Frame Logic under Answer Set Semantics

Mario Alviano and Francesco Calimeri and Giovambattista Ianni and Alessandra Martello

Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy.
{alviano,calimeri,ianni,a.martello}@mat.unical.it

Abstract. Frame logic is well known as a useful ontology modeling formalism, mainly appreciated for its object-oriented features and its non-monotonic variant, capable to deal with typical nonmonotonic features such as object oriented inheritance.
This paper presents a preliminary work defining a framework for coping with frame-like syntax and higher order reasoning within an answer set programming environment. Semantics is defined by means of a translation to an ordinary answer set program. A working prototype, together with usage examples, is presented.
The works paves the way to further deeper studies about the usage of frame logic-like constructs under Answer Set Semantics.

1 Introduction

Frame Logic (F-logic) [1, 2] is a knowledge representation and ontology language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modelling capabilities supported by the object oriented data model.
The basic idea behind F-logic is to consider complex data types as in object-oriented databases, combine them with logic and use the result as a programming language. Some of the desirable features of F-logic are:

– it eases the burden of declaring properties of instance data. With respect to the flat relational model (and to its logic counterpart, where tables are modelled as predicates), it is possible to “talk” about all the facts related to a single individual within a single structure, i.e. the *molecule*.
– it eases the burden of reasoning about classes. Higher order reasoning is native in F-logic, so that the border between the notion of class and individual is smooth. With respect to the usual logic programming paradigm, it is then possible to reason about classes with the same ease of use as classic logic programming offers when it is necessary to reason on individuals.

On the other hand, Answer Set Programming (ASP) languages and systems (among the variety of such systems we recall here DLV [3] and GNT/smodels [4]), offer several other desirable features such as declarativity (they are fully compliant with a strongly assessed model-theoretic semantics [5]) and nondeterminism

(possibility to specify, in a declarative way, search spaces, strong and soft constraints [6], and more). Also, ASP shares with F-logic the possibility to reason about ontologies using nonmonotonic constructs, included nonmonotonic inheritance, as it is done in some ASP extensions conceived for modelling ontologies [7].
Nonetheless, ASP misses the useful F-logic syntax and higher order reasoning capabilities. This paper aims at extending answer set programming with some of the F-logic features. In particular, our contributions are:

1. We present the family of Frame Answer Set Programs (FAS programs), allowing usage of frame-like constructs, and of higher order atoms. Interestingly, frames may appear both in the head and in the body of rules and can be nested. Nested frames might be negated, allowing, to some extent, the same liberality in writing programs typical of nested logic programs [8].
2. We provide the semantics of FAS programs in terms of a translation to a higher order answer set program.
3. We show some example describing the ease of use of the language.
4. We present a system (DLT), able to deal with programs in F-logic like syntax, and featuring higher order reasoning. DLT is a front-end, i.e., such programs are translated in the syntax accepted by several solvers, thus enabling F-logic features under Answer Set Semantics.

The remainder of the paper is structured as follows: Section 2 introduces a "running example" that will help the reader understanding our approach. Section 3 introduces the syntax of the language FAS (Frame Answer Set). Section 4 contains a formalization of the semantics of FAS programs, while Section 5 describes how to use frame-like syntax for modeling the running example. Eventually, the system supporting FAS programs is described in Section 6, and conclusions are drawn in Section 7.

2 Running example

In order to make the capabilities of the system and the range of applications clearer, in the rest of this paper we will refer to an example herein introduced. The example refers to a typical *team building* problem. A leader of a given project needs to build a team from a set of employees, everyone having some skills. The team must be selected according to the following specifications:

s1. The team consists of a given fixed number of employees.
s2. At least a given set of different skills must be present in the team.
s3. The sum of the salaries of the employees working in the team must not exceed a given budget.
s4. The salary of each individual employee is within a specified limit.
s5. The number of women working in the team has to reach at least a given number.

s6. Possibly, overlappings on desired skills should be avoided, therefore at most one of employee with some required skills is preferable.
s7. Possibly, employees with undesired skills should be not selected.

We decided to model employees by instances of the class named *employee*. This class has *properties* corresponding to the actual employee description and skill. Properties are mapped to several binary predicates (*gender, surname, married, skill, salary*). A project is encoded by instances of the *project* class, while specified properties of the project are stored in apposite predicates (*name, budget, numEmployee, maxSalary, numFemale, wantedSkills, nonWantedSkills*).

3 Syntax

We present here the syntax of FAS (Frame Answer Set) programs. We will assume the reader to be familiar with basic notions concerning with Answer Set Programming (ASP) [5].
Let $\mathcal{C}$ be a set of constant and predicate symbols. Let $\mathcal{X}$ be a set of variables. We conventionally denote variables with uppercase first letter (e.g. X, $Project$), while constant with lowercase first letter (e.g. x, $brown$, $nonWantedSkill$). A Frame Answer Set *program* (FAS program) is a set of *rules*, of the form

$$a_1 \vee, \ldots, \vee a_l \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m.$$

where $a_1, \ldots, a_l$ and $b_1, \ldots, b_k$ are *literals*, $not\, b_{k+1}, \ldots, not\, b_m$ are *naf-literals*, and $l \geq 0$, $k \geq 0$, $m \geq k$. An atom can be a *standard* or a *frame* atom. A *standard atom* is of the form $t_0(t_1, \ldots, t_n)$, where $t_0, \ldots, t_n$ are *terms*, and t_0 represents the *predicate name* of the atom. A *term* is either a variable from $\mathcal{X}$, or a constant symbol from $\mathcal{C}$. A *literal* is either an atom p, or an expression of the form $\neg p$ (called "strongly negated" atom), where p is an atom. A *naf-literal* is either of the form b, or of the form $not\, b$, where b is a literal.
A *frame atom*, or *molecule*, can be of one of the following three forms:

i. $obj[a_1, \ldots, a_n]$

ii. $obj : class$

iii. $obj : class[a_1, \ldots, a_n]$

where *obj* is a term, called the *subject* of the frame and used to define an object, *class* is a term that defines the *class* which *obj* belongs to, and $a_1, \ldots, a_n$ is a list of *attribute expressions* used to define *properties* of objects.
Informally, a *frame molecule* asserts that the object has some properties as specified by the attribute expressions listed inside the brackets.
An attribute expression defines an association between an *attribute name* and one or multiple *values* than it can take. We use the *auxiliary symbols* "$\rightarrow$" and "$\twoheadrightarrow$" respectively to define the single value and the set-valued mapping. A *positive attribute expression a* can be of one of the following three forms:

i. *name*
ii. *name op value* $\quad [op \in \{\rightarrow, \twoheadrightarrow\}]$
iii. *name* $\twoheadrightarrow$ *values*

where *name* is a term (or a strongly-negated term) representing the name of the property, *value* is either a molecule or a term, and *values* is a non-empty set of *value*. If more values are given for *multi-valued attributes* (e.g, case (*iii*)), the values must be enclosed in curly brackets. Note that, when only one element appears inside curly brackets, we may omit these.
A *negative attribute expression* is a negated positive attribute expression. An *attribute expression* is either a positive attribute expression or a negative attribute expression.
A *plain higher order* ASP program contains only standard atoms, while a *plain* ASP program contains only standard atoms $t_0(t_1, \ldots, t_n)$ where t_0 is a constant symbol.
Every object name refers to *exactly one* object, although molecules starting with the same *subject* may be combined. Since the value referred to an object attribute can be frames, molecules can be *nested.*
As an example, the following is a frame molecule:

$$\begin{aligned} brown : employee[\; & surname \rightarrow \text{“Mr. Brown”},\\ & skill \twoheadrightarrow \{java, asp\},\\ & salary \rightarrow 800,\\ & gender \rightarrow male,\\ & married \rightarrow pink\,] \end{aligned}$$

This defines membership of the subject *brown* to the *employee* class and asserts some values corresponding to the properties bind to this object. This frame molecule says *brown* is *male* (as expressed by the value of the attribute *gender*), is *married* to another employee identified by the subject *pink*. *brown* knows *java* and *asp* languages, as suggests the values of the *skill* property, while he has a *salary* equal to *800.* We may define a new frame molecule, like this, collecting information about the employee encoded by the subject *pink*, but we can also combine this information nesting frame molecules, as follows:

$$\begin{aligned} brown : employee[\; & surname \rightarrow \text{“Mr. Brown”},\\ & skill \twoheadrightarrow \{java, asp\},\\ & salary \rightarrow 800,\\ & gender \rightarrow male,\\ & married \rightarrow pink : employee[\; surname \rightarrow \text{“Mrs. Pink”},\\ & \qquad\qquad\qquad\qquad\qquad skill \twoheadrightarrow \{html, asp, javascript\},\\ & \qquad\qquad\qquad\qquad\qquad salary \rightarrow 900,\\ & \qquad\qquad\qquad\qquad\qquad gender \rightarrow female,\\ & \qquad\qquad\qquad\qquad\qquad married \rightarrow brown\,]\\ ]. \end{aligned}$$

The following is an example of logic rule defining the profile a particular employee must have in order to be selected for project $p3$. We encoded this rule using *strong* and *naf* nested negation:

$$\begin{aligned} E[\mathit{inProject} \twoheadrightarrow\ p3] \vee E[\neg \mathit{inProject} \twoheadrightarrow\ p3] \leftarrow\ & X : \mathit{employee}, \\ & E : \mathit{employee}[\mathit{skill} \twoheadrightarrow\ \{c++, \mathit{perl}\}, \\ & \mathit{not\ married} \rightarrow X : \mathit{employee}[\\ & \mathit{not\ skill} \twoheadrightarrow\ \{c++, \mathit{perl}\}\,]]. \end{aligned}$$

This means that candidates to the project team *p3* are employees knowing *c++* and *perl* programming languages, but not married to another employee not knowing the same programming languages.

4 Semantics

We provide here the semantics of FAS programs in terms of a translation to a higher order ASP program. Thus we first provide the semantics of plain higher order ASP programs.

4.1 Semantics of plain higher order programs

Semantics of higher order programs is defined in terms of the traditional Gelfond-Lifshitz reduct for a ground disjunctive logic program with classical negation [5]. Given a plain higher order program P, its ground version $\mathit{grnd}(P)$ is given by grounding rules of P by all the possible substitutions that can be obtained using consistently elements of $\mathcal{C}$. A ground rule thus contains only ground atoms; the set of all possible ground atoms that can be constructed combining predicates and terms occurring in the program is usually referred to as *Herbrand base* (B_P). We remark that the grounding process substitutes also nonground predicates names with symbols from $\mathcal{C}$ (e.g., a valid ground instance of the atom $H(\mathit{brown}, X)$ is $\mathit{married}(\mathit{brown}, \mathit{pink})$): however, $\mathit{grnd}(P)$ is a standard ASP ground program.
An *interpretation* for P is a set of ground atoms, that is, an interpretation is a subset $I \subseteq B_P$. I is said to be *consistent* if $\forall a \in I$ we have that $\neg a \notin i$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal *not* A is *true* w.r.t. I if A is false w.r.t. I; otherwise *not* A is false w.r.t. I.
Given a ground rule $r \in \mathit{grnd}(P)$, the head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.
A *model* for P is an interpretation M for P such that every rule $r \in \mathit{grnd}(P)$ is true w.r.t. M. A model M for P is *minimal* if no model N for P exists such that N is a proper subset of M. The set of all minimal models for P is denoted by $\mathrm{MM}(P)$.

Given a program P and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of P w.r.t. I, denoted P^I, is the set of positive rules of the form $\{a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k\}$ such that $\{a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, not\, b_{k+1}, \cdots, not\, b_m\}$ is in $\mathrm{grnd}(P)$ and $b_i \notin I$, for all $k < i \leq m$. An interpretation I for a program P is an *answer set* for P if $I \in \mathrm{MM}(P^I)$ (i.e., I is a minimal model for the positive program P^I) *[9, 5]*. The set of all answer sets for P is denoted by $ans(P)$.

4.2 From FAS programs to higher order programs

We show how to reduce the frame logic-like formalism embedded in our hybrid framework to ASP, thus allowing to manipulate frames with logic programming techniques. This operational semantics is defined through a suitable algorithm which is able, given a FAS programs containing frame structures, to produce an equivalent plain higher order ASP program.

Roughly, the idea is to introduce new predicate names wrapping properties and classes. Classes are mapped to unary predicates, while properties are mapped to unary or binary predicates. Then, a FAS program is *unfolded* in order to replace frame atoms with their equivalent predicates.

The algorithm providing the semantics is called *Standardize Algorithm* ($\mathcal{S}$): it takes as input a *FAS program P* containing frame atoms; the output is a plain higher order program R. The Answer Sets of P are defined as to be the answer sets of R. The algorithm is sketched in Figure 1.

In order to better explain how S works, we show how a frame structure is examined and processed. For instance, if we consider the following frame:

$$\begin{aligned} E[inProject \twoheadrightarrow p3] \vee E[-inProject \twoheadrightarrow p3] \leftarrow\ & X : employee, \\ & E : employee[\\ & skill \twoheadrightarrow \{c++, perl\}, \\ & not\ married \rightarrow X\,]. \end{aligned}$$

The application of S generates this output:

$$\begin{aligned} inProject(E,\, p3) \vee -inProject(E,\, p3) \leftarrow\ & employee(X), skill(E,\, c++), \\ & skill(E,\, perl), employee(E), \\ & not\, aux\_e(E,\, X). \\ aux\_e(E,\, X) \leftarrow\ & married(E,\, X). \end{aligned}$$

```
Standardize (INPUT: $P$ containing frames  OUTPUT: $R$ without frames)
Let $R = P$;
while $R$ contains frame literals do
      let $r \in R$ be a rule containing frame atoms;
      while $r$ contains frame literals do
          remove frame $f$ from $r$;
          let $o$ be the subject, $L$ the set of attributes, and $\mathbf{X}$ the set of variables of $f$;
          case $f$ appeared in the body of $r$:
              if $f$ is positive then
                  if $f$ has class $c$ then
                      add $c(o)$ to the body of $r$;
                  for each attribute expression $e \in L$ do
                      let $a$ be the name, and $V$ the set of values of $e$;
                      if $e$ is positive then
                          if $V$ is empty then
                              add $a(o)$ to the body of $r$;
                          else
                              for each term $t \in V$ do
                                  add $a(o, t)$ to the body of $r$;
                              for each molecule $m \in V$ with subject $s$ do
                                  add $a(o, s)$ and $m$ to the body of $r$;
                      else
                          let $e$ be in the form $not\, e'$;
                          add the frame $not\, o[e']$ to the body of $r$;
              else (Let $f$ in form $not\, f'$)
                  add to $r$ a new fresh literal not $aux_f(\mathbf{X})$;
                  add to $R$ a new rule $aux_f(\mathbf{X}) \leftarrow f'$;
          case $f$ appeared in the head of $r$:
              if $f$ is in the form $o : c$ (resp. in the form $o[a \rightarrow v]$) then
                  add to the head of $r$ the literal $c(o)$ (resp. $a(o, v)$);
              else
                  add to the head of $r$ a new atom $aux_f(\mathbf{X})$;
                  if $f$ has class $c$ then
                      add $c(o) \leftarrow aux_f(\mathbf{X})$ to $R$;
                  for each attribute expression $e \in L$ do
                      let $a$ be the name, and $V$ the set of values of $e$;
                      if $V$ is empty then
                          add $a(o) \leftarrow aux_f(\mathbf{X})$ to $R$;
                      else
                          for each term $t \in V$ do
                              add $a(o, v) \leftarrow aux_f(\mathbf{X})$ to $R$;
                          for each molecule $m \in V$ with subject $s$) do
                              add $a(o, s) \leftarrow aux_f(\mathbf{X})$ and $m \leftarrow aux_f(\mathbf{X})$ to $R$;
```

Fig. 1. The Standardize Algorithm.

5 Examples

In this section we will provide several examples aiming at showing how to use the frame-like language.

Higher order reasoning enables the possibility to quantify over predicate names. The rule

$$related(X, Y) \leftarrow X[Z \rightarrow Y].$$

relates all the X and Y for which some property Z holds. Also higher order reasoning enables the possibility to enforce axioms that must hold on predicates, such as

$$X : C : -subClassOf(C, D), X : D.$$

this enforces membership of an individual X to the class C when it is known that D is a subclass of C and that X is member of C.
Ease of use of frames can be seen by looking at our running example. Employees might be encoded as:

$$\begin{array}{rl}
brown : employee[& surname \rightarrow \text{“Mr. Brown”}, \\
& skill \twoheadrightarrow \{java, asp\}, \\
& salary \rightarrow 800, \\
& gender \rightarrow male, \\
& married \rightarrow pink]. \\
red : employee[& surname \rightarrow \text{“Mrs. Red”}, \\
& skill \twoheadrightarrow \{java, php, perl, python\}, \\
& salary \rightarrow 1200, \\
& gender \rightarrow female, \\
& married \rightarrow black]. \\
pink : employee[& surname \rightarrow \text{“Mrs. Pink”}, \\
& skill \twoheadrightarrow \{html, asp, javascript\}, \\
& salary \rightarrow 900, \\
& gender \rightarrow female, \\
& married \rightarrow brown]. \\
black : employee[& surname \rightarrow \text{“Mr. Black”}, \\
& skill \twoheadrightarrow \{c++, asp, asp, perl, php, python\}, \\
& salary \rightarrow 1900, \\
& gender \rightarrow male, \\
& married \rightarrow red].
\end{array}$$

While projects might be described as:

$$\begin{array}{rl}
p1 : project[& name \rightarrow \text{“A System for a Really Nice Web Site”}, \\
& budget \rightarrow 1800, \\
& numEmployee \rightarrow 2, \\
& maxSalary \rightarrow 1000, \\
& numFemale \rightarrow 1, \\
& wantedSkills \twoheadrightarrow \{html, java\}, \\
& nonWantedSkills \twoheadrightarrow \{c++\}].
\end{array}$$

$$
\begin{aligned}
p2 : project[\ & name \rightarrow \text{“A Semantic-Web-Oriented Extension of DLV”},\\
& budget \rightarrow 3000,\\
& numEmployee \rightarrow 2,\\
& maxSalary \rightarrow 1800,\\
& numFemale \rightarrow 1,\\
& wantedSkills \twoheadrightarrow \{c++, html\},\\
& nonWantedSkills \twoheadrightarrow \{java\}\,].
\end{aligned}
$$

Membership of a given employee E, in a given project P, can be *guessed* with the following disjunctive rule:

$$E[inProject \twoheadrightarrow P] \vee E[-inProject \twoheadrightarrow P] \leftarrow E : employee, P : project.$$

Conditions $(s1)$, ..., $(s7)$ can be addressed by the following FAS constraints.

$$
\begin{aligned}
(s1) \leftarrow\ & P : project[numEmployee \rightarrow N_{empl}],\\
& not\, \#count\{E : inProject(E,P)\} = N_{empl}.\\
(s2) \leftarrow\ & P : project[wantedSkills \twoheadrightarrow W_{sk}],\\
& not\, \#count\{E : skill(E, W_{sk}), inProject(E,P)\} \geq 1.\\
(s3) \leftarrow\ & P : project[budget \rightarrow B],\\
& not\, \#sum\{S,\, E : salary(E,S), inProject(E,P)\} \leq B.\\
(s4) \leftarrow\ & P : project[maxSalary \rightarrow M_{sal}],\\
& E : employee[inProject \twoheadrightarrow P, salary \rightarrow S], not\, S \leq M_{sal}.\\
(s5) \leftarrow\ & P : project[numFemale \rightarrow N_{fem}],\\
& not\, \#count\{E : gender(E, female), inProject(E,P)\} \leq N_{fem}.\\
(s6) :\sim\ & E1 : employee[skill \twoheadrightarrow S, inProject \twoheadrightarrow P[wantedSkill \twoheadrightarrow S]],\\
& E2 : employee[skill \twoheadrightarrow S, inProject \twoheadrightarrow P], E1 \neq E2.\ [1:]\\
(s7) :\sim\ & E : employee[skill \twoheadrightarrow S,\\
& inProject \twoheadrightarrow P[nonWantedSkill \twoheadrightarrow S]].\ [1:]
\end{aligned}
$$

It is worth noting that in the above set of constraints we take advantage of some peculiar features of the DLV system, namely weak constraints [6] and aggregates [10].
Intuitively, a weak constraint W induces an ordering among the answer sets of a given program, depending on the number of ground instances that violate W (the lesser W is violated in an answer set A, the more A is preferred). The aggregates atoms *#count* and *#sum*, on the other hand, are exploited in order to count the number of values a given conjunction of atoms may have, and for summing up numeric terms appearing in a given conjunction of atoms, respectively. For the sake of simplicity, such constructs were not included in the formal syntax and semantics definition given in Section 3 and 4. We refer the reader to the cited literature for further details.

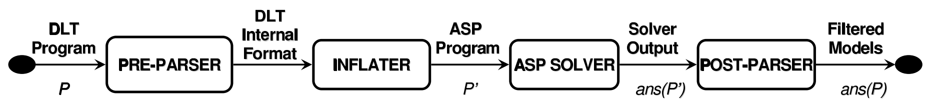

Fig. 2. Architecture of DLT System.

6 System Overview

FAS programs have been implemented within the DLT environment [11]. DLT extends the DLV system (and, to some extent, any other ASP solver) with Template predicates, Frame Logic and Higher Order predicates. The current version of the system is freely available on the DLT Web page[1].
The overall architecture of the system is shown in Figure 2. Roughly, the DLT system works as follows. A FAS program P is sent to a DLT *pre-parser*, which performs *syntactic checks*, converts frame syntax to plain syntax (by applying the algorithm $\mathcal{S}$ defined in Section 4), and builds an internal representation of P. The DLT *Inflater* produces an equivalent program P' by processing and eliminating other special constructs of the language, such as templates; P' is piped towards an answer set solver. The answer sets $ans(P')$ of P', computed by the solver are then converted in a readable format through the *Post-parser* module, which filters out from $ans(P')$ information about predicates and rules that were internally generated.

Additional features of the system

As a front-end system, DLT features several other constructs that can be used for enriching an answer set solver. Furthermore, although the syntax of produced programs is compliant with DLV, if special constructs of DLV (like disjunction, aggregate atoms, weak constraints) are avoided, DLT is compliant with any other solver supporting the traditional, prolog-like, syntax. Among others, DLT features are:

Template definitions. A DLT program may contain *template atoms*, that allow to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This feature provides a succinct and elegant way for quickly introducing new constructs using the DLT language. Syntax and semantics of template atoms are described in [11].

Frame Spaces. A Frame Space directive tells how frames are mapped to regular atoms, and can be used for defining modules where each predicate has local scope within a given frame space. The directive has syntax *@name*. From the point after the directive each frame is interpreted as belonging to the frame space *name*, and local to this. For referring to a predicate or frame belonging to

[1] http://dlt.gibbi.com.

a given frame space it is possible to use the syntax $atom@framespace$, like in e.g. $person(gibbi)@local$.
Internally, a frame like
$X[f \rightarrow Y]$ is rewritten as *f(X,Y,name)*.
When the directive "*@.*" is used, the systems switchs to the default frame space, thus triggering the traditional behavior of the system.

Solvers support. DLT can virtually support any solver that accepts inputs in the format generated by DLT. Models produced by the external solver are then parsed back to the DLT syntax. The `-solver=[pathname]` option allows to specify the path of the solver. Compatibility with the systems DLV [3], DLV-EX [12], DLV-HEX [13] is provided; compatibility with S-models is guaranteed within a subset of the language. A detailed compatibility table is available on the DLT web site.

Function Symbols. Besides constant and variable terms, the DLT parser allows also functional terms. Solvers allowing function symbols are thus ready to be coupled with DLT.

7 Conclusions

We have presented a framework that allows to enrich an Answer Set Programming language with frame-like syntax and higher order reasoning. While preliminary, our work paves the way to a more formal investigation of possible semantics for F-logic under stable models semantics.

References

1. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42**(4) (1995) 741–843
2. Yang, G., Kifer, M.: Inheritance in Rule-Based Frame Systems: Semantics and Inference. Journal on Data Semantics VII (2006) 79–135
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3) (2006) 499–562
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). Volume 2923 of Lecture Notes in AI (LNAI)., Fort Lauderdale, Florida, USA, Springer (2004) 331–335
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
6. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE Transactions on Knowledge and Data Engineering **12**(5) (2000) 845–860
7. Ricca, F., Leone, N., Bonis, V.D., Dell'Armi, T., Galizia, S., Grasso, G.: A dlp system with object-oriented features. In: LPNMR. (2005) 432–436
8. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. Annals of Mathematics and Artificial Intelligence **25**(3–4) (1999) 369–389

9. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. New Generation Computing **9** (1991) 401–424
10. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 406–411
11. Calimeri, F., Ianni, G.: Template programs for disjunctive logic programming: An operational semantics. AI Communications **19**(3) (2006) 193–206
12. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence (2007. To appear.)
13. Eiter, T., Ianni, G., Tompits, H., Schindlauer, R.: A uniform integration of higher-order reasoning and external evaluations in answer set programming. In: IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburh, UK August 2-5, 2005. (2005) 90–96