8-1998

Towards Combining Fuzzy and Logic Programming Techniques

Hung T. Nguyen

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Daniel E. Cooke

Luqi

Olga Kosheleva
The University of Texas at El Paso, olgak@utep.edu

Comments:
Technical Report: UTEP-CS-98-26
In: Nguyen Hoang Phuong and Ario Ohsato (eds.), *Proceedings of the Vietnam-Japan Bilateral Symposium on Fuzzy Systems and Applications VJFUZZY'98*, HaLong Bay, Vietnam, 30th September-2nd October, 1998, pp. 482-489.

Towards Combining Fuzzy and Logic Programming Techniques

Hung T. Nguyen[1], Vladik Kreinovich[2], Daniel E. Cooke[2], Luqi[3], and Olga Kosheleva[4]

[1]Department of Mathematical Sciences, New Mexico State University
Las Cruces, NM 88003, USA, email hunguyen@nmsu.edu
[2,4]Departments of [2]Computer Science and [4]Electrical and Computer Engineering,
University of Texas at El Paso, El Paso, TX 79968, USA,
emails {vladik,dcooke}@cs.utep.edu, olga@ece.utep.edu
[4]Department of Computer Science, Naval Postgraduate School
Monterey, CA 93943, USA

Abstract

Many problems from AI have been successfully solved using fuzzy techniques. On the other hand, there are many other AI problems, in which logic programming (LP) techniques have been very useful. Since we have two successful techniques, why not combine them?

1 Introduction

Two techniques, each successful: why not combine?

- Many problems from AI have been successfully solved using fuzzy techniques.
- On the other hand, there are many other AI problems, in which logic programming (LP) techniques have been very useful.

Since we have two successful techniques, why not combine them? Hopefully, with this combination, we will then be able to solve problems which cannot be easily solved by fuzzy techniques only:

- fuzzy part will attack the part of the problem on which it works best;
- on the other hand, LP part will attack the remaining parts of the original problem.

They are different but consistent. At first glance, these two techniques may look different (and even inconsistent). However, in our previously published papers, we showed that, e.g., interval-valued fuzzy expert systems and LP-based expert systems has the same underlying logic and are, thus, consistent [14, 15]. We have also shown that such logically unusual fuzzy techniques as Mamdani's approach to fuzzy control become very natural if we consider LP instead of classical logic [9, 15].

What do we gain by adding LP techniques to fuzzy? In addition to a better *understanding* of fuzzy techniques (as with Mamdani's approach), we can *improve* these techniques. Indeed, when designing a formalism for representing uncertainty, we want this formalism to be:

- adequate for representing how we actually think;
- adequate for describing the real world, and
- fast and easy to process.

We show that by adding LP, we can gain in all three objectives:

- LP clearly describes how we think.
- LP gives a better understanding of physics. Namely, we show how a LP representation of physical knowledge automatically leads to a physical phenomenon known as spontaneous symmetry violation and thus, enables us to naturally explain the shapes of celestial bodies, faults in solid bodies, etc., and the evolution of such shapes.
- LP, by its very origin, is a modification of logic whose aim was to make it easily computable. Prolog and other LP languages make LP efficient. In addition to this *sequential* efficiency, LP has a natural *parallelization*. This parallelization is so powerful that even such algorithmic breakthroughs as Fast Fourier Transform, breakthroughs that required ingenious ideas to invent, can be naturally obtained by parallelizing the corresponding LP. We also show that the existing semantics of LP is consistent with this parallelization.

Beyond Fuzzy Prolog. One natural combination of fuzzy techniques and LP is Fuzzy Prolog. But what we propose goes deeper than fuzzy prolog:

- we propose to use not only the algorithms of LP, but also its semantics and foundations;

- we also propose to use the inherent parallelism of LP (which is not yet reflected in fuzzy prolog) to speed up the corresponding knowledge processing.

Comment. The examples we give in this paper explain how the collaboration between fuzzy logic and logic programming can benefit fuzzy logic. We believe that this collaboration will benefit logic programming as well, and we can give two arguments in favor of this belief:

- Many heuristic methods in logic programming, in its algorithms and its foundations, can be formalized and thus made understandable if we use fuzzy logic (see, e.g., [16]).
- In applying each knowledge representation formalism, it is important to be sure that the formalism can be applied. In fuzzy control and, more generally, in fuzzy system modeling, such a possibility follows from the *universal approximation* theorems; at present, there are so many theorems covering different possible problems and different approximation criteria that a recent survey [10] takes more than 60 pages (without proofs!). Until recently, no such theorems were known for logic programming approach. Currently, there are some universal representation and approximation results [1, 8], but there are still many open problems (some of them mentioned in [1]), and the experience of universal approximation results in fuzzy logic can definitely help.

2 Logic programming gives a better understanding of physics

In this section, we show that a LP representation of physical knowledge automatically leads to a physical phenomenon known as spontaneous symmetry violation and thus, enables us to naturally explain the shapes of celestial bodies, faults in solid bodies, etc.

Shapes of celestial bodies: a physical problem. Celestial bodies such as galaxies, stellar clusters, planetary systems, etc., have different geometric shapes (e.g., galaxies can be spiral or circular, etc.). Usually, complicated physical theories are used to explain these shapes; for example, several dozen different theories explain why many galaxies are of spiral shape; see, e.g., [2, 22, 23, 25]. Some rare shapes are still unexplained. How can we explain these shapes?

How have the shapes been formed? To find out how shapes have been formed, let us start from the beginning of the Universe (for a detailed physical description, see, e.g., [26]). The only evidence about the earliest stages of the Universe is the cosmic 3K background radiation. This radiation is highly homogeneous and isotropic; this means that initially, the distribution of matter in the Universe was highly homogeneous and isotropic. In mathematical terms, the initial distribution of matter was invariant w.r.t. arbitrary shifts and rotations.

We can also say that the *initial distribution was invariant* w.r.t. dilations if in addition to dilation in space (i.e., to changing the units of length), we accordingly change the units of mass. In the following text, we will denote the corresponding transformation group (generated by arbitrary shifts $x \to x + a$, rotations, and dilation $x \to \lambda \cdot x$) by G_0.

On the astronomical scale, of all fundamental forces (strong, weak, etc.) only two forces are non-negligible: gravity and electromagnetism. The equations that describe these two forces are invariant w.r.t. arbitrary shifts, rotations, and dilations in space. In other words, these interactions are *invariant* w.r.t. our group G_0. The *initial distribution* was *invariant* w.r.t. G_0; the *evolution equations are* also *invariant*; hence, we will get G_0-invariant distribution of matter for all moments of time. But our world is not homogeneous. Why?

The reason why we do not see this homogeneous distribution is that this highly symmetric distribution is known to be *unstable*: If, due to a small perturbation, at some point a in space, density becomes higher than in the neighboring points, then this point a will start attracting matter from other points. As a result, its density will increase even more, while the density of the surrounding areas will decrease. So, arbitrarily small perturbations cause drastic changes in the matter distribution: matter concentrates in some areas, and shapes are formed. In physics, such symmetry violation is called *spontaneous*.

Spontaneous symmetry violations: an important result from statistical physics. In principle, it is possible to have a perturbation that changes the initial highly symmetric state into a state with no symmetries at all, but statistical physics teaches us that it is much more probable to have a *gradual* symmetry violation: first, some of the symmetries are violated, while some still remain; then, some other symmetries are violated, etc. (Similarly, a (highly organized) solid body normally goes through a (somewhat organized) liquid phase before it reaches a (completely disorganized) gas phase.) At the end, we get the only stable shape: rotating ellipsoid.

Before we reach the ultimate ellipsoid stage, perturbations are invariant w.r.t. some subgroup G' of the initial group G_0. If a certain perturbation concentrates matter, among other points, at some point a, then, due to invariance, for every transformation $g \in G'$, we will observe a similar concentration at the point $g(a)$. Therefore, the shape of the resulting concen-

tration contains, with every point a, the entire *orbit* $G'a = \{g(a) \mid g \in G'\}$ of the group G'. Hence, the resulting *shape consists of* one or several *orbits* of a group G'.

This result from statistical physics explains the shapes of celestial objects. As a result, we can now describe all possible shapes as all possible *orbits* of subgroups G' of the group G_0 (= all shifts, rotations, and dilations), and the most likely evolution of the shape as a transition from a group G' to a largest possible subgroup $G'' \subset G'$. It turns out that this explanation indeed describes all observed celestial shapes, from the most widely observed ones like logarithmic spirals of the spiral galaxies to the most rare and most weird ones like conic spirals or parallel line-like bars, and not only describes the shapes themselves, but also their relative frequency and evolution (see [3, 4, 5] for details).

For example, from the original group G_0, the most likely transition is to a symmetry group corresponding to a plane, and from there – to a group corresponding to the logarithmic spiral (which explains why spiral galaxies are so widely spread).

The problem with this explanation. To explain the observed shapes, we started with the theories which used very specific physical equations to explain them, and ended up with a new explanation which is (almost) physics-free. The new explanation only uses the symmetries and geometry. The problem is in the word "almost": in addition to geometric features of shapes and symmetries, our explanation also uses a rather complicated (and, at least at first glance) not very intuitive fact from statistical physics – that after a spontaneous symmetry violation, the most probable states are the ones with the largest remaining symmetry group.

Logic programming helps. We will show that this fact becomes very intuitively clear if we described it in terms of logic programming.

In logic programming, knowledge is represented by *facts* (elementary statements), and if-then *rules* of the type "if A_1 and ... and A_n then B". To distinguish between these "commonsense" (intuitive) if-then rules and the formal-logic (sometimes counterintuitive) implication, researchers in logic programming usually denote their if-then rules by an inverted arrow $B \leftarrow A_1, \ldots, A_n$.

We start with a set G of symmetries, and we want to describe which symmetries remain after the spontaneous symmetry violation. We will describe the fact that g remains a symmetry after the violation, i.e., that the matter distribution remains *invariant* w.r.t. g, as $inv(g)$. Clearly, if the distribution is invariant w.r.t. g_1 and g_2, it will remain invariant if we apply both transformations, i.e., their composition $g_1 \circ g_2$, or the reverse transformation g_1^{-1}. Thus, we get the following rules:

$$inv(g_1 \circ g_2) \leftarrow inv(g_1), inv(g_2). \quad (1)$$

$$inv(g^{-1}) \leftarrow inv(g). \quad (2)$$

To these rules, we may also want to add the fact that typically, unless forced out, the invariance stays. In logic programming, such facts are traditionally described by using a special predicate "abnormal" (*ab* for short), as

$$inv(g) \leftarrow not\ ab(g). \quad (3)$$

One of the possible interpretation of commonsense rules like (1)–(3) is through the so-called *circumscription* (see, e.g., [13]). According to circumscription, common sense corresponds to models in which the set of all abnormal objects is the smallest possible (in some reasonable sense, e.g., in the sense that no proper subset of it can serve as a set of abnormal objects here). Let us apply circumscription to our knowledge base (1)–(3). According to the rules (1)–(2), the set of all transformations g for which $inv(g)$ is true is closed under composition and taking the inverse element, and is, therefore, a *subgroup* G' of the original group G. According to the rule (3), every non-invariant transformation g is abnormal. Since we are minimizing the set of all abnormal elements, we can make two conclusions:

- first, that when G' is fixed, the smallest possible set of abnormal transformations is when only non-invariant transformations (i.e., $g \notin G'$) are abnormal and none else;
- because of that, to *minimize* the set of abnormal elements, we must select the *largest* possible subgroup G'.

This is exactly what we were trying to explain. Thus, logic programming indeed leads to an intuitive explanation of shapes of celestial bodies.

First physical comment. Similar shapes happen not only in celestial bodies, but also in fracture theory: for a symmetric body, each fault (crack, etc.) is a spontaneous symmetry violation [24]. This fact not only *explains* the shapes of the faults [24], it enables us to describe the best sensor locations for *detecting* these faults [18, 19, 20].

A knowledge representation comment. In many cases, abnormal elements are rare, so that some researchers even proposed to interpret "abnormal" as "having a small probability" and thus, use probability theory instead of logic programming. McCarthy [13], the author of circumscription, strongly opposes this identification, correctly claiming that there are some cases when most objects are abnormal. Here, we have an extreme example, where typically, the group G' is of smaller dimension than G and thus, not only most, but even *almost all* elements of G are abnormal. Thus, we have an example where a straightforward probabilistic interpretation would not work.

Second physical comment. Another physical property that can be thus explained is Laplace's *principle of insufficient reason* [12, 21], according to which if we have no reasons to assume that two elements a, b have different probability $P(a), P(b)$, then they should have the same probability. This principle is one of the foundations of statistical applications, but, if formulated too generally, as $P(a) = P(b)$ for all pairs of events (a, b) about which we have no reasons to believe that $P(a) \neq P(b)$, i.e., as

$$P(a) = P(b) \leftarrow not\ B(P(a) \neq P(b))$$

(where B stands for "believe"), it can lead to inconsistencies [12, 21]. Since the most general formulation of this principle is inconsistent, there sometimes exist abnormal pairs for which there are no reason to believe that $P(a) \neq P(b)$ but nevertheless for which $P(a) \neq P(b)$. With these abnormal pairs in mind, we can formalize this principle as

$$P(a) = P(b) \leftarrow not\ B(P(a) \neq P(b)), not\ ab(a, b).$$

Thus, we have a *consistent* formalization of this principle, in which the set of abnormal pairs is the smallest possible and thus, the set of pairs for which $P(a) = P(b)$ is the largest possible.

Similarly, we can consistently formalize *Occam's razor*, according to which, if we have no reasons to believe that two quantities are different, they should be equal. Again, a straightforward formalization of this principle leads to a contradiction [6]: e.g., if we know that $a \in [0, 1]$, $b \in [1, 2]$, and $c \in [2, 3]$, then it is possible that $a = b$ and it is possible that $b = c$, but we cannot conclude that $a = b$ and $b = c$, because then we would have to conclude that $a = c$, which contradicts to the fact that the intervals of possible values for a and c have no common elements. We can, however, consistently formalize it as

$$a = b \leftarrow not\ B(a \neq b), not\ ab(a, b).$$

In this formalization, in the above example, we would conclude that either $a = b$ or $b = c$ (but not $a = c$).

3 LP is naturally parallelizable

Another aspect of LP is that it is often very algorithmically efficient. This is not surprising because LP, by its very origin, is a modification of logic whose aim was to make it easily computable.

In this section, we will show that in addition to this *sequential* efficiency, LP has a natural *parallelization*, and that this parallelization is so powerful that even such algorithmic breakthroughs as Fast Fourier Transform, breakthroughs that required ingenious ideas to invent, can be naturally obtained by parallelizing the corresponding LP.

Indeed, many signal processing techniques are based on the use of *Fourier transform* [17], i.e., a transformation which transforms a sequence of values $x_1, \ldots, x_n$ into a sequence of Fourier coefficients

$$\hat{x}_j = \sum_k x_k \cdot \exp(\mathrm{i}\frac{k \cdot j}{n} \cdot \pi).$$

How can we compute Fourier coefficients?

First of all, we must input the coefficients $x_1, \ldots, x_n$. In Prolog (and in most other logic programming languages), the only available datatype is a *list*. Hence, we must present the original data as a list $[x_1, \ldots, x_n]$ (we use capital letters because in Prolog, variables are capitalized, while constants are not). The only list operation of Prolog is the operation that enables us to pick up the first element ("head") x_1 of the list and leave the remaining list (called "tail") $[x_2, \ldots, x_n]$.

Let us consider what will happen if we have *two* processors which can work in parallel. Since the only thing we can do with the list is take the first element out, this is what one of the processors (e.g., the first one) will have to start with. While the first processor picks up the element x_1, the second processor has nothing to do, so it is idle. After the element is picked, the first processor can then do some processing with this first element (e.g., multiply it by an appropriate coefficient, to prepare for computing the above-given Fourier sum). While the first processor starts processing its element x_1, the second processor can now use the list. The only thing it can do is pick up the second element from the list, and start processing it. When the second processor has picked up the element x_2, the first processor can come back to the list and pick the third one, etc.

Thus, in a natural parallelization, the first processor will pick up odd-numbered elements $x_1, x_3, \ldots$, while the second processor will pick up even-numbered elements $x_2, x_4, \ldots$. Thus, we reduce the original problem of computing the Fourier transform of this sequence $x_1, \ldots, x_n$ to the two problems of processing odd and even half-lists. This idea, which naturally occurs in a parallel logic programming setting, is, actually, the exact idea behind the well-known Fast Fourier Transform (FFT) algorithm [17]: to compute the Fourier transform of a list, we compute Fourier transforms of two half-lists and then combine the resulting Fourier transforms. (If we have more processors at our disposal, we can repeat the same parallelization for each half-lists, and reduce computing for each of them to computing FFT for quarter lists, etc.)

Thus, FFT, the main signal processing breakthrough algorithm, can be naturally obtained by parallelizing the corresponding logic program.

4 The existing semantics of LP is consistent with parallelization

Simple logic programs are straightforward and easy to understand, and do not require any clarification. More complicated ones require special *semantics*. The existing semantics were developed with sequential computers in mind. Thus, it is not *a priori* clear whether parallelization is consistent with these semantics. In this section, we will show that parallelization is consistent with these semantics. Before we prove it, let us describe the standard semantics for logic programs.

Basic logic programming: a natural choice of semantics. The description of what answers the system should return for each formal specification (and for each queried property) is called the *semantics* of the specification language.

For *basic* logic programs (i.e., programs without negation), semantics immediately follow from the fact that these programs are actually a particular case of formulas of first order logic. In general, if specifications are described by an arbitrary first order formula F, then for every queried property Q, we have one of the following three situations:

- The property Q follows from the formula F; in this case, this property Q is *true* for every program that satisfies this formula (or, in logical terms, in all *models* of this formula F).
- The negation $\neg Q$ of the property Q follows from the formula F; in this case, this property Q is *false* in all *models* of the formula F.
- Neither the property Q, nor its negation follow from the formula F; in this case, this property is *true* for some models that satisfy these formula but *false* for the other models that also satisfy the same formula F.

(Of course, theoretically, there is a fourth possibility: that the formula F are inconsistent.)

This classification can be also applied to rules and facts that form a basic logic program. Fortunately, basic logic programs are a particularly *simple* case of general first order formulas, and due to this simplicity, for first order formulas, we get a simplification of this classification. Indeed, all facts and rules that form a basic logic program remain true if we simply consider a model in which all elementary properties are true. Therefore, whichever of these properties Q we ask about, it is always possible that this property is true. In other words, for basic logic programs, instead of the above *three* possibilities, we have only *two* possibilities:

- First, it is possible that the property Q is *true* for all models of this logic program.
- Second, it is possible that in some models of the logic program F, this property Q is *false*.

The actual Prolog compiler, given a logic program F and a query Q, decides which of these two cases holds. Of course, it makes no practical sense to let the compiler return either of these two *long* messages that describe the corresponding cases. Therefore, only the *shortened* messages are returned:

- In the first case, the compiler returns the shortened message "true" (or, even shorter, "yes");
- in the second case, it returns a shortened message "false" (or, even shorter, "no").

Semantics of generalized logic programs. In logic programs, the negative condition *not* C is interpreted as: "if we have no reasons to believe in C" (this interpretation is called *negation as failure*). For *generalized* logic programs, with negation, we also need to determine a semantics, i.e., we also need to be able to determine, for a given program F, whether a given query Q is true or not.

Negation as failure is not a typical logical connective, and therefore, in contrast to the case of basic logic programs, we cannot directly *deduce* the semantics of a generalized logic program from the known semantics of first order logic. However, we can still deduce this semantic *indirectly*, by checking the *consistency* of the resulting assignment of "true" and "false" to different elementary statements from the program.

Indeed, let us assume that F is a generalized logic program, and that to every elementary statement from this program, we somehow assign "true" or "false". In mathematical terms, this means that we have selected, in the set of all atoms of the original logic program, a subset T formed by those atoms that our semantic deems "true".

In this case, we can determine which rules with negation are applicable and which are not and thus, transform the original rules into new rules that do not contain negation as failure. This transformation can be done as follows:

- If one of the conditions of a rule is *not* C for some atom C that is true, then this rule is not applicable, and we can safely delete it. (Informally, the presence of the condition *not* C means that this rule is only applicable in *normal* situations, in which there is no way to prove C; the fact that C is true means that we have an *exceptional* situation, and thus, the rule is not applicable.)
- If for all exception-type conditions *not* C_i of a rule, C_i is not true (i.e., $C_i \notin T$), then this rule is indeed applicable and therefore, we can simply delete these conditions *not* C_i from the list of conditions. (Informally: this situation is indeed *non-exceptional*, so the rule *is* applicable.)

As a result of this transformation, we get a new logic

program without negation as failure. For this new basic logic program, we can use the above-described semantics and find the resulting set $T_{\rm res}$ of true atoms (i.e., of elementary statements that are true according to this transformed logic program). This set should, of course, coincide with the original set T.

This consistency requirement $T_{\rm res} = T$ only holds for *some* sets of atoms T. Sets of atoms for which $T = T_{\rm res}$, i.e., sets that remain stable (do not change) under this transformation from T to $T_{\rm res}$, are called *stable models* [7] of the original logic program.

Splittings of logic programs. How can we actually *compute* the stable models? In many cases, a logic program has a natural structure which allows us to reduce this problem to the problem of computing stable models for simpler programs. The most general types of this structure was defined in [11] under the name of a *splitting*.

In that paper, this notion was defined for class of logic programs that is more general than we have allowed: namely, for the so-called *disjunctive* logic programs that allow an additional connective "or" in the conclusion of "if"–"then" rules, i.e., which allows rules of the type

$$B_1|\ldots|B_d \leftarrow A_1, \ldots, A_n, not\ C_1, \ldots, not\ C_m,$$

where "$B_1|\ldots|B_d$" means "B_1 or $\ldots$ or B_d". (For such rules, instead of a stable model, we need a more general definition of an *answer set*.)

The main definition from [11] can be reformulated as follows: for every two literals a and b from a logic program, let us denote $a \geq b$ if in one of the rules, a appears in the head (i.e., in the conclusion part of the rule), and b appears elsewhere in this rule, i.e., either in its head (as one of the possible rule's conclusions) or in its body (as one of the conditions of the rule). By a *splitting* of the program, we mean a mapping s from the set of all literals into a linearly well-ordered set (i.e., into the set of all ordinal numbers that are smaller than some ordinal number μ) for which $a \geq b$ implies $s(a) \geq s(b)$. In other words, to every literal a, we assign a *level* $s(a)$.

Comment. For finite programs, we do not need *infinite* ordinal numbers; since finite ordinal numbers are exactly natural numbers $0, 1, 2, 3, \ldots$, a reader who does not feel comfortable with general ordinal numbers can use natural numbers instead. In this case, instead of a *transfinite* recursion, i.e., recursion over ordinal numbers, the reader can substitute normal recursion, i.e., recursion over natural numbers.

Answer sets for split logic programs: intuitive description. Intuitively, the existence of a splitting sequence means that rules that define literals from level α only use literals from this and lower levels. Thus, e.g., rules that define literals of level 0 only use literals from the same level and thus, these rules are self-sufficient to define which of the literals of level 0 are true and which are not.

As soon as we have defined the truth values of all literals of level 0, we can define the truth values of literals of level 1, etc.

Answer sets for split logic programs: a formal description. Formally, this process corresponds to *transfinite recursion*, i.e., recursion over all possible levels (in our case, over all ordinal numbers $< \mu$; if μ is finite, this becomes a simple recursion):

- First, we take all the rules whose heads are of level 0. By definition of a splitting, all conditions from these rules are also of level 0. Then, we find an answer set A_0 for the the corresponding logic program.
- If for some level α, we have already described the answer sets A_ν for all levels $\nu < \alpha$, then we can define the answer set A_α corresponding to this level α as follows:
  - consider the union $A_{<\alpha}$ of all already defined sets A_ν;
  - select all the rules whose heads contain only literals of levels α and lower;
  - delete all the rules in which one of conclusions in the head is a literal from the set $A_{<\alpha}$ (because these rules are automatically true);
  - if one of the conclusions of a rule is a literal of level $< \alpha$ that does not belong to $A_{<\alpha}$, we delete this literal from the conclusion (because this literal cannot be true);
  - delete all the rules in which one of the conditions is *not* p for some literal p included in $A_{<\alpha}$ (intuitively, since $p \in A_{<\alpha}$, the condition p is true and thus, the opposite condition *not* p is not satisfied);
  - delete all the rules in which one of the conditions is p for some literal p of level $< \alpha$ that is not included in $A_{<\alpha}$ (intuitively, since $p \notin A_{<\alpha}$, the condition p is not true and thus, the rule is not applicable);
  - from each of the remaining rules, delete all conditions p and *not* p that are literals of levels $< \alpha$ (after our previous deletions, all these conditions are automatically true);

  As a result, we get a logic program which only contains literals of level α. If this logic program has an answer set, we add all literals from this answer set to $A_{<\alpha}$ and get A_α. (If this logic program turns out to be empty, we take simply $A_{<\alpha}$ as A_α.)

As a result of this procedure, we get a set A_μ.

The main result about split logic programs. The main theorem from [11] consists of the following two statements:

- if this set A_μ is consistent, then it is an answer set of the original logic program;
- vice versa, every consistent answer set A to the original logic program can be obtained by this transfinite recursive procedure.

Parallelization. Splitting helps to compute the answer sets on a *sequential* computer, but it does not help much for *parallel* computers, because splitting means that we cannot process the next level until we are done with the previous one. A natural parallelization occurs if, instead of linearly ordered levels, we only have *partial* order. Then, it is natural to handle unrelated levels in parallel. Let us give a simple example: we have a bottom layer, with the rules

$$a \leftarrow not\ b.\ \ b \leftarrow not\ a.$$

the first layer, with rules

$$c \leftarrow a.\ \ c \leftarrow b.$$

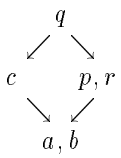
the second layer, with rules

$$p \leftarrow not\ r.\ \ r \leftarrow a, b.$$

and the final layer:

$$q \leftarrow c, p.$$

In this case, we have a *partial* order:



It is, therefore, desirable to *generalize* the result from [11] to logic programs for which the splitting map maps literals into a *partially* ordered set.

This generalization can be obtained by a simple modification of the original proof. As above, for every two literals a and b from a logic program, we use the denotation $a \geq b$ to indicate that in one of the rules, a appears in the head (i.e., in the conclusion part of the rule), and b in the body of this rule.

Let us recall that a (partially) ordered set M is called *well-ordered* if it does not have an infinite monotonically decreasing sequence $m_1 > m_2 > \ldots > m_n > \ldots$

Definition. *By a generalized splitting of a logic program, we mean a mapping s from the set of all literals into a well-ordered set (not necessarily linearly ordered) for which $a \geq b$ implies $s(a) \geq s(b)$.*

For a program that allows a generalized splitting, we can, almost literally, repeat the above construction, and get the following result:

Theorem. *If a logic program allows a generalized splitting, then:*

- *if a set A_μ obtained by the above-described transfinite recursion is consistent, then it is an answer set of the original logic program;*
- *vice versa, every consistent answer set A to the original logic program can be obtained by the above transfinite recursive procedure.*

Proof. There are two main possibilities to prove this result:

- One possibility is to simply repeat the proof from [11].
- Another possibility is to take into consideration the fact that every well-ordering can be extended to a *linear* well ordering and therefore, we can apply the original theorem from [11] to prove our result. From the recursive construction, it easily follows that if in the original ordering, levels α and β were unrelated by the ordering relation, then the corresponding reduced logic programs on stages α and β do not depend on each other.

Acknowledgments. This work was supported in part by NASA under cooperative agreement NCC5-209, by AFOSR under contracts F49620-93-1-0152 and F49620-95-1-0518, by NSF grants No. CCR-9058453, CDA-9015006, CDA-9522207, and DUE-9750858, by the United Space Alliance grant No. NAS 9-20000 (PWO C0C67713A) by the ARO under grant number ARO-145-91, and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518.

The authors are thankful to Michael Gelfond for valuable discussions.

References

[1] C. Baral, O. Kosheleva and M. Gelfond, "Expanding queries to incomplete databases by interpolating general logic programs", *Journal of Logic Programming*, 1998, Vol. 35, pp. 195–230.

[2] J. Binney, "Stellar dynamics", in: I. Appenzeller, H. J. Habing, and P. Léna (eds.), *Evolution of galaxies: astronomical observations*, Springer Lecture Notes in Physics, Vol. 333, Berlin, Heidelberg, 1989, pp. 95–146.

[3] A. Finkelstein, O. Kosheleva, and V. Kreinovich, "Astrogeometry, error estimation, and other applications of set-valued analysis", *ACM SIGNUM Newsletter*, 1996, Vol. 31, No. 4, pp. 3–25.

[4] A. Finkelstein, O. Kosheleva, and V. Kreinovich, "Astrogeometry: towards mathematical foundations", *International Journal of Theoretical Physics*, 1997, Vol. 36, No. 4, pp. 1009–1020.

[5] A. Finkelstein, O. Kosheleva, and V. Kreinovich, "Astrogeometry: geometry explains shapes of celestial bodies", *Geombinatorics*, 1997, Vol. VI, No. 4, pp. 125–139.

[6] B. H. Friesen and V. Kreinovich, "Ockham's razor in interval identification", *Reliable Computing*, 1995, Vol. 1, No. 3, pp. 225–238.

[7] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming" In: R. Kowalski and K. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, Seattle, Washington, August 15–19, 1988, pp. 1070–1080.

[8] O. Kosheleva, "An arbitrary first order theory can be represented by a logic program: a theorem", *Proceedings of the NASA University Research Centers Conference*, Albuquerque, New Mexico, February 16–19, 1997, pp. 431–436.

[9] O. Kosheleva, V. Kreinovich, and H. T. Nguyen, "Mamdani's Rule: a "weird" use of "and" as implication justified by modern logic", *Sixth International Fuzzy Systems Association World Congress*, San Paulo, Brazil, July 22–28, 1995, Vol. 1, pp. 229–232.

[10] V. Kreinovich, G. C. Mouzouris, and H. T. Nguyen, "Fuzzy rule based modeling as a universal control tool", In: H. T. Nguyen and M. Sugeno (eds.), *Fuzzy Systems: Modeling and Control*, Kluwer, Boston, MA, 1998, pp. 135–195.

[11] V. Lifschitz and H. Turner, "Splitting a Logic Program", In: Pascal van Hentenryck (ed.), *Logic Programming. Proceedings of the Eleventh International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1994, pp. 23–37.

[12] R. D. Luce and H. Raifa, *Games and decisions*, Dover, N.Y., 1989.

[13] J. McCarthy, *Formalizing common sense*, Ablex, Norwood, NJ, 1989.

[14] H. T. Nguyen, O. M. Kosheleva, and V. Kreinovich, "Is the success of fuzzy logic really paradoxical? Or: Towards the actual logic behind expert systems", *International Journal of Intelligent Systems*, 1996, Vol. 11, No. 5, pp. 295–326.

[15] H. T. Nguyen and V. Kreinovich, "Using Gelfond-Przymusinska's Epistemic Specifications to Justify (Some) Heuristic Methods Used in Expert Systems and Intelligent Control", *Soft Computing*, 1997, Vol. 1, No. 4, pp. 198–209.

[16] H. T. Nguyen, V. Kreinovich, and B. Bouchon-Meunier, "Soft Computing Explains Heuristic Numerical Methods in Data Processing and in Logic Programming", *Working Notes of the AAAI Symposium on Frontiers in Soft Computing and Decision Systems*, Boston, MA, November 8–10, 1997, pp. 40–45.

[17] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[18] R. Osegueda, C. Ferregut, M. J. George, J. M. Gutierrez, and V. Kreinovich, "Non-Equilibrium Thermodynamics Explains Semiotic Shapes: Applications to Astronomy and to Non-Destructive Testing of Aerospace Systems", *Proceedings of the International Conference on Intelligent Systems and Semiotics (ISAS'97)*, National Institute of Standards and Technology Publ., Gaithersburg, MD, 1997, pp. 378–382.

[19] R. Osegueda, C. Ferregut, M. J. George, J. M. Gutierrez, and V. Kreinovich, "Computational geometry and artificial neural networks: a hybrid approach to optimal sensor placement for aerospace NDE", In: C. Ferregut, R. Osegueda, and A. Nuñez (eds.), *Proceedings of the International Workshop on Intelligent NDE Sciences for Aging and Futuristic Aircraft*, El Paso, TX, September 30–October 2, 1997, pp. 59–71.

[20] R. Osegueda, C. Ferregut, M. J. George, J. M. Gutierrez, and V. Kreinovich, "Maximum entropy approach to optimal sensor placement for aerospace nondestructive testing", In: G. Erickson (ed.), *Maximum Entropy and Bayesian Methods*, Kluwer, Dordrecht, 1998 (to appear).

[21] L. J. Savage, *The foundations of statistics,* Wiley, N.Y., 1954.

[22] S. E. Strom and K. M. Strom, "The evolution of disk galaxies", *Scientific American*, April 1979; reprinted in P. W. Hodge (ed.), *The Universe of galaxies*, Freeman and Co., N.Y., 1984, pp. 44–54.

[23] A. Toomre and J. Toomre, "Violent tides between galaxies", *Scientific American*, December 1973; reprinted in P. W. Hodge (ed.), *The Universe of galaxies*, Freeman and Co., N.Y., 1984, pp. 55–65.

[24] A. A. Vakulenko and V. Kreinovich. "Physico-geometrical investigation of brittle fracture during creep," *Journal of Applied Mathematics and Mechanics*, 1989, Vol. 53, No. 5, pp. 660–665.

[25] B. A. Vorontsov-Veliaminov, *Extragalactic astronomy*, Harwood Academic Publishers, Chur, Switzerland, London, 1987.

[26] Ya. B. Zeldovich and I. D. Novikov, *Relativistic Astrophysics. Part 2. The structure and evolution of the Universe*, The University of Chicago Press, Chicago and London, 1983.