamente medido y la intensidad (computada previamente). También es necesario el uso de una media móvil para la *burstiness* porque esta noción se considera una característica de un core, dada por el tipo de máquinas virtuales alojadas en él, de modo que usamos la media móvil para mantener esta característica estable a lo largo del tiempo.

Mediante la llamada a *actualize*, los predicados difusos *cpuIntensive*, *memIntensive*, *netIntensive*, *cpuBursty*, *memBursty* y *netBursty* proporcionan información actualizada sobre la *intensidad* y *burstiness* de cada recurso y core. Más adelante, estos predicados son usados en el cálculo de los cores más adecuados para alojar las nuevas máquinas virtuales.

Las llamadas anteriores, *initialize* y *actualize*, sirven al propósito de establecer los valores en la base de datos del sistema. Sin embargo, la llamada fundamental es *assign*, para la cual se han proporcionado tres versiones (*assign*, *assign2* y *assign3*). Esta llamada está diseñada para devolver un grupo de cores y su idoneidad, en términos de grados de verdad. La máquina virtual a ser ubicada se pasa como parámetro de esta llamada en forma de un término MALP `vm/3`, donde cada término es una lista de utilización prevista de recursos (donde dichos recursos son CPU, memoria y red) en un período de tiempo. Otros parámetros para *assign* (todas sus versiones) son el número de cores a asignar y la lista de cores descartados. Con este último parámetro se pretende implementar tres estándares de servicio para máquinas virtuales, esto es, algunas máquinas virtuales podrán ejecutarse en cores exclusivos que estarán vedados a otras máquinas virtuales, de manera que se mejora el rendimiento de los primeros. Esta implementación es lo bastante flexible como para tratar con múltiples clases de privilegios para máquinas virtuales.

A continuación yuxtaponemos los tres predicados para cada versión de *assign* con objeto de compararlos.

```
assign (Vm, Nc, Dc, Distance,  Group).
assign2(Vm, Nc, Dc, Threshold, Group).
assign3(Vm, Nc, Dc, Threshold, Group).
```

Ya hemos detallado el significado de los parámetros comunes a las tres versiones, que son `Vm` (*Máquina virtual*), `Nc` (*Número de cores*) and `Dc` (*Cores descartados*). También es común el último parámetro, `Group`. En él se devuelve el grupo de cores –en forma de una lista de números– que el sistema considera más adecuado como respuesta a la llamada. Considérese, por último, el parámetro anterior a éste. En la primera versión de `assign`, este parámetro (`Distance`) se usa como un límite –dado por el usuario– para la suma de distancias entre los cores de la solución. El sistema emplea este límite para cortar ramas en el árbol de búsqueda tan pronto como los cores considerados en ellas se hallen demasiado separados para que se les pueda considerar una buena solución.

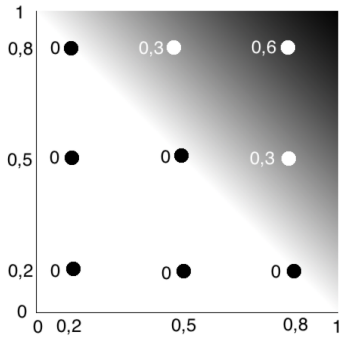


Figura 5.25: Grado de verdad del agregador *over*

Aún siendo útil este parámetro, también se pueden considerar otras opciones para elegir un buen grupo. Por ejemplo, es posible que un grupo ligeramente por encima del límite de distancia sea más adecuado que todos los grupos por debajo de él (ya sea esto porque incluye los cores más libres de cada cluster o por cualquier otra razón). Para contemplar esta posibilidad se han introducido las versiones segunda y tercera de esta llamada. Ambas aceptan, en lugar del parámetro `Distance`, un parámetro *Threshold* donde el usuario indica un umbral para el grado de verdad de la solución. Este umbral es un grado de verdad usado por el sistema para cortar ramas en cuanto su idoneidad caiga por debajo de él. Ambas llamadas otorgan a este parámetro el mismo significado.

La única diferencia entre *assign2* y *assign3* es el modo en que buscan la solución. Mientras que *assign2* devuelve el primer grupo por encima del umbral –esto es, emplea el umbral como un límite de la idoneidad–, *assign3* conserva la intención de la primera versión de ofrecer el grupo de cores globalmente más adecuado, y utiliza el umbral para ganar eficiencia (del mismo modo que se empleaba el parámetro `distance`). Nótese que *assign2* es la llamada más rápida y, si bien no devuelve necesariamente la mejor solución, responde con un grupo considerado "suficientemente bueno". Por otra parte, *assign* y *assign3* procuran devolver la mejor solución posible buscando entre un gran número de ellas, empleando criterios distintos para cortar ramas (uno emplea la suma de las distancias y el otro, el umbral de idoneidad).

Obsérvese que estas llamadas son puramente difusas, puesto que la respuesta (el grupo) se acompaña de un grado de verdad. Internamente también son implementadas en términos de otros predicados difusos. Particularmente, los predi-

cados `trygroup` (para la versión basada en distancias) y `trygroupThr` (para la versión basada en umbralización sobre el grado de verdad), se basan ambos en `suitabilityOfGroup`, que se satisface si un grupo es adecuado, lo que es nuevamente una noción difusa de satisfacción. Este predicado llama a los previamente mencionados *cpuIntensive*, *memIntensive*, *netIntensive*, *cpuBursty*, *memBursty* y *netBursty* (que, por mor de la brevedad, llamamos de ahora en adelante *ci*, *mi*, *ni*, *cb*, *mb* y *nb*, respectivamente) del siguiente modo:

$$
\begin{aligned}
&suitabilityOfGroup(vm, gr) <- \\
&\ \&_{prod}(\\
&\quad @_{not}(ci(vm)\ @_{over}\ ci(gr))\ \&_{luka} \\
&\quad @_{not}(ni(vm)\ @_{over}\ ni(gr))\ \&_{luka} \\
&\quad @_{very}(@_{not}(mi(vm)\ @_{over}\ mi(gr))) \\
&\ , \\
&\quad @_{not}(@_{aver}(ci(vm), cb(vm))\ @_{over}\ @_{aver}(ci(gr), cb(gr)))\ \&_{godel} \\
&\quad @_{not}(@_{aver}(ni(vm), nb(vm))\ @_{over}\ @_{aver}(ni(gr), nb(gr)))\ \&_{godel} \\
&\quad @_{very}(@_{not}(@_{aver}(mi(vm), mb(vm))\ @_{over}\ @_{aver}(mi(gr), mb(gr)))) \\
&\ )
\end{aligned}
$$

La expresión anterior se puede analizar dividiéndola en dos partes. La primera trata únicamente con la intensidad. Para cada recurso, se computa la intensidad en su uso por parte de la máquina virtual y del grupo de cores, y se combinan ambas mediante la conectiva *over*, que describimos más adelante. Obsérvese también que, dado que la memoria es un recurso especialmente sensible, modulamos su valor mediante el agregador *very* (lo cual implica que requerimos mayores valores para la memoria que para otros recursos). Los grados de verdad tomados en esta parte de la fórmula se combinan con la conjunción de *Łukasiewicz*, una conjunción pesimista para asegurar la estabilidad del sistema. La segunda parte de `suitabilitiOfGroup` trata el *burstiness* del grupo. Realizamos una media aritmética del *burstiness* y de la intensidad para obtener el peor escenario posible (donde coincide un pico en el uso de un recurso tanto en la máquina virtual como en el grupo de cores) y continuamos como en la primera parte. En esta parte se emplea la conjunción de *Gödel* para combinar los grados de verdad, pues esta conjunción, que es más optimista, deja un poco de espacio a la colisión de picos, pues esto no se considera una amenaza grave para el sistema. Finalmente, el resultado de ambas partes se combian mediante la conjunción del *Producto*, una conjunción intermedia –realista– para obtener el grado de verdad final del predicado.

Con respecto al agregador *over*, éste está conectado con la noción de sobrecarga. De hecho, la expresión $x@_{over}y$, donde x es la intensidad en el uso de un recurso por parte de la máquina virtual, e y es la intensidad en el uso de ese recurso por parte de un core, se interpreta como 0 si el core tiene espacio suficiente para la máquina virtual. De otro modo, la expresión devuelve un valor tan grande como grave sea la sobrecarga (potencialmente 1 si el core está al límite de capacidad y la máquina virtual pide todos los recursos). La Figura 5.25 representa el uso de este agregador con tres máquinas virtuales y tres cores.

En el siguiente apartado hacemos uso de la aplicación que hemos detallado, e ilustramos los resultados experimentales que confirman las ventajas de usar un sistema lógico difuso.

5.4.3. Experimentos

Presentamos aquí los experimentos realizados para evaluar nuestra propuesta de un planificador difuso basado en afinidad en un entorno con overbooking junto con el módulo presentado en la sección anterior. En primer lugar describimos el conjunto de tests y la carga de trabajo empleada. A continuación, evaluamos mediante un conjunto de experimentos el planificador propuesto con respecto a la utilización alcanzada y el rendimiento de las aplicaciones.

Banco de pruebas y carga de trabajo

Los experimentos se han realizado en dos máquinas. Una de ellas alberga las aplicaciones y la otra genera la carga de trabajo. Ambas están conectadas con un ancho de banda de 100 MB. La primera máquina es un servidor que consta de un total de 32 cores (AMD Opteron^{TM} 6272 a 2.1 GHz) y 56 GB de memoria, donde las aplicaciones se despliegan dentro de máquinas virtuales. La otra máquina es un ordenador de 4 núcleos (Intel Core^{TM} i5 processor a 3.4 GHz) con 16 GB de memoria.

Los diferentes tipos de VMs se mezclan con objeto de generar una carga de trabajo representativa en cloud [RTG+12]. En estos experimentos hay dos máquinas virtuales grandes (8 cores, 14GM RAM cada una) ejecutando aplicaciones interactivas por un largo período de tiempo (en este caso, todo el experimento). Para esto se han usado dos benchmarks populares en cloud: *RUBiS* y *RUBBoS*. RUBiS [RUB27b] es una web de subastas modelado tomando como modelo a eBay, mientras que RUBBoS [RUB27a] es un tablón de anuncios modelado a partir de Slashdot. La

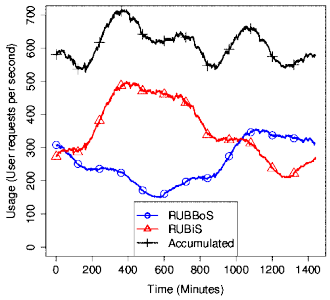


Figura 5.26: Cargas de trabajo para aplicaciones interactivas (VMs de larga vida)

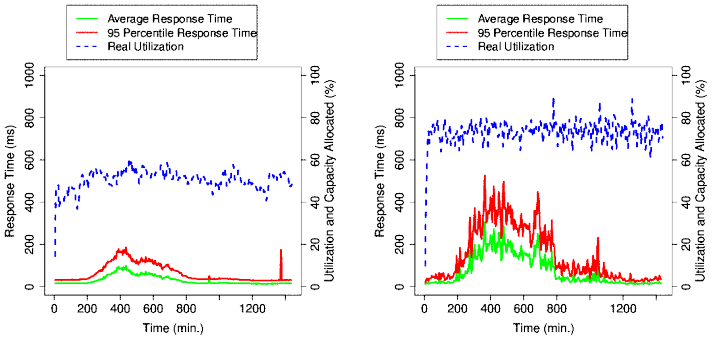
carga de trabajo de estas aplicaciones es un número de consultas recibidas en función del tiempo, generadas usando información extraída de las trazas de Wikipedia [Pag27], pero seleccionando dos días diferentes y modificando la carga original de 12 horas para RUBiS para crear tendencias y picos distintos (véase la Figura 5.26). Esto genera patrones de uso diario variados. Las consultas son generadas mediante la herramienta `httpmon`[7].

Por otro lado, también generamos VMs ejecutando aplicaciones no interactivas. Este tipo de aplicaciones tiene, típicamente, una vida relativamente corta y no son periódicas. Por ejemplo, VMs ejecutando tareas computacionales con unos requisitos muy heterogéneos y variantes en el tiempo [TT13b]. Estas VMs ejecutan numerosos shell scripts para generar *burstiness* en las diferentes dimensiones de capacidad (CPU, memoria o red). Adicionalmente, para obtener medidas de rendimiento de este tipo de aplicaciones, hemos empleado una aplicación que resuelve sudokus de forma continua [8] y devuelve el tiempo de procesamiento alcanzado sobre el tiempo –debiendo completar un número determinado antes del tiempo límite. El patrón de llegada de estas VMs de vida corta se genera mediante una distribución de Poisson con $\lambda = 20$ segundos.

Evaluación

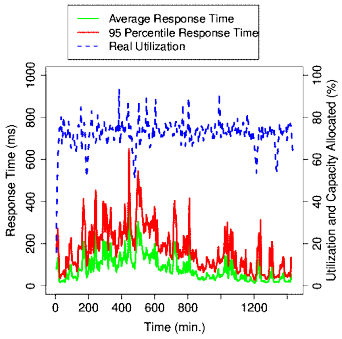
Con la carga de trabajo explicada anteriormente, el objetivo aquí consiste en medir el comportamiento de las aplicaciones interactivas (esto es, su tiempo de respuesta en cada intervalo) cuando otras VMs son ubicadas junto a ellas, así como conocer en qué medida la planificación propuesta reduce las interferencias entre ellas. Aquí nos

[7] https://github.com/cloud-control/httpmon
[8] http://norvig.com/sudoku.html



(a) No-Over KVM

(b) Over KVM



(c) AaFS

Figura 5.27: Rendimiento de RUBiS VM a lo largo del tiempo

centramos en las aplicaciones interactivas, puesto que éstas son las más afectadas por otras aplicaciones. Estas aplicaciones deben mantener su rendimiento todo el tiempo, al contrario que las aplicaciones intensivas en computación que, generalmente, no sufren en tanta medida los efectos de no disponer del máximo poder de computación unos pocos segundos, en tanto que terminen a tiempo.

Comparamos dos escenarios distintos: (1) El planificador kvm estándar, donde, por defecto, las VMs (vcpus) no están asignadas a cores específicos (pcpus) y kvm está a cargo de decidir, dinámicamente, qué vcpu tendrá acceso a qué pcpu a lo largo del tiempo. Esto se muestra con y sin overbooking (los correspondientes pies de figura son "No-Over KVM" y "Over KVM" en las siguientes figuras, respectivamente); y (2) nuestro planificador difuso basado en afinidad (con el pie de figura "AaFS").

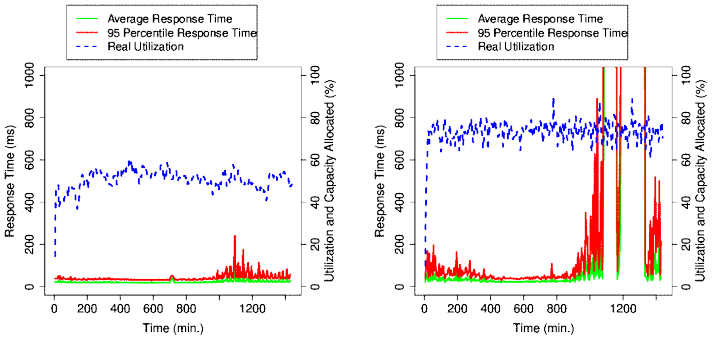
La Figura 5.27 ilustra tanto los tiempos de respuesta (medios y el percentil 95 %) para el servicio RUBiS con la carga de trabajo de Wikipedia, como la utilización

del auténtico servidor respecto al tiempo –nótese que el servidor está totalmente reservado desde el punto de vista de la capacidad nominal. Las Figuras 5.27a y 5.27b muestran el resultado de emplear el planificador kvm por defecto, tanto con overbooking como sin él, respectivamente. La Figura 5.27c muestra los resultados obtenidos por el planificador basado en afinidad propuesto. En los tres escenarios, la aplicación RUBiS se comporta como debe (manteniendo tiempos de respuesta por debajo de un segundo). Las principales diferencias se basan en si se aplica overbooking (Over-KVM y AaFS) o no (No-Over-KVM). Gracias al overbooking la utilización de recursos crece de un 50 % a un 75 %, pero a expensas de sufrir mayores tiempos de respuesta.

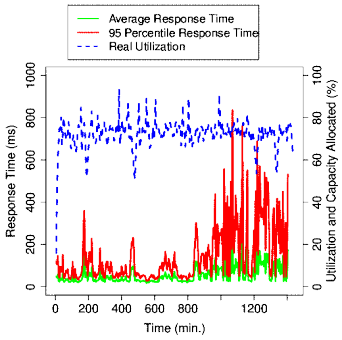
No observamos grandes diferencias entre las figuras correspondientes a los escenarios con overbooking (5.27a y 5.27b). Ambas presentan tiempos de respuesta aceptables todo el tiempo. De hecho, los valores medios obtenidos de ellas son bastante similares: las medias de los tiempos de respuesta medios son 78 y 84 ms, respectivamente; y las medias para el percentil 95 % son 152 y 177 ms, respectivamente. Las diferencias principales entre ambos escenarios radica en que AaFS presenta unos tiempos de respuesta ligeramente más planos respecto al tiempo, lo que significa que algunos de los cores sufrieron algo más de overbooking cuando hubo menos usuarios, y menos overbooking cuando la carga de trabajo era mayor. Esto permite a otras VMs ubicadas junto a ella mantener su rendimiento, como vemos en la Figura 5.28c.

En la Figura 5.28 se muestra la misma información que en la Figura 5.27 pero, en este caso, para la aplicación RUBBoS. Esta aplicación presenta un comportamiento menos lineal con respecto al número de peticiones, lo que la expone más a las interferencias de co-ubicación. Como antes, los tiempos de respuesta son realmente bajos para el escenario sin overbooking (Figura 5.28a), pero esto lleva a ratios de utilización bastante bajos. Al comparar los escenarios donde se ha realizado overbooking (Figuras 5.28b y 5.28c), destaca el hecho de que el planificador KVM por defecto no es capaz de manejar satisfactoriamente todas las VMs. Esto lleva a la degradación de rendimiento de algunas aplicaciones, espcialmente aquellas con mayor tendencia a ser afectadas por acciones de overbooking. En este caso, la máquina virtual de RUBBoS, cuyo tiempo de respuesta sobrepasa claramente el límite de un segundo establecido para esta aplicación, coincide con el pico de peticiones de usuarios (de 1000 a 1400).

En contraste, con nuestra propuesta (Figura 5.28c), el planificador es capaz de encontrar buenos vecinos a lo largo del tiempo tanto para RUBiS como para RUB-



(a) No-Over KVM (b) Over KVM



(c) AaFS

Figura 5.28: Rendimiento de RUBBoS VM a lo largo del tiempo

BoS. Esto conlleva a un uso compartido de las cpus físicas más eficiente, lo cual, a su vez, permite a la VM RUBBoS proporcionar el rendimiento deseado todo el tiempo. Además de ello, los valores globales obtenidos para los tiempos de respuesta también mejoran los obtenidos por Over-KVM. Los tiempos de respuesta medios se reducen de 363 ms a sólo 55 ms, así como el percentil 95 % decrece de 526 ms a 141 ms.

5.5. Conclusiones

En este capítulo hemos mostrado cuatro aplicaciones diferentes en las que hemos propuesto soluciones basadas en la programación lógica difusa multi-adjunta.

En primer lugar, recuérdese que XPath fue diseñado como un lenguaje de consulta para texto XML donde el camino del árbol subyacente en un documento XML

se usa para describir la consulta (nodos subsiguientes en las expresiones XPath se separan por una barra `'/'` o una barra doble `'//'`, siendo este último caso útil para sobrepasar varios nodos). Más aún, las expresiones XPath se pueden adornar con condiciones booleanas (entre corchetes `'[]'`) en nodos y hojas para restringir el número de respuestas dee la consulta. Por ejemplo, hemos usado la herramientan de XPath online `http://www.xpathtester.com/test` para ejecutar la consulta "`//node[goal='top']/substitution`" contra el fichero XML mostrado en la Figura 5.29, generada por $\mathcal{FLOPER}$ al producir el árbol de demostración de la Figura 5.15, devolviendo así como salida el nuevo documento XML mostrado en la Figura 5.30. Como se ilustra en la Figura 5.29, véase que los ficheros XML que representan árboles de demostración exportados por $\mathcal{FLOPER}$ siempre tienen como raíz la etiqueta `node`, cuyos hijos se basan en cuatro tipos de 'etiquetas' (esta estructura se anida tanto como sea necesario):

- `rule`, que indica la regla del programa que se ha evaluado para alcanzar el nodo actual (la regla virtual `R0` indica sólo el primer nodo),
- `goal`, que contiene la expresión MALP que esta siendo evaluada, esto es, la fórmula que el sistema está tratando de demostrar en ese momento. Nótese que cuando dicho valor es `top` se ha alcanzado un modelo, en que los valores asignados a los símbolos proposicionales de la fórmula se recogen en la siguiente etiqueta,
- `substitution`, que acumula los enlaces de variables realizados a lo largo de la derivación lógica difusa (esto es, la cadena de pasos computacionales a lo largo de la rama del árbol de ejecución) y cuyo significado revela el modo de interpretar las proposiciones contenidas en la fórmula para obtener sus modelos (véase la Figura 5.30 donde se muestran en el documento XML nueve soluciones con esta etiqueta, indicando cada uno los valores de verdad para las variables proposicionales que satisfacen la fórmula con el mayor grado de verdad), y
- `children`, que contiene el conjunto de nodos subyacentes en el árbol de modo anidado.

Como acabamos de mostrar, el uso combinado de la programación lógica difusa junto con el lenguaje estándar XPath para la obtención de datos XML, admite una retroalimentación interesante aplicable a la búsqueda automática de modelos de fórmulas difusas para las que planeamos introducir capacidades nuevas mediante

```
<node>
    <rule>R0</rule>
    <goal>or_godel(i(P),i(Q))</goal>
    <substitution>{}</substitution>
    <children>
        <node>
            <rule>R1</rule>
            <goal>or_godel(bottom,i(Q))</goal>
            <substitution>{P/bottom}</substitution>
            <children>
                <node>
                    <rule>R1</rule>
                    <goal>or_godel(bottom,bottom)</goal>
                    <substitution>
                        {Q/bottom,P/bottom}</substitution>
                    <children>
                        ...
                    </children>
                </node>
                ...
```

Figura 5.29: Parte del fichero XML que representa el árbol de ejecución mostrado en la Figura 5.15.

el uso de los recursos flexibles de nuestro sistema FuzzyXPath desarrollado con $\mathcal{FLOPER}$.

En particular, en la Sección 5.1 detallamos los beneficios mutuos entre diferentes herramientas difusas desarrolladas en nuestro grupo de investigación, siendo éstas el entorno de programación $\mathcal{FLOPER}$ y el intérprete de FuzzyXPath (disponible en `http://dectau.uclm.es/fuzzyXPath`). Recuérdese que, inicialmente, $\mathcal{FLOPER}$ fue concebido como una herramienta para implementar aplicaciones software flexibles –como es el caso de FuzzyXPath– codificadas en el lenguaje MALP. Así, esta herramienta ofrece opciones para compilar reglas difusas a cláusulas estándar, ejecutar objetivos y dibujar árboles de ejecución. Dichos árboles, una vez modelados en formato XML por la propia herramienta $\mathcal{FLOPER}$, se pueden analizar con el

```
<root>
  <substitution>{Q/top,P/bottom}</substitution>
  <substitution>{Q/top,P/alpha}</substitution>
  <substitution>{Q/top,P/beta}</substitution>
  <substitution>{Q/top,P/gamma}</substitution>
  <substitution>{Q/bottom,P/top}</substitution>
  <substitution>{Q/alpha,P/top}</substitution>
  <substitution>{Q/beta,P/top}</substitution>
  <substitution>{Q/gamma,P/top}</substitution>
  <substitution>{Q/top,P/top}</substitution>
</root>
```

Figura 5.30: XML file obtained after evaluating a XPath query.

intérprete FuzzyXPath para descubrir detalles (tales como las respuestas computadas, la posible infinitud de las ramas, etc.) del comportamiento computacional de los programas MALP al ser ejecutados en $\mathcal{FLOPER}$. En este sentido, planeamos incluir una opción dentro del menú de $\mathcal{FLOPER}$ para permitir la depuración de tareas basado en FuzzyXPath. Este desarrollo fue descrito en primera instancia en [ALMV13, ALMV14].

En este capítulo hemos abordado también dos problemas diferentes pero relacionados relativos a la demostración automática de fórmulas difusas proposicionales. En particular, mientras que se ha usado un resolvedor SMT para chequear la satisfacibilidad, se ha introducido una técnica alternativa basada en la programación lógica difusa para hallar en conjunto completo de interpretaciones que son modelos para una fórmula dada [BMVV15, BMVV13]. En el futuro planeamos introducir mejoras en ambos métodos, con respecto al conjunto de grados de verdad recolectados en el retículo usado para interpretar una fórmula dada. En el caso de SMT, planeamos investigar cómo tratar con retículos equipados con relaciones de orden parcial entre sus elementos, mientras que para el método basado en programación lógica difusa, trataremos de diseñar una técnica más flexible que el usado en esta sección (basada en considerar sólo unos pocos grados de verdad del espacio infinito) para tratar con retículos infinitos. En este sentido, algunas *reglas de parada* y *cortes de rama* pueden ser necesarios (tal vez a través de *cortes alfa*) o incluso podría ser interesante estudiar cómo obtener todos los modelos de una fórmula mediante una restricción

(como $x + y = 1$ para el ejemplo dado en la Introducción sobre el chip analógico) o un conjunto de restricciones.

Como se ha mostrado formalmente en la Subsección 5.2.2, el número de interpretaciones crece exponencialmente con respecto al conjunto de proposiciones y conectivas incluidos en una fórmula difusa. Por ello, conforme se evalúan fórmulas más grandes, se hace más difícil comprobar una a una todas las soluciones del árbol de ejecución correspondiente. Por ello, en el trabajo [ABL+15] posterior, hemos integrado este trabajo con el que recogemos en la Sección 5.1 previo, de modo que la exploración de los árboles de prueba SMT se pueda realizar automáticamente mediante FuzzyXPath.

Finalmente, hemos abordado un interesante trabajo en el área de computación cloud para dar respuesta a los problemas de la evaluación del riesgo en un entorno de overbooking [VTMT13], y la ubicación de máquinas virtuales en la infraestructura cloud [VMTT15].

Las técnicas de control de admisión que aplican overbooking son soluciones prometedoras para el bajo uso de recursos en los data center, un problema que surge de la naturaleza elástica de las aplicaciones cloud. Sin embargo, este overbooking puede llevar a degradaciones del rendimiento si no se planifica cuidadosamente. Hemos propuesto un control de admisión que basa su decisión de aceptación o rechazo en la información sobre el riesgo que se toma. Un motor lógico difuso provee la información que permite al control de admisión estimar el riesgo a largo plazo de aceptar una solicitud entrante. Dicha valoración del riesgo es una combinación de diversos parámetros de acuerdo a la relación entre la capacidad disponible y la solicitada, así como los picos de rendimiento surgidos cuando la capacidad se excede, proveyendo diferentes grados de riesgo que llevan a decisiones más (o menos) agresivas respecto a la aceptación del trabajo.

La evaluación de esta técnica muestra mejoras significativas en la utilización de recursos obtenida por nuestros métodos de control de admisión difusos. Incluso para las estimaciones más optimistas, los recursos disponibles se agotan tan poco como un 0.84 % del tiempo, mientras que la utilización se incrementa por un 138 %. Así, nuestros métodos difusos son una aproximación prometedora para ayudar al control de admisión a evaluar los riesgos asociados con aceptar un nuevo servicio.

En el futuro, este trabajo se puede extender teniendo en cuenta para la valoración del riesgo la información SLA. Dicha extensión podría especificar diferentes costes dependiendo del riesgo tomado o usando valores diferentes de riesgo dependiendo de

la penalización a pagar en caso de violación SLA, por ejemplo, cuanto mayor sea la penalización, más pesimista debería ser el control de admisión.

El segundo desarrollo en el ámbito cloud que hemos realizado sigue avanzando por la vía de mejorar la utilización de recursos en los centros de datos. Más allá del diseño de un mecanismo de control de admisión –como el realizado en la Sección 5.3– es esencial establecer un criterio para ubicar las máquinas virtuales aceptadas en el sistema, especialmente en entornos con overbooking.

Para resolver este problema, proponemos un modelo basado en programación lógica difusa que evalúa y clasifica los cores del servidor de acuerdo a su intensidad e irregularidad en el uso de recursos, lo que más adelante permite determinar cuál de ellos es más afín a la máquina virtual que debe ser alojada. Este método permite reducir las interferencias entre las máquinas virtuales y, por tanto, el rendimiento del sistema es más estable en el tiempo. De este modo, nuestra solución permite a los operadores cloud mejorar el uso de sus infraestructuras proporcionando aún así el rendimiento necesario a las aplicaciones.

Para el futuro próximo nos planteamos mejorar la herramienta incorporando el uso de las similaridades, esto es, pasando del lenguaje MALP al lenguaje FASILL [IMPV15] (esta capacidad, por ejemplo, podría ser útil para establecer la existencia de máquinas virtuales o cores con características similares y aliviar el cálculo). Además, actualmente estamos trabajando en la recogida de información en el rendimiento de la infraestructura cloud como valor de retroalimentación para mejorar el funcionamiento del módulo difuso.

Capítulo 6

El Lenguaje FASILL

Como se vio en el Capítulo 1, el área de investigación de la *Programación Lógica Difusa* está dedicada a la introducción de conceptos de la *lógica difusa* en la *programación lógica* para tratar explícitamente la vaguedad y la incertidumbre. A lo largo de las tres últimas décadas, esta área ha producido una amplia variedad de dialectos de Prolog. Estos lenguajes, llamados *lenguajes lógicos difusos*, se pueden clasificar (entre otros criterios) de acuerdo al modo de difuminar los mecanismos originales de unificación o resolución. Así, mientras que algunas aproximaciones introducen la noción de difuminación mediante el uso de relaciones de similaridad en tiempo de unificación [FGS00, Arc02, Ses02], otras extienden el principio operacional (manteniendo la unificación sintáctica) para emplear una amplia variedad de conectivas difusas y grados de verdad en las reglas, más allá de los clásicos *verdadero* o *falso* [KS92, MOV04, MCS11a].

Si bien a lo largo de esta tesis nos hemos centrado en la segunda línea de integración (como pone de relieve la preeminencia del lenguaje MALP en estas páginas), en este capítulo cobra especial relevancia la primera línea, donde al algoritmo de unificación sintáctica se le dota de la habilidad para manejar relaciones de similaridad/proximidad. Éstas relacionan los elementos de un conjunto con un determinado grado de aproximación, debilitando la noción de igualdad y, así, permitiendo tratar con información vaga, concretamente, aquella que hace referencia a entidades semánticamente próximas. El trabajo relacionado con respecto a esta línea de integración se puede resumir como sigue:

- En primer lugar, los artículos pioneros [AFG96, FGS99, FGS00] y [AF02],

donde se desarrolló por primera vez el concepto de unificación por similaridad. Obsérvese que en esta tesis compartimos sus objetivos si bien, al contrario que nuestra propuesta, en esos artículos se utilizan las nociones exóticas de *nubes*, *sistemas de nubes* y *operadores de clausura* en la definición del algoritmo de unificación, que pueden hacer peligrar la eficiencia de la semántica operacional derivada.

- Más próximo a nuestra propuesta es el trabajo presentado en [Ses02] por María Sessa. Allí se define una extensión del principio de SLD-resolución que incorpora un procedimiento de unificación basado en similaridad, que es una reformulación del algoritmo de Martelli y Montanari [MM82] donde los símbolos unifican si son similares (en vez de si son sintácticamente idénticos). El algoritmo resultante emplea una noción generalizada del unificador más general que proporciona un valor numérico, que es tanto una medida del grado de aproximación como una noción graduada de consecuencia lógica. La aproximación de Sessa puede considerarse nuestro punto de partida.

Desde un punto de vista práctico, las aproximaciones basadas en similaridades han producido tres implementaciones experimentales. El primer prototipo descrito en la literatura es un intérprete en Prolog del lenguaje LIKELOG (*LIKEness in LOGic*) [AF99b] que emplea los conceptos previamente mencionados de *nube* y *clausura* descritos en [AF02, AFG96, FGS00, FGS99]). El segundo, SiLog [LSS01], es un intérprete escrito en Java basado en las ideas introducidas en [Ses02]. Ni LIKELOG ni SiLog están públicamente disponibles, lo que impide una evaluación independiente de estos sistemas. Por otra parte, Bousi~Prolog [JR09b, JRG09a] es el primer sistema de programación lógica difusa que es una verdadera extensión de Prolog y no un simple intérprete capaz de ejecutar la SLD-resolución débil. También es el primer lenguaje de programación lógico difuso que propuso el uso de relaciones de proximidad como una generalización de las relaciones de similaridad [JR09a]. Merece la pena añadir que, para tratar con relaciones de proximidad, se ha desarrollado para Bousi~Prolog una nueva base teórica [JR11] y conceptual [RJ14].

Un marco relacionado es el de la *Programación Lógica Cualificada* (QLP por sus siglas en inglés, *Qualified Logic Programming*), que deriva del de la *Programación Lógica Cuantitativa* de van Emden [vE86] y de la *Programación Lógica Anotada* [KS92]. En QLP un programa se asocia a un dominio de cualificación D, y sus reglas se anotan con valores de cualificación, lo que resulta en un marco paramétrico: QLP(D). En [CRR08] se introducen relaciones de similaridad en el marco QLP(D)

mediante el uso de una aproximación transformacional. El nuevo esquema QLP(D) basado en similaridad, llamado SQLP(D), transforma una relación de similaridad en un conjunto de reglas QLP(D) capaces de emular la unificación por similaridad. En [RR08a, CRR14] se da un paso más allá al integrar restricciones y relaciones de proximidad en su esquema genérico, obteniendo un marco de programación flexible llamado SQCLP.

Para terminar esta introducción, es importante remarcar que nuestro grupo de investigación ha estado envuelto tanto en el desarrollo de sistemas de programación lógica basada en similaridades como en los que extienden el principio de resolución, como revela el diseño del lenguaje Bousi~Prolog[1] [JR09a, JR10b, RJ14], donde las cláusulas cohabitan con ecuaciones de similaridad/proximidad, y el desarrollo del sistema $\mathcal{FLOPER}$[2] que se ha descrito profusamente en el Capítulo 4.

Nuestra aproximación de unificación está inspirada por [CRR14], pero en nuestro marco admitimos un conjunto más amplio de conectivas en el cuerpo de las reglas de programa.

En este capítulo empezamos mostrando, en la Sección 6.1, la sub-clase de los $\top$-programas MALP que emplean versiones de modelo y paso computacional más sencillas que las de MALP. A continuación, en la Sección 6.2, introducimos una ténica sencilla para mapear cualquier programa MALP a otro en esta sub-clase, eliminando así la necesidad de usar pares adjuntos en la semántica de MALP [MPV13].

Razonamos de manera similar en la Sección 6.3 para construir la super-clase X-MALP de programas lógicos difusos más allá de MALP, donde se abandona definitivamente el uso de pares adjuntos en tanto que la semántica se basa en retículos mucho menos restringidos (retículos completos) que los multi-adjuntos. Más aún, detallamos la semántica de punto fijo del nuevo marco en la Sección 6.4. El lenguaje X-MALP, que describimos por primera vez en [MPV14a], sirve como punto de partida para introducir la noción de similaridad, produciendo un nuevo lenguaje integrado denominado FASILL.

En las Secciones 6.5 y 6.6 de este capítulo detallamos e ilustramos la sintáxis y la semántica operacional, respectivamente, del lenguaje FASILL. Más adelante, la Sección 6.7 trata la implementación del lenguaje en el sistema $\mathcal{FLOPER}$, de acuerdo a como hicimos en [IMPV15, JMPV14]. En la Sección 6.8 definimos la semántica

[1] Hay dos entornos de programación de Bousi~Prolog disponibles en `http://dectau.uclm.es/bousi/`.

[2] La herramienta se puede acceder libremente en el sitio web `http://dectau.uclm.es/floper/`, y en `http://dectau.uclm.es/floper/?q=sim/test` pueden probarse sus funcionalidades on-line.

Programa lógico multi-adjunto:

$$\mathcal{P} = \left\{ \begin{array}{lllll} \mathcal{R}_1: & \langle p(X) & \leftarrow_{\mathtt{P}} & q(X,Y)\ \&_{\mathtt{G}}\ (r(Y)\ |_{\mathtt{L}}\ s(Y)) & ;\ 0.8\rangle \\ \mathcal{R}_2: & \langle q(a,Y) & \leftarrow & & ;\ 0.9\rangle \\ \mathcal{R}_3: & \langle r(b) & \leftarrow & & ;\ 1\rangle \end{array} \right.$$

Derivación admisible:

$$\begin{array}{ll} \langle \underline{p(X)}; id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} \\ \langle 0.8\ \&_{\mathtt{P}}\ (\underline{q(X_1,Y_1)}\ \&_{\mathtt{G}}\ (r(Y1)\ |_{\mathtt{L}}\ s(Y1)));\{X/X_1\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\ \langle 0.8\ \&_{\mathtt{P}}\ (0.9\ \&_{\mathtt{G}}\ (\underline{r(Y2)}\ |_{\mathtt{L}}\ s(Y2)));\{X/a, X_1/a, Y_1/Y_2\}\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\ \langle 0.8\ \&_{\mathtt{P}}\ (0.9\ \&_{\mathtt{G}}\ (1\ |_{\mathtt{L}}\ \underline{s(Y2)}));\{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle & \rightarrow_{AS3} \\ \langle 0.8\ \&_{\mathtt{P}}\ (0.9\ \&_{\mathtt{G}}\ (1\ |_{\mathtt{L}}\ 0));\{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle & \end{array}$$

Derivación interpretativa:

$$\begin{array}{ll} \langle 0.8\ \&_{\mathtt{P}}\ (0.9\ \&_{\mathtt{G}}\ \underline{(1\ |_{\mathtt{L}}\ 0)});\{X/a\}\rangle & \rightarrow_{IS} \\ \langle 0.8\ \&_{\mathtt{P}}\ \underline{(0.9\ \&_{\mathtt{G}}\ 1)};\{X/a\}\rangle & \rightarrow_{IS} \\ \langle \underline{0.8\ \&_{\mathtt{P}}\ 0.9};\{X/a\}\rangle & \rightarrow_{IS} \\ \langle 0.72;\{X/a\}\rangle & \text{— f.c.a. significa que } p(X) \text{ es cierto con} \\ & \text{grado de verdad 0.72 cuando } X = a\text{''.} \end{array}$$

Modelo mínimo de Herbrand difuso:

$$\mathcal{I}_{\mathcal{P}}(p(a)) = 0.72,\ \mathcal{I}_{\mathcal{P}}(q(a,a)) = \mathcal{I}_{\mathcal{P}}(q(a,b)) = 0.9,\ \mathcal{I}_{\mathcal{P}}(r(b)) = 1.$$

Figura 6.1: Ejemplos ilustrativos de la sintáxis y semántica de MALP

declarativa del lenguaje FASILL. Concluimos el capítulo con Sección 6.9 aportando también ideas de trabajo futuro en este marco.

Para realizar comparaciones entre los distintos lenguajes presentados en este capítulo, introducimos en la Figura 6.1 un ejemplo de programa MALP que cuenta con tres reglas, dos de las cuales son hechos, y del que se aporta también su derivación admisible, interpretativa y su modelo mínimo de Herbrand.

6.1. Una sub-clase de MALP independiente de la adjunción

En adelante, empleando nuestra notación de [MPV13], llamamos $\mathcal{M}_\top$ al conjunto de los programas MALP cuyas reglas se etiquetan siempre con el elemento $\top$ de sus retículos asociados. Hablamos, así, de $\top$-programas, $\top$ -reglas, etc. Para ejecutar estos programas se puede concebir una semántica operacional más sencilla que la descrita en la Sección 3.2 (véase la Definición 3.2.1).

Definición 6.1.1 ($\top$-paso admisible)**.** *Sea $\mathcal{Q}$ un objetivo, y σ una sustitución. El par $\langle \mathcal{Q}; \sigma \rangle$ es un* estado*. Dado un $\top$-programa $\mathcal{P} \in \mathcal{M}_\top$, una* $\top$-computación admisible *se formaliza como una sistema de transición de estados cuya relación de transición, $\overset{AS^\top}{\rightsquigarrow}$, es la menor relación que satisface las siguientes* $\top$-reglas admisibles*:*

1) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\mathcal{B}])\theta; \sigma\theta \rangle$ *si* $\theta = mgu(\{H = A\})$, $\langle H \leftarrow_i \mathcal{B}; \top \rangle$ *en* $\mathcal{P}$ *y* $\mathcal{B}$ *no es vacía.*

2) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta \rangle$ *si* $\theta = mgu(\{H = A\})$, *y* $\langle H \leftarrow_i ; \top \rangle \in \mathcal{P}$.

3) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$ *si no hay $\top$-regla en $\mathcal{P}$ cuya cabeza unifique con A (esta regla trata con posibles ramas de fallo).*

Una $\top$-*derivación admisible* es una secuencia $\langle \mathcal{Q}; id \rangle \overset{\text{AS}^\top}{\rightsquigarrow}{}^* \langle \mathcal{Q}'; \theta \rangle$. Usaremos los símbolos $\overset{\text{AS1}^\top}{\rightsquigarrow}$, $\overset{\text{AS2}^\top}{\rightsquigarrow}$ y $\overset{\text{AS3}^\top}{\rightsquigarrow}$ para distinguir entre pasos de computación realizados por la aplicación de cada regla admisible específica. Obsérvese que esta definición (que es muy próxima a la clásica de la SLD-resolución usada en PROLOG) difiere de la Definición 3.2.1 únicamente en la primera regla, en tanto que no hace uso de la conjunción $\&_i$ adjunta a la implicación $\leftarrow_i$ de las $\top$-reglas.

El siguiente resultado establece que, en lo que respecta a los $\top$-programas, las derivaciones construidas con pasos admisibles (junto a los consiguientes pasos interpretativos) dados por la Definición 3.2.1, así como aquellos basados en pasos $\top$-admisibles (e interpretativos) dados por la Definición 6.1.1, llevan al mismo conjunto de respuestas computadas difusas.

Teorema 6.1.2. *Sea $\mathcal{P} \in \mathcal{M}_\top$ un $\top$-programa* MALP *con retículo asociado L, $\mathcal{Q}$ un objetivo, σ una sustitución y $v \in L$. Entonces,*

$$\langle \mathcal{Q}; id \rangle \rightarrow_{AS}{}^* \cdots \rightarrow_{IS}{}^* \langle v; \sigma \rangle \quad \textit{sii} \quad \langle \mathcal{Q}; id \rangle \overset{AS^\top}{\rightsquigarrow}{}^* \cdots \rightarrow_{IS}{}^* \langle v; \sigma \rangle$$

Demostración. Para demostrar este teorema, basta contemplar que los efectos producidos por los pasos $\rightarrow_{AS}$ sobre un estado genérico de la forma $\langle \mathcal{Q}[A]; \sigma \rangle$, son idénticos a los generados por pasos $\overset{\text{AS}^\top}{\rightsquigarrow}$ y viceversa. Consideramos tres casos distintos:

1) Si $\langle H \leftarrow_i \mathcal{B}; \top \rangle \in \mathcal{P}$, donde $\mathcal{B}$ no es vacía y $\theta = mgu(\{H = A\})$, entonces $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS1} \langle (\mathcal{Q}[A/\top \&_i \mathcal{B}])\theta; \sigma\theta \rangle$ sii $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS1}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\mathcal{B}])\theta; \sigma\theta \rangle$, ya que $\top \&_i \mathcal{B} \equiv \mathcal{B}$ y, por tanto, $(\mathcal{Q}[A/\top \&_i \mathcal{B}])\theta \equiv (\mathcal{Q}[A/\mathcal{B}])\theta$.

2) Si $\langle H \leftarrow_i ; \top \rangle \in \mathcal{P}$, donde $\theta = mgu(\{H = A\})$, entonces $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS2} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta \rangle$ sii $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS2}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta \rangle$.

3) Si no hay una $\top$-regla en $\mathcal{P}$ cuya cabeza unifique con A entonces, $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS3} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$ sii $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS3}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$.

□

Continuamos aportando aspectos de la semántica declarativa relacionados con $\top$-programas.

Definición 6.1.3 ($\top$-modelo)**.** *Una interpretación $\mathcal{I}$ $\top$-satisface (o es* $\top$-modelo *de) una $\top$-regla $\langle H \leftarrow_i \mathcal{B}; \top \rangle$ si, y sólo si, $\mathcal{I}(\mathcal{B}) \leq \mathcal{I}(H)$. Una interpretación $\mathcal{I}$ es $\top$-modelo de un $\top$-programa $\mathcal{P}$ si, y sólo si, todas las $\top$-reglas en $\mathcal{P}$ son $\top$-satisfechas por $\mathcal{I}$.*

Definición 6.1.4 ($\top$-modelo mínimo difuso de Herbrand)**.** *Sea $\mathcal{P} \in \mathcal{M}_\top$ un $\top$-programa* MALP *con retículo asociado $(L, \leq)$. La interpretación $\mathcal{I}_\mathcal{P}^\top = inf\{\mathcal{I}_j : \mathcal{I}_j$ es $\top$-modelo de $\mathcal{P}\}$ es el menor $\top$-modelo de Herbrand difuso*[3] *de $\mathcal{P}$.*

En el siguiente resultado demostramos que la noción de modelo mínimo de Herbrand difuso de un $\top$-programa MALP dado es idéntico a su $\top$-modelo mínimo de Herbrand difuso.

Teorema 6.1.5. *El modelo mínimo de Herbrand difuso de un $\top$-programa* MALP *$\mathcal{P} \in \mathcal{M}_\top$ coincide con su $\top$-modelo mínimo de Herbrand difuso, esto es, $\mathcal{I}_\mathcal{P} = \mathcal{I}_\mathcal{P}^\top$.*

Demostración. Considérese una $\top$-regla genérica $\mathcal{R} : \langle H \leftarrow_i \mathcal{B}; \top \rangle \in \mathcal{P}$ y una interpretación $\mathcal{I}$. Para demostrar que $\mathcal{I}$ satisface $\mathcal{R}$ si y sólo si $\top$-satisface $\mathcal{R}$, es suficiente

[3] A veces diremos únicamente modelo mínimo difuso o modelo mínimo.

$\top$-programa lógico multi-adjunto:

$$\mathcal{P}' = \mathcal{P}^{\mathcal{M}_\top} = \begin{cases} \mathcal{R}'_1: & p(X) & \leftarrow & 0.8 \;\&_{\mathtt{P}}\; (q(X,Y) \;\&_{\mathtt{G}}\; (r(Y) \;|_{\mathtt{L}}\; s(Y)) \\ \mathcal{R}'_2: & q(a,Y) & \leftarrow & 0.9 \\ \mathcal{R}'_3: & r(b) & \leftarrow & \end{cases}$$

$\top$-derivación admisible:

$$\begin{array}{ll} \langle \underline{p(X)}; id\rangle & \overset{\text{AS1}^\top}{\rightsquigarrow} \mathcal{R}'_1 \\ \langle 0.8 \;\&_{\mathtt{P}}\; (\underline{q(X_1,Y_1)} \;\&_{\mathtt{G}}\; (r(Y1) \;|_{\mathtt{L}}\; s(Y1)));\{X/X_1\}\rangle & \overset{\text{AS1}^\top}{\rightsquigarrow} \mathcal{R}'_2 \\ \langle 0.8 \;\&_{\mathtt{P}}\; (0.9 \;\&_{\mathtt{G}}\; (\underline{r(Y2)} \;|_{\mathtt{L}}\; s(Y2)));\{X/a, X_1/a, Y_1/Y_2\}\rangle & \overset{\text{AS2}^\top}{\rightsquigarrow} \mathcal{R}'_3 \\ \langle 0.8 \;\&_{\mathtt{P}}\; (0.9 \;\&_{\mathtt{G}}\; (1 \;|_{\mathtt{L}}\; \underline{s(Y2)}));\{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle & \overset{\text{AS3}^\top}{\rightsquigarrow} \\ \langle 0.8 \;\&_{\mathtt{P}}\; (0.9 \;\&_{\mathtt{G}}\; (1 \;|_{\mathtt{L}}\; 0));\{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle & \end{array}$$

Figura 6.2: Ejemplos de los conceptos definidos en las Secciones 6.1 y 6.2

considerar que $\top \leq \mathcal{I}(H \leftarrow_i \mathcal{B})$ pasa a ser $\mathcal{I}(\top \&_i \mathcal{B}) \leq \mathcal{I}(H)$ debido a la propiedad adjunta. Así, esa expresión se puede simplificar a $\mathcal{I}(\mathcal{B}) \leq \mathcal{I}(H)$ (puesto que, obviamente, $\top \&_i v = v$, para $v \in L$), como queríamos. Así, dado que el conjunto de modelos para un $\top$-programa $\mathcal{P}$ dado es el mismo que su conjunto de $\top$-modelos, el ínfimo de dicho conjunto es exactamente $\mathcal{I}_\mathcal{P} = \mathcal{I}^\top_\mathcal{P}$, lo que concluye esta demostración. □

Nótese que, en las definiciones anteriores relacionadas con $\top$-satisfactibilidad, $\top$-modelos y $\mathcal{I}^\top_\mathcal{P}$, no se requiere que el retículo asociado al $\top$-programa MALP sea multi-adjunto (de hecho, ni tan siquiera se hace uso de los pares adjuntos). Ocurre de igual modo en las definiciones de la nueva semántica operacional para $\top$-programas. Por este motivo, en adelante podemos simplificar la sintaxis de las $\top$-reglas eliminando la etiqueta de sus símbolos de implicación, así como sus pesos (o grados de verdad asociados), esto es, en lugar de $\langle H \leftarrow_i \mathcal{B}; \top\rangle$ escribiremos simplemente $H \leftarrow \mathcal{B}$.

6.2. Mapeado de programas MALP a $\mathcal{M}_\top$

La siguiente definición representa un sencillo preprocesado sintáctico por el que, haciendo uso de pares adjuntos, se pueden transformar programas MALP a $\top$-

programas.

Definición 6.2.1. *Definimos una aplicación que asocia a cada programa* MALP $\mathcal{P}$ *un* $\top$*-programa en* $\mathcal{M}_\top$ *del siguiente modo:*

$$\mathcal{P}^{\mathcal{M}_\top} = \{\mathcal{R}^{\mathcal{M}_\top} \ : \ \mathcal{R} \in \mathcal{P}\}$$

donde para cada $\top$*-regla* $\mathcal{R} : \langle H \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}$*, la aplicación se define como:*

$$\mathcal{R}^{\mathcal{M}_\top} \quad = \quad \begin{cases} H \leftarrow v \&_i \mathcal{B} & \textit{if} \quad v \neq \top \\ H \leftarrow \mathcal{B} & \textit{otherwise} \end{cases}$$

En la Figura 6.2 ilustramos esta definición así como otros conceptos introducidos previamente. Nótese que:

- El $\top$-programa $\mathcal{P}'$ coincide con la transformación del programa MALP $\mathcal{P}$ visto en la Figura 6.1, esto es, $\mathcal{P}' = \mathcal{P}^{\mathcal{M}_\top}$, puesto que $\mathcal{R}'_1 = \mathcal{R}_1^{\mathcal{M}_\top}$, $\mathcal{R}'_2 = \mathcal{R}_2^{\mathcal{M}_\top}$ y $\mathcal{R}'_3 = \mathcal{R}_3^{\mathcal{M}_\top}$. En este último caso, únicamente hemos tenido que eliminar el peso de la regla (ya que éste es 1, es decir, el elemento supremo del retículo $[0, 1]$) y tanto $\mathcal{R}_1$ como $\mathcal{R}'_1$ son hechos en $\mathcal{P}$ y $\mathcal{P}'$, respectivamente.

- Por otra parte, obsérvese que incluso siendo $\mathcal{R}_2 \in \mathcal{P}$ un hecho, $\mathcal{R}'_2 \in \mathcal{P}'$ no lo es, dado que su cuerpo no es vacío (está compuesto únicamente por un grado de verdad). Por este motivo, mientras que el segundo paso admisible de la derivación admisible de la Figura 6.1 es del tipo $\rightarrow_{AS2}$, el equivalente segundo $\top$-paso admisible de la $\top$ derivación admisible de la Figura 6.2 no es un paso $\overset{\text{AS2}^\top}{\rightsquigarrow}$, sino $\overset{\text{AS1}^\top}{\rightsquigarrow}$.

- La secuencia de estados en la derivación admisible de la Figura 6.1 coincide con la secuencia de estados en la $\top$-derivación admisible de la Figura 6.2, y tras aplicar exactamente la misma secuencia de pasos interpretativos de la Figura 6.1 (por este motivo la hemos omitido en la Figura 6.2), se alcanza la misma respuesta computada difusa.

- Obsérvese que aunque la noción de $\top$-modelo[4] es más sencilla que la de modelo, el $\top$-modelo mínimo de Herbrand difuso coincide con $\mathcal{I}_\mathcal{P}$ en la Figura 6.1, como queríamos.

[4] Recordemos que este concepto no hace uso de pares adjuntos ni pesos en las reglas de programa.

El siguiente resultado establece que las derivaciones construidas con pasos admisibles (junto con sus subsiguientes pasos interpretativos) llevan al mismo conjunto de respuestas computadas que las derivaciones construidas con $\top$-pasos admisibles (e interpretativos) sobre los $\top$-programas obtenidos del programa MALP previo, una vez transdormado de acuerdo a la Definición 6.2.1.

Teorema 6.2.2. *Sea $\mathcal{P}$ un programa* MALP *con retículo asociado L, $\mathcal{Q}$ un objetivo, σ una sustitución y $v \in L$. Entonces,*

$$\langle \mathcal{Q}; id\rangle \rightarrow_{AS}{}^* \cdots \rightarrow_{IS}{}^* \langle v; \sigma\rangle \quad sii \quad \langle \mathcal{Q}; id\rangle \overset{AS^\top}{\rightsquigarrow}{}^* \cdots \rightarrow_{IS}{}^* \langle v; \sigma\rangle$$

Demostración. Distinguimos cuatro casos para mostrar cómo los efectos producidos por los pasos $\rightarrow_{AS}$, sobre un estado genérico de la forma $\langle \mathcal{Q}[A]; \sigma\rangle$, son replicados por pasos $\overset{\text{AS}^\top}{\rightsquigarrow}$ y viceversa.

1) Obsérvese que $\langle H \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}$, donde $\mathcal{B}$ no es vacío y $\theta = mgu(\{H = A\})$, si y sólo si $\langle H \leftarrow v \&_i \mathcal{B}\rangle \in \mathcal{P}^{\mathcal{M}_\top}$, y, por tanto, $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS1} \langle (\mathcal{Q}[A/v\&_i\mathcal{B}])\theta; \sigma\theta\rangle$ si y sólo si $\langle \mathcal{Q}[A]; \sigma\rangle \overset{\text{AS1}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/v\&_i\mathcal{B}])\theta; \sigma\theta\rangle$.

2) Entonces, $\langle H \leftarrow; v\rangle \in \mathcal{P}$, donde $v \neq \top$, $\theta = mgu(\{H = A\})$ si y sólo si $\langle H \leftarrow v\rangle \in \mathcal{P}^{\mathcal{M}_\top}$, y, por tanto, $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS2} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta\rangle$ si y sólo si $\langle \mathcal{Q}[A]; \sigma\rangle \overset{\text{AS1}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta\rangle$ (queda probada la correspondencia entre $\rightarrow_{AS2}$ y $\overset{\text{AS1}^\top}{\rightsquigarrow}$).

3) Nótese que $\langle H \leftarrow; \top\rangle \in \mathcal{P}$, donde $\theta = mgu(\{H = A\})$, si y sólo si $\langle H \leftarrow\rangle \in \mathcal{P}^{\mathcal{M}_\top}$, y, por tanto, $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS2} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta\rangle$ si y sólo si $\langle \mathcal{Q}[A]; \sigma\rangle \overset{\text{AS2}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta\rangle$ (hasta aquí queda demostrada la equivalencia entre los pasos $\rightarrow_{AS2}$ y $\overset{\text{AS2}^\top}{\rightsquigarrow}$).

4) Para finalizar, no existe en $\mathcal{P}$ una regla cuya cabeza unifique con A si y sólo si no existe una regla en $\mathcal{P}^{\mathcal{M}_\top}$ cuya cabeza unifique con A y, por tanto, $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS3} \langle (\mathcal{Q}[A/\bot]); \sigma\rangle$ si y sólo si $\langle \mathcal{Q}[A]; \sigma\rangle \overset{\text{AS3}^\top}{\rightsquigarrow} \langle (\mathcal{Q}[A/\bot]); \sigma\rangle$.

□

En el siguiente resultado demostramos que la noción de modelo mínimo de Herbrand difuso de los programas MALP es el mismo que el $\top$-modelo mínimo de Herbrand difuso de los $\top$-programas obtenidos por la aplicación del proceso de transformación descrito en la Definición 6.2.1.

Teorema 6.2.3. *El modelo mínimo de Herbrand difuso de un programa* MALP $\mathcal{P}$ *coincide con el* $\top$*-modelo mínimo de Herbrand difuso de su* $\top$*-programa asociado* $\mathcal{P}^{\mathcal{M}_\top}$*, esto es,* $\mathcal{I}_\mathcal{P} = \mathcal{I}^\top_{\mathcal{P}^{\mathcal{M}_\top}}$.

Demostración. Considérese la regla genérica $\mathcal{R} : \langle H \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}$ y, correspondientemente, $\mathcal{R}^{\mathcal{M}_\top} : H \leftarrow v \&_i \mathcal{B} \in \mathcal{P}^{\mathcal{M}_\top}$. Se asume la existencia de una interpretation $\mathcal{I}$ tal que $\mathcal{I}$ satisface $\mathcal{R}$ si y sólo si $\mathcal{I}$ $\top$-satisface $\mathcal{R}^{\mathcal{M}_\top}$, puesto que por la propiedad adjunta $v \leq \mathcal{I}(H \leftarrow_i \mathcal{B})$ si y sólo si $\mathcal{I}(v \&_i \mathcal{B}) \leq \mathcal{I}(H)$ y, por tanto, el conjunto de modelos de $\mathcal{P}$ coincide con el conjunto de $\top$-modelos de $\mathcal{P}^{\mathcal{M}_\top}$ y, en particular, el ínfimo de dicho conjunto es $\mathcal{I}_\mathcal{P}$ así como $\mathcal{I}^\top_{\mathcal{P}^{\mathcal{M}_\top}}$, como queríamos. □

6.3. La clase extendida de programas X-MALP

Como se recoge en [MPV14a], en adelante llamamos X-MALP a la *programación lógica multi-adjunta relajada* (en inglés, *relaXed Multi-Adjoint Logic Programming*). En contraste con el uso de retículos multi-adjuntos por parte de los programas MALP (y también de los $\top$-programas MALP), en la nueva clase X-MALP los grados de verdad se modelan en un tipo de retículos más relajado (concretamente, retículos completos), que no requieren de la presencia de pares adjuntos, lo que justifica que realmente MALP sea un subconjunto de X-MALP (véase la Figura 6.3). En general, la sintaxis de las reglas de programa X-MALP coincide con la de los $\top$-programas MALP, esto es, son expresiones de la forma $H \leftarrow_i \mathcal{B}$, donde H es una fórmula atómica (la *cabeza*) y $\mathcal{B}$ (el *cuerpo*) es una fórmula construida a partir de fórmulas atómicas $B_1, \ldots, B_n$ ($n \geq 0$), grados de verdad del retículo asociado, y conjunciones, disyunciones y agregadores. Además de ello, se permite también la presencia de reglas con pesos, como en las reglas MALP, pero debe tenerse en cuenta que en el nuevo marco, dichos pesos son únicamente "azúcar sintáctico" en el sentido de que cualquier expresión de la forma $\mathcal{R} : \langle H \leftarrow_i \mathcal{B}; v\rangle$ se refiere a la regla X-MALP $\mathcal{R} : H \leftarrow v \&_i \mathcal{B}$ donde en el retículo asociado sólo se requiere que esté definido el símbolo de conjunción $\&_i$, pero no que conforme un par adjunto $\langle \leftarrow_i, \&_i\rangle$ (de hecho, el símbolo de implicación $\leftarrow_i$ no se utiliza en ningún momento para definir la sintaxis de X-MALP, en claro contraste con MALP). La siguiente definición muestra la sintaxis general de los programas X-MALP, remarcando el poder expresivo de este estilo de programación lógica difusa, que cubre fácilmente otros marcos bien establecidos.

Definición 6.3.1. *Un* programa X-MALP $\mathcal{P}$*, asociado a un retículo completo,* $(L, \leq)$*, es un conjunto de reglas* $A \leftarrow \mathcal{B}$ *que verifican:*

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

ii) $\mathcal{B}$ *es una fórmula arbitraria (*cuerpo*) construida mediante átomos* $B_1, \ldots, B_n$*,* $n \geq 0$ *y cualesquiera conjunciones, disyunciones, agregadores y grados de verdad (esto es, elementos tomados del retículo* L *subyacente).*

Además, las reglas de $\mathcal{P}$ también se expresan mediante la sintaxis $A \leftarrow f(B_1, \ldots, B_n)$, donde f es una función computable que resulta de combinar todas las conectivas presentes en el cuerpo (esta misma sintaxis puede encontrarse explícitamente en [Voj01, LS02a, LS09], y también se acepta en muchos otros marcos de programación lógica difusa). Las reglas de programa se interpretan en un retículo completo $(L, \leq)$, previamente asociado a $\mathcal{P}$.

En este punto es necesario adaptar todas las definiciones presentes en la Sección 6.1 para referirnos a partir de ahora a programas X-MALP en lugar de $\top$-programas MALP.

Definición 6.3.2 (Paso admisible para programas X-MALP)**.** *Sean* $\mathcal{Q}$ *un objetivo y* σ *una sustitución. El par* $\langle \mathcal{Q}; \sigma \rangle$ *es un* estado*. Dado un programa* X-MALP $\mathcal{P}$*, una* computación admisible *se formaliza como un sistema de transición de estados cuya relación de transición* $\overset{AS}{\hookrightarrow}$ *es la menor relación que satisface las siguientes* reglas admisibles*:*

1) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS}{\hookrightarrow} \langle (\mathcal{Q}[A/\mathcal{B}])\theta; \sigma\theta \rangle$ *si* $\theta = mgu(\{H = A\})$*,* $\langle H \leftarrow_i \mathcal{B}; \top \rangle$ *en* $\mathcal{P}$ *y* $\mathcal{B}$ *no es vacío.*

2) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS}{\hookrightarrow} \langle (\mathcal{Q}[A/\top])\theta; \sigma\theta \rangle$ *si* $\theta = mgu(\{H = A\})$*, y* $\langle H \leftarrow_i; \top \rangle$ *en* $\mathcal{P}$*.*

3) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{AS}{\hookrightarrow} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$ *si no hay ninguna regla en* $\mathcal{P}$ *cuya cabeza unifique con* A *(este caso trata con posibles ramas fallidas).*

Al igual que en el caso de MALP, una vez explotados todos los átomos de un objetivo dado –mediante la aplicación de tantos pasos admisibles como sean necesarios durante la fase operacional–, éste se convierte en una fórmula sin átomos que se puede interpretar con respecto al retículo completo L para obtener el conjunto de respuestas computadas difusas de acuerdo a la Definición 3.2.6.

En lo que sigue, introducimos formalmente la noción semántica de modelo de Herbrand para programas X-MALP, de modo similar a como hicimos para MALP.

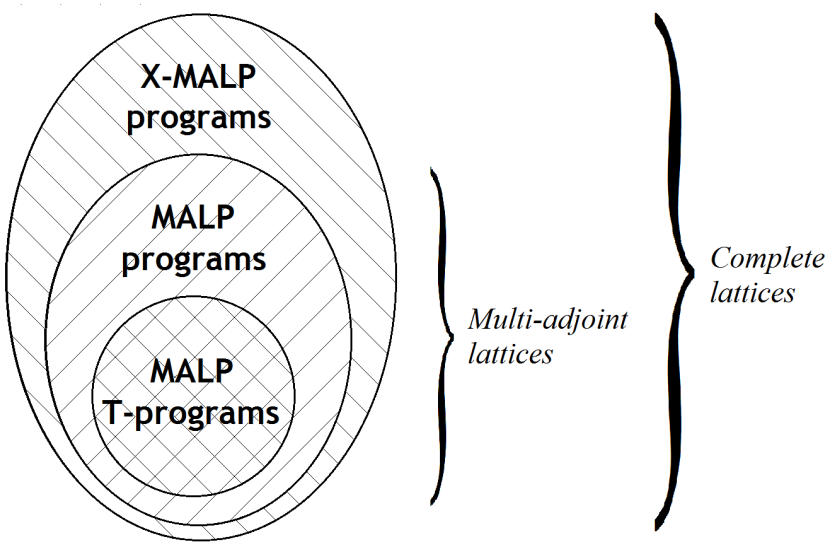


Figura 6.3: Comparación de programas MALP, ⊤-programas MALP y programas X-MALP

Basaremos esta nueva semántica declarativa en el concepto de interpretación de Herbrand difusa, tal y como se considera, entre otros, en [VP96].

Definición 6.3.3 (Interpretación de Herbrand)**.** *Una interpretación de Herbrand es una aplicación* $\mathcal{I} : B_\mathcal{P} \to L$*, en la que* $B_\mathcal{P}$ *es la base de Herbrand del programa* X-MALP $\mathcal{P}$ *y* $(L, \leq)$ *es el retículo completo asociado a* $\mathcal{P}$*.*

$\mathcal{I}$ se extiende de forma natural al conjunto de fórmulas básicas del lenguaje. Para interpretar una fórmula no básica A (cerrada y universalmente cuantificada en el caso del lenguaje X-MALP) basta con tomar $\mathcal{I}(A) = inf\{\mathcal{I}(A\xi) : A\xi$ *es una instancia básica de* $A\}$. Sea $\mathcal{H}$ el conjunto de interpretaciones de Herbrand cuyo orden está inducido por el de L

$$\mathcal{I}_j \leq \mathcal{I}_k \iff \mathcal{I}_j(F) \leq \mathcal{I}_k(F), \forall F \in B_\mathcal{P}$$

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

Definición 6.3.4 (Modelo de Herbrand de un programa X-MALP)**.** *Una interpretación de Herbrand* $\mathcal{I}$ *satisface (o es un* modelo de Herbrand *de) una regla* X-MALP

$H \leftarrow \mathcal{B}$ si y sólo si $\mathcal{I}(\mathcal{B}) \leq \mathcal{I}(H)$. Una interpretación $\mathcal{I}$ es un modelo de Herbrand de un programa X-MALP *$\mathcal{P}$ si y sólo si todas las reglas de $\mathcal{P}$ son satisfechas por $\mathcal{I}$.*

Definición 6.3.5 (Modelo mínimo de Herbrand difuso de un programa X-MALP)**.** *Sea $\mathcal{P}$ un programa* X-MALP *con un retículo (completo) asociado $(L, \leq)$. El modelo mínimo de Herbrand difuso* [5] *de $\mathcal{P}$ es la interpretación $\mathcal{I}_\mathcal{P} = inf\{\mathcal{I}_j : \mathcal{I}_j$ es modelo de Herbrand de $\mathcal{P}\}$.*

El siguiente resultado justifica que la interpretación $\mathcal{I}_\mathcal{P}$ anterior pueda ser tomada como el modelo mínimo de Herbrand difuso.

Teorema 6.3.6. *Sea $\mathcal{P}$ un programa lógico* X-MALP *con retículo (completo) asociado L. La aplicación $\mathcal{I}_\mathcal{P} = inf\{\mathcal{I}_j : \mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}\}$ es el modelo mínimo de Herbrand de $\mathcal{P}$.*

Demostración. En lo que sigue, denotamos por $\mathcal{M}$ el conjunto de interpretaciones de Herbrand que son modelos de $\mathcal{P}$, es decir, $\mathcal{M} = \{\mathcal{I}_j : \mathcal{I}_j$ *es modelo de Herbrand de* $\mathcal{P}\}$. Téngase en cuenta que $\mathcal{M}$ es un conjunto no vacío, dado que la interpretación de Herbrand $\mathcal{I} = sup(\mathcal{H})$, definida sobre cada $A \in \mathcal{B}_\mathcal{P}$ por $\mathcal{I}(A) = sup(L)$ es un modelo de Herbrand de (todas de las reglas) de $\mathcal{P}$.

Entonces, como $\mathcal{I}_\mathcal{P} = inf(\mathcal{M})$, $\mathcal{I}_\mathcal{P}$ es una interpretación de Herbrand, por ser el conjunto de interpretraciones de Herbrand $(\mathcal{H}, \leq)$ un retículo completo, existe el ínfimo del subconjunto $\mathcal{M}$ y es un elemento de $\mathcal{H}$.

Veamos, ahora, que $\mathcal{I}_\mathcal{P}$ es modelo Herbrand de $\mathcal{P}$, es decir, que satisface todas las reglas del programa.

Sea $\mathcal{R} : H \leftarrow_i \mathcal{B}$ una regla de $\mathcal{P}$. Puesto que $\mathcal{I}_\mathcal{P}$ es el ínfimo de $\mathcal{M}$, se tiene que $\mathcal{I}_\mathcal{P} \leq \mathcal{I}_j$, para cada modelo $\mathcal{I}_j$ de $\mathcal{P}$. Por tanto, $\mathcal{I}_\mathcal{P}(A) \leq \mathcal{I}_j(A)$ para cada átomo A. Por otra parte, dado que $\mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}$, $\mathcal{I}_j$ satisface la regla $\mathcal{R}$, es decir, $\mathcal{I}_j(\mathcal{B}) \leq \mathcal{I}_j(H)$.

Haciendo uso ahora de la definición de ínfimo, se tiene:

$$\mathcal{I}_\mathcal{P}(\mathcal{B}) = inf\{\mathcal{I}_j(\mathcal{B}) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} \leq$$
$$inf\{\mathcal{I}_j(H) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} = \mathcal{I}_\mathcal{P}(H)$$

En consecuencia, $\mathcal{I}_\mathcal{P}$ satisface la regla $\mathcal{R}$ considerada y (por satisfacer análogamente todas las reglas de $\mathcal{P}$) es un modelo de $\mathcal{P}$, como queríamos. Finalmente, como $\mathcal{I}_\mathcal{P} = inf(\mathcal{M})$, usando de nuevo la definición de ínfimo, $\mathcal{I}_\mathcal{P} \leq \mathcal{I}_j, \forall j$ por lo que $\mathcal{I}_\mathcal{P}$ es el menor modelo de $\mathcal{P}$, lo que termina la prueba. □

[5] A veces diremos únicamente modelo mínimo difuso o modelo mínimo.

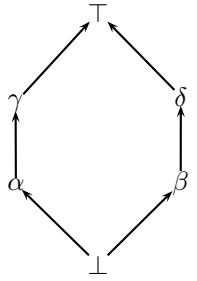
La semántica declarativa que representa el modelo mínimo de Herbrand $\mathcal{I}_{\mathcal{P}}$ extiende a la correspondiente de la programación multi-adjunta que definimos en [JMP09c] de la manera más amable que cabe esperar: si el retículo L es multi-adjunto, el modelo $\mathcal{I}_{\mathcal{P}}$ anterior coincide con el correspondiente de éste marco. Análogamente, $\mathcal{I}_{\mathcal{P}}$ extiende el concepto de modelo mínimo de la programación lógica formulado en [Llo87]; a la hora de buscar una semejanza completa entre ambos conceptos puede usarse la expresión conjuntista de $\mathcal{I}_{\mathcal{P}}$ que figura en la posterior Proposición 6.3.8.

En el ejemplo que sigue se obtiene el modelo mínimo de Herbrand de un programa X-MALP.

Ejemplo 6.3.7. *Sea* $(\&_{\mathtt{G}}, \leftarrow_{\mathtt{G}})$ *un par de conectivas que siguen la lógica intuicionista de* Gödel*, es decir, las funciones de verdad de* $\&_{\mathtt{G}}$ *y* $\leftarrow_{\mathtt{G}}$ *son:*

$$\dot{\&}_{\mathtt{G}}(x,y) = \text{ínf}\{x,y\} \qquad y \qquad \dot{\leftarrow}_{\mathtt{G}}(y,x) = \begin{cases} \top, & si\ x \leq y \\ y, & si\ x > y \end{cases}$$

En el retículo $(L, \leq)$*, cuyo diagrama de Hasse se muestra más abajo, estas dos conectivas no conforman un par adjunto, puesto que, si bien* $\alpha \dot{\&} \delta \leq \beta$*, no se cumple que* $\alpha \leq \beta \dot{\leftarrow} \delta$*.*



	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$	$\mathcal{I}_4$	$\mathcal{I}_5$	$\mathcal{I}_6$	$\mathcal{I}_7$	$\mathcal{I}_8$	$\mathcal{I}_9$
p	$\bot$	α	β	γ	δ	$\top$	α	γ	$\top$
q	δ	δ	δ	δ	δ	δ	$\top$	$\top$	$\top$

Por tanto, el programa $\mathcal{P} = \{\mathcal{R}_1, \mathcal{R}_2\}$*, siendo* $\mathcal{R}_1 : \langle p \leftarrow_{\mathtt{G}} q; \alpha \rangle$ *y* $\mathcal{R}_2 : \langle q \leftarrow_{\mathtt{G}} ; \delta \rangle$ *no es válido en el marco multi-adjunto, pues está asociado a un retículo que no es multi-adjunto. Considérese ahora el programa* X-MALP $\mathcal{P}' = \{\mathcal{R}'_1, \mathcal{R}'_2\}$*, donde* $\mathcal{R}'_1 : p \leftarrow \alpha \&_{\mathtt{G}} q$ *y* $\mathcal{R}'_2 : q \leftarrow \delta$*, que es en esencia equivalente al programa* MALP $\mathcal{P}$*. La tabla de la figura resume el conjunto* $\mathcal{M} = \{\mathcal{I}_1, \ldots, \mathcal{I}_9\}$ *de modelos de Herbrand para* $\mathcal{P}'$*. Detallamos a continuación cómo hemos obtenido esa tabla. Obsérvese que, por definición de modelo de Herbrand, cada* $\mathcal{I}_j$ *es un modelo de Herbrand de la regla* $\mathcal{R}'_2$ *si y sólo si* $\delta \leq \mathcal{I}_j(q)$*, lo que conlleva que* $\mathcal{I}_j(q)$ *admite el conjunto de valores* $\delta, \top \in L$*. Además,* $\mathcal{I}_j$ *es un modelo de Herbrand de* $\mathcal{R}'_1$ *si y sólo si* $\mathcal{I}_j(\alpha \&_{\mathtt{G}} q) \leq \mathcal{I}_j(p)$*, esto es,* $\alpha \dot{\&}_{\mathtt{G}} \mathcal{I}_j(q) \leq \mathcal{I}_j(p)$*. Entonces, una vez asociado* $\mathcal{I}_j(q)$ *a cada uno de los grados de verdad anteriores, se añade la condición* $\alpha \dot{\&}_{\mathtt{G}} \mathcal{I}_j(q) \leq \mathcal{I}_j(p)$*, que determina el*

conjunto de valores válidos para $\mathcal{I}_j(p)$ y, consiguientemente, establece el conjunto final de modelos de Herbrand $\mathcal{I}_j \in \mathcal{M}$. Finalmente, es fácil ver que $\mathcal{M}$ tiene un elemento ínfimo (la interpretación $\mathcal{I}_{\mathcal{P}'}$ tal que $\mathcal{I}_{\mathcal{P}'}(p) = \bot$ y $\mathcal{I}_{\mathcal{P}'}(q) = \delta$), así que $\mathcal{I}_{\mathcal{P}'} \in \mathcal{M}$ es el modelo mínimo de Herbrand de $\mathcal{P}'$.

Los conceptos anteriores de interpretación de Herbrand y de modelo de Herbrand pueden ser expresados en términos conjuntistas, entendiendo una interpretación de $\mathcal{P}$, en lugar de como una aplicación, como la correspondiente relación binaria. Damos a continuación el resultado elemental que justifica este hecho.

Bajo este punto de vista, puesto que cada modelo de Herbrand de $\mathcal{P}$ puede verse como un conjunto de pares (un subconjunto de $B_{\mathcal{P}} \times L$), el modelo mínimo de Herbrand difuso admite una segunda caracterización que incluye una propiedad aportada ya en [Llo87] para el caso de los modelos mínimos de Herbrand de programas lógicos puros. Esta nueva caracterización se establece en términos del siguiente teorema, cuya demostración es inmediata, donde consideramos que $\cap\mathcal{I}_j = \{(A_i; \alpha) : (A_i, \alpha_i) \in \mathcal{I}_j, \forall j, \alpha = \text{ínf}\{\alpha_i\} \in L\}$.

Proposición 6.3.8. *Sea $\mathcal{P}$ un programa* X-MALP *con retículo asociado $(L, \leq)$. Sea $\mathcal{I}$ una interpretación de Herbrand de $\mathcal{P}$, es decir, una aplicación $\mathcal{I} : \mathcal{B}_{\mathcal{P}} \to L$. Entonces $\mathcal{I}$ determina una única relación binaria $R_{\mathcal{I}} \subset \mathcal{B}_{\mathcal{P}} \times L$.*

Demostración. Es suficiente considerar que la aplicación $\mathcal{I}$ está determinada por su conjunto de imágenes, y la relación $R_{\mathcal{I}}$ es un conjunto de pares del tipo $(A, \mathcal{I}(A))$, $A \in \mathcal{B}_{\mathcal{P}}$. En consecuencia, la aplicación $\mathcal{I}$ puede entenderse como una relación binaria, es decir, como un cierto conjunto pares ordenados cuya primera componente es una fórmula básica de la base de Herbrand y cuya segunda componente es un elemento del retículo L.

□

En virtud del resultado anterior, cabe dar una interpretación de Herbrand $\mathcal{I}$ por su expresión conjuntista $R_{\mathcal{I}}$ que, cuando no sea necesario enfatizar este aspecto conjuntista, denotaremos también por $\mathcal{I}$.

En consecuencia, sin más que pensar cada modelo de Herbrand de $\mathcal{P}$ como un conjunto (subconjunto de $\mathcal{B}_{\mathcal{P}} \times L$), esto es, $\mathcal{I}_j = \{(A, \alpha) : A \in \mathcal{B}_{\mathcal{P}}, \alpha \in L\}$, el modelo mínimo de Herbrand difuso puede caracterizarse también por el siguiente resultado.

Teorema 6.3.9. *El modelo mínimo de Herbrand difuso de $\mathcal{P}$ es la intersección de todos los modelos de Herbrand $\mathcal{P}$, es decir, $\mathcal{I} = \cap\mathcal{I}_j$, donde $\mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}$, para todo j.*

6.4. Semántica de punto fijo para X-MALP

Con esta sección terminamos de recoger el contenido de [MPV14a]. Definimos aquí el operador $T_{\mathcal{P}}$ de punto fijo para un programa $\mathcal{P}$ X-MALP, extendiendo el correspondiente operador de punto fijo de un programa multi-adjunto considerado en [MOV04]. Además, concretamos sobre algunos ejemplos particulares los puntos fijos de este operador.

En un retículo completo $(L, \leq)$, una función $f : L \to L$ es monótona si, y sólo si, $\forall x, y \in L, x \leq y \Rightarrow f(x) \leq f(y)$. Un punto fijo de f es un elemento $x \in L$ tal que $f(x) = x$. El resultado básico para el estudio de puntos fijos de funciones sobre retículos es el siguiente teorema de Knaster-Tarski (véase [Tar55]). Otros teoremas de punto fijo pueden verse en [KM97, Sto04].

Teorema 6.4.1. *Sea f una función monótona sobre un retículo completo $(L, \leq)$. Entonces, f tiene un punto fijo.*

Además, el conjunto de puntos fijos de f es un retículo completo, por lo que es posible tomar el menor punto fijo de f. Éste puede obtenerse iterando f sobre $\bot \in L$, es decir, es el supremo de la sucesión no decreciente $x_0, \ldots, x_i, x_{i+1}, \ldots, x_\lambda, \ldots$ verificando que para cualquier $i \geq 0, x_0 = \bot$, $x_{i+1} = f(x_i)$, mientras que para un cierto índice λ, $y_\lambda = sup\{y_i : f(y_i) = y_i, i > \lambda\}$.

En [MOV01d, MOV04] se extiende el operador $T_{\mathcal{P}}$, definido por [EK76], para el lenguaje multi-adjunto. La siguiente definición aporta un operador de punto fijo para el lenguaje X-MALP que extiende el del marco MALP y que representa la semántica (por punto fijo) de un programa X-MALP $\mathcal{P}$, tomada ésta como el menor punto fijo de $T_{\mathcal{P}}$. Además, justificaremos en el Teorema 6.4.11 la equivalencia entre esta construcción y la noción de modelo mínimo de Herbrand difuso de la Definición 3.3.13.

Definición 6.4.2. *Sea $\mathcal{P}$ un programa lógico* X-MALP *con retículo asociado $(L, \leq)$ e $\mathcal{I}$ una interpretación de Herbrand de $\mathcal{P}$. Definimos el operador $T_{\mathcal{P}}$ como una aplicación en el conjunto de interpretaciones de Herbrand de $\mathcal{P}$ tal que, a cada átomo básico $A \in \mathcal{B}_{\mathcal{P}}$ le asocia el elemento del retículo (completo) L*

$$T_{\mathcal{P}}(\mathcal{I})(A) = sup\{\mathcal{I}(\mathcal{B}\theta) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta\}$$

Es sencillo comprobar que $T_{\mathcal{P}}$ está bien definido. Los siguientes teoremas garantizan, para la programación X-MALP, resultados que son clásicos en la LP para

el operador de punto fijo y que extienden también los correspondientes del marco multi-adjunto.

En particular, es sencillo comprobrar que el operador $T_{\mathcal{P}}$ anterior extiende al operador de punto fijo de [MOV04] de la manera más satisfactoria: si el programa $\mathcal{P}$ es multi-adjunto nuestro operador de punto fijo coincide con el debido a Medina et all.

Teorema 6.4.3. *El operador $T_{\mathcal{P}}$ es monótono.*

Demostración. $T_{\mathcal{P}}$ es el morfismo $T_{\mathcal{P}} : (\mathcal{H}, \leq) \to (\mathcal{H}, \leq)$ tal que sobre cada $\mathcal{I} \in \mathcal{H}$ determina la aplicación en la base de Herbrand $T_{\mathcal{P}}(\mathcal{I}) : \mathcal{B}_{\mathcal{P}} \to \mathcal{B}_{\mathcal{P}}$ definida por $T_{\mathcal{P}}(\mathcal{I})(A) = sup\{\mathcal{I}(\mathcal{B}\theta) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta\}$. Entonces, dadas las interpretaciones de Herbrand $\mathcal{I}_1, \mathcal{I}_2 \in \mathcal{H}$, si suponemos que $\mathcal{I}_1 \leq \mathcal{I}_2$, y como el supremo conserva el orden del retículo $(L, \leq)$ asociado a $\mathcal{P}$, se obtiene que $T_{\mathcal{P}}(\mathcal{I}_1) \leq T_{\mathcal{P}}(\mathcal{I}_2)$, es decir, $T_{\mathcal{P}}$ es monótono.

□

Definición 6.4.4. *Sea $(L, \leq)$ un retículo completo y $f : L \to L$ una aplicación. f es* continua *si, y sólo si, conserva el supremo de conjuntos dirigidos*[6]*, es decir, para todo conjunto dirigido $X \subset L$ se cumple $f(sup(X)) = sup\{f(x) : x \in X\}$. Además, la aplicación $f : L^n \to L$ se dice continua si lo es en cada variable por separado.*

El siguiente lema es instrumental. Su demostracción es sencilla, basta proceder por inducción en el número de fórmulas atómicas del cuerpo de las reglas.

Lema 6.4.5. *Sea $\mathcal{P}$ un programa* X-MALP *y $\mathcal{B}$ el cuerpo de una regla arbitraria del mismo. Si todas las funciones de verdad de las conectivas de $\mathcal{B}$ son continuas, para todo conjunto dirigido $X \subset \mathcal{H}$ se verifica $sup(X)(\mathcal{B}) = sup\{\mathcal{I}(\mathcal{B}) : \mathcal{I} \in X\}$.*

El teorema posterior garantiza la continuidad del operador de punto fijo bajo hipótesis esperadas. La condición suficiente enunciada es también una condición necesaria para dicho resultado.

Teorema 6.4.6. *Sea $\mathcal{P}$ un programa* X-MALP*. Si todas las funciones de verdad de las conectivas de los cuerpos de las reglas de $\mathcal{P}$ son continuas, el operador $T_{\mathcal{P}}$ es continuo.*

[6]Por definición, $X \subset (L, \leq)$ es un conjunto dirigido si para todo subconjunto $\{x_1, \ldots, x_n\} \subset X$, $sup\{x_1, \ldots, x_n\} \in X$.

Demostración. Sea X un conjunto dirigido del retículo completo, $(\mathcal{H}, \leq)$, de interpretaciones de Herbrand de $\mathcal{P}$. Probemos que $T_{\mathcal{P}}(sup(X)) = sup\{T_{\mathcal{P}}(\mathcal{I})\}$; en efecto, para cada $A \in \mathcal{B}_{\mathcal{P}}$,

$$\begin{aligned} T_{\mathcal{P}}(sup(X))(A) &= sup\{sup(X)(\mathcal{B}) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta\} \\ &= sup\{sup\{\mathcal{I}(\mathcal{B}) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta, \mathcal{I} \in X\}\} \\ &= sup\{T_{\mathcal{P}}(\mathcal{I})(A) : \mathcal{I} \in X\} \end{aligned}$$

sin más que hacer uso del Lema 6.4.5. □

A continuación demostramos el siguiente resultado por el que $T_{\mathcal{P}}$ permite caracterizar las interpretaciones que son modelo de $\mathcal{P}$. Previamente, enunciamos el lema básico que sigue.

Lema 6.4.7. *Sea $(L, \leq)$ un retículo completo. Para cualesquiera subconjuntos A, B de L se verifica que si $A \subset B$, entonces $inf(B) \leq inf(A)$.*

Demostración. Es suficiente considerar la definición de ínfimo y el carácter completo del retículo $(L, \leq)$. □

Obsérvese que, en virtud de lema anterior, resulta que $\mathcal{I}(A\theta) \geq \mathcal{I}(A)$, para toda sustitución θ, para toda interpretación de Herbrand $\mathcal{I}$, toda vez que el conjunto de instancias básicas de la fórmula $A\theta$ está contenido en el conjunto de instancias básicas de A.

Teorema 6.4.8. *Una interpretación de Herbrand $\mathcal{I}$ es un modelo de Herbrand de un programa* X-MALP *$\mathcal{P}$ si, y sólo si, $T_{\mathcal{P}}(\mathcal{I}) \leq \mathcal{I}$.*

Demostración. Sea $\mathcal{I}$ un modelo de Herbrand de $\mathcal{P}$ y veamos que $T_{\mathcal{P}}(\mathcal{I}) \leq \mathcal{I}$. Si $H \leftarrow_i \mathcal{B}$ es una regla de $\mathcal{P}$, por definición de modelo y por el Lema 6.4.7 se tiene $\mathcal{I}(\mathcal{B}\theta) \leq \mathcal{I}(H) \leq \mathcal{I}(H\theta) = \mathcal{I}(A)$, de donde obtenemos que $T_{\mathcal{P}}(\mathcal{I})(A) \leq \mathcal{I}(A)$ por la definición de supremo, como queríamos.

Recíprocamente, supongamos ahora que $T_{\mathcal{P}}(\mathcal{I}) \leq \mathcal{I}$ y probemos que $\mathcal{I}$ es un modelo de Herbrand de $\mathcal{P}$, es decir, es modelo de una regla arbitraria $\mathcal{R} : H \leftarrow_i \mathcal{B} \in \mathcal{P}$. Fijada esta regla, sea $A \in \mathcal{B}_{\mathcal{P}}$ tal que $H\theta = A$[7]. Como por hipótesis se tiene $T_{\mathcal{P}}(\mathcal{I})(A) = sup\{\mathcal{I}(\mathcal{B}\theta) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta\} \leq \mathcal{I}(A)$, resulta que $\mathcal{I}(\mathcal{B}) \leq \mathcal{I}(\mathcal{B}\theta) \leq sup\{\mathcal{I}(\mathcal{B}\theta) : H \leftarrow_i \mathcal{B} \in \mathcal{P}, A = H\theta\} \leq \mathcal{I}(A)$, por lo que $\mathcal{I}$ es modelo de la regla $\mathcal{R}$. □

[7] La existencia de éste átomo está garantizada por la definición de universo y base de Herbrand de $\mathcal{P}$.

En virtud del teorema anterior, para todo modelo de Herbrand $\mathcal{I}$ de $\mathcal{P}$ tenemos $T_{\mathcal{P}}(\mathcal{I})(A) = sup\{\mathcal{I}(\mathcal{B}\theta) : H \leftarrow \mathcal{B} \in \mathcal{P}, A = H\theta\} \leq \mathcal{I}(A)$ y, además, la igualdad $T_{\mathcal{P}}(\mathcal{I})(A) = \mathcal{I}(A)$ no se alcanza en general, como ocurre en el caso de la programación lógica pura; podemos sugerir que cuanto más se itere el operador $T_{\mathcal{P}}$ sobre un átomo, mejor es la cota superior obtenida para la correspondiente respuesta correcta.

Por otra parte, tanto en la programación lógica pura como en la programación lógica X-MALP puede existir más de un punto fijo. El siguiente ejemplo ilustra este hecho para el caso clásico.

Ejemplo 6.4.9. *Sea $\mathcal{P}$ el programa lógico formado por la cláusula $p(a) \leftarrow p(a)$. La base de Herbrand de $\mathcal{P}$ es el conjunto $\mathcal{B}_{\mathcal{P}} = \{p(a)\}$ y los subconjuntos de la base de Herbrand $\mathcal{I}_1 = \varnothing, \mathcal{I}_2 = \{p(a)\}$ son los únicos modelos de $\mathcal{P}$. Ambos son puntos fijos del operador (de punto fijo) definido en [Llo87].*

En el contexto X-MALP podríamos tomar el mismo ejemplo, toda vez que cualquier programa lógico definido puede ser expresado como un programa X-MALP. Con la intención de considerar un caso más específico aportamos también el siguiente ejemplo.

Ejemplo 6.4.10. *Sea $\mathcal{P}$ el programa* X-MALP *con la única regla $p(a) \leftarrow 0.5 \&_{\mathtt{G}} p(a)$ con retículo asociado el intervalo $([0,1], \leq)$, y donde la función de verdad de la conectiva $\&_{\mathtt{G}}$ está dada por $\dot{\&}_{\mathtt{G}}(x,y) = inf\{x,y\}$. Entonces, las interpretaciones $\mathcal{I}_1$, $\mathcal{I}_2$ definidas por $\mathcal{I}_1(p(a)) = 0$, $\mathcal{I}_2(p(a)) = 0.5$ verifican:*

$$T_{\mathcal{P}}(\mathcal{I}_1)(p(a)) = sup\{0.5\dot{\&}\mathcal{I}_1(p(a))\} = sup\{0.5\dot{\&}0\} = 0 = \mathcal{I}_1(p(a))$$

$$T_{\mathcal{P}}(\mathcal{I}_2)(p(a)) = sup\{0.5\dot{\&}\mathcal{I}_2(p(a))\} = sup\{0.5\dot{\&}0.5\} = 0.5 = \mathcal{I}_2(p(a))$$

y, por tanto, ambos son puntos fijos del operador $T_{\mathcal{P}}$, resultando $\mathcal{I}_1$ el menor punto fijo. Además, es sencillo concretar que el operador $T_{\mathcal{P}}$ tiene infinitos puntos fijos: todas las interpretaciones de Herbrand $\mathcal{I}$ definidas por $\mathcal{I}(p(a)) = z$, donde $z \in [0, 0.5]$.

Veremos que la semántica declarativa de modelo mínimo de Herbrand difuso es equivalente a la semántica de punto fijo para la programación X-MALP.

Teorema 6.4.11. *Dado el programa* X-MALP *$\mathcal{P}$ con retículo (completo) asociado $(L, \leq)$, $\mathcal{I}_{\mathcal{P}}$ es modelo mínimo de Herbrand difuso si, y sólo si, $\mathcal{I}_{\mathcal{P}}$ es el menor punto fijo de $T_{\mathcal{P}}$.*

Demostración. Por el Teorema 6.3.6, $\mathcal{I}_\mathcal{P}$ es el modelo mímino de Herbrand difuso si, y sólo si, $\mathcal{I}_\mathcal{P} = \inf\{\mathcal{I}_j : \mathcal{I}_j \text{ es modelo de } \mathcal{P}\}$ y, por el Teorema 6.4.8, $\mathcal{I}_\mathcal{P} = \inf\{\mathcal{I}_j : T_\mathcal{P}(\mathcal{I}_j) \leq \mathcal{I}_j\}$. Además, por ser $T_\mathcal{P}$ un operador monótono en el retículo completo L (véase el Teorema 6.4.3), existe el menor punto fijo de $T_\mathcal{P}$ y coincide con el $\inf\{\mathcal{I}_j : T_\mathcal{P}(\mathcal{I}_j) \leq \mathcal{I}_j\}$ (véase [Llo87]). En consecuencia, $\mathcal{I}_\mathcal{P}$ es el menor punto fijo del operador $T_\mathcal{P}$. □

Este último resultado afirma que la semántica declarativa de un programa X-MALP $\mathcal{P}$ puede obtenerse por iteración de $T_\mathcal{P}$ sobre la interpretación mínima. Debe tenerse en cuenta que $T_\mathcal{P}$ puede no ser continuo, en cuyo caso –a diferencia de la programación lógica pura– puede ser necesario un número infinito no numerable de iteraciones para alcanzar el punto fijo.

6.5. El lenguaje FASILL

Recogemos en esta sección y las siguientes el lenguaje FASILL, descrito en primer lugar en [JMPV14, IMPV15]. FASILL (de sus siglas en inglés, *Fuzzy Aggregators and Similarity Into a Logic Language*) es un lenguaje de primer orden construido sobre una signatura Σ que contiene los elementos de un conjunto infinito contable de variables $\mathcal{V}$, símbolos de función y de predicado con una aridad asociada –expresados comúnmente como pares f/n o p/n donde n representa la aridad de f o p–, el símbolo de implicación ($\leftarrow$) y un amplio conjunto de conectivas. El lenguaje combina los elementos de Σ formando términos, átomos, reglas y fórmulas. Una *constante* c es un símbolo de función de aridad cero. Un *término* es una variable, una constante o un símbolo de función f/n aplicado a n términos $t_1, \ldots, t_n$, y se denota por $f(t_1, \ldots, t_n)$. Permitimos también los elementos de un retículo completo L como parte de la signatura Σ. Por tanto, una fórmula bien formada puede ser:

- r, si $r \in L$
- $p(t_1, \ldots, t_n)$, si $t_1, \ldots, t_n$ son términos y p/n es un predicado n-ario. Esta fórmula se llama *átomo*. Particularmente, los átomos que no contienen variables se llaman *átomos básicos*, y los átomos construidos a partir de predicados nularios se llaman *variables proposicionales*
- $\varsigma(\mathcal{F}_1, \ldots, \mathcal{F}_n)$, si $\mathcal{F}_1, \ldots, \mathcal{F}_n$ son fórmulas bien formadas y ς es una conectiva n-aria con función de verdad $\dot{\varsigma} : L^n \to L$

Definición 6.5.1 (Retículo completo)**.** *Un retículo completo es un conjunto parcialmente ordenado $(L, \leq)$ tal que cada subconjunto S de L tiene elementos ínfimo y supremo. Por tanto, es un retículo acotado, esto es, tiene elementos ínfimo y supremo, denotados por $\bot$ y $\top$, respectivamente. Se dice que L es el conjunto soporte del retículo, y $\leq$ su relación de orden.*

El lenguaje está equipado con un conjunto de *conectivas*[8] interpretadas en el retículo, como

- agregadores, denotados por @, cuya función de verdad $\dot{@}$ cumple la condición de frontera:$\dot{@}(\top,\top) = \top$, $\dot{@}(\bot, \bot) = \bot$, y de monotonía: $(x_1, y_1) \leq (x_2, y_2) \Rightarrow \dot{@}(x_1, y_1) \leq \dot{@}(x_2, y_2)$.

- t-normas y t-conormas [NW06] (también llamadas conjunciones y disyunciones, que denotamos por & y |, respectivamente) cuyas funciones de verdad cumplen las siguientes propiedades:

  · Conmutativa: $\dot{\&}(x, y) = \dot{\&}(y, x)$ $\quad \dot{|}(x, y) = \dot{|}(y, x)$

  · Asociativa: $\dot{\&}(x, \dot{\&}(y, z)) = \dot{\&}(\dot{\&}(x, y), z)$ $\quad \dot{|}(x, \dot{|}(y, z)) = \dot{|}(\dot{|}(x, y), z)$

  · Elemento neutro: $\dot{\&}(x, \top) = x$ $\quad \dot{|}(x, \bot) = x$

  · Monotonía en cada argumento:

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

Ejemplo 6.5.2. *A lo largo de este capítulo, emplearemos el retículo $([0, 1], \leq)$, donde $\leq$ es la relación de orden usual en los números reales, junto a tres conjuntos de conectivas que corresponden a las lógicas difusas de Gödel, Łukasiewicz y Producto, definidas en la Figura 6.4, donde las etiquetas* `L`*,* `G` *y* `P` *significan, respectivamente,* lógica de Łukasiewicz*,* lógica de Gödel *y* lógica del producto *(con diferentes capacidades para modelar escenarios* optimistas*,* realistas *y* pesimistas*).*

También es posible incluir otras conectivas. Por ejemplo, la media aritmética, definida por la conectiva $@_{aver}$ (con función de verdad $\dot{@}_{aver}(x, y) \triangleq \frac{x+y}{2}$) y que es una conectiva bien establecida y fácil de entender que no pertenece a ninguna lógica concreta. También se pueden incluir conectivas con aridades diferentes de 2, como el agregador $@_{very}$, definido por $\dot{@}_{very}(x) \triangleq x^2$, que es una conectiva unaria.

[8] Aquí, las conectivas son operaciones binarias, pero normalmente las generalizamos para un número arbitrario de argumentos.

$\dot{\&}_{\mathtt{P}}(x,y) \triangleq x * y$	$\dot{\mid}_{\mathtt{P}}(x,y) \triangleq x + y - xy$	*Producto*
$\dot{\&}_{\mathtt{G}}(x,y) \triangleq \text{mín}(x,y)$	$\dot{\mid}_{\mathtt{G}}(x,y) \triangleq max(x,y)$	*Gödel*
$\dot{\&}_{\mathtt{L}}(x,y) \triangleq \text{máx}(0, x+y-1)$	$\dot{\mid}_{\mathtt{L}}(x,y) \triangleq \text{mín}(x+y, 1)$	*Łukasiewicz*

Figura 6.4: Conjunciones y disyunciones en $[0,1]$ para las lógicas difusas del *Producto*, *Łukasiewicz*, y *Gödel*.

Definición 6.5.3 (Relación de similaridad)**.** *Dado un dominio $\mathcal{U}$ y un retículo L con una t-norma $\wedge$, una relación de similaridad $\mathcal{R}$ es una relación binaria en $\mathcal{U}$, esto es, un subconjunto difuso en $\mathcal{U} \times \mathcal{U}$ (dicho de otro modo, una aplicación $\mathcal{R} : \mathcal{U} \times \mathcal{U} \to L$), tal que cumple las siguientes propiedades*[9]*:*

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

Ciertamente, estamos interesados en relaciones binarias difusas sobre un dominio sintáctico. Definimos en primer lugar similaridades sobre los símbolos de una signatura, Σ, de un lenguaje de primer orden. Esto hace posible tratar indistintamente dos símbolos sintácticos relacionados por la relación de similaridad $\mathcal{R}$. Además, una relación de similaridad $\mathcal{R}$ sobre el alfabeto de un lenguaje de primer orden se puede extender al conjunto de términos por inducción estructural del modo usual [Ses02]. Esto es, la extensión, $\hat{\mathcal{R}}$, de una relación de similaridad $\mathcal{R}$ se define como:

1. Sea x una variable, $\hat{\mathcal{R}}(x,x) = \mathcal{R}(x,x) = 1$,
2. Sean f y g dos símbolos de función o predicado n-arios, y $t_1, \ldots, t_n$, $s_1, \ldots, s_n$ términos,

   $\hat{\mathcal{R}}(f(t_1,\ldots,t_n), g(s_1,\ldots,s_n)) = \mathcal{R}(f,g) \wedge (\bigwedge_{i=1}^{n} \hat{\mathcal{R}}(t_i,s_i))$
3. En otro caso, el grado de similaridad entre dos términos es el ínfimo del retículo (cero, en el retículo $([0,1], \leq)$).

Se procede análogamente para fórmulas atómicas. En lo que sigue, no distinguiremos entre $\mathcal{R}$ y su extensión $\hat{\mathcal{R}}$.

[9] Por conveniencia, $\mathcal{R}(x,y)$, también denotado por $x\mathcal{R}y$, se refiere tanto a la expresión sintáctica (que simboliza que los elementos $x, y \in \mathcal{U}$ están relacionados por $\mathcal{R}$) como al grado de verdad $\mu_{\mathcal{R}}(x,y)$ (el grado de afinidad del par $(x,y) \in \mathcal{U} \times \mathcal{U}$ con el predicado verbal $\mathcal{R}$).

Ejemplo 6.5.4. *Definimos una relación de similaridad $\mathcal{R}$ sobre los elementos de $\mathcal{U} = \{vanguardista,\ elegante,\ metro,\ taxi,\ bus\}$ mediante la siguiente matriz:*

$\mathcal{R}$	*vanguardista*	*elegante*	*metro*	*taxi*	*bus*
vanguardista	*1*	*0.6*	*0*	*0*	*0*
elegante	*0.6*	*1*	*0*	*0*	*0*
metro	*0*	*0*	*1*	*0.4*	*0.5*
taxi	*0*	*0*	*0.4*	*1*	*0.4*
bus	*0*	*0*	*0.5*	*0.4*	*1*

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

Ahondando en el ejemplo, los términos $elegante(taxi)$ y $vanguardista(metro)$ son similares mediante extensión $\hat{\mathcal{R}}$ de $\mathcal{R}$, pues:

$$\hat{\mathcal{R}}(elegante(taxi), vanguardista(metro)) =$$
$$\mathcal{R}(elegante, vanguardista) \wedge \hat{\mathcal{R}}(taxi, metro) =$$
$$0.6 \wedge \mathcal{R}(taxi, metro) = 0.6 \wedge 0.4 = 0.4$$

Definición 6.5.5 (Regla)**.** *Una* regla *tiene la forma $A \leftarrow \mathcal{B}$, donde A es una fórmula atómica llamada* cabeza *y $\mathcal{B}$, llamado* cuerpo*, es una fórmula bien formada (construida en última instancia por fórmulas atómicas $B_1, \ldots, B_n$, grados de verdad de L y conectivas)*[10]*. Particularmente, cuando el cuerpo de una regla es $r \in L$ (un elemento del retículo L), dicha regla se llama* hecho *y se puede escribir como $A \leftarrow r$ (o simplemente como A si $r = \top$).*

Definición 6.5.6 (Programa)**.** *Un programa $\mathcal{P}$ es una tupla $\langle \Pi, \mathcal{R}, L \rangle$, donde Π es un conjunto de reglas, $\mathcal{R}$ es una relación de similaridad entre los elementos de Σ, y L es un retículo completo.*

Ejemplo 6.5.7. *El conjunto de reglas Π mostrado abajo, la relación de similaridad $\mathcal{R}$ del Ejemplo 6.5.4 y el retículo $L = ([0,1], \leq)$ del Ejemplo 6.5.2, conforman un programa. $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$.*

[10] Para englobar las convenciones sintácticas de MALP, en los programas FASILL admitimos también *reglas con pesos*, con la forma $A \leftarrow_i \mathcal{B}$ *with* v, que se transforma internamente en $A \leftarrow v \&_i \mathcal{B}$ (esta transformación preserva el significado de las reglas, como se recoge en la sección anterior).

$$\Pi \begin{cases} R_1: \ vanguardista(hydropolis) & \leftarrow 0.9 \\ R_2: \ elegante(ritz) & \leftarrow 0.8 \\ R_3: \ cercano(hydropolis, taxi) & \leftarrow 0.7 \\ R_4: \ buen\_hotel(x) & \leftarrow @_{aver}(elegante(x), @_{very}(cercano(x, metro))) \end{cases}$$

6.6. Semántica operacional de FASILL

Las reglas en los programas FASILL tienen el mismo rol que las cláusulas en PROLOG (o que las reglas en programas MALP [MOV04, JMP09c, MPV14c]), que es el de establecer que cierto predicado relaciona unos términos (la *cabeza*) si se cumplen unas condiciones (el *cuerpo*).

Como otros lenguajes lógicos, FASILL incorpora el concepto de sustitución, unificador y unificador más general (*mgu*). Algunos de ellos en una versión extendida para tratar con similaridades. Concretamente, siguiendo la línea de BOUSI~PROLOG [JR09a], el unificador más general se reemplaza por el concepto de *unificador débil más general* (por sus siglas en inglés, w.m.g.u) e introduce un algoritmo de unificación débil para computarlo. El *algoritmo de unificación débil* establece que dos *expresiones* (términos o fórmulas atómicas) $f(t_1, \ldots, t_n)$ y $g(s_1, \ldots, s_n)$ unifican débilmente si sus símbolos radicales f y g son similares con un determinado grado de verdad ($\mathcal{R}(f, g) = r > \bot$) y cada uno de sus argumentos t_i y s_i unifican débilmente. Por tanto, existe un unificador débil para dos expresiones incluso si los símbolos en sus raíces no son iguales sintácticamente ($f \not\equiv g$).

Técnicamente hablando, el algoritmo de unificación débil que utiliza FASILL es una reformulación/extensión del que aparece en [Ses02] para retículos completos arbitrarios. Lo formalizamos como un sistema de transición de estados basado en la relación de unificación basada en similaridades "$\Rightarrow$". La unificación de dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ se obtiene por la secuencia de estados que compienza por el estado inicial $\langle G \equiv \{\mathcal{E}_1 \approx \mathcal{E}_2\}, id, \alpha_0\rangle$, donde id es la sustitución identidad y $\alpha_0 = \top$ es el supremo de $(L, \leq)$: $\langle G, id, \alpha_0\rangle \Rightarrow \langle G_1, \theta_1, \alpha_1\rangle \Rightarrow \cdots \Rightarrow \langle G_n, \theta_n, \alpha_n\rangle$. Cuando se alcanza el estado final $\langle G_n, \theta_n, \alpha_n\rangle$, donde $G_n = \emptyset$ (momento en que se han resuelto todas las ecuaciones del estado inicial), las expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ son unificables por similaridad con w.m.g.u. θ_n y *grado de unificación* α_n. Por tanto, el estado final $\langle \emptyset, \theta_n, \alpha_n\rangle$ apunta a que la unificación ha tenido éxito. Por otro lado, cuando las expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ no son unificables, la secuencia de estados termina en fallo ($G_n = Fail$).

La *relación de unificación basada en similaridad*, "$\Rightarrow$", se define como la menor

relación derivada del siguiente conjunto de reglas de transición (donde $\mathcal{V}ar(t)$ denota el conjunto de variables de un término dado t)

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

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

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

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

La regla 1 descompone dos expresiones y anota la relación entre los símbolos de función (o predicado) de sus raíces. La segunda regla elimina información espúrea, y la cuarta intercambia la posición de los símbolos para facilitar su manejo por las otras reglas. Las reglas tercera y quinta realizan un chequeo de ocurrencia (en inglés, *occur check*) de la variable X en un término t. En caso de éxito, genera una sustitución $\{X/t\}$; en caso contrario el algoritmo termina en fallo. Puede terminar igualmente en fallo si la relación entre los símbolos de función (o predicado) en $\mathcal{R}$ es $\bot$, como establece la regla 6.

Por lo general, dadas dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$, si hay una secuencia de estados que acaba en éxito, $\langle\{\mathcal{E}_1\approx\mathcal{E}_2\},id,\top\rangle\Rightarrow^\star\langle\emptyset,\theta,r\rangle$, entonces escribimos que $wmgu(\mathcal{E}_1,\mathcal{E}_2)=\langle\theta,r\rangle$, siendo θ el *unificador débil más general* de $\mathcal{E}_1$ y $\mathcal{E}_2$, y r su *grado de unificación*.

Finalmente, obsérvese que en general un w.m.g.u de dos expresiones $\mathcal{E}_1$ y $\mathcal{E}_2$ no es único [Ses02]. Ciertamente, el algoritmo de unificación débil computa sólo un representante de una clase w.m.g.u, en el sentido de que, si $\theta=\{x_1/t_1,\ldots,x_n/t_n\}$ es un w.m.g.u., con grado β, entonces, por definición, cualquier sustitución $\theta'=\{x_1/s_1,\ldots,x_n/s_n\}$, que satisfaga $\mathcal{R}(s_i,t_i)>\bot$, para cada $1\leq i\leq n$, es también un w.m.g.u. con grado de similaridad $\beta'=\beta\wedge(\bigwedge_1^n\mathcal{R}(s_i,t_i))$, donde "$\wedge$" es la t-norma seleccionada. Sin embargo, nótese que el representante de la clase de w.m.g.u.'s computado por el algoritmo de unificación débil tiene un grado de similaridad mayor o igual que cualquier otro w.m.g.u. Como ocurre con el algoritmo de unificación clásico, nuestro algoritmo siempre termina devolviendo un éxito o un fallo.

A continuación ilustramos el proceso de unificación débil con el siguiente ejemplo.

Ejemplo 6.6.1. *Considérese el retículo* $L = ([0,1], \leq)$ *del Ejemplo 6.5.2 y la relación* $\mathcal{R}$ *del Ejemplo 6.5.4. Dados los términos* $elegante(taxi)$ *y* $vanguardista(metro)$*, el siguiente proceso de unificación débil puede tener lugar:*

$$\langle \{elegante(taxi) \approx vanguardista(metro)\}, id, 1\rangle \overset{1}{\Rightarrow} \langle \{taxi \approx metro\}, id, 0.6\rangle \overset{1}{\Rightarrow}$$
$$\langle \{\}, id, 0.6 \wedge 0.4\rangle = \langle \{\}, id, 0.4\rangle$$

También es posible la unificación de los términos $elegante(taxi)$ *y* $vanguardista(X)$*, pues:*

$$\langle \{elegante(taxi) \approx vanguardista(X)\}, id, 1\rangle \overset{1}{\Rightarrow} \langle \{taxi \approx X\}, id, 0.6\rangle \overset{4}{\Rightarrow}$$
$$\langle \{X \approx taxi\}, id, 0.6\rangle \overset{3}{\Rightarrow} \langle \{\}, \{X/taxi\}, 0.6\rangle$$

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

Para describir la semántica operacional del lenguaje FASILL, en adelante denotamos por $\mathcal{C}[A]$ una fórmula donde A es una sub-expresión (comúnmente un átomo) que ocurre en el contexto –posiblemente vacío– $\mathcal{C}[]$ mientras que $\mathcal{C}[A/A']$ significa el reemplazo de A por A' en el contexto $\mathcal{C}[]$. Además, $\mathcal{V}ar(s)$ denota el conjunto de variables distintas que aparecen en el objeto sintáctico s y $\theta[\mathcal{V}ar(s)]$ se refiere a la sustitución obtenida de θ restringiendo su dominio a $\mathcal{V}ar(s)$. En la siguiente definición, consideramos siempre que A es el átomo seleccionado en un objetivo $\mathcal{Q}$ y L es el retículo completo asociado al programa Π.

Definición 6.6.2 (Paso de computación)**.** *Sean* $\mathcal{Q}$ *un objetivo y* σ *una sustitución. El par* $\langle \mathcal{Q}; \sigma\rangle$ *es un* estado*. Dado un programa* $\langle \Pi, \mathcal{R}, L\rangle$ *y una t-norma* $\wedge$ *en* L*, una* computación *se formaliza como un sistema de transición de estados, cuya relación de transición* $\rightsquigarrow$ *es la menor relación que satisface estas reglas:*
1) Paso de éxito *(en inglés, successful step, denotado por* $\overset{SS}{\rightsquigarrow}$*):*

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

2) Paso de fallo *(en inglés, failure step, denotado por* $\overset{FS}{\rightsquigarrow}$*):*

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

3) Paso interpretativo *(en inglés, Interpretive step, denotado por* $\overset{IS}{\rightsquigarrow}$*):*

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

Una *derivación* es una secuencia de longitud arbitraria $\langle \mathcal{Q}; id\rangle \rightsquigarrow^* \langle \mathcal{Q}'; \sigma\rangle$. Como es usual, las reglas se renombran en cada paso. Cuando $\mathcal{Q}' = r \in L$, el estado $\langle r;\sigma\rangle$ se llama *respuesta computada difusa* (f.c.a. por sus siglas en inglés, *fuzzy computed answer*) para esa derivación.

Ejemplo 6.6.3. *Considérese el objetivo* $\mathcal{Q} = buen\_hotel(X)$ *y el programa* $\mathcal{P} = \langle \Pi, \mathcal{R}, L\rangle$ *del Ejemplo 6.5.7. Es posible realizar estas dos derivaciones para* $\mathcal{P}$ *y* $\mathcal{Q}$*:*

$$\begin{array}{lll}
D_1: & \langle buen_hotel(X), id\rangle & \overset{SS^{R4}}{\rightsquigarrow} \\
& \langle @_{aver}(elegante(X), @_{very}(cercano(X, metro))), \{X_1/X\}\rangle & \overset{SS^{R2}}{\rightsquigarrow} \\
& \langle @_{aver}(0.8, @_{very}(cercano(ritz, metro))), \{X_1/ritz, X/ritz\}\rangle & \overset{FS}{\rightsquigarrow} \\
& \langle @_{aver}(0.8, @_{very}(0)), \{X_1/ritz, X/ritz\}\rangle & \overset{IS}{\rightsquigarrow} \\
& \langle @_{aver}(0.8, 0), \{X_1/ritz, X/ritz\}\rangle & \overset{IS}{\rightsquigarrow} \\
& \langle 0.4, \{X_1/ritz, X/ritz\}\rangle & \\
D_2: & \langle buen_hotel(X), id\rangle & \overset{SS^{R4}}{\rightsquigarrow} \\
& \langle @_{aver}(elegante(X), @_{very}(cercano(X, metro))), \{X_1/X\}\rangle & \overset{SS^{R1}}{\rightsquigarrow} \\
& \langle @_{aver}(\&_{godel}(0.9, 0.6), @_{very}(cercabo(hydropolis, metro))), \{\ldots\}\rangle & \overset{SS^{R3}}{\rightsquigarrow} \\
& \langle @_{aver}(\&_{godel}(0.9, 0.6), @_{very}(\&_{godel}(0.7, 0.4))), \{\ldots\}\rangle & \overset{IS}{\rightsquigarrow} \\
& \langle @_{aver}(0.6, @_{very}(0.4)), \{X_1/hydropolis, X/hydropolis\}\rangle & \overset{IS}{\rightsquigarrow} \\
& \langle @_{aver}(0.6, 0.16), \{X_1/hydropolis, X/hydropolis\}\rangle & \overset{IS}{\rightsquigarrow} \\
& \langle 0.38, \{X_1/hydropolis, X/hydropolis\}\rangle &
\end{array}$$

con respuestas computadas difusas $\langle 0,4, \{X/ritz\}\rangle$ *y* $\langle 0.38, \{X/hydropolis\}\rangle$*, respectivamente.*

6.7. Implementación de FASILL en $\mathcal{FLOPER}$

Como indicamos en el Capítulo 4, durante los últimos años hemos desarrollado la herramienta $\mathcal{FLOPER}$ que, inicialmente, manipulaba únicamente programas MALP[11].

[11] El lenguaje MALP ha quedado totalmente subsumido por el nuevo lenguaje FASILL, ya que, dado un programa FASILL $\mathcal{P} = \langle \Pi, \mathcal{R}, L\rangle$, si $\mathcal{R}$ es la relación identidad (esto es, aquella en la que cada elemento de una signatura Σ es similar únicamente a sí mismo, con el máximo grado

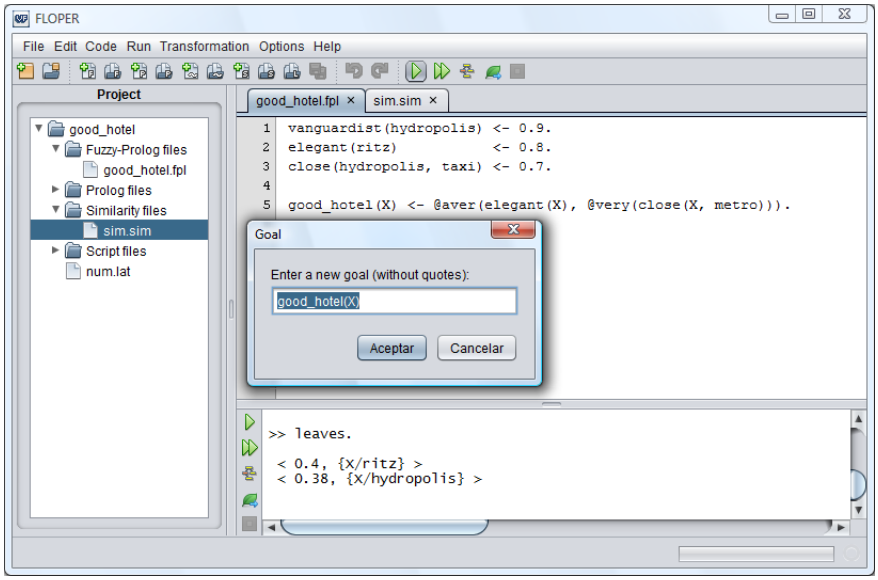


Figura 6.5: Captura de pantalla de una sesión de trabajo en $\mathcal{FLOPER}$

En su actual estado de desarrollo, $\mathcal{FLOPER}$ ha sido equipado con nuevas características para tratar con lenguajes más expresivos, particularmente FASILL, como detallamos en [IMPV15, JMPV14]. La nueva versión de $\mathcal{FLOPER}$ se puede descargar gratuitamente a través de la url `http://dectau.uclm.es/floper/?q=sim`, donde también es posible efectuar ejecución online. $\mathcal{FLOPER}$ ha sido ya profusamente descrito en el Capítulo 4, de modo que en esta sección nos limitamos a indicar las características introducidas al respecto del nuevo lenguaje FASILL. En primer lugar, en el menú del intérprete de comandos de $\mathcal{FLOPER}$ hemos introducido un nuevo submenú titulado **Similarity Menu**, que agrupa los siguientes comandos:

- **sim**: permite al usuario introducir un fichero de similaridad (con extensión "`.sim`", y cuya sintaxis detallamos más adelante).

- **tnorm**: establece la conjunción, definida en el fichero del retículo, que se usará para realizar la clausura transitiva de la relación.

de similaridad) y L es un retículo completo que contiene *pares adjuntos* [MOV04], entonces $\mathcal{P}$ es también un programa MALP.

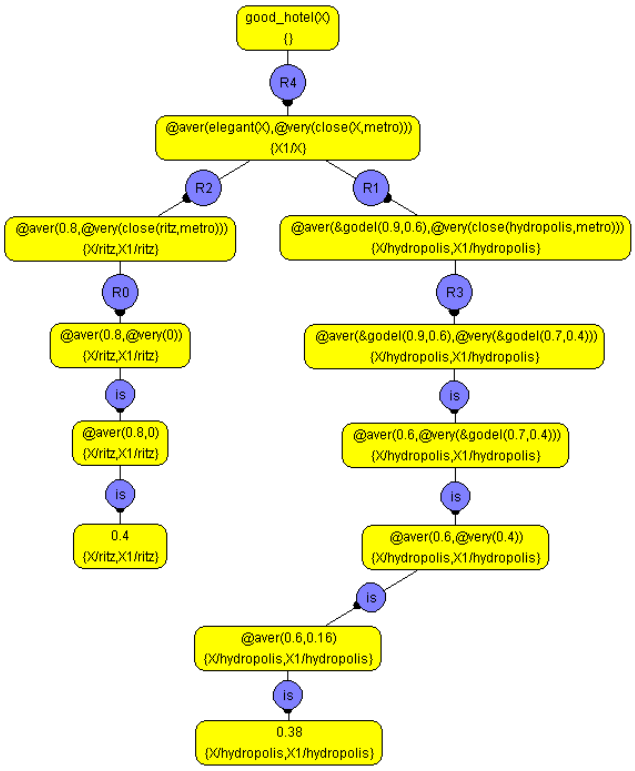


Figura 6.6: Árbol de ejecución mostrado por el sistema $\mathcal{FLOPER}$

La sintaxis de FASILL presentada en la Sección 6.6, muy similar a la de MALP, se puede adaptar fácilmente para ser escrita en un ordenador. Como es usual en los lenguajes lógicos, la variables se escriben como identificadores que empiezan por una letra mayúscula o el caracter subrayado "`_`", mientras que los símbolos de función y de predicado se expresan con identificadores que empiezan por una letra en minúscula, y los números son literales. Los términos y los átomos tienen la sintaxis usual (al símbolo de función o de predicado, si no es nulario, se le sigue de sus argumentos entre paréntesis, separados cada uno por una coma). Las conectivas se etiquetan con su nombre. El símbolo de implicación se escribe como "`<-`", y cada regla termina con un punto. Adicionalmente es posible incluir expresiones Prolog puras en el cuerpo de una regla, encapsulándolas entre llaves "`{}`", y cláusulas Prolog entre símbolos

de dólar "`$`", junto con las reglas FASILL.

La interfaz gráfica (hecha en Java) permite una interacción intuitiva con el usuario, como se refleja en la Figura 6.5. Para incluir el lenguaje FASILL, La interfaz gráfica de $\mathcal{FLOPER}$ ha sido modificada en los siguientes puntos:

- Se han añadido opciones en el menú contextual y en el menú **Code** para crear un nuevo fichero de similaridades ("`.sim`") y para abrir un fichero de similaridades ya presente en un directorio.
- Se ha incluido también un botón en la barra de atajos para crear o abrir un fichero de similaridades.
- Los proyectos $\mathcal{FLOPER}$ incorporan un nuevo directorio `sim` que almacena los ficheros de similaridades asociados al mismo.

A continuación pasamos a describir la principal novedad introducida en $\mathcal{FLOPER}$, que es la posibilidad de usar relaciones de similaridad. Efectivamente, el sistema admite ahora, a través de la opción **sim**, ficheros con extensión `.sim` (mencionados anteriormente), que representan un subconjunto de una relación de similaridad $\mathcal{R}$ siguiendo una sintaxis concreta:

$\langle Relation \rangle$ ::= $\langle Sim \rangle$ $\langle Relation \rangle$ | $\langle Sim \rangle$
$\langle Sim \rangle$ ::= $\langle Id_f \rangle$['/' $\langle Int_n \rangle$] '$\sim$' $\langle Id_g \rangle$['/' $\langle Int_n \rangle$] '=' $\langle r \rangle$ '.'
$\langle Sim \rangle$ | '$\sim$' 'tnorm' '=' $\langle tnorm \rangle$

La opción *Sim* traduce expresiones de la forma $f \sim g = r$, donde f y g son variables proposicionales o constantes, y r es un elemento de L. Admite también la inclusión de aridades en la expresiones, como $f/n \sim g/n = r$ (aquí, f y g son símbolos de función o predicado). En este caso, ambas aridades deben ser iguales. Estas expresiones reciben el nombre de *ecuaciones de similaridad*. También se puede indicar, mediante la directiva $\sim tnorm = \langle label \rangle$, la conjunción que debe usarse en la construcción de la clausura transitiva de la relación. Internamente $\mathcal{FLOPER}$ almacena cada relación como un hecho r en un módulo ad hoc *sim* como $r(f/n, g/n, r)$, donde $n = 0$ si no se especifica de otro modo (esto es, el símbolo se considera como una constante). El fichero `.sim` puede contener sólo un pequeño conjunto de ecuaciones de similaridad que, más tarde, $\mathcal{FLOPER}$ completa llevando a cabo las clausuras reflexiva, simétrica y transitiva. La primera de ellas consiste únicamente en la aserción del hecho $r(A, A, \top)$. La clausura simétrica produce, para cada $r(a, b, r)$, la aserción de su simétrica $r(b, a, r)$ –siempre que no exista ya una $r(b, a, r')$ donde $r \leq r'$ (en este caso $r(a, b, r)$ se reescribirá como $r(a, b, r')$ al considerar $r(b, a, r)$). La clausura

transitiva se computa mediante el siguiente algoritmo[12], donde $\wedge$ representa la conjunción especificada por la directiva "*tnorm*", y "*assert*" y "*retract*" son predicados auto-explicativos definidos en PROLOG:
Es importante tener en cuenta que incluso si el usuario introdujera ecuaciones de similaridad incompletas o aparentemente inconsistentes, $\mathcal{FLOPER}$ obtiene una relación de similaridad coherente y completa tras aplicar las clausuras arriba mencionadas a dichas ecuaciones. Por ejemplo, si el usuario proporciona las siguientes ecuaciones: $a \sim b = 0.8$, $b \sim c = 0.6$ y $a \sim c = 0.3$, tras la aplicación de nuestro algoritmo para la construcción de la similaridad, resulta el siguiente conjunto de ecuaciones: $a \sim b = 0.8$, $b \sim c = 0.6$ and $a \sim c = 0.6$, que preservan la propiedad transitiva[13].

Transitive Closure
forall r(A,B,r_1) **in** `sim`
  forall r(B,C,r_2) **in** `sim`
    $r = r_1 \wedge r_2$
    if r(A,C,r') **in** `sim` **and** $r' < r$
      retract r(A,C,r') **from** `sim`
      retract r(C,A,r') **from** `sim`
    end if
    if r(A,C,r') **not in** `sim`
      assert r(A,C,r) **in** `sim`
      assert r(C,A,r) **in** `sim`
    end if
  end forall
end forall

Ejemplo 6.7.1. *Sea L un retículo* $([0,1], \leq)$. *Para ilustrar la expresividad mejorada de* FASILL*, considérese el programa* $\langle \Pi, \mathcal{R}, L \rangle$ *que modela el concepto de* buen hotel*, esto es, un hotel elegante y muy próximo a una entrada de metro, como se aprecia en la Figura6.5. Aquí, usamos un* agregador de la media aritmética*, definido como* $\dot{@}_{avg}(x,\ y) \triangleq (x + y)/2$*, donde* very *es un modificador lingüístico, así como un agregador (de aridad 1). con función de verdad* $\dot{@}_{very}\ x \triangleq x^2$*. El programa* FASILL

[12]Es importante notar que este algoritmo debe ejecutarse inmediatamente después de las clausuras simétrica y reflexiva.

[13]Por mor de la simplicidad, hemos omitido las ecuaciones obtenidas durante las clausuras reflexiva y simétrica.

que modela este ejemplo es el siguiente:

```
vanguardista(hydropolis) <- 0.9.
eleganta(ritz)           <- 0.9.
buen_hotel(X) <- elegante(X) @aver @very(próximo(X, metro)).
```

La relación de similaridad $\mathcal{R}$ establece que elegante *es similar a* vanguardista*, y que* metro *lo es a* bus *y (por transitividad) a* taxi*:*

```
~tnorm  = godel                          metro ~ bus  = 0.5.
elegante/1 ~ vanguardista/1  = 0.6.      bus   ~ taxi = 0.4.
```

Indicamos también que la t-norma empleada para hacer la clausura transitiva es la conjunción de Gödel (esto es, el ínfimo de dos elementos). Para este programa, (que incluye junto con las reglas presentadas anteriormente, el retículo L y la relación de similaridad, $\mathcal{R}$), el objetivo `buen_hotel(X)` *produce dos respuestas computadas difusas:*

```
< 0.4, {X/ritz} >
< 0.38, {X/hydropolis} >
```

Cada una de ellas corresponde a las hojas del árbol mostrado en la Figura 6.6. Obsérvese que para alcanzar estas soluciones, se ha realizado un paso de fallo *en la derivación de la rama más a la izquierda del árbol, mientras que en la rama más a la derecha (y esta es la principal novedad con respecto a las versiones anteriores de la herramienta $\mathcal{FLOPER}$) se dan dos pasos de* éxito *que explotan la relación de similaridad. El primero relaciona* elegante *con* vanguardista*, y el segundo, por transitividad,* metro *con* taxi *al resolver el átomo* `próximo(hydropolis,metro)`*, lo que ilustra la flexibilidad de nuestro sistema.*

Para terminar esta sección, cabe precisar que nuestra aproximación difiere de la presentada en [CRR14], en el sentido de que allí se emplea una combinación de técnicas de transformación para extraer, en primer lugar, la definición del predicado "$\sim$", simulando la unificación débil en términos de un conjunto de reglas de programa que extienden al programa original. Finalmente, el predicado "$\sim$" en dicho trabajo se reduce a un operador de unificación por proximidad/similaridad (en este caso no se implementa mediante reglas y es más próximo a nuestra implementación del algoritmo de unificación débil), que mejora enormemente la eficiencia de sus sistemas de programación previos.

6.8. Semántica declarativa de FASILL

En programación lógica, la semántica declarativa de un programa se formula, tradicionalmente, sobre la base del modelo mínimo de Herbrand (concebido como el ínfimo del conjunto de interpretaciones). En esta sección, introducimos formalmente las nociones semánticas de interpretación de Herbrand, modelo de Herbrand y modelo mínimo de Herbrand para un programa FASILL $\mathcal{P}$, con objeto de caracterizar la semántica declarativa de esta clase de programas difusos.

Seguimos el mismo proceso de [Llo87] y, además, generalizamos la semántica por teoría de modelos definida en [JMP09c] para programas lógicos multi-adjuntos, y en [MPV14a] para programas X-MALP[14] que recogemos en la Sección 6.4. Esto es, si L es un retículo multi-adjunto y $\mathcal{P}$ es un programa multi-adjunto, o bien L es un retículo completo y $\mathcal{P}$ es un programa X-MALP, nuestro modelo de Herbrand $\mathcal{I}$ coincide con el que corresponde a su marco.

En lo que sigue, consideramos que $\mathcal{B}_\mathcal{P}$ es la base de Herbrand del programa FASILL $\mathcal{P}$, esto es, el conjunto de todos los átomos básicos que se pueden formar con símbolos de Π y la relación de similaridad $\mathcal{R}$ de $\mathcal{P}$.

Definición 6.8.1 (Interpretación de Herbrand)**.** *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L\rangle$ un programa* FASILL*. Una interpretación de Herbrand es una aplicación $\mathcal{I} : \mathcal{B}_\mathcal{P} \to L$, donde $\mathcal{B}_\mathcal{P}$ es la base de Herbrand de $\mathcal{P}$.*

Una interpretación de Herbrand $\mathcal{I}$ se puede extender de forma natural al conjunto de fórmulas básicas del lenguaje haciendo uso de la siguiente definición:

$$\mathcal{I}(\varsigma(F_1, \ldots, F_n)) = \dot{\varsigma}(\mathcal{I}(F_1), \ldots, \mathcal{I}(F_n))$$

donde ς es una conectiva arbitraria y $\dot{\varsigma}$ su función de verdad. Nótese que, por abuso del lenguaje, usamos el mismo símbolo para la interpretación de Herbrand y para su extensión.

Para interpretar una fórmula no básica (cerrada y universalmente cuantificada) A, basta tomar

$$\mathcal{I}(A) = inf\{\mathcal{I}(A\theta) : A\theta \text{ es una instancia básica de } A\}$$

Sea $\mathcal{H}$ el conjunto de interpretaciones de Herbrand cuyo orden está inducido por el orden en L.

$$\mathcal{I}_1 \leq \mathcal{I}_2 \iff \mathcal{I}_1(F) \leq \mathcal{I}_2(F), \forall F \in \mathcal{B}_\mathcal{P}$$

[14] Recuérdese que los programas X-MALP no dependen de *pares adjuntos*.

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

Definición 6.8.2 (Modelo de Herbrand)**.** *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa* FASILL*. Una interpretación de Herbrand $\mathcal{I}$ satisface, o es un* modelo de Herbrand[15] *de una regla $A \leftarrow \mathcal{B} \in \mathcal{P}$ si, y sólo si, verifica:*

i) $\mathcal{I}(A) \geq \mathcal{I}(\mathcal{B})$

ii) si A' es un átomo tal que $\mathcal{R}(A, A') = r$, entonces $\mathcal{I}(A'\theta) \geq r \wedge \mathcal{I}(A\theta)$, para todas las instancias básicas $A\theta, A'\theta$ de A y A'.

Más aún, una interpretación de Herbrand $\mathcal{I}$ es un modelo de Herbrand *de $\mathcal{P}$ si y sólo si $\mathcal{I}$ es un modelo de Herbrand de todas las reglas en $\mathcal{P}$ (esto es, todas las reglas de $\mathcal{P}$ son satisfechas por $\mathcal{I}$).*

Obsérvese que, en este sentido, dados dos átomos A y A' (donde A es la cabeza de una regla de $\mathcal{P}$) tales que $\mathcal{R}(A, A') = r$ es muy próximo al supremo de L, ambos serán interpretados por cualquier modelo $\mathcal{I}$ de $\mathcal{P}$ con valores muy próximos. En particular, si $\mathcal{R}(A, A') = \top$, la interpretación de A y A' debe ser la misma para cualquier modelo, puesto que $\mathcal{I}(A\theta) \geq \top \wedge \mathcal{I}(A'\theta) = \mathcal{I}(A'\theta)$ y $\mathcal{I}(A'\theta) \geq \top \wedge \mathcal{I}(A\theta) = \mathcal{I}(A\theta)$, esto es, $\mathcal{I}(A\theta) = \mathcal{I}(A'\theta)$. Este es el resultado esperado cuando A y A' son sintacticamente iguales, pues, entonces, $\mathcal{R}(A, A') = \top$.

Por otra parte, si $\mathcal{R}(A, A') = \bot$ (esto es, si A y A' no son similares), no se requiere que los modelos de $\mathcal{P}$ den valores parecidos para A y A', ya que $\mathcal{I}(A\theta) \geq \bot \wedge \mathcal{I}(A'\theta) = \bot$ y $\mathcal{I}(A'\theta) \geq \bot \wedge \mathcal{I}(A\theta) = \bot$. La definición de modelo, en este caso, requiere únicamente que $\mathcal{I}(A)$ y $\mathcal{I}(A')$ sean mayores o iguales que $\bot$, cosa que cumple cualquier valor. De nuevo, este es el resultado esperado, puesto que definimos A y A' como no similares en absoluto, de modo que sus respectivos grados de verdad no guardan ninguna relación.

Definición 6.8.3 (Modelo mínimo de Herbrand)**.** *Sea $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ un programa* FASILL*. La interpretación $\mathcal{I}_{\mathcal{P}} = inf\{\mathcal{I}_j : \mathcal{I}_j$ es modelo de Herbrand de $\mathcal{P}\}$ se llama modelo mínimo difuso de Herbrand*[16] *de $\mathcal{P}$.*

El siguiente resultado justifica que la interpretación anterior $\mathcal{I}_{\mathcal{P}}$ pueda ser entendida realmente como el modelo mínimo de Herbrand difuso.

[15] A veces abreviaremos diciendo "modelo difuso" o, simplemente, "modelo".
[16] A veces abreviamos escribiendo "modelo mínimo difuso" o, simplemente, "modelo mínimo".

Teorema 6.8.4. *Sea* $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ *un programa* FASILL. *La interpretación de Herbrand* $\mathcal{I}_\mathcal{P} = inf\{\mathcal{I}_j : \mathcal{I}_j$ *es modelo de Herbrand de* $\mathcal{P}\}$ *es el menor modelo de Herbrand de* $\mathcal{P}$.

Demostración. Sea $\mathcal{M}$ el conjunto de modelos de Herbrand de $\mathcal{P}$, esto es, $\mathcal{M} = \{\mathcal{I}_j : \mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}\}$. $\mathcal{M}$ no es vacío, puesto que al menos contiene la interpretación de Herbrand $\mathcal{I} = sup(\mathcal{H})$, definida en cada $A \in \mathcal{B}_\mathcal{P}$ por $\mathcal{I}(A) = sup(L)$, que es un modelo de Herbrand de $\mathcal{P}$.

Entonces, si denotamos $\mathcal{I}_\mathcal{P} = \text{ínf}(\mathcal{M})$, $\mathcal{I}_\mathcal{P}$ es una interpretación de Herbrand. Esto es así porque, dado que $(\mathcal{H}, \leq)$ es un retículo completo, existe el ínfimo del subconjunto $\mathcal{M} \subset \mathcal{H}$, que es, además, miembro de $\mathcal{H}$. Demostramos a continuación que $\mathcal{I}_\mathcal{P}$ es un modelo de Herbrand de $\mathcal{P}$, esto es, que satisface todas las reglas de $\mathcal{P}$. Considérese una regla $R = H \leftarrow \mathcal{B} \in \Pi$.Dado que $\mathcal{I}_\mathcal{P}$ es el ínfimo de $\mathcal{M}$, $\mathcal{I}_\mathcal{P} \leq \mathcal{I}_j$, para todo modelo $\mathcal{I}_j$ de $\mathcal{P}$. Por tanto, $\mathcal{I}_\mathcal{P}(A) \leq \mathcal{I}_j(A)$ para cada átomo A. Por otra parte, dado que $\mathcal{I}_j$ es un modelo de Herbrand de $\mathcal{P}$, $\mathcal{I}_j$ satisface cada regla R, esto es,

i) $\mathcal{I}_j(H) \geq \mathcal{I}_j(\mathcal{B})$

ii) si H' es una fórmula tal que $\mathcal{R}(H, H') = r$, entonces $\mathcal{I}_j(H'\theta) \geq r \wedge \mathcal{I}_j(H\theta)$, para toda instancia básica $H\theta, H'\theta$ de H y H'.

Haciendo uso, de nuevo, de la definición de ínfimo, tenemos que:

$$\begin{array}{rcll} \mathcal{I}_\mathcal{P}(\mathcal{B}) & = & \text{ínf}\{\mathcal{I}_j(\mathcal{B}) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} & \\ & \leq & \text{ínf}\{\mathcal{I}_j(H) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} & = \mathcal{I}_\mathcal{P}(H) \end{array}$$

Considérese ahora una fórmula H' que verifica *ii*). Mediante la monotonía de $\wedge$,

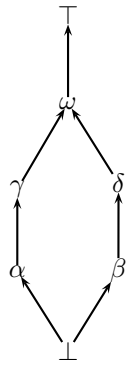
$$\begin{array}{rcll} \mathcal{I}_\mathcal{P}(H'\theta) & = & inf\{\mathcal{I}_j(H'\theta) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} & \\ & \geq & inf\{r \wedge \mathcal{I}_j(H\theta) : \mathcal{I}_j \text{ es modelo de Herbrand de } \mathcal{P}\} & \\ & \geq & r \wedge inf\{\mathcal{I}_j(H\theta) : \mathcal{I}_j \text{ es modelo de Herbrand de} \mathcal{P}\} & = r \wedge \mathcal{I}_\mathcal{P}(H\theta) \end{array}$$

Así, las condiciones $\mathcal{I}_\mathcal{P}(H) \geq \mathcal{I}_\mathcal{P}(\mathcal{B})$ y $\mathcal{I}_\mathcal{P}(H'\theta) \geq r \wedge \mathcal{I}_\mathcal{P}(H\theta)$ se cumplen, y $\mathcal{I}_\mathcal{P}$ satisface la regla R y (puesto que, análogamente, satisface todas las reglas de $\mathcal{P}$) es un modelo de Herbrand de $\mathcal{P}$, como se esperaba. Por último, puesto que $\mathcal{I}_\mathcal{P} = \text{ínf}(\mathcal{M})$, por medio de la definición de ínfimo, $\mathcal{I}_\mathcal{P} \leq \mathcal{I}_j, \forall j$ y, así, $\mathcal{I}_\mathcal{P}$ es el modelo mínimo de Herbrand de $\mathcal{P}$, lo que concluye esta demostración. □

En el siguiente ejemplo ilustramos el modo de calcular el modelo mínimo de Herbrand $\mathcal{I}_\mathcal{P}$ de un programa FASILL y, en general, el modo de calcular modelos $\mathcal{I}$ de $\mathcal{P}$.

Ejemplo 6.8.5. *Sea* $\mathcal{P} = \langle \Pi, \mathcal{R}, L \rangle$ *un programa* FASILL *donde* Π *y el retículo* $(L, \leq)$ *se dan en el siguiente diagrama (el retículo es dado por su diagrama de Hasse) y la relación* $\mathcal{R}$ *es la que establece que* $\mathcal{R}(a, a') = \alpha$ *y* $\mathcal{R}(b, b') = \beta$.

Si asumimos que la función de verdad $\dot{\vee}_{\mathtt{G}}$ *para la conectiva* $\vee_{\mathtt{G}}$ *se define como* $\dot{\vee}_{\mathtt{G}}(x, y) = \sup\{x, y\}$, *el modelo mínimo de Herbrand* $\mathcal{I}_{\mathcal{P}}$ *se determina por* $\mathcal{I}_{\mathcal{P}}(p(a)) = \gamma$, $\mathcal{I}_{\mathcal{P}}(p(b)) = \delta$, $\mathcal{I}_{\mathcal{P}}(p(a')) = \alpha$, $\mathcal{I}_{\mathcal{P}}(p(b')) = \beta$, $\mathcal{I}_{\mathcal{P}}(q(a)) = \gamma$, $\mathcal{I}_{\mathcal{P}}(q(a'))$ $= \alpha$, $\mathcal{I}_{\mathcal{P}}(r(b)) = \delta$, $\mathcal{I}_{\mathcal{P}}(r(b')) = \beta$, *como se muestra en la siguiente tabla (donde hemos omitido los átomos que se interpretan como* $\bot$*).*



$$\mathcal{P} = \begin{cases} R_1: & r(b) & \leftarrow & \delta \\ R_2: & q(a) & \leftarrow & \gamma \\ R_3: & p(x) & \leftarrow & q(x) \vee_{\mathtt{G}} r(x) \end{cases}$$

	$p(a)$	$p(a')$	$p(b)$	$p(b')$	$q(a)$	$q(a')$	$r(b)$	$r(b')$
$\mathcal{I}_{\mathcal{P}}$	γ	α	δ	β	γ	α	δ	β

De hecho, por la Definición 6.8.2,

$$\mathcal{I} \text{ es modelo de Herbrand de } R_1 \text{ sii } \begin{cases} \mathcal{I}(r(b)) \geq \delta \\ \mathcal{I}(r(b')) \geq \mathcal{R}(b, b') \wedge \mathcal{I}(r(b)) \geq \beta \wedge \delta = \beta \end{cases}$$

$$\mathcal{I} \text{ es modelo de Herbrand de } R_2 \text{ sii } \begin{cases} \mathcal{I}(q(a)) \geq \gamma \\ \mathcal{I}(q(a')) \geq \mathcal{R}(a, a') \wedge \mathcal{I}(q(a)) \geq \alpha \wedge \gamma = \alpha \end{cases}$$

$$\mathcal{I} \text{ es modelo de Herbrand de } R_3 \text{ sii } \begin{cases} \mathcal{I}(p(a)) \geq \mathcal{I}(q(a) \vee r(a)) = \gamma \\ \mathcal{I}(p(b)) \geq \mathcal{I}(q(b) \vee r(b)) = \delta \\ \mathcal{I}(p(a')) \geq \mathcal{R}(a, a') \wedge \mathcal{I}(p(a)) = \alpha \\ \mathcal{I}(p(b')) \geq \mathcal{R}(b, b') \wedge \mathcal{I}(p(b)) = \beta \end{cases}$$

Obsérvese que este proceso permite calcular el modelo mínimo de Herbrand $\mathcal{I}_{\mathcal{P}}$ *y sugiere, además, el modo de obtener todos los modelos de Herbrand* $\mathcal{I}$ *de* $\mathcal{P}$.

	$\mathcal{I}_\mathcal{P}$
vanguardista(hydropolis)	0.9
vanguardista(ritz)	0.6
elegante(hydropolis)	0.6
elegante(ritz)	0.8
cercano(hydropolis, taxi)	0.7
cercano(hydropolis, metro)	0.4
cercano(hydropolis, bus)	0.5
buen_hotel(hydropolis)	0.38
buen_hotel(ritz)	0.4

Siguiendo la metodología explicada hasta este punto, se puede comprobar fácilmente que el modelo mínimo de Herbrand del programa mostrado en las Secciones 6.6 y 6.8 es el que se da en la tabla adjunta (donde se asume que las interpretaciones de los átomos omitidos es 0).

6.9. Conclusiones y Trabajo Futuro

El alto poder expresivo del lenguaje MALP (e, incluso, su propio nombre), recae en la posibilidad de usar múltiples pares adjuntos en el diseño de los programas. Aunque, como hemos mostrado en el Capítulo 3, la propiedad adjunta juega un papel muy importante al definir las propiedades de MALP, restringe (aunque sea desde un punto de vista puramente teórico) la clase de retículos que se pueden asociar a los programas y, por tanto, la variedad de nociones de verdad aceptadas en este marco.

En este capítulo hemos recogido nuestro anterior trabajo en [MPV13]. Allí mostramos una técnica de transformación sintáctica que, preservando la semántica, produce programas MALP de sintaxis más sencilla y que no dependen de las restricciones de los pares adjuntos, abriendo así la puerta a futuros desarrollos orientados a incrementar el rango de programas lógicos difusos más allá de MALP.

A partir de ahí, hemos construdo sobre MALP el marco de la "*Programación lógica multi-adjunta relajada*" [MPV14a], X-MALP para abreviar, donde se elimina totalmente la dependencia de pares adjuntos. Hemos descrito esta clase de programas difusos basada en retículos completos (no necesariamente multi-adjuntos) a través de su sintaxis y semántica operacional y declarativa. En este sentido, hemos provisto caracterizaciones basadas en teoría de modelos y en punto fijo (y probado su

equivalencia) de los modelos difusos de Herbrand para programas X-MALP.

A continuación hemos introducido el lenguaje FASILL, que combina los adelantos sintácticos y semánticos de X-MALP (esto es, la independencia de los retículos multi-adjuntos para modelar la noción de verdad) con la capacidad expresiva de los lenguajes basados en similaridades. Puede verse FASILL, por tanto, como un lenguaje integrador de los dos modelos principales de difuminación de la programación lógica clásica. FASILL (acrónimo de "Fuzzy Aggregators and Similarity Into a Logic Language") es un lenguaje de programación lógica difusa con anotación implícita/explícita de grados de verdad, una gran variedad de conectivas y unificación por similaridad. En los recientes artículos [JMPV14, IMPV15] se ha revelado su sintáxis, semántica operacional y detalles de implementación[17] de este lenguaje que, en esencia, integra y extiende las características tanto de MALP (*Multi-Adjoint Logic Programming*, un lenguaje de programación lógica difusa que anota reglas explícitamente) y Bousi~Prolog (que emplea un algoritmo de unificación débil). Por tanto, trabaja con similaridades y grados de verdad en un marco que combina de forma natural los beneficios de ambos lenguajes. Tras introducir las características sintácticas y operacionales de FASILL, hemos descrito su implementación en el renovado sistema $\mathcal{FLOPER}$, donde nos hemos centrado en la descripción del *módulo de similaridad*, y hemos detallado la representación interna de una relación de similaridad y su construcción automática mediante algoritmos de clausura. Hemos descrito finalmente la herramienta *FLOPER online* que permite la ejecución de programas FASILL a través de la web. A continuación, en la Sección 6.8 introducimos la semántica declarativa por modelo mínimo de FASILL, lo cual contemplamos como un paso necesario en el desarrollo de este marco.

Para el futuro queremos, como ya hiciéramos en [MPV14c] (donde se avanzó en el diseño de semánticas declarativas y/o propiedades de corrección de acuerdo al desarrollo de lenguajes lógicos basados en relaciones de similaridad/proximidad, Bousi~Prolog, o retículos altamente expresivos para modelar grados de verdad, MALP), establecer propiedades análogas –pero reforzadas– para el lenguaje FASILL. Otra tarea pendiente para el futuro inmediato consiste en establecer las conexiones entre nuestra nueva versión difusa del modelo mínimo de Herbrand y la semántica operacional de programas FASILL, con objeto de demostrar la corrección del marco.

[17] La última versión del sistema $\mathcal{FLOPER}$, ya actualizada para trabajar con este lenguaje, se puede descargar gratuitamente de `http://dectau.uclm.es/floper/?q=sim` o bien se puede probar online a través de `http://dectau.uclm.es/floper/?q=sim/test`.

Capítulo 7

Conclusions and future work

In this chapter we summarize the work developed in this thesis. In the first place, Section 7.1 enumerates the main contributions of this work and, in the second place, in Section 7.2 we discuss some of the future work suggested by the performed research.

Our main goals, that we have enumerated in Section 1.1, have been successfully addressed throughout this work. As a brief summary, by our research on multi-adjoint lattices we have found new classes of programs able to be modelled by MALP; we have used this language to provide a fuzzy solution to real-life problems; we have researched the theoretical basis of fuzzy logic programming and its connections with category theory; we have expanded our starting point language to FASILL, that inherits the similarity-based equality from the Bousi~Prolog language; and, finally, we have refined and completed a programming environment, $\mathcal{FLOPER}$, able to work with these developments.

7.1. Conclusions

We proceed in this section to detail the contributions achieved in this thesis. We divide this section according the nature of the contributions, either theoretical or practical.

Works related to MALP

The first contributions of this thesis focused on the MALP language. MALP is a

very general fuzzy extension of the PROLOG language. Here, every atom is interpreted not in the $\{true, false\}$ classical bi-valued lattice, but in any complete lattice that fulfils the property of having at least one adjoint pair (that is, a conjunction and an implication that are related in a specific way). Furthermore, the set of connectives in this language is not restricted to just the classical conjunction (represented in PROLOG by a comma), and any monotonous connective is valid, as far as it is defined over the elements of the previously mentioned multi-adjoint lattice. Each rule in a MALP program is accompanied with a truth degree that states the confidence of the programmer in the truth of the entire rule. From the syntactic point of view, this truth degree is added after the label `with`, that, together with the use of different connectives instead the comma, constitutes the only difference in appearance with PROLOG.

The operational semantic of MALP is based in a state transition system and computation does not end when some atom is interpreted as 0 (or the bottom element) as in PROLOG, but it can continue and the interpretation of the entire formula be different from 0, since the "failed" atom may be connected to the rest of the formula by a disjunction or another aggregator. Consider goal $?- p \vee q$ and program q *with* 1. Although PROLOG would stop as soon as p fails, MALP interprets p as 0 and continues evaluating q as 1. Finally, our language interprets the formula $0 \vee 1$, that is necessarily 1.

From the semantic point of view, note that the declarative semantics of MALP is non-refutational and has been established through model theory as well as fix-point operator. With respect to this declarative semantics, in [MPV12a] we provided the notion of *logical consequence* and related it with that of *fuzzy correct answer* in the MALP framework. Furthermore, in [MPV14c] we proposed a description of MALP based on fuzzy sets (particularly, interpretations were seen as fuzzy subsets of the Herbrand base of a program), which can be understood as a reinforcement of the fuzzy quality of the language.

Recall that for this language the multi-adjointness of lattices is an essential property, since it can only use multi-adjoint lattices to represent truth degrees. Therefore, we provided a proof of the multi-adjointness of rich lattices, like the String lattice [MMPV12b, MMPV12c] (the set of all words for certain alphabet with operation "append" as a conjunction) and the cartesian product of lattices [MMPV12a, MMPV11b] (lattices whose elements are tuples containing elements of other lattices), as we detail in Section 3.6. This is an important contribution for

the MALP framework since it enriches the class of programs to be modelled in it, as well as allows to report different information about the execution of a goal. As a result of this research, it is possible to model lattices and truth degrees in such a way that declarative traces can be introduced into the fuzzy computed answers [MMPV11a, MMPV11c]. These traces include the number of computational steps to reach the f.c.a, or a list (in the form of a string) of the program rules used in each computational step. This information is useful for debugging tasks (for instance, unreachable code is easier to find) and also for cost measurements.

With respect to equality models, the richer notion of equality called *similarity-based strict equality* has been researched [MPV12b]. Strict equality differs from purely syntactic equality (used in PROLOG and MALP) in the fact that terms have to be evaluated in order to be compared. This is why it is used mainly in the functional paradigm with lazy evaluation. The introduction of similarity relations into the strict equality model just implies that different constants can be said to be similar (i.e., equal at a certain truth degree), while using a classical ("crisp") equality model they are absolutely different. It is possible to model the notion of *similarity-based strict equality* using fuzzy rules and, so, fuzzy programs can benefit from this model of equality as well as from the syntactic equality, as we do in [MPV12b]. Furthermore, in [MPV14b, JMV15] we provide a method to automatically generate the fuzzy rules necessary to model the similarity-based strict equality for a certain similarity relation (expressed as a set of similarity equations). This work can be freely tested through the url `http://dectau.uclm.es/sse/`. This point is closely related to an ulterior work where similarity equations expressing a similarity relation are also used, that we discuss further.

Applications of fuzzy logic programming

One of the goals of this thesis is to apply fuzzy logic programming to real-life problems both in the theoretical and the practical areas.

In the first place, it is noteworthy that MALP, through $\mathcal{FLOPER}$, had already been successfully used to develop FUZZYXPATH [ATM12, ATM14], a fuzzy version of the XPATH language for the flexible retrieval of information of XML documents. FUZZYXPATH is based on an extended syntax of that of XPATH, with the additions of fuzzy connectives (such as the conjunctions of Gödel, Product and Łukasiewicz, in the name of *and+*, *and* and *and-*, respectively) and directives `DEEP` and `DOWN` to graduate the suitability of answers according to its positions in the XML file. Then, as a first practical application, we used the execution trees produced by $\mathcal{FLOPER}$

in an XML file to test the tool FuzzyXPath. These trees are adequate for this task because they include deeply nested tags and tags with a large amount of children, that makes them a good benchmark to test the tool. As a result, FuzzyXPath has become a promising tool for debugging MALP (and FASILL) derivation trees, as in [ALMV13, ALMV14], referred in Section 4.1. In particular, FuzzyXPath can be used to obtain from a derivation tree of a MALP (of FASILL) program and a goal these information:

- Unreachable code, in the form of unused program rules.
- Fuzzy computed answers.
- Potentially infinite branches.

In a totally different research field, we have also found applications in the SMT (Satisfiability Modulo Theory) area. SMT is the fuzzy version of the well known SAT (propositional satisfiability problem), that consists on determining whether a propositional formula holds or not. SMT focuses on fuzzy propositional formulae. In Section 5.2, that refers the works in [BMVV13, BMVV15], we successfully used our $\mathcal{FLOPER}$ environment as an SMT tool. To do so, we provided a method to build propositional fuzzy formulae in terms of a MALP program. Concretely, such a program includes a fact for every element of the lattice modelling truth degrees, and the propositional fuzzy formula to be proven is represented in the goal of the program. For instance, formula

$$((\neg In)\&Out)|(In\&(\neg Out))$$

is translated as `(@not(i(In)) & i(Out)) | (i(In) & @not(i(Out)))`. The benefits of using MALP as an SMT solver includes the possibility of using the wide range of connectives and definitions for truth degrees that this language allows. In particular, we tested our method with a fuzzy formula interpreted in a partially ordered lattice of five elements with connectives defined ad-hoc. As a drawback, $\mathcal{FLOPER}$ is less efficient in this task than those tools designed specifically for this purpose. And although our system excels in SMT problems with finite lattices, it is flawed when there are infinite truth degrees as in the $[0, 1]$ case.

We have merged the two previous works in [ABL$^+$15], where FuzzyXPath serves as an automatic browser for solutions in the execution tree corresponding to an SMT formula.

To end the application subsection, we found that fuzzy logic programming can have a major role in fields as far as cloud computing. In a cloud infrastructure, the admission control accepts works until the amount of resources requested reaches the total number of resources of the infrastructure. If the system does not accept any other work, it is bound to have unused resources since it is well known that a work does not use all its requested resources constantly. To enhance resource utilization, it is common to use overbooking of resources, that is, to accept more VMs than the limit. The evident risk of this policy is evident: the higher the number of works accepted over the limit, the more probable some of them cannot access their requested resources because they are in use by some other work, and more SLA (Service Level Agreement) are violated. In [VTMT13] we use a fuzzy module (a MALP program) for evaluating the risk of accepting incoming works in a cloud data center. The fuzzy module uses a prediction of the behaviour of the work (with respect to resource utilization) and the current state of resource utilization in the system. It takes into account the different risks associated to all three resources (CPU, memory and network), since a clash on CPU is considered less harmful than a clash on memory (in terms of performance degradation).

Our module, described in Section 5.3, provides three measures of risk, optimistic, realistic and pessimistic, with the optimistic estimation returning lesser risk values than the realistic one, and that, in turn, lesser values than the pessimistic one. The numerous tests of this approach reveal that the cloud infrastructure, equipped with our fuzzy risk estimator, highly increases resource utilization (even as high as a 2.38 times) with very little resource exhaustion (only 0.84 % of the time).

We have developed further developments to the fuzzy module for overbooking risk in cloud infrastructure above mentioned. This new development focuses on a problem that arises once the work has been accepted into the network, that is known as the *allocation problem*. The question of the allocation consists on determining which core or group of cores, into the cloud infrastructure, are better suited for holding the recently accepted work. To answer this question the system has to consider issues like the actual topology of the network (keeping track of the costs in terms of time of communication between each pair of cores) and special requisites of certain works (that require that certain cores are dedicated full-time for certain works). Also, the fuzzy system has to allocate a work in a core (or group) that can attend its demand of resources. We have addressed this problem in Section 5.4, that reports the work

in [VMTT15].

Ulterior theoretical researches

In my stay in Umea I had the opportunity to work with Prof. Patrik Eklund in an interesting approach to fuzzy logic programming from the point of view of Category Theory. We have provided a preliminary description of fuzzy logic programming via category theory. In particular, we have addressed the definition of fuzzy data and fuzzy sort, and made an online tool to test this experimental definitions that can be accessed in the url `http://dectau.uclm.es/FLUS/`. In [EGH$^+$13a, EGH$^+$13b], we translate the notions of fuzzy logic programming to a categorical notation and provide important results such as the declarative semantics based on the fixpoint operator. The earlier experience of [GMV10] was very useful for this development.

On the other hand, in Chapter 6, we addressed the goal of integrating fuzzy paradigms in the second part of this thesis. To do so, and with the intention of providing a more general language than MALP, we proceeded by finding a subset of MALP for which the adjoint property (of the lattice) could be omitted [MPV13]. In this subset the weights of the rules are always the top element of the lattice (e.g., 1 for the $[0, 1]$ interval). Thanks to this, the adjoint property becomes unnecessary during the operational phase, since there is no weight of the rule to be connected with the body in the substitution of the selected atom, as we wanted. Then, we provided an algorithm to transform each MALP program to a program in the aforementioned subset. This is performed by including the weight of the rule into its body, using the conjunction adjoint to the implication of the rule to connect the weight with the formula of the body. This means that the above mentioned subset of programs includes all the semantics of the language in itself. Furthermore, we designed a new language, X-MALP, with the same form of that subset of MALP, such that every MALP program has an equivalent X-MALP program (a program with the same f.c.a for the same goal). The advantage of the new language is that its notion of truth degree is not constraint to a multi-adjoint lattice, but to the wider concept of complete lattice. Hence, the new language is an enhancement of MALP. Note that the operational and declarative semantics of X-MALP are inherited from MALP through the above mentioned subset of programs.

As a final step, we introduced the notion of similarity. The Bousi~Prolog language (which is, in turn, another product of our research group) was used as a model of these class of languages. In particular, we used its notion of similarity relation, closure and, most importantly, weak most general unifier, in our research.

In this respect, we modified the notion of similarity equation, thus diverging from BOUSI~PROLOG. For this language, the similarity equation $p \sim q = 1$ relates all symbols p and q independently of their arity. This makes impossible to discriminate between predicates with the same name but different arity. In our approach, we take this into account by annotating implicitly the arity of the related symbols, so the previous equation can be divided in $p/1 \sim q/1 = 1$, $p/2 \sim q/2 = 0.5$. When arities are not indicated, they are supposed to be 0 (that is, the symbols are constants).

We call FASILL [MPV14a] the language resulting from merging similarity in the way of BOUSI~PROLOG (with the indicated modification) and X-MALP. It generalizes both languages based on weighted rules –with an extremely wide range of connectives and truth degrees–, and languages based on similarity relations.

The fuzzy logic programming environment $\mathcal{FLOPER}$
All the previously mentioned developments have been tested through the $\mathcal{FLOPER}$ programming environment. This tool was created as a project of end of studies in 2005 in our research group. At that time, the tool were able to translate MALP code into PROLOG code –only with the $[0, 1]$ lattice–, thus producing as an output a PROLOG program that could be executed by every PROLOG interpreter. In subsequent projects and works, the tool was equipped with newer options and capabilities. By 2010, the tool included a graphical interface and it could not only translate code, but also evaluate fuzzy goals and display execution trees. In that year, we generalised the lattice to any multi-adjoint lattice [MMPV10a, MMPV10c] and implemented the interpretative phase [MMPV10b]. Under the work in this thesis, $\mathcal{FLOPER}$ has undergo the following modifications:

- The tool is now able to manage the new X-MALP and FASILL languages the same way it does with MALP [IMPV15, JMPV14].
- Also, an on-line instance of the environment has been provided under the name "FLOPER online" through the link `http://dectau.uclm.es/floper/?q=sim/test`.
- The visual interface includes, in the project tree, a "sim" folder to hold the similarity files of the program.

With respect to the tool, see [MV14] for a detailed description of its current state of development.

7.2. Future work

In this section we detail the current research being developed in our group and also its possible derivation in the near future.

Theoretical research lines

With respect to the newly designed language FASILL, although it is already well established, we have still mayor points to address. We focus on proving soundness (that is, the property for which every fuzzy computed answer is also a fuzzy correct answer) and completeness. This last property can be seen as symmetric to the soundness, and consists on the fact that for all fuzzy correct answer for a program and a goal, there is a fuzzy computed answer –reached through the operational semantics– more general than that. In the case that completion was non provable or, even worst, false for this framework, we would prove the quasicompletion property of FASILL, that consists on the existence of a succession of fuzzy computed answers whose limit, although non reachable in a finite amount of time– would be a fuzzy correct answer.

Similarly to MALP, it is interesting to apply the richer notions of FASILL to solve practical problems. In particular, similarity can play a significant role in the above mentioned cloud module to evaluate risks, for instance, taking into account the similarities between different works and using this information to short-cut acceptance decision.

Furthermore, as previously developed for MALP in our group, the application of transformation techniques for FASILL, such as folding/unfolding, would enhance its suitability for solving the complex problems of real life. Other techniques, such as tabulation, has also to be taken into account.

In order to develop folding/unfolding techniques to FASILL–and, possibly, X-MALP–, it is mandatory to provide it with the notion of (fuzzy) reductant. Reductants are a theoretical device necessary to formulate the completion of our fuzzy programs when the lattice associated to them may not be totally ordered. In this case, the operational semantics is not able to compute the supreme of two answers (although it is indeed a correct answer following the definitions given by the declarative semantics). A reductant for a certain atom in a program is a rule that combines all the bodies of the program such that they appear in rules whose heads unify with the previously mentioned atom. This way, the reductant for an atom accumulates all the contributions to the truth degree of that atom (in particular, the supremum of these contributions). So, in order to prove the completion of a fuzzy program, we

consider the set of reductants of that program, and not the program itself.

Furthermore, the introduction of the negation and, more broadly, non-monotone aggregators, in the language would bring an immediate benefit to its modelling power, since it would be able to represent negative information as well as positive one.

A very promising research line is to equip FASILL with notions from the field of *fuzzy control*, that is, to provide a more natural way to introduce fuzzy sets, linguistic modifiers and linguistic variables into the language. This would allow the production of fuzzy programs in an easier and more efficient way.

To end the purely theoretical future work, there is a possible research line in the field of Artificial Intelligence in the form of induction of FASILL programs, that would produce a FASILL program from a set of positive and negative examples.

Practical developments

We are nowadays working on an extension of the previously detailed cloud developments. In particular, we want the fuzzy allocation module to take into account some level of feedback with respect to the result of its decision, whether has it been good or bad. This feedback value can then be used to fine-tune future decisions of the module and, then, enhance in a natural and transparent way the behaviour of the system.

In addition to this, the other two applications of fuzzy logic programming devised in Sections 5.1 and 5.2 can be merged together in a new application where the execution trees from an SMT program can be inspected through FuzzyXPath. By doing so, it will be possible to detect automatically the interpretations of a fuzzy formula that satisfies it at certain truth degrees.

Also, as we foresee in the Section 5.1, we plan to integrate an option based on FuzzyXPath into $\mathcal{FLOPER}$ to ease debugging tasks.

Bibliografía

[ABL+15] J. M. Almendros-Jiménez, M. Bofill, A. Luna, G. Moreno, C. Vázquez, M. Villaret. Fuzzy XPath for the Automatic Search of Fuzzy Formulae Models. In *Proc. of the 9th International Conference on Scalable Uncertainty Management, SUM'15. Quebec, Canada, September 16-18.* P. 14 (in press). Springer LNCS, 2015.

[ABMV12] C. Ansótegui, M. Bofill, F. Manyà, M. Villaret. Building Automated Theorem Provers for Infinitely-Valued Logics with Satisfiability Modulo Theory Solvers. In *Proceedings of the 42nd IEEE International Symposium on Multiple-Valued Logic, ISMVL 2012.* Pp. 25–30. 2012.

[Ack67] R. Ackerman. *Introduction to Many Valued Logics.* Dover, New York, 1967.

[ACT95] C. Alsina, J. L. Castro, E. Trillas. On the characterization of S and R implications. In *Proc. of the 6th International Fuzzy Systems Association World Congress, Sao Paulo.* Volume 1, pp. 317–319. 1995.

[Acz48] J. Aczél. On mean values. *Bulletin of the American Mathematical Society* 54(2):392–400, 1948.

[Ada76] J. B. Adams. Probabilistic reasoning and certainty factors. *Mathematical Biosciences* 32:177–186, 1976.

[AF99a] F. Arcelli, F. Formato. Likelog: A Logic Programming Language for Flexible Data Retrieval. In *Proc. of the ACM Symposium on Applied Computing, SAC'99, San Antonio.* Pp. 260–267. ACM, Artificial Intelligence and Computational Logic, 1999.

[AF99b] F. Arcelli, F. Formato. Likelog: A Logic Programming Language for Flexible Data Retrieval. In *Proc. of the 1999 ACM Symposium on Applied Computing, SAC'99, San Antonio, Texas*. Pp. 260–267. 1999.

[AF02] F. Arcelli, F. Formato. A similarity-based resolution rule. *International Journal of Intelligent Systems* 17(9):853–872, 2002.

[AFG96] F. Arcelli, F. Formato, G. Gerla. Similitude-based unification as a foundation of fuzzy logic programming. In *Proc. of Int. Workshop of Logic Programming and Soft Computing, Bonn*. 1996.

[AG93] K. Atanassov, C. Georgiev. Intuitionistic fuzzy Prolog. *Fuzzy Sets Systems* 53(2):121–128, 1993.

[AG98] T. Alsinet, L. Godo. Fuzzy Unification Degree. In *Proc. 2th International Workshop on Logic Programming and Soft Computing'98, in conjunction with JICSLP'98, Manchester*. P. 18. 1998.

[ALM11a] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In Bassiliades et al. (eds.), *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*. Pp. 186–193. Springer Verlag, Lectures Notes in Computer Science 6826, 2011.

[ALM11b] J. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. In Arenas et al. (eds.), *Actas de las XI Jornadas sobre Programación y Lenguajes, PROLE'11, A Coruña, Septiembre 5-7*. Pp. 154–168. Universidade da Coruña ISBN 978-84-9749-487-8, 2011.

[ALMV13] J. M. Almendros-Jiménez, A. Luna, G. Moreno, C. Vázquez. Analyzing Fuzzy Logic Computations with Fuzzy XPath. In Fredlund (ed.), *Proc. of XIII Spanish Conference on Programming and Languages, PROLE'2013, Madrid, Spain, September 18-20*. Pp. 136–150. ECEASST, 2013.

[ALMV14] J. Almendros-Jiménez, A. Luna, G. Moreno, C. Vázquez. Analyzing Fuzzy Logic Computations with Fuzzy XPath. *Electronic Communi-*

cations of the EASST (European Association of Software Science and Technology), pp. 1–19 (in press), 2014.

[Ama30] Amazon Elastic Compute Cloud (Amazon EC2). Web page at `http://aws.amazon.com/ec2/`, Visited 2013-05-30.

[AMM07a] J. Abietar, P. Morcillo, G. Moreno. Building a Fuzzy Logic Programming Tool. In Pimentel (ed.), *Actas de las VII Jornadas sobre Programación y Lenguajes, PROLE'07, Zaragoza, Septiembre 12-14*. Pp. 215–222. Thomson-Paraninfo, 2007. ISBN 978-84-9732-599-8.

[AMM07b] J. Abietar, P. Morcillo, G. Moreno. Designing a Software Tool for Fuzzy Logic Programming. In Simos and Maroulis (eds.), *Proc. of the ICCMSE'07 International Conference of Computational Methods in Sciences and Engineering, Corfu, Greece, September 25-30*. Volume 2, pp. 1117–1120. American Institute of Physics, 2007. Distributed by Springer Verlag. ISBN 978-0-7354-0478-6.

[Apt90] K. R. Apt. Introduction to Logic Programming. In Leeuwen (ed.), *Handbook of Theoretical Computer Science*. Volume B: Formal Models and Semantics, pp. 493–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[Apt97] K. R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science, Prentice Hall, 1997.

[Arc02] F. Arcelli. Likelog for flexible query answering. *Soft Computing* 7(2):107–114, 2002.

[ATM12] J. M. Almendros-Jiménez, A. L. Tedesqui, G. Moreno. Debugging Fuzzy XPath Queries. In Gallardo et al. (eds.), *Actas de las XII Jornadas sobre Programación y Lenguajes, PROLE'12 (Jornadas SISTEDEs Almería 17-19 Sept. 2012)*. Pp. 119–133. Universidad de Almería ISBN:978-84-15487-27-2, 2012.

[ATM14] J. M. Almendros-Jiménez, A. L. Tedesqui, G. Moreno. Dynamic Filtering of Ranked Answers When Evaluating Fuzzy XPath Queries. In Cornelis et al. (eds.), *Rough Sets and Current Trends in Computing*. Lecture Notes in Computer Science 8536, pp. 319–330. Springer International Publishing, 2014.

[BB13] A. Beloglazov, R. Buyya. Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers Under Quality of Service Constraints. *IEEE Transactions on Parallel and Distributed Systems* 24(7):1366–1379, 2013.

[BDa03] P. Barham, B. Dragovic, et al. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.* 37(5):164–177, 2003.

[BDE+12] D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson, I. Shapira. SLA-aware resource over-commit in an IaaS cloud. In *Proc. of 8th Intl. Conference on Network and Service Management (CNSM)*. Pp. 73–81. 2012.

[BMP95] J. F. Baldwin, T. P. Martin, B. W. Pilsworth. *Fril-Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., 1995.

[BMVV13] M. Bofill, G. Moreno, C. Vázquez, M. Villaret. Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT. In Fredlund (ed.), *Proc. of XIII Spanish Conference on Programming and Languages, PROLE'2013, Madrid, Spain, September 18-20*. Pp. 151–165. ECEASST, 2013.

[BMVV15] M. Bofill, G. Moreno, C. Vázquez, M. Villaret. Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT. In Fredlund (ed.), *Electronic Communications of the EASST (European Association of Software Science and Technology)*. Pp. 1–19. ISSN 01863-2122, 2015.

[Cao00] T. H. Cao. Annotated fuzzy logic programs. *Fuzzy Sets and Systems* 113(2):277–298, 2000.

[CBM99] T. Calvo, B. D. Baets, R. Mesiar. Weighted sums of aggregation operators. *Mathware & soft computing* 6(1):33–47, 1999.

[CF95] C. Carlsson, R. Fullér. On fuzzy screening system. In Mainz (ed.), *Proc. of the 3th European Congress on Intelligent Techniques and Soft Computing, EUFIT'95, Aachen*. Pp. 1261–1264. 1995.

[CFF97] C. Carlsson, R. Fullér, S. Fullér. Possibility and necessity in weighted aggregation. In Yager and Kacprzyk (eds.), *The ordered weighted*

averaging operators: Theory, Methodology and Applications. Pp. 18–28. Kluwer Academic Publishers, 1997.

[Cha58] C. C. Chang. Algebraic analysis of many valued logics. *Transactions of the American Mathematical Society* 88:467–490, 1958.

[CHHM01] O. Cordón, F. Herrera, F. Hoffmann, L. Magdalena. Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases. In *Advances in Fuzzy Systems Applications and Theory*. Volume 19. World Scientific, 2001.

[CKKM02] T. Calvo, A. Kolesárová, M. Komorníková, R. Mesiar. Aggregation operators: properties, classes and construction methods. In Calvo et al. (eds.), *Aggregation operators: new trends and applications*. Pp. 3–104. Physica-Verlag GmbH, Heidelberg, 2002.

[Coh85] P. R. Cohen. *Heuristic reasoning about uncertainty: an artificial intelligence approach*. Pitman Publishing, Inc., Marshfield, Massachusetts, 1985.

[CRR08] R. Caballero, M. Rodríguez-Artalejo, C. A. Romero-Díaz. Similarity-based reasoning in qualified logic programming. In *PPDP'08: Proc. of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*. Pp. 185–194. ACM, New York, 2008.

[CRR14] R. Caballero, M. Rodríguez-Artalejo, C. A. Romero-Díaz. A Transformation-based implementation for CLP with qualification and proximity. *Theory and Practice of Logic Programming* 14(1):1–63, 2014.

[DHN90] R. O. Duda, P. E. Hart, N. J. Nilsson. Subjective Bayesian methods for rule-based inference systems. In *Readings in uncertain reasoning*. Pp. 274–281. Morgan Kaufmann Publishers Inc., San Francisco, 1990.

[DHR96] D. Driankov, H. Hellendoorn, M. Reinfrank. *An introduction to control fuzzy*. Springer-Verlag, 1996.

[Dil39] R. Dilworth. Residuated Lattices. *Transactio6ns of the American Mathematical Society* 45:335–354, 1939.

[DK14] C. Delimitrou, C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. 2014.

[DLP91] D. Dubois, J. Lang, H. Prade. Towards possibilistic logic programming. In *Proc. of the 8th International Confenrence on Logic Programming, ICLP'91*. Pp. 581–595. The MIT Press, 1991.

[DM00] C. V. Damásio, L. Moniz-Pereira. Hybrid Probabilistic Logic Programs as Residuated Logic Programs. In *JELIA '00: Proc. of the European Workshop on Logics in Artificial Intelligence*. Pp. 57–72. Springer-Verlag, London, 2000.

[DM01a] C. V. Damásio, L. Moniz-Pereira. Antitonic Logic Programs. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'01, Vienna*. Pp. 379–392. Springer-Verlag, 2001.

[DM01b] C. V. Damásio, L. Moniz-Pereira. Monotonic and residuated logic programs. In Benferhat and Besnard (eds.), *Proc. of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU'01, Toulouse*. Pp. 748–759. Lecture Notes in Artificial Intelligence 2143, 2001.

[DM02] C. V. Damásio, L. Moniz-Pereira. Hybrid Probabilistic Logic Programs as Residuated Logic Programs. *Lecture Notes in Computer Science* 1919:57–72, 2002.

[DM04] C. V. Damásio, L. Moniz-Pereira. Sorted Monotonic Logic Programs and their Embeddings. In *Proc. of the 10th International Conference on Information Processing and Managment of Uncertainty in Knowledge-Based Systems, IPMU'04, Perugia*. Pp. 807–814. 2004.

[DMO04a] C. V. Damásio, J. Medina, M. Ojeda-Aciego. Sorted multi-adjoint logic programs: termination results and applications. In *Proc. of Logics in Artificial Intelligence, JELIA'04, Lisbon*. Pp. 260–273. Lecture Notes in Artificial Intelligence 3229, 2004.

[DMO04b] C. V. Damásio, J. Medina, M. Ojeda-Aciego. A tabulation proof procedure for residuated logic programming. *In Proc. of the European*

Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications 110:808–812, 2004.

[DMO04c] C. V. Damásio, J. Medina, M. Ojeda-Aciego. Termination results for sorted multi-adjoint logic programming. In *Proc. of the 10th International Conference on Information Processing and Managment of Uncertainty in Knowledge-Based Systems, IPMU'04, Perugia.* Pp. 1879–1886. 2004.

[DMO07] C. V. Damásio, J. Medina, M. Ojeda-Aciego. Termination of logic programs with imperfect information: applications and query procedure. *Journal of Appplied Logic* 5:435–458, 2007.

[DP80] D. Dubois, H. Prade. *Fuzzy Sets and Systems: Theory and Applications.* Academic Press, 1980.

[DP84] D. Dubois, H. Prade. Criteria aggregation and ranking of alternatives in the framework of fuzzy set theory. *TIMS Studies in the Management Sciences* 20:209–240, 1984.

[DP85] D. Dubois, H. Prade. A review of fuzzy sets aggregation connectives. *Information Sciences* 36:85–121, 1985.

[DP86] D. Dubois, H. Prade. Weighted minimun and maximum operations in fuzzy sets theory. *Information Sciences* 39:205–210, 1986.

[DS00] A. Dekhtyar, V. S. Subrahmanian. Hybrid Probabilistic Programs. *Journal of Logic Programming* 43(3):187–250, 2000.

[DSMK07] F. Durante, C. Sempi, R. Mesiar, E. P. Klement. Conjunctors and their residual implicators: characterizations and construction methods. *Mediterranean Journal of Mathematics* 4(3):343–356, 2007.

[Ebr01] R. Ebrahim. Fuzzy logic programming. *Fuzzy Sets and Systems* 117(2):215–230, 2001.

[EGH+13a] P. Eklund, M. A. G. García, R. Helgesson, J. Kortelainen, G. Moreno, C. Vázquez. Towards Categorical Fuzzy Logic Programming. In al. (ed.), *Workshop on Fuzzy Logic and Applications (WILF' 13). November 19-22, Genoa, Italy.* Pp. 109–121. LNCS 8256, 2013.

[EGH+13b] P. Eklund, M. A. G. García, R. Helgesson, J. Kortelainen, G. Moreno, C. Vázquez. Towards Categorical Fuzzy Logic Programming. In *Proc. of XIII Jornadas sobre Programación y Lenguajes, PROLE'13, Madrid, Spain, September 17-20.* Pp. 126–135. Technical University of Madrid 8256, 2013. ISBN 978-84-695-8331-9.

[EGHK11] P. Eklund, M. Galán, R. Helgesson, J. Kortelainen. Paradigms for many-sorted non-classical substitutions. In *2011 41st IEEE International Symposium on Multiple-Valued Logic (ISMVL 2011).* Pp. 318–321. 2011.

[EGHK14] P. Eklund, M. Galán, R. Helgesson, J. Kortelainen. Fuzzy terms. *Fuzzy Sets and Systems* 256:211–235, 2014.

[EGOV00] P. Eklund, M. A. Galán, M. Ojeda-Aciego, A. Valverde. Set functors and generalised terms. *Proc. IPMU 2000, 8th Information Processing and Management of Uncertainty in Knowledge-Based Systems Conference* III:1595–1599, 2000.

[EK66] S. Eilenberg, G. M. Kelly. Closed categories. In al. (ed.), *Proceedings of the Conference on Categorical Algebra, La Jolla 1965.* Pp. 421–562. Springer-Verlag, 1966.

[EK76] M. H. van Emden, R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM* 23(4):733–742, 1976.

[EM45] S. Eilenberg, S. MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society* 58(2):231–294, 1945.

[vE86] M. H. van Emden. Quantitative Deduction and its Fixpoint Theory. *Journal of Logic Programming* 3(1):37–53, 1986.

[Esc15] S. Escobar (ed.). *Proc. XIV Jornadas sobre Programación y Lenguajes, PROLE 2015, Cádiz, Spain.* EPTCS 173. 2015.

[FC98] J. Fodor, T. Calvo. Aggregation functions defined by t-norms and t-conorms. In Bouchon-Meunier (ed.), *Aggregation and Fusion of Imperfect Information.* Pp. 36–48. Physica Verlag, 1998.

[FGS99] F. Formato, G. Gerla, M. I. Sessa. Extension of Logic Programming by Similarity. In *Proc. of the Italian-Portuguese-Spanish Joint Conference on Declarative Programming, APPIA-GULP-PRODE'99, L'Aquila*. Pp. 397–410. 1999.

[FGS00] F. Formato, G. Gerla, M. I. Sessa. Similarity-based Unification. *Fundamenta Informaticae* 40(4):393–414, 2000.

[Fit91] M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming* 11:91–116, 1991.

[Fit96] M. Fitting. *First-order logic and automated theorem proving*. Springer Verlag, 1996.

[FR92] J. Fodor, M. Roubens. Aggregation and scoring procedures in multicriteria decision making methods. In *Proc. of the IEEE International Conference on Fuzzy Systems 1992*. Pp. 1261–1267. 1992.

[FY94] J. Fodor, R. R. Yager. Fuzzy Preference Modelling and Multicriteria Decision Support. In *Theory and Decision Library, Series D, System Theory, Knowlege engineering and Problem Solving*. Volume 14. Kluwer Academic, Kluwer, 1994.

[GCWK12] P. X. Gao, A. R. Curtis, B. Wong, S. Keshav. It's Not Easy Being Green. In *Proc. of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. Pp. 211–222. 2012.

[Ger04] G. Gerla. Representation theorems for fuzzy orders and quasi-metrics. *Soft Computing* 8(8):571–580, 2004.

[Ger05] G. Gerla. Fuzzy Logic Programming and fuzzy control. *Studia Logica* 79:231–254, 2005.

[Gin88] M. L. Ginsberg. Multi-valued logics: a uniform approach to reasoning in artificial intelligence. *Computational Intelligence* 4:265–316, 1988.

[GM08a] J. Guerrero, G. Moreno. Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding. *In J. Visser et al, editors, Proc. of the 8th.Int. Workshop on Rule-Based Programming, RULE'07, Electronic*

Notes in Theoretical Computer Science, Elsevier Science, Amsterdam 219:19–34, 2008.

[GM08b] J. Guerrero, G. Moreno. Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding. *Electronic Notes in Theoretical Computer Science* 219:19–34, 2008.

[GMV04] S. Guadarrama, S. Muñoz, C. Vaucheret. Fuzzy Prolog: A New Approach Using Soft Constraints Propagation. *Fuzzy Sets and Systems, Elsevier* 144(1):127–150, 2004.

[GMV10] J. Guerrero, G. Moreno, C. Vázquez. Una implementación del λ-cálculo en Prolog. In Gulías et al. (eds.), *Actas del II Taller de Programación Funcional, TPF'10, Valencia, Septiembre 7*. Pp. 7–14. Garceta grupo editorial, 2010.

[Gog69] J. A. Goguen. The logic of inexact concepts. *Synthese* 19:325–373, 1969.

[GS00] D. Gilbert, M. Schroeder. FURY: Fuzzy unification and resolution based on edit distance. In *Proc of BIBE2000*. 2000.

[Hý98] P. Hájek. *Metamathematics of Fuzzy Logic*. Kluwer Academic Publishers, Dordrecht, 1998.

[Haj06] P. Hajek. Fuzzy Logic. In Zalta (ed.), *The Stanford Encyclopedia of Philosophy (Summer 2008 Edition)*. The MRL and the CSLI, Stanford University, 2006.

[Hel13] R. Helgesson. *Generalized General Logics*. PhD thesis, Umea University, Department of Computing Science, 2013.

[Her30] J. Herbrand. *Recherches sur la Theorie de la Demostration*. Travaoux de la Societe des Sciences et des Lettres de Varsovie, 1930.

[HGGG12] S. He, L. Guo, M. Ghanem, Y. Guo. Improving Resource Utilisation in the Cloud Environment Using Multivariate Probabilistic Models. In *Proc of 5th Intl. Conference on Cloud Computing (CLOUD)*. Pp. 574–581. 2012.

[HHV96] F. Herrera, E. Herrera-Viedma, J. L. Verdegay. Direct approach processes in group decision making using linguistic OWA operators. *Fuzzy Sets and Systems* 79(2):175–190, 1996.

[Hin86] C. Hinde. Fuzzy prolog. *International Journal Man-Machine Studies* 24:569–595, 1986.

[IK85] M. Ishizuka, N. Kanai. Prolog-ELF incorporating fuzzy logic. In Joshi (ed.), *Proc. of the 9th International Joint Conference on Artificial Intelligence, IJCAI'85. Los Angeles, 1985.* Pp. 701–703. Morgan Kaufmann, 1985.

[IMPV15] P. J. Iranzo, G. Moreno, J. Penabad, C. Vázquez. A Fuzzy Logic Programming Environment for Managing Similarity and Truth Degrees. Pp. 71–86 in [Esc15].
`http://dx.doi.org/10.4204/EPTCS.173.6`

[JA07] P. Julián, M. Alpuente. *Programación Lógica. Teoría y Práctica*. Pearson Educación, S.A., Madrid, 2007.

[Jen04] S. Jenei. How to construct left-continuous triangular norms: state of the art. *Fuzzy Sets and Systems* 143:27–45, 2004.

[Jen06] S. Jenei. On the convex combination of left-continuous t-norms. *Aequationes Mathematicae* 72:47–59, 2006.

[JM03] S. Jenei, F. Montagna. A general method for constructing left-continuous t-norms. *Fuzzy Sets and Systems* 136:263–282, 2003.

[JMP04] P. Julián, G. Moreno, J. Penabad. Unfolding Fuzzy Logic Programs. In *Proc. of the Fourth International Conference on Intelligent Systems Design and Applications, ISDA'04 (Sponsored by IEEE). Budapest (Hungary), August 26-28.* Pp. 595–600. 2004.

[JMP05a] P. Julián, G. Moreno, J. Penabad. On Fuzzy Unfolding. A Multi-Adjoint Approach. *Fuzzy Sets and Systems, Elsevier* 154:16–33, 2005.

[JMP05b] P. Julián, G. Moreno, J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. In López-Fraguas (ed.), *Actas de las V Jornadas sobre Programación y Lenguajes, PROLE'05, Granada, Septiembre 14-16.* Pp. 239–248. Universidad de Granada, 2005.

[JMP06a] P. Julián, G. Moreno, J. Penabad. Efficient Reductants Calculi using Partial Evaluation Techniques with Thresholding. In Lucio (ed.), *Actas*

de las VI Jornadas sobre Programación y Lenguajes, PROLE'06, Sitges, Octubre 10-12. Pp. 275–289. Universidad Politécnica de Cataluña, 2006. ISBN 84-95999-84-6.

[JMP06b] P. Julián, G. Moreno, J. Penabad. Evaluación Parcial de Programas Lógicos Multi-adjuntos y Aplicaciones. In Fernández (ed.), *Proc. of Campus Multidisciplinar en Percepción e Inteligencia, CMPI'06, Albacete, Spain, July 10-14*. Pp. 712–724. Universidad de Castilla-La Mancha, 2006. (ISBN 84-689-9560-6). An extended version published in *Fuzzy Sets and Systems* can be found in [JMP09b].

[JMP06c] P. Julián, G. Moreno, J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science* 12:1679–1699, 2006. Extended version of [JMP05b].

[JMP07a] P. Julián, G. Moreno, J. Penabad. Efficient Reductants Calculi using Partial Evaluation Techniques with Thresholding. In Lucio (ed.), *Electronic Notes in Theoretical Computer Science*. Volume 188, pp. 77–90. Elsevier, 2007. Extended version of [JMP06a].

[JMP07b] P. Julián, G. Moreno, J. Penabad. Measuring the Interpretive Cost in Fuzzy Logic Computations. In all (ed.), *Proc. of 7th. International Whorkshop on Fuzzy Logic and Applications WILF'07, Portofino, July 07-10*. Pp. 28–36. Springer Verlag, Lectures Notes in Artificial Intelligence 4578, 2007.

[JMP09a] P. Julián, G. Moreno, J. Penabad. An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques. *Fuzzy Sets and Systems, Elsevier* 160:162–181, 2009.

[JMP09b] P. Julián, G. Moreno, J. Penabad. An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques. *Fuzzy Sets and Systems* 160:162–181, 2009.

[JMP09c] P. Julián, G. Moreno, J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In al (ed.), *Bio-Inspired Systems: Computational and Ambient Intelligence, Proc. of 10th International Work-Conference on Artificial Neural Networks, IWANN'09, Sa-*

lamanca, June 10-12, 2009. Lecture Notes in Computer Science 5517, pp. 253–260. Springer, 2009.

[JMPV14] P. Julián-Iranzo, G. Moreno, J. Penabad, C. Vázquez. A Fuzzy Logic Programming Environment for Managing Similarity and Truth Degrees. Pp. 145–154 in [Esc15].

[JMV15] P. Julián-Iranzo, G. Moreno, C. Vázquez. Similarity-based Strict Equality in a Fully Integrated Fuzzy Logic Language. In *Proc. of the 9th International Web Rule Symposium, RuleML'15. Berlin, Germany, August 2-5.* Pp. 193–207. Springer LNCS, 2015.

[JR06a] P. Julián, C. Rubio. Introducing Fuzzy Unification into the Warren Abstract Machine. In Sirlantzis (ed.), *In Proc. of the 6th International Conference on Recent Advances in Soft Computing, RASC'06. Canterbury, UK, July 10-12*. Pp. 36–41. University of Kent, 2006. (ISBN: 978-1-902671-42-0).

[JR06b] P. Julián, C. Rubio. A WAM Implementation for Flexible Query Answering. In Pobil (ed.), *In Proc. of the 10th IASTED International Conference on Artificial Intelligence and Soft Computing, ASC'06, August 28-30, 2006, Palma de Mallorca*. Pp. 262–267. ACTA Press, 2006.

[JR09a] P. Julián, C. Rubio. A declarative semantics for Bousi∼Prolog. In Porto and López-Fraguas (eds.), *Proc. of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'09, September 7-9, 2009, Coimbra*. Pp. 149–160. ACM, 2009.

[JR09b] P. Julián, C. Rubio. A Similarity-Based WAM for Bousi∼Prolog. In al (ed.), *Bio-Inspired Systems: Computational and Ambient Intelligence, Proc. of 10th International Work-Conference on Artificial Neural Networks, IWANN'09, Salamanca, June 10-12, 2009*. Lecture Notes in Computer Science 5517, pp. 245–252. Springer, 2009.

[JR09c] P. Julián, C. Rubio. UNICORN: A Programming Environment for Bousi∼Prolog. In P. Lucio and Peña (eds.), *Actas de las IX Jornadas*

sobre Programación y Lenguajes, PROLE'09, San Sebastián, Septtiembre 8-11. Pp. 99–108. Universidad del País Vasco, 2009. ISBN 978-84-692-4600-9.

[JR10a] P. Julián, C. Rubio. BOUSI PROLOG - A Fuzzy Logic Programming Language for Modeling Vague Knowledge and Approximate Reasoning. In *International Conference on Fuzzy Computation, Proc. of ICFC'10, Valencia*. P. 10. INSTICC–The Institute for Systems and Technologies of Information, Control and Communication, 2010.

[JR10b] P. Julián, C. Rubio. An Efficient Fuzzy Unification Method and its Implementation into the Bousi∼Prolog System. In IEEE (ed.), *2010 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE'10, July 18-23, Barcelona*. Pp. 658–665. 2010.

[JR11] P. Julián-Iranzo, C. Rubio-Manzano. A Sound Semantics for a Similarity-Based Logic Programming Language. In *Proc. of 11th Int. Work-Conference on Artificial Neural Networks, IWANN'11, Part II, Torremolinos, Málaga, Spain, 2011*. Lecture Notes in Computer Science 6692, pp. 421–428. Springer, 2011.

[JRG08] P. Julián, C. Rubio, J. Gallardo. Bousi∼Prolog: a Prolog extension language for flexible query answering. In Almendros (ed.), *Actas de los VIII Jornadas sobre Programación y Lenguajes, PROLE'08, Gijón, Octubre 7-10*. Pp. 41–55. Fundación Universidad de Oviedo, 2008. ISBN 978-84-612-5819-2.

[JRG09a] P. Julián, C. Rubio, J. Gallardo. Bousi∼Prolog: a Prolog Extension Language for Flexible Query Answering. *Electronic Notes in Theoretical Computer Science* 248:131–147, 2009.

[JRG09b] P. Julián, C. Rubio, J. Gallardo. Inclusión de Conjuntos Borrosos en el Núcleo del Lenguaje Bousi∼Prolog. In Mateos (ed.), *Proc. of the First Workshop on Computational Logics and Artificial Intelligence, CLAI'09, Sevilla, November 9*. Pp. 81–90. Universidad de Sevilla, 2009.

[JRG10] P. Julián, C. Rubio, J. Gallardo. Inclusión de Conjuntos Borrosos en el Núcleo del Lenguaje Bousi∼Prolog. In al. (ed.), *Actas del XV Congreso Español de Tecnología y Lógica Fuzzy, ESTYLF'10, 3 a 5 de Febrero,*

Huelva. Pp. 211–216. Universidad de Huelva, 2010. ISBN 978-84-92944-02-6 (Versión extendida de [JRG09b]).

[Jul00] P. Julián. *Especialización de Programas Lógico-Funcionales Perezosos*. Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia. Tesis Doctoral, 2000.

[Jul04] P. Julián. *Lógica simbólica para informáticos*. RA-MA, Madrid, 2004.

[KK94] F. Klawonn, R. Kruse. A Łukasiewicz logic based Prolog. *Mathware & Soft Computing* 1(1):5–29, 1994.

[KK99] A. Kolesárová, M. Komorníková. Triangular norm-based iterative compensatory operators. *Fuzzy Sets and Systems* 104(1):109–120, 1999.

[KL88] M. Kifer, A. Li. On the Semantics of Rule-Based Expert Systems with Uncertainty. In *Proc. of the 2th International Conference on Database Theory, ICDT'88*. Pp. 102–117. Springer-Verlag, London, 1988.

[KLM+02] S. Krajci, R. Lencses, J. Medina, M. Ojeda-Aciego, A. Valverde, P. Vojtás. Non-commutativity and Expressive Deductive Logic Databases. In *Proc. of the European Conference on Logics in Artificial Intelligence, JELIA'02*. Pp. 149–160. Springer-Verlag, London, 2002.

[KLV02] S. Krajči, R. Lencses, P. Vojtáš. A data model for annotated programs. *ADBIS Research Communications*, pp. 141–154, 2002.

[KLV04] S. Krajči, R. Lencses, P. Vojtáš. A comparison of fuzzy and annotated logic programming. *Fuzzy Sets and Systems* 144:173–192, 2004.

[KM97] M. A. Khamsi, D. Misane. Fixed point theorems in logic programming. *Annals of Mathematics and Artificial Intelligence* 21:231–243, 1997.

[KMP00] E. Klement, R. Mesiar, E. Pap. *Triangular Norms*. Trends in logic, Studia logica library. Springer, 2000.
`http://books.google.es/books?id=rIyqcjfKMN4C`

[KMP04] E. P. Klement, R. Mesiar, E. Pap. Triangular norms. General constructions and parameterized families. *Fuzzy Sets and Systems* 145(1):411–438, 2004.

[Kow74] R. A. Kowalski. Predicate Logic as a Programming Language. *Information Processing* 74:569–574, 1974.

[KS92] M. Kifer, V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming* 12:335–367, 1992.

[KY95] G. J. Klir, B. Yuan. *Fuzzy Sets and Fuzzy Logic*. Prentice-Hall, 1995.

[Lee72] R. C. T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM* 19(1):119–129, January 1972.

[Lin65] C. Ling. Representation of associative functions. *Publicationes Mathematicae Debrecen* 12:189–212, 1965.

[LL90] D. Li, D. Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

[Llo84] J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1984.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[LMM88] J. L. Lassez, M. J. Maher, K. Marriott. Unification Revisited. In Minker (ed.), *Foundations of Deductive Databases and Logic Programming*. Pp. 587–625. Morgan Kaufmann, Los Altos, 1988.

[LS94] L. V. S. Lakshmanan, F. Sadri. Probabilistic Deductive Databases. In *Symposium on Logic Programming*. Pp. 254–268. 1994.

[LS01a] L. V. S. Lakshmanan, F. Sadri. On a theory of probabilistic deductive databases. *Theory and Practice of Logic Programming* 1(1):5–42, 2001.

[LS01b] L. V. S. Lakshmanan, N. Shiri. A Parametric Approach to Deductive Databases with Uncertainty. *IEEE Transactions on Knowledge and Data Engineering* 13(4):554–570, 2001.

[LS02a] Y. Loyer, U. Straccia. Uncertainty and Partial Non-uniform Assumptions in Parametric Deductive Databases. In *Proc. of the European Conference on Logics in Artificial Intelligence*. Pp. 271–282. Springer-Verlag, London, 2002.

[LS02b] Y. Loyer, U. Straccia. The Well-Founded Semantics in Normal Logic Programs with Uncertainty. In *Proc. of the 6th International Symposium on Functional and Logic Programming*. Pp. 152–166. Springer-Verlag, London, 2002.

[LS03] Y. Loyer, U. Straccia. The Approximate Well-founded Semantics for Logic Programs with Uncertainty. In *Proc. of the 28th International Symposium on Mathematical Foundations of Computer Science*. Volume 2747, pp. 541–550. Lecture Notes in Computer Science, 2003.

[LS04] Y. Loyer, U. Straccia. Epistemic Foundation of the Well-Founded Semantics over Bilattices. In *Proc. of the 29th International Symposium on Mathematical Foundations of Computer Science, MFCS'04*. Volume 3153, pp. 513–524. Springer, Heidelberg, 2004.

[LS05] Y. Loyer, U. Straccia. Approximate Well-founded Semantics, Query Answering and Generalized Normal Logic Programs over Lattices. Technical report ISTI-2005-TR-xx, I.S.T.I. - C.N.R., 2005.

[LS06] Y. Loyer, U. Straccia. Epistemic foundation of stable model semantics. *Theory and Practice of Logic Programming* 6(4):355–393, 2006.

[LS09] Y. Loyer, U. Straccia. Approximate well-founded semantics, query answering and generalized normal logic programs over lattices. *Annals of Mathematics and Artificial Intelligence* 55(3-4):389–417, 2009.

[LSS01] V. Loia, S. Senatore, M. I. Sessa. Similarity-based SLD Resolution and Its Implementation in An Extended Prolog System. In *Proc. of the 10th IEEE International Conference of Fuzzy Systems FUZZ-IEEE'01, Melbourne*. Pp. 650–653. 2001.

[Lu96] J. J. Lu. Logic programming with signs and annotations. *Journal of Logic and Computation* 6(6):755–778, 1996.

[Luk01] T. Lukasiewicz. Probabilistic logic programming with conditional constraints. *ACM Transactions on Computational Logic* 2(3):289–339, 2001.

[ML71] S. Mac Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, Sept. 1971.

[Mam93] E. Mamdani. Twenty Years of Fuzzy Control: Experiences Gained and Lessons Learnt. In *Proc. of the 2th IEEE International Conference on Fuzzy Systems*. Pp. 339–344. 1993.

[MBP87] T. P. Martin, J. F. Baldwin, B. W. Pilsworth. The implementation of fprolog–a fuzzy prolog interpreter. *Fuzzy Sets Systems* 23(1):119–129, 1987.

[MC97] G. Mayor, T. Calvo. Extended aggregation functions. In *Proc. Seventh International Fuzzy Systems Association congress, IFSA'97, Prague*. Volume I, pp. 281–285. Academy Sciences, 1997.

[MCS09] S. Muñoz-Hernández, V. P. Ceruelo, H. Strass. RFuzzy: An Expressive Simple Fuzzy Compiler. In Cabestany et al. (eds.), *Proc. of 10th International Work-Conference on Artificial Neural Networks, IWANN'09*. Lecture Notes in Computer Science 5517, pp. 270–277. Springer, 2009.

[MCS11a] S. Muñoz-Hernández, V. P. Ceruelo, H. Strass. RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog. *Information Sciences* 181(10):1951–1970, 2011.

[MCS11b] S. Muñoz-Hernández, V. P. Ceruelo, H. Strass. RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog. *Information Sciences* 181(10):1951–1970, 2011.

[MIK+10] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, D. Pendarakis. Efficient resource provisioning in compute clouds via VM multiplexing. In *Proc. of the Intl. Conference on Autonomic Computing (ICAC)*. Pp. 11–20. 2010.

[Miz89a] M. Mizumoto. Pictorial representations of fuzzy connectives, part I: cases of t-norms, t-conorms and averaging operators. *Fuzzy Sets and Systems* 31(2):217–242, 1989.

[Miz89b] M. Mizumoto. Pictorial representations of fuzzy connectives, part II: cases of compensatory operators and self-dual operators. *Fuzzy Sets and Systems* 32(1):45–79, 1989.

[MM82] A. Martelli, U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems* 4:258–282, 1982.

[MM08a] P. Morcillo, G. Moreno. FLOPER, a Fuzzy Logic Programming Environment for Research. In Almendros (ed.), *Actas de las VIII Jornadas sobre Programación y Lenguajes, PROLE'08, Gijón, Octubre 7-10.* Pp. 259–263. Fundación Universidad de Oviedo, 2008. ISBN 978-84-612-5819-2.

[MM08b] P. Morcillo, G. Moreno. The Fuzzy Logic Programming Environment FLOPER. In Leuschel (ed.), *15th International Symposium on Formal Methods, (FM'08),* "Posters and Research Tools"*, May 26-30, Turku.* 2008.

[MM08c] P. Morcillo, G. Moreno. Programming with Fuzzy Logic Rules by Using the FLOPER Tool. In *Proc. of Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08, Orlando, October 30-31.* Pp. 119–126. Springer, Lectures Notes in Computer Science 5321, 2008.

[MM08d] P. Morcillo, G. Moreno. Using FLOPER for Running/Debugging Fuzzy Logic Programs. In Magdalena et al. (eds.), *In Proc. of the 12th International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems, IPMU'08, June 22-27, 2008, Málaga.* Pp. 481–488. U. Málaga ISBN 978-84-612-3061-7, 2008.

[MM09a] P. Morcillo, G. Moreno. Modeling Interpretive Steps in Fuzzy Logic Computations. In Gesù et al. (eds.), *Proc. of the 8th Int. Workshop on Fuzzy Logic and Applications, WILF'09. Palermo, June 9-12.* Pp. 44–51. Springer Verlag, Lectures Notes in Artificial Intelligence 5571, 2009.

[MM09b] P. Morcillo, G. Moreno. Modeling Interpretive Steps in Fuzzy Logic Computations. In Lucio et al. (eds.), *Actas de las IX Jornadas sobre Programación y Lenguajes, PROLE'09, San Sebastián, Septiembre 8-11.* P. 353. Universidad del País Vasco ISBN 978-84-692-4600-9, 2009. (Presentado dentro de la sección de "trabajos de alto nivel ya publicados en revistas o congresos internacionales de prestigio").

[MM09c] P. Morcillo, G. Moreno. On Cost Estimations for Executing Fuzzy Logic Programs. In Arabnia et al. (eds.), *Proc. of the 2009 International Conference on Artificial Intelligence, ICAI'09, July 13-16, 2009, Las Vegas Nevada, USA.* Pp. 217–223. CSREA Press, 2009.

[MM09d] P. Morcillo, G. Moreno. Unfolding Connective Definitions in Multi-adjoint Logic Programs. In Mateos (ed.), *Proc. of the First Workshop on Computational Logics and Artificial Intelligence, CLAI'09 (integrated in CAEPIA'09), SevillA, November 9*. Pp. 59–70. Universidad de Sevilla, 2009.

[MMPV10a] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In al. (ed.), *Proc. of 4th International Symposium on Rule Interchange and Applications, RuleML'10. Washington, October 21–23*. Lecture Notes in Computer Science 6403, pp. 119–126. Springer Verlag, Lectures Notes in Computer Science 6403, 2010.

[MMPV10b] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Modeling Interpretive Steps into the FLOPER Environment. In Arabnia et al. (eds.), *Proc. of the 2010 International Conference on Artificial Intelligence, ICAI'10, July 12-15, 2010, 2 Volumes*. Pp. 16–22. CSREA Press, 2010.

[MMPV10c] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Multi-Adjoint Lattices for Manipulating Truth-Degrees into the FLOPER System. In Gulías et al. (eds.), *Actas de las X Jornadas sobre Programación y Lenguajes, PROLE'10, Valencia, Septiembre 7-10*. Pp. 151–161. Garceta grupo editorial, 2010.

[MMPV11a] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Declarative Traces into Fuzzy Computed Answers. In Bassiliades et al. (eds.), *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, July 19–21*. Pp. 170–185. Springer Verlag, Lectures Notes in Computer Science 6826, 2011.

[MMPV11b] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-Macneille Completion and Multi-Adjoint Lattices. In Vigo-Aguiar (ed.), *Proc. of the 11th International Conference on Mathematical Methods in Science and Engineering, CMMSE'11, June 26-30, Benidorm*. Pp. 846–857. ISBN 978-84-614-6167-7, 2011.

[MMPV11c] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In al. (ed.), *Advances in*

Computational Intelligence - Proc of the 11th International Work-Conference on Artificial Neural Networks, IWANN'11. Pp. 445–452. Springer Verlag, Lectures Notes in Computer Science 6692, 2011.

[MMPV12a] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-MacNeille completion and Cartesian product of multi-adjoint lattices. *International Journal of Computer Mathematics* 89(13-14):1742–1752, 2012.

[MMPV12b] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations. In Gallardo et al. (eds.), *Actas de las XII Jornadas sobre Programación y Lenguajes, PROLE'12 (Jornadas SISTEDEs Almería 17-19 Sept. 2012)*. Pp. 177–191. Universidad de Almería ISBN:978-84-15487-27-2, 2012.

[MMPV12c] G. Moreno, P. J. Morcillo, J. Penabad, C. Vázquez. String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations. *Electronic Communications of the European Association of Software Science and Technology (EASST)* 55:1–17, 2012.

[MO02] J. Medina, M. Ojeda-Aciego. A new approach to completeness for multi-adjoint logic programming. In *Proc. of 9th Information Processing and Management of Uncertainty in Knowledge-Based Systems Conference, IPMU'02, Annecy*. 2002.

[MO04] J. Medina, M. Ojeda-Aciego. Multi-adjoint logic programming. In *In Proc. of the 10th International Conference on Information Processing and Managment of Uncertainty in Knowledge-Based Systems, IPMU'04, Perugia*. Pp. 823–830. 2004.

[MOR05] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño. Fuzzy logic programming via multilattices: first results and prospects. In *Proc. of Lógica Fuzzy & Soft Computing, LFSC'05, Granada*. Pp. 19–26. Thomson, 2005.

[MOR06a] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño. On the ideal semantics of multilattice-based logic programs. In *Proc. of the 11th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU'06, Paris*. Pp. 463–470. 2006.

[Mor06b] G. Moreno. Building a Fuzzy Transformation System. In Wiedermann et al. (eds.), *Proc. of the 32th Conference on Current Trends in Theory*

and Practice of Computer Science, SOFSEM'06. Merin, January 21-27. Pp. 409–418. Springer Lectures Notes in Computer Science 3831, 2006.

[MOR07a] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño. A fixed-point theorem for multi-valued functions with application to multilattice-based logic programming. *Lecture Notes in Artificial Intelligence* 4578:37–44, 2007.

[MOR07b] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño. Fuzzy logic programming via multilattices. *Fuzzy Sets and Systems* 158(6):674–688, 2007.

[MOR07c] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño. On reachability of minimal models of multilattice-based logic programming. *Lecture Notes in Artificial Intelligence* 4827:271–282, 2007.

[Mor13] P. J. Morcillo. *Semánticas Operacionales Avanzadas para Programas Lógicos Difusos*. Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha. Tesis Doctoral, Marzo 2013.

[MOV01a] J. Medina, M. Ojeda-Aciego, P. Vojtas. A completeness theorem for multi-adjoint logic programming. In *Proc. of 10th IEEE International Conference on Fuzzy Systems, IEEE Press, Melbourne*. Volume 2, pp. 1031–1034. 2001.

[MOV01b] J. Medina, M. Ojeda-Aciego, P. Vojtás. A Multi-adjoint Logic Approach to Abductive Reasoning. In *Proc. of the 17th International Conference on Logic Programming, EPIA'01, Lectures Notes in Artificial Intelligence 2258*. Pp. 269–283. Springer-Verlag, London, 2001.

[MOV01c] J. Medina, M. Ojeda-Aciego, P. Vojtás. A Procedural Semantics for Multi-adjoint Logic Programming. In *Proc. of the 10th Portuguese Conference on Artificial Intelligence on Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, EPIA'01, Lectures Notes in Artificial Intelligence 2258*. Pp. 290–297. London, 2001.

[MOV01d] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Proc. of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Vienna, Lecture Notes in Artificial Intelligence 2173*. Pp. 351–364. 2001.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems, Elsevier* 146:43–62, 2004.

[MPS11] S. Muñoz-Hernández, V. Pablos-Ceruelo, H. Strass. RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog. *Information Sciences* 181(10):1951 – 1970, 2011.

[MPV12a] G. Moreno, J. Penabad, C. Vázquez. On Fuzzy Correct Answers and Logical Consequences in Multi-Adjoint Logic Programming. In Vigo-Aguiar (ed.), *Proc. of 12th International Conference on Mathematical Methods in Science and Engineering, CMMSE'12. La Manga, Spain, July 2-5.* Volume III, pp. 864–875. 2012.

[MPV12b] G. Moreno, J. Penabad, C. Vázquez. SSE: Similarity-based Strict Equality for Multi-adjoint Logic Programs. In Vigo-Aguiar (ed.), *Proc. of the 12th International Conference on Mathematical Methods in Science and Engineering, CMMSE 2012.* Volume 3, pp. 876–887. 2012.

[MPV13] G. Moreno, J. Penabad, C. Vázquez. Relaxing the Role of Adjoint Pairs in Multi-Adjoint Logic Programming. In Vigo-Aguiar (ed.), *Proc. of the 13th International Conference on Mathematical Methods in Science and Engineering, CMMSE'13, July 23-27, Cabo de Gata, Almería, Spain.* Volume 3, pp. 1056–1067. ISBN 978-84-616-2723-3, 2013.

[MPV14a] G. Moreno, J. Penabad, C. Vázquez. Beyond Multi-Adjoint Logic Programming. *International Journal of Computer Mathematics* 92:1956–1975, 2014.

[MPV14b] G. Moreno, J. Penabad, C. Vázquez. Fuzzy Logic Rules Modeling Similarity-based Strict Equality. In Denisiuk (ed.), *Proc. of 9th International Symposium Advances in Artificial Intelligence and Applications (AAIA'14), in the Federated Conference on Computer Science and Information Systems (FedCSIS). September 7-10, Warsaw, Poland.* Pp. 119–128. 2014.

[MPV14c] G. Moreno, J. Penabad, C. Vázquez. Fuzzy Sets for a Declarative Description of Multi-adjoint Logic Programming. In *Proc. of Rough Sets and Current Trends in Soft Computing - 9th International Conference*

(RSCTC'14) (Special session on "Fuzzy Logic and Rough Sets: Tools for Imperfect Information"). July 9-13, Granada and Madrid, Spain. Pp. 71–82. Springer LNCS 8536, 2014.

[MSD89] M. Mukaidono, Z. Shen, L. Ding. Fundamentals of Fuzzy Prolog. *International Journal Approximate Reasoning* 3(2):179–193, 1989.

[MTK99] B. Moser, E. Tsiporkova, E. P. Klement. Convex combination in terms of triangular norms: a characterization of idempotent, bisymmetrical and self-dual compensatory operators. *Fuzzy Sets and Systems, Special Issue: Triangular norms* 104:97–108, 1999.

[Muk82] M. Mukaidono. Fuzzy inference of resolution style. *Fuzzy Sets and Possibility Theory*, pp. 224–231, 1982.

[MV14] G. Moreno, C. Vázquez. Fuzzy Logic Programming in Action with FLOPER. *Journal of Software Engineering and Applications* 7:273–298, 2014.

[Ngu02] H. T. Nguyen. *A First Course in Fuzzy and Neural Control*. CRC Press, Inc., Boca Raton, 2002.

[NPM99] V. Novak, I. Perfilieva, J. Mockor. *Mathematical principles of fuzzy logic*. Kluwer, Boston, 1999.

[NS91] R. Ng, V. S. Subrahmanian. Stable Model Semantics for Probabilistic Deductive Databases. In *Proc. of the 6th International Symposium on Methodologies for Intelligent Systems, ISMIS'91*. Pp. 162–171. Springer-Verlag, Charlotte, 1991.

[NS92] R. Ng, V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation* 101(2):150–201, 1992.

[NW06] H. T. Nguyen, E. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, 2006. third edition.

[Pag27] Page view statistics for Wikimedia projects. Web page at `http://dumps.wikimedia.org/other/pagecounts-raw/`, Visited 2014-01-27.

[Pav79] J. Pavelka. On fuzzy logic I, II, III. *Zeitschrift fur Mathemathische Logik und Grundlagen der Mathematik* 25:45–52, 119–134, 447–464, 1979.

[PDH97] R. Palmn, D. Driankov, H. Hellendoorn. *Model-Based Fuzzy Control*. Springer-Verlag, 1997.

[Pen10] J. Penabad. *Desplegado de Programas Lógicos Difusos*. Departamento de Sistemas Informáticos, Universidad de Castilla-La Mancha. Tesis Doctoral, Junio 2010.

[PG98] W. Pedrycz, F. Gomide. *Introduction to fuzzy sets*. MIT Press, Cambridge, 1998.

[PS05] G. Pajares, M. Santos. *Inteligencia Artificial e Ingeniería del Conocimiento*. Ra-Ma, 2005.

[RB85] D. E. Rydeheard, R. M. Burstall. A Categorical Unification Algorithm. In Pitt et al. (eds.), *CTCS*. Lecture Notes in Computer Science 240, pp. 493–505. Springer, 1985.

[RBa09] B. Rochwerger, D. Breitgand, et al. The reservoir model and architecture for open federated cloud computing. *IBM J. Res. Dev.* 53(4):535–545, 2009.

[RD08] M. Rodríguez-Artalejo, C. R. Diaz. Quantitative logic programming revisited. *Springer Lectures Notes in Computer Science 4989*, pp. 272–288, 2008.

[RJ14] C. Rubio-Manzano, P. Julián-Iranzo. A Fuzzy linguistic prolog and its applications. *Journal of Intelligent and Fuzzy Systems* 26(3):1503–1516, 2014.

[Rob65] J. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12(1):23–41, January 1965.

[RR08a] M. Rodríguez-Artalejo, C. A. Romero-Díaz. A declarative semantics for clp with qualification and proximity. *Theory and Practice of Logic Programing* 10:627–642, 2008.

[RR08b] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Quantitative Logic Programming Revisited. In *Proc. of the 9th International Symposium on Functional and Logic Programming, FLOPS'08*. Pp. 272–288. Springer-Verlag, Lecture Notes in Computer Science 4989, 2008.

[RR09] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Qualified Logic Programming with Bivalued Predicates. *Electronic Notes in Theoretical Computer Science* 248:67–82, 2009.

[RTG+12] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, M. A. Kozuch. Towards understanding heterogeneous clouds at scale:Google trace analysis. Technical report ISTC–CC–TR–12–101, Carnegie Mellon University, Pittsburgh, PA, USA, Apr. 2012. `http://www.istc-cc.cmu.edu/publications/papers/2012/ISTC-CC-TR-12-101.pdf`.

[RUB27a] RUBBoS: Bulletin Board Benchmark. Web page at `http://jmob.ow2.org/rubbos.html`, Visited 2014-01-27.

[RUB27b] RUBiS: Rice University Bidding System. Web page at `http://rubis.ow2.org/`, Visited 2014-01-27.

[Rub11] C. Rubio. *Diseño e Implementación de un Lenguaje de Programación Lógica Borrosa con Unificación Débil*. Departamento de Tecnologías y Sistemas de Información, Universidad de Castilla-La Mancha. Tesis Doctoral, Julio 2011.

[RZ01] W. Rounds, G. Q. Zhang. Clausal Logic and Logic Programming in Algebraic Domains. *Information and Computation* 171:183–200, 2001.

[SB75] E. H. Shortliffe, B. G. Buchanan. A model of inexact reasoning in medicine. *Mathematical Biosciences* 23:351–379, 1975.

[Sco82] D. S. Scott. Domains for denotational semantics. *Lecture Notes in Computer Science* 140:577–610, 1982.

[Ses02] M. I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Fuzzy Sets and Systems* 275:389–426, 2002.

[Sha76] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[Sha83] E. Y. Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proc. of the 8th International Joint Conference on Artificial Intelligence, IJCAI'83, Karlsruhe*. Pp. 529–532. 1983.

[SOD09] U. Straccia, M. Ojeda-Aciego, C. V. Damásio. On Fixed-Points of Multivalued Functions on Complete Lattices and Their Application to Generalized Logic Programs. *SIAM Journal on Computing* 38(5):1881–1911, 2009.

[SS83] B. Schweizer, A. Sklar. *Probabilistic Metric Spaces*. North-Holland, New York, 1983.

[Sti88] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated reasoning* 4(4):353–380, 1988.

[Sto04] A. Stouti. A fuzzy version of Tarski's fixpoint theorem. *Archivum Mathematicum* 40(3):273–279, 2004.

[Str05a] U. Straccia. Query Answering in Normal Logic Programs Under Uncertainty. In Godó (ed.), *Proc. of 8th. European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU'05, Barcelona*. Pp. 687–700. Lecture Notes in Computer Science 3571, 2005.

[Str05b] U. Straccia. Uncertainty Management in Logic Programming: Simple and Effective Top-Down Query Answering. In Khosla et al. (eds.), *Proc. 9th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, Part II*. Pp. 753–760. Lecture Notes in Computer Science 3682, 2005.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5:285–309, 1955.

[TAT95] E. Trillas, C. Alsina, J. M. Terricabras. *Introducción a la lógica borrosa*. Ariel, S.A., Barcelona, 1995.

[TCC00] E. Trillas, C. del Campo, S. Cubillo. When QM-Operators Are Implication Functions and Conditional Fuzzy Relations. *International Journal of Intelligent Systems* 15:647–655, 2000.

[The30] The NIST Definition of Cloud Computing. Web page at `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, Visited 2013-05-30.

[TT89] J. M. Terricabras, E. Trillas. Some Remarks on Vague Predicates. *Theoria* 10:1–12, 1989.

[TT13a] L. Tomás, J. Tordsson. Cloudy with a Chance of Load Spikes: Admission Control with Fuzzy Risk Assessments. In *6th IEEE/ACM Intl. Conference on Utility and Cloud Computing*. Pp. 155–162. 2013.

[TT13b] L. Tomás, J. Tordsson. Improving Cloud Infrastructure Utilization through Overbooking. In *Proc. of the ACM Cloud and Autonomic Computing Conference (CAC), Miami, USA*. 2013.

[TT14a] L. Tomás, J. Tordsson. An Autonomic Approach to Risk-Aware Data Center Overbooking. *IEEE Transactions on Cloud Computing* 2(3):292–305, 2014.

[TT14b] L. Tomás, J. Tordsson. Cloud Service Differentiation in Overbooked Data Centers. In *7th IEEE/ACM Intl. Conference on Utility and Cloud Computing*. Pp. 541–546. 2014.

[Tur92] I. B. Turksen. Interval-valued fuzzy sets and "compensatory AND". *Fuzzy Sets Systems* 51(3):295–307, 1992.

[TV85] E. Trillas, L. Valverde. On mode and implication in approximate reasoning. In Gupta et al. (eds.), *Approximate reasoning in expert systems*. North Holland, Elsevier Science Publishers B. V., 1985.

[VBG12] A. Vidal, F. Bou, L. Godo. An SMT-Based Solver for Continuous t-norm Based Logics. In *Proceedings of the 6th International Conference on Scalable Uncertainty Management*. Lecture Notes in Computer Science 7520, pp. 633–640. 2012.

[VGM02] C. Vaucheret, S. Guadarrama, S. Muñoz. Fuzzy Prolog: A Simple General Implementation Using $CLP(R)$. In Baaz and Voronkov (eds.), *Proc. of Logic for Programming, Artificial Intelligence and Reasoning, LPAR'02, Tbilisi*. Pp. 450–463. Lectures Notes in Artificial Intelligence 2514, 2002.

[VMTT15] C. Vázquez, G. Moreno, L. Tomás, J. Tordsson. A Cloud Scheduler Assisted by a Fuzzy Affinity-Aware Engine. In *Proc. of the 2015 IEEE*

International Conference on Fuzzy Systems, FUZZIEEE'15. Istanbul, Turkey, August 2-5. P. 8 (in press). 2015.

[Voj01] P. Vojtáš. Fuzzy Logic Programming. *Fuzzy Sets and Systems, Elsevier* 124(1):361–370, 2001.

[VP96] P. Vojtáš, L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In Dyckhoff and al (eds.), *Proc. of the Workshop on Extensions of Logic Programming, ELP'96, Leipzig.* Pp. 289–301. Lecture Notes in Computer Science 1050, 1996.

[VTMT13] C. Vázquez, L. Tomás, G. Moreno, J. Tordsson. A fuzzy approach to cloud admission control for safe overbooking. In *Proc. of 10th International Workshop on Fuzzy Logic and Applications, WILF 2013.* Pp. 212–225. Springer Verlag, LNAI 8256, 2013.

[VVB13] S. Verboven, K. Vanmechelen, J. Broeckhove. Black Box Scheduling for Resource Intensive Virtual Machine Workloads with Interference Models. *Future Gener. Comput. Syst.* 29(8):1871–1884, 2013.

[VZ96] L. Valverde, L. A. Zadeh. Del control analítico al control borroso. *Universitat de les Illes Balears*, 1996.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, 1983.

[WTL93] T. J. Weigert, J. J. P. Tsai, X. Liu. Fuzzy Operator Logic and Fuzzy Resolution. *Journal of Automated Reasoning* 10(1):59–78, 1993.

[Yag88] R. R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Transactions on Systems, Man and Cybernetics* 18(1):183–190, 1988.

[Yag93a] R. R. Yager. Families of OWA operators. *Fuzzy Sets and Systems* 59(2):125–148, 1993.

[Yag93b] R. R. Yager. Fuzzy Screening Systems. In Owen and Roubens (eds.), *Fuzzy logic: state of the art.* Pp. 251–261. Kluwer Academic Publishers, Dordrecht, 1993.

[Yag94a] R. R. Yager. Aggregation operators and fuzzy systems modeling. *Fuzzy Sets and Systems* 67(2):129–145, 1994.

[Yag94b] R. R. Yager. On weighted median aggregation. *International Journal of Uncertainty* 2:101–113, 1994.

[Yin02] M. Ying. Implication operators in fuzzy logic. *IEEE Transactions on Fuzzy Systems* 10:88–91, 2002.

[Zad65a] L. A. Zadeh. Fuzzy logic and approximate reasoning. *Synthese* 30:407–428, 1965.

[Zad65b] L. A. Zadeh. Fuzzy Sets. *Information and Control* 8(3):338–353, 1965.

[Zad73] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man and Cybernetics* 3(1), 1973.

[Zad75] L. A. Zadeh. The concept of a Linguistic Variable and Its Applications to Aproximate Reasoning. *Information Sciences*, pp. 199–249, 1975.

[Zad96] L. A. Zadeh. Nacimiento y evolución de la lógica borrosa, el soft computing y la computación con palabras: un punto de vista personal. *Psicothema* 8(2):421–429, 1996.

[Zad08] L. A. Zadeh. Is there a need for fuzzy logic? *Information Sciences* 178:2751–2779, 2008.

[ZHa11] M. Zaharia, B. Hindman, et al. The datacenter needs an operating system. In *Proc. of the 3rd USENIX conference on Hot topics in cloud computing*. 2011.

[ZTH+13] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*. Pp. 379–391. 2013.

[ZZ80] H. Zimmermann, P. Zysno. Latent Connectives in Human Decision Making. *Fuzzy Sets and Systems* 4:37–51, 1980.