Enhancing ASP by Functions: Decidable Classes and Implementation Techniques*

Francesco Calimeri and **Susanna Cozza** and **Giovambattista Ianni** and **Nicola Leone**
Department of Mathematics, University of Calabria
I-87036 Rende (CS), Italy
e-mail: {calimeri, cozza, ianni, leone}@mat.unical.it

Abstract

This paper summarizes our line of research about the introduction of function symbols (functions) in Answer Set Programming (ASP) – a powerful language for knowledge representation and reasoning. The undecidability of reasoning on ASP with functions, implied that functions were subject to severe restrictions or disallowed at all, drastically limiting ASP applicability. We overcame most of the technical difficulties preventing this introduction, and we singled out a highly expressive class of programs with functions ($\mathcal{FG}$-programs), allowing the (possibly recursive) use of function terms in the full ASP language with disjunction and negation. Reasoning on $\mathcal{FG}$-programs is decidable, and they can express any computable function (causing membership in this class to be semi-decidable). We singled out also $\mathcal{FD}$-programs, a subset of $\mathcal{FG}$-programs which are effectively recognizable, while keeping the computability of reasoning. We implemented all results into the DLV system, thus obtaining an ASP system allowing to encode any computable function in a rich and fully declarative KRR language, ensuring termination on every $\mathcal{FG}$ program. Finally, we singled out the class of DFRP programs, where decidability of reasoning is guaranteed and Prolog-like functions are allowed.

1 Introduction

Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Marek and Truszczyński 1999), is a powerful formalism for knowledge representation and reasoning (KRR), which is widely used in AI especially for its high expressiveness and for its ability to deal also with incomplete knowledge (Baral 2003). Several systems supporting ASP are nowadays available, making ASP actually usable. Experimenting with ASP in real-world applications, from the one hand, has confirmed its usefulness; but, on the other hand, has evidenced some limitations. One of the most noticeable shortcoming is the lack of an adequate support for complex terms like functions, sets, and lists. Indeed, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. This need has been clearly perceived in the ASP community, and the latest years witness a big effort for embedding functions in the context of ASP (Syrjänen 2001; Bonatti 2004; Gebser, Schaub, and Thiele 2007; Lin and Wang 2008; Baselice, Bonatti, and Criscuolo 2009; Eiter and Simkus 2009; Lierler and Lifschitz 2009). Nevertheless, no ASP system allows for a reasonably unrestricted usage of function terms. Functions are either required to be nonrecursive or subject to severe syntactic limitations. The goal of our research is to overcome the above mentioned limitations. The main results can be summarized as follows.

- ▶ We formally define the class of *finitely-ground ($\mathcal{FG}$)* programs, allowing the (possibly recursive) use of functions in the ASP language with disjunction and negation. We prove that $\mathcal{FG}$ programs guarantee decidability for both brave and cautious reasoning, and computability of answer sets. We show that the language is highly expressive, as *any* computable function can be encoded by an $\mathcal{FG}$ program (Calimeri et al. 2008).
- ▶ We prove that membership in the class of $\mathcal{FG}$ programs is semi-decidable: for applications where termination needs to be "a priori" (statically) determined, we define the subclass of *finite-domain ($\mathcal{FD}$)* programs, where membership is decidable, while the computability of the reasoning tasks is kept (Calimeri et al. 2008).
- ▶ We enrich ASP with list and set terms, and implement the full (extended) language into the DLV system, obtaining DLV-Complex, a new system allowing to exploit the full expressiveness of $\mathcal{FG}$ programs, or to require the finite-domain check, getting the guarantee of termination. The system is publicly available (Calimeri et al. since 2008) and already in use in many universities and research centers throughout the world. A rich library of built-in functions for list and set manipulation complements the system and further improves its usability (Calimeri et al. 2009a).
- ▶ For users familiar with top-down computation, we single out the class of disjunctive finitely recursive positive programs (DFRP), where the allowed functions are the top-down decreasing ones, à la Prolog, and head-unsafe variables are permitted. We prove that ground querying

*This paper summarizes the results appearing in (Calimeri et al. 2008, Calimeri et al. 2009a, Calimeri et al. 2009b).

is decidable on DFRP programs. We design a suitable magic set transformation which turns ground querying on a DFRP program into answer sets computation on an $\mathcal{FG}$-program (Calimeri et al. 2009b), making also DFRP programs usable in practice.

Function symbols allow us to aggregate atomic data and manipulate complex data structures. For instance, in the area of self-healing Web Services, our DLV-Complex system is already exploited for the computation of minimum cardinality diagnoses (Friedrich and Ivanchenko 2008), functional terms are there employed to replace existential quantification. The introduction of functions offers a significant gain in terms of knowledge-modeling power and program clarity. In some cases, the better problem encoding obtained through functions, can bring also a significant computational gain (see, for instance, the encoding for Tower of Hanoi reported in DLV-Complex web site against the classical ASP encoding).

2 DLP with Functions

DLP programs are finite set of disjunctive rules of the form $r : \alpha_1 \vee \cdots \vee \alpha_k$:- $\beta_1, \ldots, \beta_n$, $\texttt{not}\,\beta_{n+1}, \ldots, \texttt{not}\,\beta_m$. where $m \geq 0$, $k \geq 0$, and $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_m$ are atoms. If $k = 1$ and $m = 0$ then r is referred to as a *fact*. An *atom* is of the form $p(t_1, \ldots, t_k)$, where p is a predicate symbol of arity $k \geq 0$ and each t_i is a term. A *term* is either a *simple term* (constant or variable) or a *functional term*, which is of the form $f(t_1, \ldots, t_n)$, for f a function symbol, and $t_1, \ldots, t_n$ terms. $\alpha_1 \vee \cdots \vee \alpha_k$ is the *head* of the rule; while $\beta_1, \ldots, \beta_n$, $\texttt{not}\ \beta_{n+1}, \ldots, \texttt{not}\ \beta_m$ is the *body*. The sets $\{\alpha_1, \ldots, \alpha_k\}$ and $\{\beta_1, \ldots, \beta_n\}$ are denoted by $H(r)$ and $B^+(r)$, respectively. We assume (except in Section 6) that each rule r is safe: each variable of r appears also in $B^+(r)$. Given a program P, a predicate defined only by facts is an ***EDB*** predicate, the remaining predicates are ***IDB*** predicates. The set of all facts in P is denoted by $Facts(P)$; the set of instances of all EDB predicates in P is denoted by $EDB(P)$. The set of all head atoms in P is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$.

Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all (ground) terms that can be built combining constants and functors appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all ground atoms obtainable from the atoms of P by replacing variables with elements from U_P. A *substitution* for a rule $r \in P$ is a mapping from the set of variables of r to the set U_P of ground terms. A *ground instance* of a rule r is obtained applying a substitution to r. The *instantiation (grounding)* $grnd(P)$ of P is defined as the set of all ground instances of its rules. Given a ground program P, an *interpretation* I for P is a subset of B_P. A positive literal $l = a$ (resp., a negative literal $l = \texttt{not}\ a$) is true w.r.t. I if $a \in I$ (resp., $a \notin I$); it is false otherwise. Given a ground rule r, we say that r is satisfied w.r.t. I if some atom appearing in $H(r)$ is true w.r.t. I or some literal appearing in $B(r)$ is false w.r.t. I. Given a ground program P, we say that I is a *model* of P, iff all rules in $grnd(P)$ are satisfied w.r.t. I. A model M is *minimal* if there is no model N for P such that $N \subset M$. The *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz 1991) of P, w.r.t. an interpretation I, is the positive ground program P^I obtained from $grnd(P)$ by: (i) deleting all rules having a negative literal false w.r.t. I; (ii) deleting all negative literals from the remaining rules. $I \subseteq B_P$ is an *answer set* for a program P iff I is a minimal model for P^I. The set of all answer sets for P is denoted by $AS(P)$.

3 Finitely-Ground Programs

Informally, a program P is $\mathcal{FG}$ if its instantiation has a finite subset S, which has the same answer sets as P and is computable.

Given a program P, a *component* of P is a maximal subset of mutually recursive predicates of P, where only positive dependencies are considered (e.g. in rule $a(X)$:- $b(X), \texttt{not}\, c(X)$, a depends on b, but not on c). We denote by $comp(p)$ the component of predicate p. A component ordering $\gamma = \langle C_1, \ldots, C_n \rangle$ for P is a total ordering of the components of P strictly respecting the positive and the stratified dependencies among predicates of different components. For instance, in presence of the rule $a(X)$:- $b(X), \texttt{not}\, c(X)$, if a, b and c belong to different components, then $comp(b)$ precedes $comp(a)$, while $comp(c)$ might not precede $comp(a)$ only if there is a mutually negative dependency between a and c. The *module* $\mathrm{P}(C_i)$ of a component C_i is the set of all rules defining some predicate $p \in C_i$ excepting those defining also some other predicate belonging to a lower component (i.e., certain C_j with $j < i$ in γ).

Definition 1 Given a rule r and a set S of ground atoms, an *S-restricted* instance of r is a ground instance r' of r such that $B^+(r') \subseteq S$. The set of all S-restricted instances of a program P is denoted as $Inst_P(S)$.

Note that, for any $S \subseteq B_P$, $Inst_P(S) \subseteq grnd(P)$. Intuitively, this helps selecting, among all ground instances, those somehow *supported* by a given set S.

Example 2 Consider the following program P:
$t(f(1))$. $t(f(f(1)))$. $p(1)$.
$p(f(X))$:- $p(X)$, $t(f(X))$.
The set $Inst_P(S)$ of all S-restricted instances of P, w.r.t. $S = Facts(P)$ is: $t(f(1))$. $t(f(f(1)))$. $p(1)$. $p(f(1))$:- $p(1)$, $t(f(1))$.

The presence of negation allows to identify some further rules which do not matter for the computation of answer sets, and to simplify the bodies of some others. This can be properly done by exploiting a modular evaluation of the program that relies on a component ordering.

Definition 3 Given a program P, a component ordering $\langle C_1, \ldots, C_n \rangle$, a set S_i of ground rules for C_i, and a set of ground rules R for the components preceding C_i, the *simplification* $Simpl(S_i, R)$ of S_i w.r.t. R is obtained from S_i by:

1. *deleting* each rule whose body contains some negative body literal $\texttt{not}\ a$ s.t. $a \in Facts(R)$, or whose head contains some atom $a \in Facts(R)$;

2. *eliminating* from the remaining rules each literal l s.t., either $l = a$ is a positive body literal and $a \in Facts(R)$, or $l = \texttt{not}\ a$ is a negative body literal, $comp(a) = C_j$ with $j < i$, and $a \notin Heads(R)$.

Assuming that R contains all instances of the modules preceding C_i, $Simpl(S_i, R)$ deletes from S_i all rules whose body is certainly false or whose head is certainly already true w.r.t. R, and simplifies the remaining rules by removing from the bodies all literals that are true w.r.t. R.

Example 4 Consider the following program P:

$t(1).\ \ s(1).\ \ s(2).\ \ q(X) \text{ :- } t(X).\ \ \ p(X) \text{ :- } s(X), \texttt{not}\ q(X).$

It is easy to see that $\langle C_1 = \{q\}, C_2 = \{p\}\rangle$ is the only component ordering for P. If we consider $R = EDB(P) = \{\ t(1).,\ s(1).,\ s(2).\ \}$ and $S_1 = \{q(1) \text{:-} t(1).\}$, then $Simpl(S_1, R) = \{q(1).\}$ (i.e., $t(1)$ is eliminated from body). Considering then $R = \{t(1).,\ s(1).,\ s(2).,\ q(1).\}$ and $S_2 = \{\ p(1) \text{:-} s(1), \texttt{not}\ q(1).,\ p(2) \text{:-} s(2), \texttt{not}\ q(2).\ \}$, after the simplification we have $Simpl(S_2, R) = \{p(2).\}$. Indeed, $s(2)$ is eliminated as it belongs to $Facts(R)$ and $\texttt{not}\ q(2)$ is eliminated because $comp(q(2)) = C_1$ precedes C_2 in the component ordering and the atom $q(2) \notin Heads(R)$; in addition, rule $p(1) \text{:-} s(1), \texttt{not}\ q(1).$ is deleted, since $q(1) \in Facts(R)$.

We are now ready to define an operator Φ that acts on a module of a program P in order to: (i) select only the ground rules whose positive body is contained in a set of ground atoms consisting of the heads of a given set of rules; (ii) perform further simplifications among these rules by means of $Simpl$ operator.

Definition 5 Given a program P, a component ordering $\langle C_1,\ \ldots,\ C_n\rangle$, a component C_i, the module $M = P(C_i)$, a set X of ground rules of M, and a set R of ground rules belonging only to EDB(P) or to modules of components C_j with $j < i$, let $\Phi_{M,R}(X)$ be the transformation defined as follows: $\Phi_{M,R}(X) = Simpl(Inst_M(Heads(R \cup X)), R)$.

Example 6 Let P be the program

$$a(1).\ \ q(g(3)).\ \ s(X) \vee t(f(X)) \text{ :- } a(X), \texttt{not}\ q(X).$$
$$p(X,Y) \text{ :- } q(g(X)),\ t(f(Y)).\ \ \ \ q(X) \text{:- } s(X),\ p(Y,X).$$

Considering the component $C_1 = \{s\}$, the module $M = P(C_1)$, and the sets $X = \emptyset$ and $R = \{a(1)\}$, we have:

$$\begin{aligned}\Phi_{M,R}(X) &= Simpl(Inst_M(Heads(R \cup X)), R) = \\ &= Simpl(Inst_M(\{a(1)\}), \{a(1).\}) = \\ &= Simpl(\{s(1) \vee t(f(1)) \text{:-} a(1), \texttt{not}\ q(1).\}, \{a(1).\}) = \\ &= \{s(1) \vee t(f(1)) \text{:-} \texttt{not}\ q(1).\}.\end{aligned}$$

The operator Φ has the following important property.

Proposition 7 $\Phi_{M,R}$ always admits a least fixpoint $\Phi^{\infty}_{M,R}(\emptyset)$.

By properly composing consecutive applications of Φ^{∞} to a component ordering, we can obtain an instantiation which drops many useless rules w.r.t. answer sets computation.

Definition 8 Given a program P and a component ordering $\gamma = \langle C_1,\ \ldots,\ C_n\rangle$ for P, the *intelligent instantiation* P^{γ} of P for γ is the last element S_n of the sequence $S_0 = EDB(P)$, $S_i = S_{i-1} \cup \Phi^{\infty}_{M_i,S_{i-1}}(\emptyset)$, where M_i is the program module $P(C_i)$.

Example 9 Let P be the program of Example 6 where the extension of EDB predicate a is $\{a(1)\}$; considering the component ordering $\gamma = \langle C_1 = \{s\}, C_2 = \{t\}, C_3 = \{p, q\}\rangle$ we have:

- $S_0 = \{a(1).\}$;
- $S_1 = S_0 \cup \Phi^{\infty}_{M_1,S_0}(\emptyset) = \{a(1).,\ s(1) \vee t(f(1)) \text{:-} \texttt{not}\ q(1).\}$;
- $S_2 = S_1 \cup \Phi^{\infty}_{M_2,S_1}(\emptyset) = \{a(1).,\ s(1) \vee t(f(1)) \text{:-} \texttt{not}\ q(1).\}$;
- $S_3 = S_2 \cup \Phi^{\infty}_{M_3,S_2}(\emptyset) = \{a(1).,\ s(1) \vee t(f(1)) \text{:-} \texttt{not}\ q(1).,$ $q(g(3)).,\ p(3,1) \text{:-} q(g(3)), t(f(1)).,\ q(1) \text{:-} s(1), p(3,1).\}$.

Thus, the resulting intelligent instantiation P^{γ} of P for γ is:

$$a(1).\qquad q(g(3)).\qquad s(1) \vee t(f(1)) \text{:-} \texttt{not}\ q(1).$$
$$p(3,1) \text{:-} q(g(3)), t(f(1)).\qquad q(1) \text{:-} s(1), p(3,1).$$

Definition 10 A program P is *finitely-ground* ($\mathcal{FG}$) if P^{γ} is finite, for every component ordering γ for P.

Example 11 The intelligent instantiation of the program:

$$q(f(f(1))).\ \ p(1).\ \ p(f(X)) \text{:- } p(X),\ not\ q(f(X)).$$

will actually include only a finite number of rules, thanks to the dropping of all the (infinite, in this case) useless rules w.r.t. answer set computation. Thus, the unique answer set is easily computable: $\{q(f(f(1))),\ p(1),\ p(f(1)).\}$

We proved that $AS(P) = AS(P^{\gamma})$ for any component ordering γ of P, and then demonstrated what follows.

Theorem 12 Given an $\mathcal{FG}$ program P, then: *i)* P has finitely many finite answer sets; *ii)* $AS(P)$ is computable; *iii)* cautious and brave reasoning over P are computable for both ground and non-ground queries.

Also, $\mathcal{FG}$ programs are very expressive:

Theorem 13 $\mathcal{FG}$ programs can encode any computable function. Given a program P, recognizing whether P is an $\mathcal{FG}$ program is R.E.-complete.

4 Finite-Domain Programs

As reported in the previous Section, membership in $\mathcal{FG}$ class is not decidable. For this reason, in (Calimeri et al. 2008) a subclass of such programs, called *finite-domain* ($\mathcal{FD}$) programs, has been singled out, which ensures the decidability of recognizing membership in the class. A program P belongs to this class if all arguments of the predicates appearing in P are included into the set of finite-domain ($\mathcal{FD}$) arguments. Such set is defined by means of some syntactical restrictions over variables and (sub)terms when recursion is involved; intuitively, such restrictions ensure that an argument can range only on a finite set of different ground values. Importantly, we have the following:

Theorem 14 $\mathcal{FD}$ programs can be recognized in polynomial time; every $\mathcal{FD}$ program is an $\mathcal{FG}$ program.

In the next section, the program of Example 15 is $\mathcal{FD}$; while the programs in Examples 16 and 17 are $\mathcal{FG}$ but are not $\mathcal{FD}$.

5 An ASP System with Complex Terms

The results reported above allow to actually introduce an effective support for function symbols into current ASP systems. Indeed, the presented language has been implemented on top of the state-of-the-art ASP system DLV (Leone et al. 2006). The resulting system, publicly available (Calimeri et al. since 2008), called DLV-Complex, features an even richer language, that, besides functions, explicitly offers native syntactic support for complex terms such as lists and sets. A large library of *built-in* functions and predicates is provided, having a predefined intended semantics. The system features an optional syntactical check for the membership to the $\mathcal{FD}$ class, that guarantees an "a priori" termination for users/applications requiring it. We report next some examples aiming at showing the KRR capabilities of the language supported by DLV-Complex; many other examples with sets and lists, and function symbols in general, can be found in (Baral 2003; Calimeri et al. since 2008).

Example 15 A program consisting of some facts modeling strings by means of list terms, plus a rule, as reported below:

$$word([a,d,a]).\ \ word([g,i,b,b,i]).\ \ word([a,n,n,a]).$$
$$palindromic(X)\ \text{:-}\ word(X), \#reverse(X) = X.$$

has a unique answer set, including the initial facts and some instances of predicate $palindromic$ modeling all palindromic strings. Please note the usage of the built-in function $\#reverse$.

Example 16 In a given graph, a path with no repeated nodes is called a *simple* path. If a path is denoted by a list of nodes, the head of the list corresponding to the head node, the following program derives all simple paths for a directed graph starting from a given $edge$ relation (please note the use "à la Prolog" ([Head|Tail]) for lists):

$$path([X,Y])\ \text{:-}\ edge(X,Y).$$
$$path([X|[Y|W]])\ \text{:-}\ edge(X,Y), path([Y|W]), \texttt{not}\ \#member(X,[Y|W]).$$

Example 17 If facts like: $sons(someone, \{son_1, ..., son_n\})$ model the association between a parent and her sons, one can obtain the names of all descendants of someone as follows:

$$ancestor(A, Ss)\ \text{:-}\ sons(A, Ss).$$
$$ancestor(A, \#union(Ds, Ss))\ \text{:-}\ ancestor(A, Ds), \#member(S, Ds), sons(S, Ss).$$

The first rule says that all sons of A are descendants of A. The second rule says that if Ds are descendants of A, S belongs to Ds, and Ss contains the sons of S, then the union of Ds and Ss is also a set of descendants for A.

6 DFRP Programs and Top-Down Querying

One of the first and relevant proposals for extending ASP with functions, is the class of finitary programs (Bonatti 2004). Such class imposes restrictions both on recursion and on the number of potential sources of inconsistency. In particular, recursion is restricted by requiring each ground atom to depend on finitely many ground atoms; if a program fulfills this requirement, is said to be *finitely recursive*.

Finitary programs are uncomparable and somehow complementary w.r.t. $\mathcal{FG}$ programs: indeed, the former class is conceived to allow decidable querying, having a top-down evaluation strategy in mind, while the latter supports the computability of answer-sets (by means of a bottom-up evaluation strategy). One of the main advantages of $\mathcal{FG}$ programs is that they can be "directly" evaluated by current ASP systems, which are based on a bottom-up computational model. However, there are also some interesting programs which are suitable for top-down query evaluation; but they do not fall in the class of $\mathcal{FG}$ programs.

In (Calimeri et al. 2009b) we focused on querying *disjunctive* finitely-recursive positive (DFRP) programs. These latter were not known to be decidable at the time of the work. We defined an appropriate magic-set rewriting technique, which turns a DFRP program P together with a query Q into a $\mathcal{FG}$ program.

Theorem 18 Given a ground query Q on a DFRP program P, let $M(P,Q)$ be the magic-set rewriting of P for Q. Then: (i) for both brave and cautious reasoning, $M(P,Q) \models Q$ iff $P \models Q$; (ii) $M(P,Q)$ is $\mathcal{FG}$ and hence computable; (iii) $M(P,Q)$ is at most linearly bigger than P.

Thus, reasoning on DFRP programs is decidable, and the theorem supplies an effective implementation strategy: (i) Rewrite the query by our magic-set technique, (ii) Evaluate the rewritten program with a solver supporting $\mathcal{FG}$ programs like DLV-Complex. It is worth noting that it is now possible to evaluate, by means of bottom-up systems, programs featuring "unsafe" variables in the head of rules, like non-ground facts, which are usual in Prolog but could not be handled by ASP solvers to date.

7 Related Work

Functional terms are widely used in logic formalisms stemming from first order logic. Introduction and treatment of functional terms (or similar constructs) have been studied indeed in several fields, such as Logic Programming and Deductive Databases. In the ASP community, the treatment of functional terms has recently received quite some attention (Syrjänen 2001; Bonatti 2004; Gebser, Schaub, and Thiele 2007; Lin and Wang 2008; Baselice, Bonatti, and Criscuolo 2009; Eiter and Simkus 2009; Lierler and Lifschitz 2009).

As a main complementary line of research we mention *finitary programs* and variants thereof as discussed in Section 6 Ground queries are decidable for both finitary and $\mathcal{FG}$ programs; however, for finitary programs, to obtain decidability one needs to additionally know ("a priori") what is the set of atoms involved in odd-cycles. We next discuss other proposals for introducing functional terms in ASP.

ω-restricted Programs (Syrjänen 2001), the earliest attempt of introducing function symbols under answer set semantics, have been implemented into the SMODELS system. The notion of ω-restricted program relies on the concept of ω-stratification; this essentially enforces that each variable appearing in a rule body also appears in a predicate belonging to a strictly lower stratum, for ω the uppermost layer containing all the unstratified portion of the program. This strong restriction allows to determine a finite domain which

a program can be grounded on. ω-restricted programs are strictly contained in $\mathcal{FD}$ (and $\mathcal{FG}$) programs.

In order to retain the decidability of standard reasoning tasks, rules in $\mathbb{FDNC}$ *programs*, and extensions thereof (Simkus and Eiter 2007; Eiter and Simkus 2009), must have a structure chosen among predefined syntactic shapes. This ensures that programs have a *forest-shaped model* property. Answer sets of $\mathbb{FDNC}$ programs are in general infinite, but have a finite representation which can be exploited for knowledge compilation and fast query answering. The class of $\mathbb{FDNC}$ programs is less expressive than both finitary and $\mathcal{FG}$ programs, and results to be incomparable from a syntactic viewpoint to both of them.

The idea of $\mathcal{FG}$ programs is also related to termination studies of SLD-resolution for Prolog programs (refer to (Schreye and Decorte 1994) as a starting point). Such works are based on the study of non-increasing trends in the size of complex terms, using several notions of measure (*norm*) for the size of terms. Such results are, however, not directly applicable to our context, given the operational nature of SLD resolution and its top-down flavor, which is complementary to our bottom-up approach.

8 Conclusions

This paper summarized our line of research on the introduction of functions in ASP. On the theoretical side, we have proven a number of decidability results for relevant program classes. On the practical side, we have designed effective implementation methods, and provided a powerful system, named DLV-Complex, implementing our results and supporting a rich ASP language with functions, lists, and sets. The system is already used in many universities and research institutes successfully.

9 Acknowledgments

Partially supported by the Regione Calabria and the EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP.

Baselice, S.; Bonatti, P. A.; and Criscuolo, G. 2009. On Finitely Recursive Programs. *TPLP* 9(2):213–238.

Bonatti, P. A. 2004. Reasoning with infinite stable models. *AI* 156(1):75–111.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2008. Computable Functions in ASP: Theory and Implementation. In *ICLP 2008*, 407–424.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2009a. An ASP System with Functions, Lists, and Sets. In *LPNMR'09*, 483–489.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. 2009b. Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In *LPNMR'09*, 71–86.

Calimeri, F.; Cozza, S.; Ianni, G.; and Leone, N. since 2008. `DLV-Complex` homepage. `http://www.mat.unical.it/dlv-complex`.

Eiter, T., and Simkus, M. 2009. Bidirectional Answer Set Programs with Function Symbols. In *IJCAI-09*, 765–771.

Friedrich, G., and Ivanchenko, V. 2008. Diagnosis from first principles for workflow executions. Technical report. `http://proserver3-iwas.uni-klu.ac.at/download\_area/Technical-Reports/technical\_report\_2008\_02.pdf`.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR'07*, 266–271.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *NGC* 9:365–385.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* 7(3):499–562.

Lierler, Y., and Lifschitz, V. 2009. One More Decidable Class of Finitely Ground Programs. In *ICLP'09*, 489–493.

Lin, F., and Wang, Y. 2008. Answer Set Programming with Functions. In *KR 2008*, 454–465.

Marek, V. W., and Truszczyński, M. 1999. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*. 375–398.

Schreye, D. D., and Decorte, S. 1994. Termination of Logic Programs: The Never-Ending Story. *JLP* 19/20:199–260.

Simkus, M., and Eiter, T. 2007. FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *LPAR 2007*, 514–530.

Syrjänen, T. 2001. Omega-restricted logic programs. In *LPNMR 2001*, 267–279.