Compiling Fuzzy Answer Set Programs to Fuzzy Propositional Theories

Jeroen Janssen[1]*, Stijn Heymans[2]**, Dirk Vermeir[1], and Martine De Cock[2]

[1] Dept. of Computer Science, Vrije Universiteit Brussel
{jeroen.janssen,dvermeir}@vub.ac.be
[2] Dept. of Applied Mathematics and Computer Science, Universiteit Gent
{stijn.heymans,martine.decock}@ugent.be

Abstract. We show how a fuzzy answer set program can be compiled to an equivalent fuzzy propositional theory whose models correspond to the answer sets of the program. This creates a basis for constructing fuzzy answer set solvers, such as solvers based on fuzzy SAT-solvers or on linear programming.

Keywords: answer set programming, fuzzy logic, Clark's completion, fuzzy ASSAT.

1 Introduction

Fuzzy answer set programming (FASP, see e.g. [1,2,3]) is a form of many-valued logic programming (see e.g. [4,5,6]) that extends answer set programming (ASP) to handle vague predicates, partial satisfaction of rules, and, in the case of [3], the notion of quality of an answer set, i.e. a solution may be an answer set to a certain degree. This makes it possible to provide approximate answers, e.g. for problems that do not have a perfect solution. For many application areas, this is a desirable feature.

As an example, consider the problem of arranging a group of people such that friends are seated close to each other. Clearly, the input predicate *friend* is vague, with $friend(a, b)$ indicating the degree of friendship between a and b, e.g. on a scale from 0 to 1. Likewise, the second input predicate, *near*, is also vague, with $near(s, z)$ representing the proximity between the seats s and z.

The following (ungrounded)[3] FASP program P_{intro} defines (and solves) the problem where "$\leftarrow$" and "," are interpreted as the indicated fuzzy implicator and t-norm respectively, and 0 and 1 stand for the minimal ("false") and maximal

* Funded by Research Foundation–Flanders
** Funded by Research Foundation–Flanders
[3] Grounding is performed as usual, except that for input predicates, the actual value of the literal is substituted, e.g. $near(s, z)$ might be replaced by .7.

("true") truth value, see Section 2.

$$\begin{array}{rrl}
(\mathit{choice}) & \mathit{sit}(P,S) \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{1} \\
(c_1) & \mathit{0} \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{sit}(P,S), \mathit{sit}(P',S), P \neq P' \\
(c_2) & \mathit{0} \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{sit}(P,S), \mathit{sit}(P,S'), S \neq S' \\
(\mathit{crisp}) & \mathit{0} \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{sit}(P,S), \mathit{not}_{\sim^s} \mathit{sit}(P,S) \\
(u) & \mathit{unhappy}(P) \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{sit}(P,S), \mathit{sit}(P',S'), \mathit{friend}(P,P'), \mathit{not}_{\sim^s} \mathit{near}(S,S') \\
(q) & \mathit{0} \leftarrow_{\rightsquigarrow^l,\curlywedge^l} & \mathit{unhappy}(P) \\
(\mathit{sit}) & \mathit{seated}(P) \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{sit}(P,S) \\
(\mathit{all}) & \mathit{0} \leftarrow_{\rightsquigarrow^m,\curlywedge^m} & \mathit{not}\ \mathit{seated}(P)
\end{array}$$

The following aggregator specifies the quality of the solution[4] as a monotonic function on the degrees of satisfaction of the rules:

$$\mathcal{A}_{P_{\mathit{intro}}} = ((c_1 \curlywedge^m c_2 \curlywedge^m \mathit{crisp} \curlywedge^m \mathit{sit} \curlywedge^m \mathit{all} \curlywedge^m u) \geq 1) \curlywedge^m q$$

The (choice) rules generate a seating arrangement which is completely arbitrary, since the degree of satisfaction of the (choice) rules does not influence $\mathcal{A}_{P_{\mathit{intro}}}$. Thus, an instantiation

$$(\mathit{choice}\langle p,s\rangle) \quad \mathit{sit}(p,s) \leftarrow_{\rightsquigarrow^m,\curlywedge^m} 1$$

of (choice) with p a person and s a seat, can be used to motivate any literal $\mathit{sit}(p,s)^l$ with l an arbitrary truth value[5], hence the name for these rules. Indeed, having $\mathit{sit}(p,s)^l$ with $l \in [0..1[$ implies that ($\mathit{choice}\langle p,s\rangle$) is only satisfied to degree l, but this has no impact on the value of $\mathcal{A}_{P_{\mathit{intro}}}$, which is independent of the degrees of satisfaction of the (choice) rule instantiations.

However, the constraints (c_1), (c_2), (crisp) and (all), which $\mathcal{A}_{P_{\mathit{intro}}}$ forces to be fully satisfied, ensure that only arrangements where each person fully occupies exactly one seat can appear in an answer set. Why particular operators are being used for specific rules is explained in Section 3. The overall quality of the solution is represented by the degree of satisfaction of the (q) constraint which itself depends on the vague output predicate *unhappy* defined by the (u) rules. Thus, friends that sit far apart will weaken the satisfaction of a (q) rule and hence give rise to an answer set with a lower aggregated value.

As an example, consider a case where there are three available seats s_1, s_2, and z, only two of which are relatively near to each other, namely $\mathit{near}(s_1,s_2)^{.8}$, and three people connected by friendship with $\mathit{friend}(a,b)^{.8}$ and $\mathit{friend}(a,c)^{.5}$. Obviously, there exists no perfect arrangement that puts a close to both of her friends. However, an arrangement such as

$$I_1 = \{\, \mathit{sit}(a,s_1)^1, \mathit{sit}(b,s_2)^1, \mathit{sit}(c,z)^1, \mathit{unhappy}(a)^{.5}, \dots \} \qquad (1)$$

[4] In the grounded version, each ungrounded rule (r) in $\mathcal{A}_{P_{\mathit{intro}}}$ is replaced by $\bigsqcap\{r_i | r_i \in r\}$ where r represents the set of grounded instances of the rule.

[5] We use l^u to denote that a literal l holds to degree u. The default value for any atom is 0.

is still better than e.g. the arrangement

$$I_2 = \{\, sit(a, s_1)^1, sit(b, z)^1, sit(c, s_2)^1, unhappy(a)^{.8}, \ldots \} \qquad (2)$$

In accordance with this intuition, I_1 yields a .5-answer set of the FASP program P_{intro}, while I_2 corresponds to a .2-answer set only, as we explain in Section 3.

In this paper, we make an important contribution towards the implementation of a fuzzy answer set solver. In particular, we show how FASP programs (in the sense of [3], but this can be readily adapted to other approaches such as e.g. [1,2] as these can be translated to the FASP framework used in this paper) can be implemented by translating them to an equivalent formula in a fuzzy propositional logic, such that the answer sets of the program correspond to models of the formula. The latter can be computed using a fuzzy satisfiability (FSAT) solver or, subject to restrictions on the choice of connectives used in the program, by translating the formula, e.g. using tableau methods such as proposed in [7,8,9], to a linear programming problem that can itself be solved using standard tools.

The remainder of the paper is organized as follows. Section 2 contains preliminaries, while Section 3 introduces the fuzzy answer set programming [3] formalism being used. A fuzzy propositional logic framework is presented in Section 4. In Section 5, we extend to FASP programs the well-known translation of regular logic programs to a propositional theory called "Clark's completion" [10]. This translation forms the basis of many algorithms for finding answer sets such as those based on linear programming [11] or those using SAT-solvers [12]. We show that, under certain conditions, the models of our fuzzy completion and the fuzzy answer sets of a program coincide.

However, as in the boolean case, not every model of the fuzzy completion is an answer set. In Section 6, we remedy the situation by adding "loop formulas" to the completion, thus extending a similar approach for traditional answer set programs from [12]. We also show that the procedure proposed by [12] to iteratively compute such loop formulas "on demand" can be extended to fuzzy answer set programs. Finally, section 7 presents conclusions and directions for further research.

Due to space restrictions, all proofs have been omitted. They can be obtained from the full paper at
`http://tinf2.vub.ac.be/~jeroen/papers/ICLP08/ICLP08-full.pdf`

2 Preliminaries

The traditional logical operations of negation, conjunction, disjunction, and implication are generalised to logical operators acting on $[0, 1]$ in the usual way (see e.g. [13]). A *negator* is any anti-monotone $[0, 1] \to [0, 1]$ mapping $\sim$ satisfying $\sim 0 = 1$ and $\sim 1 = 0$. A negator $\sim$ is called *involutive* iff $\forall x \in [0, 1] \cdot \sim\sim x = x$. A *triangular norm*, t-norm for short, is any commutative and associative $[0, 1]^2 \to [0, 1]$ (infix) operator $\curlywedge$ satisfying $\forall x \in [0, 1] \cdot 1 \curlywedge x = x$.

Moreover we require $\curlywedge$ to be increasing in both of its arguments, i.e. [6] for $x_1, x_2 \in [0,1]$, $x_1 \leq x_2$ implies $x_1 \curlywedge y \leq x_2 \curlywedge y$. Intuitively, a t-norm corresponds to conjunction. In this paper, we restrict ourselves to continuous t-norms. As the most often used t-norms are continuous, this is not a burdensome restriction.

An *implicator* $\leadsto$ is any $[0,1]^2 \to [0,1]$ (infix) operator $\leadsto$ satisfying $0 \leadsto 0 = 1$, and $\forall x \in [0,1] \cdot 1 \leadsto x = x$. Moreover $\leadsto$ must be decreasing in its first, and increasing in its second argument. Every t-norm $\curlywedge$ induces a *residual implicator* defined by $x \leadsto y = \sup\{ \lambda \in [0,1] \mid x \curlywedge \lambda \leq y \}$. When the partial mappings of a t-norm $\curlywedge$ are supmorphisms[7], then $\curlywedge$ and its residual implicator $\leadsto$ satisfy the *residual property*, i.e. $\forall x, y, z \in [0,1] \cdot x \curlywedge y \leq z \equiv x \leq y \leadsto z$. Throughout this paper we only consider such residual pairs.

Well-known fuzzy logical operators on $[0,1]$ include the *minimum t-norm* $x \curlywedge^m y = \min(x,y)$, its residual implicator (also known as the "Gödel implicator") $x \leadsto^m y = 1$ if $x \leq y$, and $x \leadsto^m y = y$ otherwise, the *Lukasiewicz t-norm* $x \curlywedge^l y = \max(x+y-1, 0)$, and its residual implicator $x \leadsto^l y = \min(1-x+y, 1)$. For negation, often the *standard negator* $\sim^s x = 1 - x$ is used.

Fuzzy equivalence is denoted as $\approx$ and defined as $a \approx b = (a \leadsto b) \curlywedge (b \leadsto a)$, where $\curlywedge$ is a t-norm and $\leadsto$ its residual implicator. If we want to denote the use of a specific t-norm together with its residual implicator, we do so by superscripting the $\approx$-symbol as in $\approx^m = (a \leadsto^m b) \curlywedge^m (b \leadsto^m a)$.

An *fuzzy set* A over some (ordinary) set X is an $X \to [0,1]$ mapping. For x in X, $A(x)$ is called the *membership degree* of x in A. We also use $\mathcal{F}(X)$ to denote the set of all fuzzy sets over X. The *support* of a fuzzy set A is defined by $supp(A) = \{ x \mid A(x) > 0 \}$. Fuzzy set inclusion is also defined as usual by $A \subseteq B$, iff $\forall x \in X \cdot A(x) \leq B(x)$. Fuzzy set intersection (union) is defined by $(A \cap B)(x) = A(x) \sqcap B(x)$ ($(A \cup B)(x) = A(x) \sqcup B(x)$). This is extended to sets of fuzzy sets in the usual way, i.e. $\bigcap\{ A_1, \ldots, A_n \} = A_1 \cap \ldots \cap A_n$ and $\bigcup\{ A_1, \ldots, A_n \} = A_1 \cup \ldots \cup A_n$. Lastly the fuzzy set difference we will be using in this paper is $(A \setminus B)(x) = |A(x) - B(x)|$.

3 Fuzzy Answer Set Programming

Fuzzy answer set programming [3] is an extension of regular answer set programming (see e.g. [14]), a declarative formalism based on the stable model semantics for logic programming [15].

Definition 1 (FASP program). . *A* literal[8] *is an atom a or a constant from* $[0,1]$. *An* extended literal *is either an atom or of the form $not_{\sim} a$, with a an atom and $\sim$ a negator, representing negation as failure (naf). A* rule *r is of the form*

$$a \leftarrow_{\leadsto, \curlywedge} b_1, \ldots, b_n, not_{\sim_1} c_1, \ldots, not_{\sim_m} c_m$$

[6] Note that the monotonicity of the second component immediately follows from that of the first component due to the commutativity.

[7] The partial mappings of a t-norm $\curlywedge$ are called supmorphisms when for an arbitrary index set J it holds that $\sup\{ x_i \curlywedge y | i \in J \} = \sup\{ x_i | i \in J \} \curlywedge y$.

[8] As usual, we will assume that programs have already been grounded.

where $n \geq 0$, $m \geq 0$ *and* a, $\{\, b_i \mid 1 \leq i \leq n \,\}$, *and* $\{\, c_j \mid 1 \leq j \leq m \,\}$ *are (sets of) literals;* $\sim_1, \ldots, \sim_m$ *are negators and* $\leadsto$ *and* $\curlywedge$ *are resp. a residual implicator and a t-norm. The literal* a *is called the* head*, denoted* r_h *of the rule* r*, while* $\{\, b_1 \ldots, b_n, not\ c_1, \ldots, not\ c_m \,\}$ *is called the* body r_b *of* r*. We use* $Lit(r_b)$ *to denote the set of regular literals* $\{\, b_1, \ldots, b_n \,\}$ *from* r_b*. A* constraint *is a rule* r *where* $r_h \in [0, 1]$*.*

For a rule r*, we use* $\curlywedge_{r_b}$ *and* $\leadsto_r$ *to denote the rule's t-norm* $\curlywedge$*, and implicator* $\leadsto$*, respectively. We also use* $\curlywedge_r$ *to denote the t-norm of which* $\leadsto_r$ *is the residual implicator.*

A (FASP) program P *is a finite set of rules. The set of all literals that occur in* P *is called the* Herbrand Base $\mathcal{B}_P$ *of* P*. The set of all rules in* P *that have the literal* l *in the head is denoted as* P_l*.*

A rule-interpretation *is a function* $\rho : P \to [0, 1]$ *that associates a degree of satisfaction* $\rho(r)$ *to each rule* $r \in P$*. With every FASP program, the programmer must define a monotonic* aggregator *function* $\mathcal{A}_P : (P \to [0, 1]) \to [0, 1]$*, which aggregates the values of all rules into a single degree of rule satisfaction for the program.*

Definition 2 (Interpretation of a FASP program). *Let* P *be a FASP program. An* interpretation *of* P *is any fuzzy set* $I \in \mathcal{F}(\mathcal{B}_P)$*. Interpretations are extended to constants, extended literals and rules in a straightforward way: for a constant* $c \in [0, 1]$*, define* $I(c) = c$*. For extended literals, we define* $I(not_{\sim} a) = \sim I(a)$*. For a rule* $r = a \leftarrow_{\leadsto, \curlywedge} b_1, \ldots, b_n, not_{\sim_1} c_1, \ldots, not_{\sim_m} c_m$*, the extension to the rule body* r_b *is defined as:* $I(r_b) = I(b_1) \curlywedge \ldots \curlywedge I(b_n) \curlywedge I(not_{\sim_1} c_1) \curlywedge \ldots \curlywedge I(not_{\sim_m} c_m)$*, yielding the degree of satisfaction of* r *as* $I(r) = I(r_b) \leadsto I(r_h)$*.*

For every interpretation I *there is a corresponding rule interpretation* I_ρ*, defined by* $I_\rho(r) = I(r)$ *for all* r *from* P*.*

Example 1. Consider a grounded version of the program P_{intro} from Section 1, with the seat constants, person constants, the *near* and *friend* predicates as given in the introduction, and the interpretations I_1 and I_2 as given in (1)-(2). Interpretation I_1 satisfies the constraint $0 \leftarrow_{\leadsto^l, \curlywedge^l} unhappy(a)$ to degree $I_1(unhappy(a)) \leadsto^l 0 = \min(1 - .5 + 0, 1) = .5$, while interpretation I_2 satisfies this constraint only to degree .2. Note that the choice of the Łukasiewicz implicator $\leadsto^l$ in this constraint is crucial to preserve the gradual character of the vague *unhappy* predicate in the rule satisfaction. Using the Gödel implicator $\leadsto^m$ e.g. would force this rule to be evaluated in a crisp way (either the rule is fully satisfied or it is not satisfied at all), hence loosing the nuance.

In the (*crisp*) rules on the contrary, the choice for the residual pair $\curlywedge^m$ and $\leadsto^m$ allows to enforce that a given person either sits on a given seat or not. Indeed, a constraint like $0 \leftarrow_{\leadsto^m, \curlywedge^m} sit(a, s_1), not_{\sim^s} sit(a, s_1)$ is only satisfied to degree 1 when the rule body is satisfied to degree 0. Since the minimum t-norm does not have zero divisors, this situation only occurs when either $I(sit(a, s_1)) = 0$, i.e. a does not sit on seat s_1, or when $I(not_{\sim^s} sit(a, s_1)) = 0$, i.e. $I(sit(a, s_1)) = 1$, in other words a sits on seat s_1.

Residual implicators adhere to the property that $x \rightsquigarrow y = 1$ iff $x \leq y$. In other words, according to Definition 2, an interpretation fully satisfies a rule whenever it satisfies the head at least as much as the body. The interpretation

$$I_3 = \{\, \mathit{sit}(a, s_1)^1, \mathit{sit}(b, s_2)^1, \mathit{sit}(c, z)^1, \mathit{unhappy}(a)^{.9}, \ldots \} \tag{3}$$

fully satisfies the rule

$$\mathit{unhappy}(a) \leftarrow_{\rightsquigarrow^m, \curlywedge^m} \mathit{sit}(a, s_1), \mathit{sit}(c, z), \mathit{friend}(a, c), \mathit{not}_{\sim^s} \mathit{near}(s_1, z)$$

since $.9 \geq \min(1, 1, .5, 1)$. However, assigning .5 to *unhappy*(a) would already be sufficient to fully satisfy the rule; in other words the desire to fully satisfy this rule does not provide sufficient justification to assign to *unhappy*(a) a degree higher than .5. To ensure that we only derive a minimal knowledge set from our programs, we use the so called "support" of an interpretation with respect to a given rule and relative to a given rule interpretation. Intuitively, this is the lowest possible value that can be assigned to the head of the rule such that the rule is satisfied to at least the degree that is required by the given rule interpretation.

Definition 3 (Support). *Let P be a FASP program. We define the* support *of an interpretation I of P with respect to the rule $r \in P$ and relative to the rule interpretation ρ of P as:*

$$I_s(r, \rho) = \inf\{\, k \in [0, 1] \mid I(r_b) \rightsquigarrow_r k \geq \rho(r) \,\}$$

We abbreviate $I_s(r, I_\rho)$ as $I_s(r)$.

Theorem 1. *Let P be a FASP program. For any interpretation I of P, rule $r \in P$, and rule interpretation ρ of P the following holds:*

$$I_s(r, \rho) = I(r_b) \curlywedge_r \rho(r)$$

For $\rho = I_\rho$ we have the following result:

$$I_s(r) = I(r_b) \sqcap I(r_h)$$

The definition of fuzzy answer sets relies on the notion of *unfounded sets*, which, intuitively, are sets of "assumption" literals that have no proper motivation from the program.

Definition 4 (Unfounded-free interpretation). *Let I be an interpretation of a program P. A set $Y \subseteq \mathcal{B}_P$ is called* unfounded *w.r.t. I iff for each literal $l \in Y$ and rule $r \in P_l$ it holds that either:*

- $Y \cap \mathit{Lit}(r_b) \neq \emptyset$ *or*
- $I(l) > I_s(r)$ *or*
- $I(r_b) = 0$

An interpretation I of P is unfounded-free *iff $\mathit{supp}(I) \cap Y = \emptyset$ for any unfounded set Y w.r.t. I.*

The first condition in Definition 4 prevents circular motivation between assumptions. The second condition prohibits assumptions motivated by rules that are not applied conservatively, i.e. a rule r is used to motivate a truth value of the head in excess of the support that is actually available (from $I_s(r)$). The third condition finally helps to ensure that Definition 4 is a proper generalization of the classical definition of unfounded sets [16].

Example 2. Consider an interpretation $I = \{ a^{0.5}, b^{0.5} \}$ for program P_2, defined below.

$$r_1 : a \leftarrow_{\rightsquigarrow^m, \curlywedge^m} b$$
$$r_2 : b \leftarrow_{\rightsquigarrow^m, \curlywedge^m} a$$

As there is no rule supporting the fact that $I(a) = 0.5$ or $I(b) = 0.5$, this interpretation contains more knowledge than what is inferable from the program and is therefore unwanted. In fact e.g. $Y = \{ a, b \}$ is an unfounded set because both $Y \cap \mathit{Lit}(r_1) \neq \emptyset$ and $Y \cap \mathit{Lit}(r_2) \neq \emptyset$. Since $\mathit{supp}(I) \cap Y \neq \emptyset$, I is not unfounded-free.

Answer sets of FASP programs are unfounded-free interpretations reflecting the intuition that each literal in an answer set should have a proper motivation in the program. Moreover, the rules of the program should be satisfied to a desired degree.

Definition 5 (y-answer set). *Let P be a FASP program and $y \in [0, 1]$. An interpretation I of P is called a y-answer set iff I is unfounded-free and $\mathcal{A}_P(I_\rho) \geq y$.*

Example 3. Consider program P_{intro} from Section 1 and its interpretations I_1, I_2, and I_3 as given in (1)–(3). The set $\{ \mathit{unhappy}(a) \}$ is unfounded w.r.t. I_3 as for each grounded instance r of the (u) rules with $\mathit{unhappy}(a)$ in the head, it holds that $I_3(\mathit{unhappy}(a)) > (I_3)_s(r)$. Hence I_3 is not unfounded-free. Interpretations I_1 and I_2 on the other hand are unfounded-free. Furthermore one can verify that $\mathcal{A}_{P_{\mathit{intro}}}((I_1)_\rho) \geq .5$ and $\mathcal{A}_{P_{\mathit{intro}}}((I_2)_\rho) \geq .2$, in other words I_1 and I_2 are resp. a .5-answer set and a .2-answer set of P_{intro}.

4 Fuzzy propositional logic

We build a fuzzy propositional logic starting from a set of t-norms $\{ \curlywedge_1, \ldots, \curlywedge_n \}$, their residual implicators $\{ \rightsquigarrow_1, \ldots, \rightsquigarrow_n \}$ and a set of negators $\{ \sim_1, \ldots, \sim_k \}$, all of which are defined over $[0, 1]$. Furthermore there is a set of variable symbols $\{ v_1, \ldots, v_l \}$. Further connectives are $\approx$, $\sqcup$ and $\sqcap$, where $\sqcup$ and $\sqcap$ are the infix supremum and infimum resp. and where $\approx$ is defined as $p \approx q = (p \rightsquigarrow q) \curlywedge (q \rightsquigarrow p)$, for $\curlywedge$ a t-norm and $\rightsquigarrow$ its residual implicator.

The syntax of this fuzzy propositional logic is defined as follows. A *proposition* is either a constant from $[0, 1]$, a variable, or an expression of one of the following forms, where p and q are propositions: $p \curlywedge_i q$, where $i \in 1 \ldots n$, $p \rightsquigarrow_i q$, where

$i \in 1 \ldots n$, $\sim_i p$, where $i \in 1 \ldots k$, $p \approx_i q$, where $i \in 1 \ldots n$, $p \sqcup q$, or $p \sqcap q$. A *theory* is a set of propositions.

The semantics of this logic is defined in a straightforward way. Let I be a fuzzy set over the variables of a proposition. Then I is inductively extended to propositions as follows: let p and q be fuzzy propositions and I an interpretation over the variables of p and q, then $I(l) = l$, where $l \in [0,1]$, $I(p \curlywedge q) = I(p) \curlywedge I(q)$, $I(p \rightsquigarrow q) = I(p) \rightsquigarrow I(q)$, $I(\sim p) = \sim I(p)$, $I(p \sqcup q) = I(p) \sqcup I(q)$ and $I(p \sqcap q) = I(p) \sqcap I(q)$. A fuzzy set over the variables of a proposition is then called an *interpretation* of this proposition.

We say that an interpretation I is a model of a theory P, whenever $\forall p \in P \cdot I(p) = 1$ and denote this as $I \models P$.

5 Fuzzy Completion

In this section we show how certain fuzzy answer set programs can be translated to fuzzy theories such that the models of these theories will be y-answer sets and vice versa.

Definition 6 (Fuzzy y-completion). *Let P be a FASP program with aggregator $\mathcal{A}_P$ and let $y \in [0,1]$. The* fuzzy completion *of the body of a rule $r \in P$, with $r = a \leftarrow_{\rightsquigarrow,\curlywedge} b_1, \ldots, b_n, not_{\sim_1} c_1, \ldots, not_{\sim_m} c_m$, is the propositional formula*

$$Comp(r_b) = b_1 \curlywedge_{r_b} \ldots \curlywedge_{r_b} b_n \curlywedge_{r_b} \sim_1 c_1 \curlywedge_{r_b} \ldots \curlywedge_{r_b} \sim_m c_m$$

The completion of the rule r is defined as:

$$Comp(r) = Comp(r_b) \sqcap r_h$$

Assume that the aggregator is representable as a fuzzy propositional formula, i.e. that a proposition $Comp_y(\mathcal{A}_P)$ exists such that for any interpretation I of P, $\mathcal{A}_P(I_\rho) \geq y$ iff $I \models Comp_y(\mathcal{A}_P)$. The fuzzy y-completion of the program P is then defined as:

$$Comp_y(P) = \{\, l \approx \bigsqcup \{\, Comp(r) \mid r \in P_l \,\} \mid l \in \mathcal{B}_P \,\} \cup Comp_y(\mathcal{A}_P)$$

for $\approx$ an arbitrary equivalence relation.

Note that the y in the completion is the same y we use for y-answer sets, thus the intention is that the models of the y-completion of a program will be the y-answer sets of the program.

Example 4. Consider the following program P:

$$r_1 : a \leftarrow_{\rightsquigarrow^m,\curlywedge^m} not_{\sim^s} b$$
$$r_2 : b \leftarrow_{\rightsquigarrow^m,\curlywedge^m} not_{\sim^s} a$$

with aggregator $\mathcal{A}_P(\rho) = \inf\{\, \rho(r) \mid r \in P \,\}$. The aggregator of this program is representable in fuzzy propositional logic as the formula $y \rightsquigarrow (\sim^s b \rightsquigarrow^m$

$a) \sqcap (\sim^s a \rightsquigarrow^m b)$. The completion of this program will then be the following fuzzy propositional theory:

$$\begin{aligned} a &\approx^m ((\sim^s b) \sqcap a) \\ b &\approx^m ((\sim^s a) \sqcap b) \\ y \rightsquigarrow^m (\sim^s b \rightsquigarrow^m &a) \sqcap (\sim^s a \rightsquigarrow^m b) \end{aligned}$$

It is easy to see that the interpretation $I = \{ a^{0.8}, b^{0.2} \}$ is a 1-answer set of this program and will also be a model of the completion $Comp_1(P)$.

Readers familiar with the completion in traditional logic programming may wonder why our completion uses $Comp(r_b) \sqcap r_h$ instead of the more usual $Comp(r_b)$ in the right-hand side of the equations. This is necessary in order to support the partial satisfaction of rules. Indeed, using $Comp(r_b)$ would force rules to be fully satisfied, while using $Comp(r_b) \sqcap r_h$ allows interpretations for which $I(r_h) < I(r_b)$, leading to $(I(r_b) \rightsquigarrow_r I(r_h)) < 1$, hence interpretations that only partially satisfy rules.

In the fuzzy y-completion of a program P, we do not introduce a separate proposition for literals l that do not appear in the head of any rule from P, since these will be subsumed by the introduction of $l \approx \bigsqcup \emptyset$ (with our choice of equivalence relations), which is equivalent to $l \approx 0$ by definition of $\bigsqcup$. No separate propositions are added for constraints, i.e. rules with a value from $[0, 1]$ in the head, either, since constraints are only used to determine the aggregated satisfaction value of the program and hence are only needed in the aggregator proposition.

Finally, the condition on aggregators to be representable in fuzzy propositional logic is necessary to solve programs using SAT-solvers. That this condition still allows for sufficient expressiveness is illustrated by the fact that the aggregator of the program in the introduction can be represented as

$$Comp_y(\mathcal{A}_{P_{intro}}) = [1 \rightsquigarrow^m (c_1 \curlywedge^m c_2 \curlywedge^m crisp \curlywedge^m sit \curlywedge^m all \curlywedge^m u)] \sqcap [y \rightsquigarrow^l q]$$

One can now show that any y-answer set of a program P is a model of its completion $Comp_y(P)$.

Theorem 2. *Let P be a FASP program. Then if the aggregator is representable in fuzzy propositional logic, any y-answer set of P is a model of $Comp_y(P)$.*

The reverse of Theorem 2 is not true in general, since it is already invalid for classical answer set programming. The problem is with the completion of programs that have "loops", as shown in the following example.

Example 5. Consider P_2 from Example 2. The y-completion of this program is:

$$\begin{gathered} a \approx b \sqcap a \\ b \approx a \sqcap b \\ y \rightsquigarrow^m (a \rightsquigarrow^m b) \sqcap (b \rightsquigarrow^m a) \end{gathered}$$

The interpretation $I = \{ a^1, b^1 \}$, is a model of $Comp_y(P_2)$, but it is not a y-answer set of P_2, as the only y-answer set (with $y > 0$) of this program is $\{ a^0, b^0 \}$.

As in the crisp case, when a program has no loops in its positive dependency graph however, the models of the y-completion and the y-answer sets do coincide. First we define what a loop of a logic program actually means and then we formally state that the aforementioned holds.

Definition 7 (Loop). *Let P be a FASP program. The* positive dependency graph *of P is then a directed graph $G_P = (\mathcal{B}_P, R)$ where $a \, R \, b \equiv \exists r \in P_a \cdot b \in Lit(r_b)$. We denote this relation also with $G_P(a, b)$ for any literals a and b in the Herbrand base of P. We call a non-empty set $L \subseteq \mathcal{B}_P$ a* loop *iff for all literals a and b in L there is a path (with length > 0) from a to b in G_P such that all vertices on this path are elements of L.*

Using this definition, one can easily see that the program from Example 2 contains the loop $L = \{ a, b \}$.

Theorem 3. *Let P be a FASP program. If P has no loops in its positive dependency graph and its aggregator is representable in fuzzy propositional logic, it holds that I is a y-answer set of P iff $I \models Comp_y(P)$.*

6 Solving the loop problem

As mentioned in the previous section, sometimes the models of the y-completion are not y-answer sets, which hinders the possibility of using the y-completion of a program to e.g. compute y-answer sets using a fuzzy satisfiability solver. In this section, we investigate how the solution for boolean answer set programming, which consists of adding loop formulas to the completion [12], can be extended to fuzzy answer set programs.

For this extension, we will start from a partition of the rules whose heads are in a loop, for a given loop L. Based upon this partition, we will then define a condition that must be fulfilled and can be expressed in fuzzy propositional logic, such that any model of the y-completion satisfying it, will no longer have the problem of attaching a value that is too high to atoms that occur in a loop.

For any program P and loop L we consider the following partition of rules with heads in the loop of P (due to [12]):

$$R_P^+(L) = \{ a \leftarrow B \mid (a \leftarrow B) \in P \wedge a \in L \wedge B \cap L \neq \emptyset \}$$
$$R_P^-(L) = \{ a \leftarrow B \mid (a \leftarrow B) \in P \wedge a \in L \wedge B \cap L = \emptyset \}$$

Intuitively, this means that $R_P^+(L)$ contains the rules that are “in” the loop L, i.e. that are responsible for the creation of the loop in the positive dependency graph, whereas the rules in $R_P^-(L)$ are the rules that are outside of this loop. We will refer to them as “loop rules”, resp. “non-loop rules”. Recalling the program from Example 2, the partitions of rules with respect to the loop $L = \{ a, b \}$ would be $R_P^+(L) = \{ a \leftarrow_{\rightsquigarrow^m, \curlywedge^m} b, b \leftarrow_{\rightsquigarrow^m, \curlywedge^m} a \}$ and $R_P^-(L) = \emptyset$.

All literals in the support of a y-answer set are derived using rules that are not contained in any loop. Therefore, like in [12], this motivates the use of "loop formulas" to eliminate any model of the completion in which the value of a literal is derived using only loop rules (or is higher than what the non-loop rules could conclude). Considering Example 2 once again, one can see that for the interpretation $I_0 = \{ a^1, b^1 \}$, the loop rules were used to attach the high values to a and b. The only interpretation that does not use the loop rules would be $I_1 = \{ a^0, b^0 \}$.

There is thus a problem when the values of literals in a loop are only supported by other literals in the loop. This is the case when their value is only supported by loop rules, as the support of these rules is by definition always based on literals in the loop. Hence to solve this problem, we should require that at least one non-loop rule supports the value of loop literals. Only one rule's support is needed as this support propagates through the loop.

Example 6. As an illustration of the above remark, consider program P_6.

$$\begin{aligned} r_1 &: a \leftarrow_{\rightsquigarrow^m, \curlywedge^m} 0.8 \\ r_2 &: a \leftarrow_{\rightsquigarrow^m, \curlywedge^m} b \\ r_3 &: b \leftarrow_{\rightsquigarrow^m, \curlywedge^m} a \end{aligned}$$

with aggregator $\mathcal{A}_P(\rho) = \inf\{ \rho(r) \mid r \in P_6 \}$.

There is a loop $L = \{ a, b \}$ in P_6, with loop sets $R_{P_6}^+(L) = \{ r_2, r_3 \}$ and $R_{P_6}^-(L) = \{ r_1 \}$. The interpretation $I = \{ a^1, b^1 \}$ is a model of $Comp_1(P_6)$ since $I \models a \approx (0.8 \sqcap a) \sqcup (a \sqcap b)$ as $(0.8 \sqcap I(a)) \sqcup (I(b) \sqcap I(a)) = 1$, $I \models b \approx b \sqcap a$ likewise and $I \models 1 \rightsquigarrow^m (0.8 \rightsquigarrow^m a) \sqcap (b \rightsquigarrow^m a) \sqcap (a \rightsquigarrow^m b)$ as $0.8 \rightsquigarrow^m I(a) = 1$ since $0.8 \leq I(a)$ and likewise $(I(b) \rightsquigarrow^m I(a)) = 1$ and $(I(a) \rightsquigarrow^m I(b)) = 1$. I is not a 1-answer set of P_6 however, as $L \cap \mathit{supp}(I) \neq \emptyset$ and L is unfounded due to $I(a) > 0.8$, $\mathit{Lit}(r_{2b}) \cap L \neq \emptyset$ and $\mathit{Lit}(r_{3b}) \cap L \neq \emptyset$. In other words, I has only used loop rules to determine the values of a and b.

The set $I' = \{ a^{0.8}, b^{0.8} \}$ is however a 1-answer set as the non-loop rule r_1 was used to derive the value of literal a. Since the value of b is derived from this non-loop-derived value of a, the use of the loop rule r_3 to determine the value of b then poses no problem.

Summing all of this up, the definition then becomes:

Definition 8 (Loop formula). *Let P be a FASP program and $L = \{ l_1, \ldots, l_m \}$ a loop in the positive dependency graph of P. Suppose that $R_P^-(L) = \{ r_1, \ldots, r_n \}$. Then the* loop formula *associated with the loop L, denoted by* $\mathrm{LF}(L, P)$*, is the following fuzzy proposition:*

$$l_1 \sqcup \ldots \sqcup l_m \rightsquigarrow Comp(r_1) \sqcup \ldots \sqcup Comp(r_n)$$

If $R_P^-(L) = \emptyset$, the loop formula becomes:

$$l_1 \sqcup \ldots \sqcup l_m \rightsquigarrow 0$$

The loop formula proposed for boolean answer set programs in [12] is of the form

$$\neg(\bigwedge B_{11} \vee \ldots \vee \bigwedge B_{1k_1} \vee \ldots \vee \bigwedge B_{n1} \vee \ldots \vee \bigwedge B_{nk_n}) \Rightarrow (\neg l_1 \wedge \ldots \wedge \neg l_m)$$

It can easily be seen that our loop formulas are a straightforward generalisation of this loop formula as the latter is equivalent to

$$(l_1 \vee \ldots \vee l_m) \Rightarrow (\bigwedge B_{11} \vee \ldots \vee \bigwedge B_{1k_1} \vee \ldots \vee \bigwedge B_{n1} \vee \ldots \vee \bigwedge B_{nk_n})$$

Furthermore, since $I \models l_1 \sqcup \ldots \sqcup l_m \rightsquigarrow 0$ only when $I(l_1) \sqcup \ldots \sqcup I(l_m) \leq 0$, it is easy to see that in the case where no rules exist outside of the loop, the maximum amount of knowledge we can derive from our program is that the literals in the loop are all "false" (0).

Example 7. Consider program P_2 from Example 2 again. There is a loop $L = \{ a, b \}$ in G_P with as loop formula $a \sqcup b \rightsquigarrow^m 0$, since the set $R_P^-(L) = \emptyset$. $I_0 = \{ a^1, b^1 \}$ is not a model of this formula, as $I_0(a) \sqcup I_0(b) \rightsquigarrow 0 = 1 \equiv I_0(a) \sqcup I_0(b) \leq 0$. Hence, only the interpretation $I_1 = \{ a^0, b^0 \}$ is a model of this loop formula, which is the intended behaviour.

Considering program P_6 from Example 6, we can see that the loop $L = \{ a, b \}$ has the loop formula $a \sqcup b \rightsquigarrow a \sqcap 0.8$. Since $I \models a \sqcup b \rightsquigarrow^m a \sqcap 0.8$ only if $a \leq 0.8$ and $b \leq a \sqcap 0.8$, interpretation I from Example 6 is eliminated as a model while I' is preserved.

We now show that by adding loop formulas to the completion of a program, we get a propositional theory that is both sound and complete with respect to the answer set semantics. First we show that this procedure is complete.

Theorem 4. *Let P be a FASP program and let $\mathrm{LF}(P)$ be the set of all loop formulas of P, i.e. the set of loop formulas for any loop L in P. Then for any interpretation I of P it holds that if I is a y-answer set of P, then $I \models \mathrm{LF}(P) \cup Comp_y(P)$.*

Secondly we show that it is sound.

Theorem 5. *Let P be a FASP program and $\mathrm{LF}(P)$ be the set of all loop formulas of P. Then for any interpretation I of P it holds that if $I \models \mathrm{LF}(P) \cup Comp_y(P)$, then I is a y-answer set of P.*

A straightforward procedure for finding answer sets would now be to extend the completion of a program with all possible loop formulas and let a fuzzy SAT solver generate models of the resulting propositional theory. The models of the propositional theory that we get this way will be y-answer sets of the program, as ensured by Theorems 4 and 5. This however has a potential drawback, as the amount of loops can grow exponentially. In [12] a procedure to overcome this problem was proposed, where loop formulas are added iteratively, when a model of the completion generated by a SAT-solver violates a loop formula. We will show that the same procedure can be used for finding fuzzy answer sets. For this, we need a characterization of answer sets in terms of the consequence operator.

Definition 9 (Consequence operator). *Let P be a FASP program and let ρ be a rule interpretation of P. The* consequence operator *of P and ρ is defined as follows:*

$$\Pi_{P,\rho} : (\mathcal{B}_P \rightarrow [0,1]) \rightarrow \mathcal{B}_P \rightarrow [0,1]$$
$$\Pi_{P,\rho}(I)(l) = \sup_{r \in P_l} I_s(r, \rho)$$

This operator is monotonic and thus has a least fixpoint [17], denoted as $lfp(\Pi_{P,\rho})$. Furthermore, a reduct is defined as follows:

Definition 10 (Reduct). *Let P be a FASP program. Then the* reduct *of a rule $r \in P$, where $r = a \leftarrow_{\rightsquigarrow,\curlywedge} b_1, \ldots, b_n, not_{\sim_1} c_1, \ldots, not_{\sim_m} c_m$, with respect to an interpretation I is denoted as r^I and defined as $r^I = a \leftarrow_{\rightsquigarrow,\curlywedge} b_1, \ldots, b_n, \sim_1 I(c_1), \ldots, \sim_m I(c_m)$. The reduct of a program P w.r.t. an interpretation I is denoted as P^I and defined as $P^I = \{ r^I \mid r \in P \}$.*

It can then be shown that a fixpoint characterisation exists for fuzzy answer sets, as follows:

Theorem 6. *Let P be a FASP program. Then I is a y-answer set of P iff $I = lfp(\Pi_{P^I, I_\rho})$ and $\mathcal{A}_P(I_\rho) \geq y$.*

Using this characterisation of fuzzy answer sets, we have a quick way of checking whether a model of the y-completion is a y-answer set. In case it is not a y-answer set, the following theorem shows us that this means there is at least one loop whose loop formula is violated. Furthermore, it identifies a set of literals that contains the loop, enabling us to reduce the search space for finding the loops.

Theorem 7. *Let P be a FASP program. If an interpretation I of P is a model of $Comp_y(P)$ and $I \neq lfp(\Pi_{P^I, I_\rho})$, then some $L \subseteq supp(I \setminus lfp(\Pi_{P^I, I_\rho}))$ must exist such that L is a loop and $I \not\models \mathrm{LF}(L, P)$.*

Now, we can extend the ASSAT-procedure proposed in [12] to fuzzy answer set programs. The main idea of this method is to use a fuzzy SAT-solver to find models of the fuzzy propositional theory constructed from the completion and the loop formulas of some maximal loops. If the model generated is not an answer set, then the loop that is violated is sought and added to the theory and the process is started again. The algorithm thus becomes:

1. Initialize $Loops = \emptyset$
2. Generate a model I of $Comp_y(P) \cup \mathrm{LF}(Loops, P)$, where $\mathrm{LF}(Loops, P)$ is the set of loop formulas of all loops in $Loops$.
3. If $I = lfp(\Pi_{P^I, I_\rho})$, return I as it is a y-answer set. Else, find the loops occurring in $supp(I \setminus lfp(\Pi_{P^I, I_\rho}))$, add them to $Loops$ and go to step 2.

As we only need to search for the loops of a subset of all literals due to Theorem 7, which only needs to be done when a model is generated that is not an answer set, this procedure does not need to add an exponential number of loop formulas at the start. Based on the experimental results in [12], we would expect a similar improvement when finding fuzzy answer sets using fuzzy SAT-solvers.

7 Conclusions and future work

We defined a fuzzy version of Clark's completion, creating a basis for different kinds of (fuzzy) answer set solvers. Furthermore, we defined loop formulas that ensure that the completion semantics coincide with the program semantics in the presence of loops in the positive dependency graph. We have also shown how, similar to the ASSAT procedure for answer set programs, loop formulas of fuzzy answer set programs can be computed "on the fly", thus avoiding a possibly exponential blow-up of the number of loop formulas to consider.

As algorithms for solving the fuzzy SAT problem, with restrictions on the operators used, have been developed [7,8,9], the results of this paper thus effectively create a basis for practical implementations of the FASP paradigm. This is enhanced by the possibility of iteratively adding loop formulas, as in the ASSAT procedure for crisp answer set programming.

In the future, we intend to investigate solving the completion proposition using a combination of a translation to linear programming and tableaux, as in [7,9], but with less restrictions on the operators. Related with this, we intend to investigate the possibilities in directly solving the program using mixed integer programming as in [11].

A first prototype FASPMIP[9] has already been developed It supports a simple concrete syntax to express a limited set of connectives and a restricted set of aggregator functions.

As an example, the source (not including the "data") for the program P_{intro} from Section 1 is shown below.

```
sit(P,S) :/ Person(P),Seat(S). % choice
:- sit(P,S),sit(PP,S), P /= PP.
:- sit(P,S),sit(P,SS), S /= SS.
:- sit(P,S),not sit(P,S). % crispify sit/2
unhappy(P) :- sit(P,S), sit(PP,SS), friend(P,PP), not near(S,SS).
:~ unhappy(P). % score
seated(P) :- sit(P,S). :- not seated(P),Person(P). % all seated
```

To compute the semantics of an input program, FASPMIP parses and grounds the rules in the usual way. Then the program is translated to a set of linear programming constraints corresponding to the y-completion of the program, see also [11]. The resulting linear programming model is then written to a file using the *MathProg* modeling language. The file serves as input for the LP/MIP *glpsol* solver[10], which computes a minimal y-answer set.

The output (for selected predicates) of FASPMIP for a 0.5-answer set of the program P_{intro} from Section 1 is shown below.

```
[(near(s1,s2),0.8), (friend(a,c),0.5),(friend(a,b),0.8),
 (unhappy(a),0.5),(sit(c,s3),1.0),(sit(b,s2),1.0), (sit(a,s1),1.0)]
```

[9] Available from `http://tinf2.vub.ac.be/faspsolver/faspmip-0.1.tar.gz`.

[10] *glpsol* is part of GLPK, the GNU Linear Programming Kit, see `http://www.gnu.org/software/glpk/glpk.html`.

References

1. Lukasiewicz, T.: Fuzzy description logic programs under the answer set semantics for the semantic web. In: Proceedings of the Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML'06), IEEE Computer Society (2006) 89–96
2. Lukasiewicz, T., Straccia, U.: Tightly integrated fuzzy description logic programs under the answer set semantics for the semantic web. In: Proceedings of the First International Conference on Web Reasoning and Rule Systems (RR'07), Springer-Verlag (2007) 289–298
3. Van Nieuwenborgh, D., De Cock, M., Vermeir, D.: An introduction to fuzzy answer set programming. Annals of Mathematics and Artificial Intelligence **50**(3-4) (2007) 363–388
4. Damásio, C., Medina, J., Ojeda-Aciego, M.: Sorted multi-adjoint logic programs: termination results and applications. In: Logics in Artificial Intelligence (JELIA'04), Springer-Verlag (2004) 260–273
5. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming **12**(3&4) (1992) 335–367
6. Straccia, U.: Annotated answer set programming. In: Proceedings of the 11th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, (IPMU'06). (2006)
7. Hähnle, R.: Many-valued logic and mixed integer programming. Annals of Mathematics and Artificial Intelligence **12**(3-4) (1994) 231–263
8. Lepock, C., Pelletier, F.J.: Fregean algebraic tableaux: Automating inferences in fuzzy propositional logic. In: Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05), Springer-Verlag (2005) 43–48
9. Straccia, U.: Reasoning and experimenting within Zadeh's fuzzy propositional logic. Technical report, Paris, France (2000)
10. Clark, K.L.: Negation as failure. In: Logic and Databases, Plenum Press, New York (1978) 293–322
11. Bell, C., Nerode, A., Ng, R.T., Subrahmanian, V.S.: Mixed integer programming methods for computing nonmonotonic deductive databases. Journal of the ACM **41**(6) (1994) 1178–1215
12. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by sat solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
13. Novák, V., Perfilieva, I., Močkoř, J.: Mathematical Principles of Fuzzy Logic. Kluwer Academic Publishers (1999)
14. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP/SLP'88), ALP, IEEE, The MIT Press (1988) 1081–1086
16. van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the Association for Computing Machinery **38**(3) (1991) 620–650
17. Tarski, A.: A lattice theoretical fixpoint theorem and its application. Pacific Journal of Mathematics **5** (1955) 285–309