Computable Functions in ASP: Theory and Implementation *

Francesco Calimeri, Susanna Cozza, 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. Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP), is a powerful formalism for knowledge representation and reasoning (KRR). The latest years witness an increasing effort for embedding functions in the context of ASP. Nevertheless, at present no ASP system allows for a reasonably unrestricted use of function terms. Functions are either required to be nonrecursive or subject to severe syntactic limitations, if allowed at all in ASP systems.
In this work we formally define the new class of finitely-ground programs, allowing for a powerful (possibly recursive) use of function terms in the full ASP language with disjunction and negation. We demonstrate that finitely-ground programs have nice computational properties: (i) both brave and cautious reasoning are decidable, and (ii) answer sets of finitely-ground programs are computable. Moreover, the language is highly expressive, as any computable function can be encoded by a finitely-ground program. Due to the high expressiveness, membership in the class of finitely-ground program is clearly not decidable (we prove that it is semi-decidable). We single out also a subset of finitely-ground programs, called finite-domain programs, which are effectively recognizable, while keeping computability of both reasoning and answer set computation.
We implement all results in DLV, further extending the language in order to support list and set terms, along with a rich library of built-in functions for their manipulation. The resulting ASP system is very powerful: any computable function can be encoded in a rich and fully declarative KRR language, ensuring termination on every finitely-ground program. In addition, termination is "a priori" guaranteed if the user asks for the finite-domain check.

1 Introduction

Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP) [1, 10, 11, 13, 15], evolved significantly during the last decade, and has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. Several systems supporting ASP have been implemented so far, thereby encouraging a number of applications in many real-world contexts ranging, e.g., from information integration, to frauds detection, to software configuration, and many others. On the one hand, the above mentioned applications have confirmed the viability of the exploitation of ASP for advanced knowledge-based tasks. On the other hand, they have evidenced some limitations of ASP languages and systems, that should be overcome to make ASP better suited for real-world applications even in industry. One of the most noticeable limitations is the fact that complex terms like functions, sets and lists, are not adequately supported by current ASP languages/systems. Therefore, even by using state-of-the-art systems, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. This is a strong limitation, both for standard knowledge-based tasks and for emerging applications, such as those manipulating XML documents.

* The material in the appendices is complementary, and will be skipped in the final version. It has been included also to ease the work of the referees.

The strong need to extend DLP by functions is clearly perceived in the ASP community, and many relevant contributions have been recently done in this direction [2, 19, 14, 21, 5]. However, we still miss a proposal which is fully satisfactory from a linguistic viewpoint (high expressiveness) and suited to be incorporated in the existing ASP systems. Indeed, at present no ASP system allows for a reasonably unrestricted use of function terms. Functions are either required to be nonrecursive or subject to severe syntactic limitations, if allowed at all in ASP systems.

This paper aims at overcoming the above mentioned limitations, toward a powerful enhancement of ASP systems by functions. The contribution is both theoretical and practical, and leads to the implementation of a powerful ASP system supporting (recursive) functions, sets, and lists, along with libraries for their manipulations. The main results can be summarized as follows:

- We formally define the new class of *finitely-ground ($\mathcal{FG}$)* DLP programs. This class allows for (possibly recursive) function symbols, disjunction and negation. We demonstrate that $\mathcal{FG}$ programs enjoy many relevant computational properties:
  - both brave and cautious reasoning are computable, even for non-ground queries;
  - answer sets are computable;
  - each computable function can be expressed by a $\mathcal{FG}$ program.
- Since $\mathcal{FG}$ programs express any computable function, membership in this class is obviously not decidable (we prove that it is semi-decidable). For users/applications where termination needs to be "a priori" guaranteed, we define the class of *finite-domain ($\mathcal{FD}$)* programs:
  - both reasoning and answer set generation are computable for $\mathcal{FD}$ programs (they are a subclass of $\mathcal{FG}$ programs), and, in addition,
  - recognizing whether a program is an $\mathcal{FD}$ program is decidable.
- We extend the language with list and set terms, along with a rich library of built-in functions for lists and sets manipulations.
- We implement all results and the full (extended) language in DLV, obtaining a very powerful system where the user can exploit the full expressiveness of $\mathcal{FG}$ programs (able to encode any computable function), or require the finite-domain check, getting the guarantee of termination. The system is available for downloading [9], and already in use in many universities and research centers throughout the world.

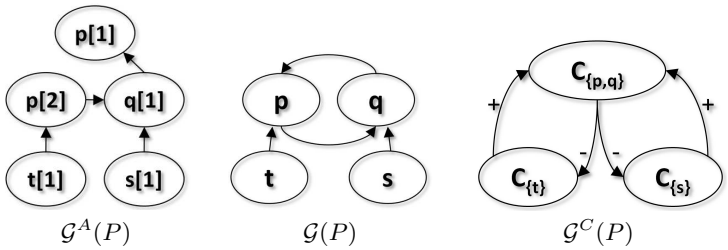
For space limitations, we cannot include detailed proofs. Further documentation and examples are available on the web site [9].

2 DLP with Functions

This section reports the formal specification of the DLP language with function symbols allowed.

2.1 Syntax and notations A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \ldots t_n$ are terms and f is a function symbol (*functor*) of arity n, then: $f(t_1, \ldots, t_n)$ is a *functional term*. We say that each t_i, $1 \leq i \leq n$, is a subterm of $f(t_1, \ldots, t_n)$. The subterm relation is reflexive and transitive, that is: (i) each term is also a subterm of itself; and (ii) if t_1 is a subterm of t_2 and t_2 is subterm of t_3 then t_1 is also a subterm of t_3.

Each predicate p has a fixed arity $k \geq 0$; by $p[i]$ we denote its i-th argument. If $t_1, \ldots, t_k$ are terms, then $p(t_1, \ldots, t_k)$ is an *atom*. A *literal* l is of the form a or `not` a, where a is an atom; in the former case l is *positive*, and in the latter case *negative*. A *rule* r is of the form $\alpha_1 \vee \cdots \vee \alpha_k$ `:-` $\beta_1, \ldots, \beta_n$, `not` $\beta_{n+1}, \ldots,$ `not` β_m. where $m \geq 0$, $k \geq 0$; $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_m$ are atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ (the *head* of r) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of r), where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ (the *positive body* of r) and $B^-(r)$



$\mathcal{G}^A(P)$ $\mathcal{G}(P)$ $\mathcal{G}^C(P)$

Fig. 1. *Argument*, *Dependency* and *Component* Graphs of the program in Example 1.

= $\{\texttt{not}\ \beta_{n+1}, \ldots, \texttt{not}\ \beta_m\}$ (the *negative body* of r). If $H(r) = \emptyset$ then r is a *constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then r is referred to as a *fact*.

A rule is safe if each variable in that rule also appears in at least one positive literal in the body of that rule. For instance, the rule $p(X, f(Y, Z))$:- $q(Y), \texttt{not}\ s(X)$. is not safe, because of both X and Z. From now on we assume that all rules are safe and there is no constraint.[1] A DLP program is a finite set P of rules. As usual, a program (a rule, a literal) is said to be *ground* if it contains no variables. Let A be a set of atoms and p be a predicate. Given a program P, according with the database terminology, a predicate occurring only in facts is referred to as an *EDB* predicate, all others as *IDB* predicates. The set of all facts of P is denoted by Facts(P); the set of instances of all EDB predicates is denoted by EDB(P) (note that EDB(P) $\subseteq$ Facts(P)). The set of all head atoms in P is denoted by $Heads(P) = \bigcup_{r \in P} H(r)$.

2.2 Semantics The most widely accepted semantics for DLP programs is based on the notion of answer-set, proposed by Gelfond and Lifschitz in [11] as a generalization of the concept of stable model [10].

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 standard 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. Given a program P 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* [11] 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 negated literal that is false w.r.t. I; (ii) deleting all negated literals from the remaining rules. $I \subseteq B_P$ is an *answer set* for a program P iff I is a minimal model for the positive program P^I. The set of all answer sets for P is denoted by $AS(P)$.

2.3 Dependency Graphs We next define three graphs that point out dependencies among arguments, predicates, and components of a program.

[1] Under Answer Set semantics, a constraint :- B(r) can be simulated through the introduction of a standard rule fail :- B(r), not fail, where fail is a fresh predicate not occurring elsewhere in the program.

Definition 1. The *Argument Graph* $\mathcal{G}^A(P)$ of a program P contains a node for each argument $p[i]$ of a predicate p of P; there is an edge $(q[j], p[i])$ if there is a rule $r \in P$ such that: (a) an atom $p(\overline{t})$ appears in the head of r; (b) an atom $q(\overline{v})$ appears in the positive body of r; (c) $p(\overline{t})$ and $q(\overline{v})$ share the same variable within the i-th and j-th term, respectively.

Given a program P, an argument $p[i]$ is said to be recursive with $q[j]$ if there exists a cycle in $\mathcal{G}^A(P)$ involving both $p[i]$ and $q[j]$. Roughly speaking, this graph keeps track of (body-head) dependencies between the arguments of predicates sharing some variable. It is actually a more detailed version of the commonly used (predicate) dependency graph, defined below.

Definition 2. The *Dependency Graph* $\mathcal{G}(P)$ of P is a directed graph whose nodes are the IDB predicates appearing in P. There is an edge (p_2, p_1) in $\mathcal{G}(P)$ iff there is some rule r with p_2 appearing in $B^+(r)$ and p_1 in $H(r)$, respectively.

The graph $\mathcal{G}(P)$ suggests to split the set of all predicates of P into a number of sets (called components), one for each strongly connected component (SCC)[2] of the graph itself. Given a predicate p, the component p belongs to is denoted by $comp(p)$; with a small abuse of notation, we define also $comp(l)$ and $comp(a)$, where l is a literal and a is an atom, accordingly.

In order to single out dependencies among components, a proper graph is defined next.

Definition 3. Given a program P and its Dependency Graph $\mathcal{G}(P)$, the *Component Graph* of P, denoted $\mathcal{G}^C(P)$, is a directed labelled graph having a node for each strongly connected component of $\mathcal{G}(P)$ and: (i) an edge (B, A), labelled "+", if there is a rule r in P such that there is a predicate $q \in A$ occurring in the head of r and a predicate $p \in B$ in a positive literal of the body of r; (ii) an edge (B, A), labelled "-", if there is a rule r in P such that there is a predicate $q \in A$ occurring in the head of r and a predicate $p \in B$ occurring in the negative the body of r, and there is no edge (B, A), with label "+".

Example 1. Consider the following program P, where a is an EDB predicate:

$q(g(3)).$ $\quad$ $s(X) \vee t(f(X))$:- $a(X)$, not $q(X)$.
$p(X, Y)$:- $q(g(X))$, $t(f(Y))$. $\quad$ $q(X)$:- $s(X)$, $p(Y, X)$.

Graphs $\mathcal{G}^A(P)$, $\mathcal{G}(P)$ and $\mathcal{G}^C(P)$ are respectively depicted in Figure 1. There are three SCC in $\mathcal{G}(P)$: $C_{\{s\}} = \{s\}$, $C_{\{t\}} = \{t\}$ and $C_{\{p,q\}} = \{p, q\}$ which are the three nodes of $\mathcal{G}^C(P)$.

An ordering among the rules, respecting dependencies pointed out by $\mathcal{G}^C(P)$, is defined next.

Definition 4. A path in $\mathcal{G}^C(P)$ is named *strong* if all its edges are labelled with "+". If, on the contrary, there is at least an edge in the path labelled with "-", the path is said to be *weak*. A *component ordering* for a given program P is a total ordering $\langle C_1, \ldots, C_n \rangle$ of all components of P s.t., for any C_i, C_j with $i < j$, both the following conditions hold: (i) there are no strong paths from C_j to C_i; (ii) if there is a weak path from C_j to C_i, then there must be a weak path also from C_i to C_j.[3]

Example 2. Consider the graph $\mathcal{G}^C(P)$ of previous example. Both $C_{\{s\}}$ and $C_{\{t\}}$ are connected to $C_{\{p,q\}}$ through a strong path, while a weak path connects: $C_{\{s\}}$ to $C_{\{t\}}$, $C_{\{t\}}$ to $C_{\{s\}}$, $C_{\{p,q\}}$ to $C_{\{s\}}$ and $C_{\{p,q\}}$ to $C_{\{t\}}$. Both $\gamma_1 = \langle C_{\{s\}}, C_{\{t\}}, C_{\{p,q\}} \rangle$ and $\gamma_2 = \langle C_{\{t\}}, C_{\{s\}}, C_{\{p,q\}} \rangle$ constitute component orderings for the program P.

[2] We recall here that a strongly connected component of a directed graph is a maximal subset S of the vertices, such that each vertex in S is reachable from all other vertices in S.

[3] Note that, given the component ordering γ, C_i stands for the i-th component in γ, and $C_i < C_j$ means that C_i precedes C_j in γ (i.e., $i < j$).

By means of the graphs defined above, it is possible to identify a set of subprograms (also called *modules*) of P, allowing for a modular bottom-up evaluation. We say that a rule $r \in P$ *defines* a predicate p if p appears in $H(r)$. Once a component ordering $\gamma = \langle C_1, \ldots, C_n \rangle$ is given, for each component C_i we define the *module* of C_i, denoted by P(C_i), as the set of all rules r defining some predicate $p \in C_i$ excepting those that define also some other predicate belonging to a lower component (i.e., certain C_j with $j < i$ in γ).

Example 3. Consider the program P of Example 1. If we consider the component ordering γ_1, the corresponding modules are:

$P(C_{\{s\}}) = \{\ s(X) \vee t(f(X))\ \texttt{:-}\ a(X),\ \texttt{not}\ q(X).\ \},$ $\qquad P(C_{\{t\}}) = \emptyset,$

$P(C_{\{p,q\}}) = \{p(X,Y)\ \texttt{:-}\ q(g(X)),\ t(f(Y)).,\ q(X)\ \texttt{:-}\ s(X),\ p(Y,X).,\ q(g(3)).\ \}.$

The modules of P are defined, according to a component ordering γ, with the aim of properly instantiating all rules. It is worth remembering that we deal only with safe rules, i.e., all variables appear in the positive body; it is therefore enough to instantiate the positive body. Furthermore, any component ordering γ guarantees that, when $r \in P(C_i)$ is instantiated, each nonrecursive predicate p appearing in $B^+(r)$ is defined in a lower component (i.e., in some C_j with $j < i$ in γ). It is also worth remembering that, according to how the modules of P are defined, if r is a disjunctive rule, then it is associated only to a unique module $P(C_i)$, chosen in such a way that, among all components C_j such that $comp(a) = C_j$ for some $a \in H(r)$, it always holds $i \leq j$ in γ (that is, the disjunctive rule is associated only to the (unique) module corresponding to the lowest component among those "covering" all predicates featuring some instance in the head of r). This implies that the set of the modules of P constitute an exact partition for it.

3 Finitely-Ground Programs

In this section we introduce a subclass of DLP programs, namely finitely-ground ($\mathcal{FG}$) programs, having some nice computational properties.

Since the ground instances of a rule might be infinite (because of the presence of function symbols), it is crucial to try to identify those that really matters in order to compute answer sets. Supposing that S contains all atoms that are potentially true, next definition singles out the relevant instances of a rule.

Definition 5. 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 4. Consider the following program P:

$t(f(1)).$ $\qquad t(f(f(1))).$ $\qquad p(1).$ $\qquad p(f(X))\ \texttt{:-}\ p(X),\ t(f(X))).$

The set $Inst_P(S)$ of all S-restricted instances of P, w.r.t. $S = Facts(P)$ is:

$t(f(1)).$ $\qquad t(f(f(1))).$ $\qquad p(1).$ $\qquad p(f(1))\ \texttt{:-}\ p(1),\ t(f(1)).$

The presence of negation allows for identifying some further rules which do not matter for the computation of answer sets, and for simplifying 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 6. 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.:
   - $l = a$ 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 5. Consider the following program P:

$$\begin{array}{lll} t(1). & s(1). & s(2). \\ q(X) \texttt{ :- } t(X). & p(X) \texttt{ :- } s(X), \texttt{not } q(X). & \end{array}$$

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 $S_1 = \{q(1) \texttt{ :- } t(1).\}$, 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 $S_2 = \{\ p(1) \texttt{ :- } s(1), \texttt{not } q(1).,\ p(2) \texttt{ :- } s(2), \texttt{not } q(2).\ \}$, 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 $q(2) \notin Heads(R)$; in addition, rule $p(1) \texttt{ :- } s(1), \texttt{not } q(1).$ 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 those 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 a further simplification among these rules by means of the $Simpl$ operator.

Definition 7. 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 of Example 1 where the extension of EDB predicate a is $\{a(1)\}$. Considering the component $C_1 = \{s\}$, the module $M = P(C_1)$, and the sets $X = \emptyset$ and $R = \{a(1)\}$, we have:

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

The operator defined above has the next important property.

Proposition 1. $\Phi_{M,R}$ always admits a least fixpoint $\Phi^{\infty}_{M,R}(\emptyset)$.

Proof. (Sketch) The statement follows from Tarski's theorem [22]), noting that $\Phi_{M,R}$ is a monotonic operator and that a set of rules forms a meet semilattice under set containment. □

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.t. $S_0 = EDB(P)$, $S_i = S_{i-1} \cup \Phi^{\infty}_{M_i, S_{i-1}}(\emptyset)$, where M_i is the program module $P(C_i)$.

Example 7. Let P be the program of Example 1 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:

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

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

$a(1).$	$q(g(3)).$	$s(1) \vee t(f(1))$:- not $q(1).$
$p(3,1)$:- $q(g(3)), t(f(1)).$	$q(1)$:- $s(1), p(3,1).$	

We are now ready to define the class of $\mathcal{FG}$ programs.

Definition 9. A program P is *finitely-ground* ($\mathcal{FG}$) if P^{γ} is finite, for every component ordering γ for P.

Example 8. The program of Example 1 is $\mathcal{FG}$: P^{γ} is finite both when $\gamma = \langle C_{\{s\}}, C_{\{t\}}, C_{\{p,q\}}\rangle$ and when $\gamma = \langle C_{\{t\}}, C_{\{s\}}, C_{\{p,q\}}\rangle$ (i.e., for both the only two component orderings for P).

4 Properties of Finitely-Ground Programs

In this section the class of $\mathcal{FG}$ programs is characterized by identifying some key properties.

The next theorem shows that we can compute the answer sets of an $\mathcal{FG}$ program by considering intelligent instantiations, instead of the theoretical (possibly infinite) ground program.

Theorem 1. Let P be an $\mathcal{FG}$ program and P^{γ} be the intelligent instantiation of P w.r.t. a component ordering γ for P. Then, $AS(P) = AS(P^{\gamma})$ (i.e., P and P^{γ} have the same answer sets).

Proof. (Sketch) Given $\gamma = \langle C_1, \ldots, C_n\rangle$, let denote, as usual, by M_i the program module $P(C_i)$, and consider the sets $S_0, \ldots, S_n$ as defined in Definition 8. Since $P = \bigcup_{i=0}^{n} M_i$ the theorem can be proven by showing that:

$$AS(S_k) = AS(\ \textstyle\bigcup_{i=0}^{k} M_i\ ) \text{ for } 1 \leq k \leq n$$

where M_0 denotes $EDB(P)$. The equation clearly holds for $k = 0$. Assuming that it holds for all $k \leq j$, we can show that it holds for $k = j + 1$. The equation above can be rewritten as:

$$AS(S_{k-1} \cup \Phi^{\infty}_{M_k,S_{k-1}}(\emptyset)) = AS(\ \textstyle\bigcup_{i=0}^{k-1} M_i\ \cup M_k)\ ) \text{ for } 1 \leq k \leq n$$

The induction hypothesis allows us to assume that the equivalence $AS(S_{k-1}) = AS(\ \bigcup_{i=0}^{k-1} M_i)$ holds. A careful analysis is needed of the impact that the addition of M_k to $\bigcup_{i=0}^{k-1} M_i$ has on answer sets of S_k; in order to prove the theorem, it is enough to show that the set $\Phi^{\infty}_{M_k,S_{k-1}}(\emptyset)$ does not drop any "meaningful" rule w.r.t. M_k.

If we disregard the application of the $Simpl$ operator, i.e. we consider the operator Φ performing only $Inst_{M_k}\ (Heads(S_{k-1} \cup \emptyset))$, then $\Phi^{\infty}_{M_k,S_{k-1}}(\emptyset)$ clearly generates all rules having a chance to have a true body in any answer set; omitted rules have a false body in every answer set, and are therefore irrelevant.

The application of $Simpl$ does not change the scenario: it relies only on previously derived facts, and on the absence of atoms from heads of previously derived ground rules.[4] If a fact q has been derived in a previous component, then any rule featuring q in the head or `not` q in the body is deleted, as it is already satisfied and cannot contribute to any answer set. The simplification operator also drops, from the bodies, positive atoms of lower components appearing as facts, as well as negative atoms belonging to lower components which do not appear in the head of any already generated ground rule. The presence of facts in the bodies is obviously irrelevant, and the deleted negative atoms are irrelevant as well. Indeed, by construction of the component dependency graph, while instantiating a module, all rules defining atoms of lower components have been already instantiated. Thus, atoms of lower components not appearing in the head of any generated rule, have no chances to be true in any answer set. □

Corollary 1. Every answer set of an $\mathcal{FG}$ program is finite.

Theorem 2. Given an $\mathcal{FG}$ program P, $AS(P)$ is computable.

Proof. Note that by Theorem 1, answer sets of P can be obtained by computing the answer sets of P^γ for a component ordering γ of choice, which can be easily computed. Then, P^γ can be obtained by computing the sequence of fixpoints of Φ specified in Definition 8. Each fixpoint is guaranteed to be finitely computable, since the program is finitely-ground. □

From this property, the main result below immediately follows.

Theorem 3. Cautious and brave reasoning over $\mathcal{FG}$ programs are computable. Computability holds even for non-ground queries.

As the next theorem shows, the class of $\mathcal{FG}$ programs allows for the encoding of any computable function.

Theorem 4. Given a recursive function f, there exists a DLP program P_f such that, for any input x for f, $P_f \cup \theta(x)$ is finitely-ground and $AS(P_f \cup \theta(x))$ encodes $f(x)$, for θ a simple function encoding x by a set of facts.

Proof. (Sketch) We can build a positive program P_f, which encodes the Turing machine M_f corresponding to f (see Appendix A). For any input x to M_f, $(P_f \cup \theta(x))^\gamma$ is finite for any component ordering γ, and $AS(P_f \cup \theta(x))$ contains an appropriate encoding of $f(x)$. □

Note that recognizing $\mathcal{FG}$ programs is semi-decidable, yet not decidable:

Theorem 5. Recognizing whether P is an $\mathcal{FG}$ program is R.E.-complete.

Proof. (Sketch) Semi-decidability is shown by implementing an algorithm evaluating the sequence given in Definition 8, and answering "yes" if the sequence converges in finite time.

On the other hand, given a Turing machine M and an input tape x, it is possible to write a corresponding program P_M and a set $\theta(x)$ of facts encoding x, such that M halts on input x iff $P_M \cup \theta(x)$ is finitely-ground. The program P_M is the same as that in the proof of Theorem 4 and reported in Appendix A. □

[4] Note that, due to the elimination of true literals performed by the simplification operator $Simpl$, the intelligent instantiation of a rule with a non empty body may generate some facts.

5 Finite-Domain Programs

In this section we single out a subclass of $\mathcal{FG}$ programs, called finite-domain ($\mathcal{FD}$) programs, which ensures the decidability of recognizing membership in the class.

Definition 10. Given a program P, the set of *finite-domain arguments ($\mathcal{FD}$ arguments)* of P is the maximal (w.r.t. inclusion) set $FD(P)$ of arguments of P such that, for each each argument $q[k] \in FD(P)$, every rule r with head predicate q satisfies the following condition. Let t be the term corresponding to argument $q[k]$ in the head of r. Then,

1. either t is variable-free, or
2. t is a subterm [5] of (the term of) some $\mathcal{FD}$ argument of a positive body predicate, or
3. every variable appearing in t also appears in (the term of) a $\mathcal{FD}$ argument of a positive body predicate which is not recursive with $q[k]$.

If all arguments of the predicates of P are $\mathcal{FD}$, then P is said to be an *$\mathcal{FD}$ program*.

Observe that $FD(P)$ is well-defined; indeed, it is easy to see that there always exists, and it is unique, a maximal set satisfying Definition 10 (trivially, given two sets A_1 and A_2 of $\mathcal{FD}$ arguments for a program P, the set $A_1 \cup A_2$ is also a set of $\mathcal{FD}$ arguments for P).

Example 9. The following is an example of $\mathcal{FD}$ program:

$$q(f(0)). \qquad q(X) \text{ :- } q(f(X)).$$

Indeed q[1] is the only argument in the program and it is an $\mathcal{FD}$ argument since the two occurrences of $q[1]$ in a rule head satisfy first and second condition of Definition 10 respectively.

Example 10. The following is not an $\mathcal{FD}$ program:

$$\begin{array}{ll} q(f(0)). & q(X) \text{ :- } q(f(X)). \\ s(f(X)) \text{ :- } s(X). & v(X) \text{ :- } q(X),\ s(X). \end{array}$$

We have that all arguments belong to $FD(P)$, except for $s[1]$. Indeed, $s[1]$ appears as head argument in the third rule with term $f(X)$, and: (i) $f(X)$ is not variable-free; (ii) $f(X)$ is not a subterm of some term appearing in a positive body $\mathcal{FD}$ argument; (iii) there is no positive body predicate which is not recursive with s and contains X.

By the following theorems we now point out two key properties of $\mathcal{FD}$ programs.

Theorem 6. Recognizing whether P is an $\mathcal{FD}$ program is decidable.

Proof. (Sketch) An algorithm deciding whether P is $\mathcal{FD}$ or not can be defined as follows. Arguments of predicates in P are all supposed to be $\mathcal{FD}$ at first. If at least one rule is found, such that for an argument of an head predicate none of the three conditions of Definition 10 holds, then P is recognized as not being an $\mathcal{FD}$ program. If no such rule is found, the answer is positive. □

Theorem 7. Every $\mathcal{FD}$ program is an $\mathcal{FG}$ program.

[5] The condition can be made less strict considering other notions, as, e.g., the *norm* of a term [6, 7, 18].

Proof. (*Sketch*) Given an $\mathcal{FD}$ program P, it is possible to find *a priori* an upper bound for the maximum nesting level[6] of the terms appearing in P^γ, for any component ordering γ for P. This is given by $max\_nl = (n+1) * m$, where m is the maximum nesting level of the terms in P, and n is the number of components in γ. Indeed, given that P is an $\mathcal{FD}$ program, it is easy to see that the maximum nesting level cannot increase because of recursive rules, since, in this case, the second condition of Definition 10 forces a sub-term relationships between head and body predicates. Hence, the maximum nesting level can increase only because of body-head dependencies among predicates of different components. We can now compute the set of all possible ground terms t obtained by combining all constants and function symbols appearing in P, such that the nesting level of t is less or equal to max_nl. This is a finite set, and clearly a superset of the ground terms appearing in P^γ. Thus, P^γ is necessarily finite. □

The results above allow us to state the following properties for $\mathcal{FD}$ programs.

Corollary 2. Let P be an $\mathcal{FD}$ program, then:

1. $AS(P)$ is computable;
2. every answer set in $AS(P)$ is finite;
3. skeptical and credulous reasoning over P are computable. Computability holds even if the query at hand is not ground.

6 An ASP System with functions, sets, and lists

In this section we briefly illustrate the implementation of an ASP system supporting the language herein presented. Such system actually features an even richer language, that, besides functions, explicitly supports also complex terms such as lists and sets, and provides a large library of built-in predicates for facilitating their manipulation. Thanks to such extensions, the resulting language becomes even more suitable for easy and compact knowledge representation tasks.

6.1 Language We next informally point out the peculiar features of the fully extended language, with the help of some sample programs.

In addition to simple and functional terms, there might be also *list* and *set* terms; a term which is not simple is said to be *complex*. A list term can be of two different forms: (i) $[t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are terms; (ii) $[h|t]$, where h (the head of the list) is a term, and t (the tail of the list) is a list term. Examples for list terms are: $[jan, feb, mar, apr, may, jun]$, $[jan \mid [feb, mar, apr, may, jun]]$, $[[jan, 31] \mid [[feb, 28], [mar, 31], [apr, 30], [may, 31], [jun, 30]]]$.

Set terms are used to model collections of data having the usual properties associated with the mathematical notion of set. They satisfy idempotence (i.e., sets have no duplicate elements) and commutativity (i.e., two collections having the same elements but with a different order represent the same set) properties. A *set term* is of the form: $\{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ are ground terms. Examples for set terms are: $\{red, green, blue\}$, $\{[red, 5], [blue, 3], [green, 4]\}$, $\{\{red, green\}, \{red, blue\}, \{green, blue\}\}$. Note that duplicated elements are ignored, thus the sets: $\{red, green, blue\}$ and $\{green, red, blue, green\}$ are actually considered as the same.

As already mentioned, in order to easily handle list and set terms, a rich set of built-in functions and predicates is provided. Functional terms prefixed by a # symbol are *built-in* functions. Such kind of functional terms are supposed to be substituted by the values resulting from the application of a functor to its arguments, according to some predefined semantics. For this reason,

[6] The nesting level of a ground term is defined inductively as follows: (i) a constant term has nesting level zero; (ii) a functional term $f(t_1, \ldots, t_n)$ has nesting level equal to the maximum nesting level among $t_1, \ldots, t_n$ plus one.

built-in functions are also referred to as *interpreted* functions. Atoms prefixed by # are, instead, instances of *built-in* predicates. Such kind of atoms are evaluated as true or false by means of operations performed on their arguments, according to some predefined semantics[7]. Some simple built-in predicates are also available, such as the comparative predicates equality, less-than, and greater-than $(=, <, >)$ and arithmetic predicates like successor, addition or multiplication, whose meaning is straightforward. A pair of simple examples about complex terms and proper manipulation functions follows. Another interesting example, i.e., the Hanoi Tower problem, is reported in Appendix B.

Example 11. Given a directed graph, a *simple path* is a sequence of nodes, each one appearing exactly once, such that from each one (but the last) there is an edge to the next in the sequence. The following program derives all simple paths for a directed graph, starting from a given $edge$ relation:

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

The first rule builds a simple path as a list of two nodes directly connected by an edge. The second rule constructs a new path adding an element to the list representing an existing path. The new element will be added only if there is an edge connecting it to the head of an already existing path. The external predicate $\#member$ (which is part of the above mentioned library for lists and sets manipulation) allows to avoid the insertion of an element that is already included in the list; without this check, the construction would never terminate in the presence of circular paths. Even if not an $\mathcal{FD}$ program, it is easy to see that this is an $\mathcal{FG}$ program; thus, the system is able to effectively compute the (in this case unique) answer set.

Example 12. Let us imagine that the administrator of a social network wants to increase the connections between users. In order to do that, (s)he decides to propose a connection to pairs of users that result, from their personal profile, to share more than two interests. If the data about users are given by means of EDB atoms of the form $user(id,\ \{interest_1, \ldots, interest_n\})$, the following rule would compute the set of common interests between all pairs of users:

$$sharedInterests(U_1, U_2, \#intersection(S_1, S_2)) \text{ :- } user(U_1, S_1), user(U_2, S_2), U_1 \neq U_2.$$

where the interpreted function $\#intersection$ takes as input two sets and returns their intersection. Then, the predicate selecting all pairs of users sharing more than two interests could be defined as follows:

$$proposeConnection(pair(U_1, U_2)) \text{ :- } sharedInterests(U_1, U_2, S), \#card(S) > 2.$$

Here, the interpreted function $\#card$ returns the cardinality of a given set, which is compared to the constant 2 by means of the built-in predicate "$>$".

6.2 Implementation The presented language has been implemented on top of the state-of-the-art ASP system DLV [12]. Complex terms have been implemented by using a couple of built-in predicates for packing and unpacking them (see below). These functions, along with the library for lists and sets manipulation have been incorporated in DLV by exploiting the framework introduced in [8].

In particular, support for complex terms is actually achieved by suitably rewriting the rules they appear in. The resulting rewritten program does not contain complex terms any more, but a number of instances of proper built-in predicates. We briefly illustrate in the following how the rewriting is performed in case of functional terms; the cases of list and set terms are treated analogously. Firstly, any functional term $t = f(X_1, \ldots, X_n)$, appearing in some rule $r \in P$, is replaced by a fresh variable Ft and then, one of the following atom is added to $B(r)$:

[7] The specification of the entire library for lists and sets manipulation is available at [9].

- $\#function\_pack(Ft, f, X_1, \ldots, X_n)$ if t appears in $H(r)$;
- $\#function\_unpack(Ft, f, X_1, \ldots, X_n)$ if t appears in $B(r)$.

This transformation is applied to the rule r until no functional terms appear in it. The role of an atom $\#function\_pack$ is to build a functional term starting from a functor and its arguments; while an atom $\#function\_unpack$ acts unfolding a functional term to give values to its arguments. So, the former binds the Ft variable, provided that all other terms are already bound, the latter binds (checks values, in case they are already bound) the $X_1, \ldots, X_n$ variables according to the binding for the Ft variable (the whole functional term).

Example 13. The rule: $p(f(f(X)))$:- $q(X, g(X, Y))$. will be rewritten as follow:

$$p(Ft_1) \text{ :- } \#function\_pack(Ft_1, f, Ft_2), \#function\_pack(Ft_2, f, X), \\ q(X, Ft_3), \#function\_unpack(Ft_3, g, X, Y).$$

Note that rewriting the nested functional term $f(f(X))$ requires two $\#function\_pack$ atoms in the body: (i) for the inner f function having X as argument and (ii) for the outer f function having as argument the fresh variable Ft_2, representing the inner functional term.

The resulting ASP system is indeed very powerful: the user can exploit the full expressiveness of $\mathcal{FG}$ programs (plus the ease given by the availability of complex terms), at the price of giving the guarantee of termination up. In this respect, it is worth stating that the system grounder fully complies with the definition of *intelligent* instantiation introduced in this work (see Section 3 and Definition 8). This implies, among other things, that the system is guaranteed to terminate and correctly compute all answer sets for any program resulting as finitely-ground. Nevertheless, the system comes equipped with a syntactic $\mathcal{FD}$ programs recognizer, based on the algorithm sketched in Theorem 6. This kind of finite-domain check, which is active by default, ensures computability for all accepted programs, without the need for knowing the membership to $\mathcal{FG}$ programs class.

The system prototype is available at [9]; besides the system itself, the above mentioned library for list and set terms manipulation is available for free download, together with a reference guide and a number of examples.

7 Related Works

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 [2, 19, 14, 21, 5]. We next focus on the main proposals for introducing functional terms in ASP.

Finitary Programs. Finitary programs [5, 2] are a major contribution to the introduction of recursive functional terms (and thus infinite domains) in logic programming under stable model semantics.

Given a normal (or-free) program P, a labelled dependency graph $LDG(P)$ is associated to $grnd(P)$. The nodes are the (infinite) atoms in B_P, there is an edge (A, B) (from A to B) if there is a rule $r \in grnd(P)$ such that $A \in H(r)$ and $B \in B(r)$; in particular, the edge is labelled $\neg$ if $B \in B^-(r)$. Program P is finitary if: ***(i)*** from any node in $LDG(P)$ only a finite sets of nodes is reachable (i.e., the program is ***finitely recursive***, as any atom depends only on a finite set of other atoms), and ***(ii)*** the dependency graph $LDG(P)$ has only a finite number of cycles with an odd number of negated ($\neg$) edges (called *odd-cycles*).

The class of finitary programs can be seen as a "dual" notion of the class of finitely-ground programs. The former is suitable for a top-down evaluation, while the latter allows for a bottom-up computation.

Comparing the computational properties of the two classes, we observe the following:

- Both finitary programs and finitely-ground programs can express any computable function.
- Ground queries are decidable for both finitary and finitely-ground 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 [3].
- Answer sets on finitely-ground programs are computable, while they are not computable on finitary programs. The same holds for nonground queries.
- Recognizing if a program is finitely-ground is semi-decidable; while recognizing if a program is finitary is undecidable.

Finitary and $\mathcal{FG}$ programs are not comparable: there are finitary programs that are not finitely-ground, and finitely-ground programs that are not finitary. The syntactic restrictions imposed by the two notions somehow come from the underlying computational approaches (top-down vs bottom-up). Finitary programs impose that all rule variables must occur in the head; while finitely-ground programs require that all rule variables occur in the positive body. Therefore, $p(X, Y)$:- $q(X)$. is safe for finitary programs, while it is not for finitely-ground programs (as Y is not range-restricted). On the contrary, $p(X, Y)$:- $q(X, V), r(V, Y)$ is safe for finitely-ground programs, while it is not admissible for finitary programs (because of the "local" variable V). Similarly, for the nesting level of the functions: it cannot increase head-to-body for finitary programs, while it cannot increase body-to-head for finitely-ground programs. For instance, $p(X)$:- $p(f(X))$. is not finitary, while $p(f(X))$:- $p(X)$. is not finitely-ground. Importantly, finitary programs are or-free; while finitely-ground programs allow for disjunctive rules. The class of finitary programs has been extended to the disjunctive case in [4]. To this end, a third condition on the disjunctive heads is added to the definition of finitary programs, in order to guarantee the decidability of ground querying.

Concluding, we observe that the bottom-up nature of the notion of finitely-ground programs allows for an immediate implementation of this class in ASP systems (as ASP instantiators are based on a bottom-up computational model). Indeed, we could enhance DLV to deal with finitely-ground by small changes in its instantiator, keeping the database optimization techniques which rely on the bottom-up model and significantly enhance the efficiency of the instantiation. While an ASP instantiator should be replaced by a top-down grounder to deal with finitary programs.

ω-restricted Programs. The class of ω-restricted logic programs [21] allows for function symbols under Answer Set semantics. It has been effectively implemented into SMODELS [20] - a very popular ASP system. The notion of ω-restricted program relies on the concept of ω-stratification. An ω-stratification corresponds, essentially, to a traditional stratification (i.e., a function mapping each predicate name to a *level* number) w.r.t. negation, extended by the (uppermost) ω-stratum, which contains all predicates depending negatively on each other (basically, this stratum contains entirely the unstratified part of the program). In order to avoid infiniteness/undecidability, programs must fulfill some syntactic conditions w.r.t. an ω-stratification. In particular, each variable appearing in a rule must also occur in a positive body literal belonging to a *strictly* lower stratum than the head. The above restrictions are strong enough to guarantee the computability of answer sets, yet losing recursive completeness. Thus, ω-restricted programs are strictly less expressive than both finitary and $\mathcal{FG}$ programs (which can express all computable functions). From a merely syntactic viewpoint, the class of ω-restricted programs is uncomparable with that of finitary programs, while it is strictly contained in the class of $\mathcal{FD}$ programs

(and thus, of $\mathcal{FG}$ programs). Indeed, if a program P is ω-restricted, then each variable appearing in a rule head fulfills Condition 3 of Definition 10 (thus, P is $\mathcal{FD}$). On the contrary, there are $\mathcal{FD}$ programs that are not ω-restricted: for instance, the $\mathcal{FD}$ program made of the single rule $p(X)$:- $p(f(X))$ is $\mathcal{FD}$ but it is not ω-restricted.

$\mathbb{FDNC}$ *programs.* In [19] the class of $\mathbb{FDNC}$ programs is introduced, which allows for function symbols in DLP programs. In order to retain the decidability of the standard reasoning tasks, the structure of any rule must be chosen among one out of seven predefined forms. These syntactic restrictions ensure 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 finitely-ground programs. From a syntactic viewpoint, $\mathbb{FDNC}$ programs are uncomparable with both finitary and finitely-ground programs. Notably, $\mathbb{FDNC}$ programs are finitely recursive, but not necessarily finitary.

Other works. Recently, in [14], functions have been proposed as a tool for obtaining a more direct and compact representation of problems, and for improving the performance of ASP computation by reducing the size of resulting ground programs. The class of programs which is considered is strictly contained in ω-restricted programs: indeed, predicates as well as functions must range over finite domains, which must be explicitly (and extensively) provided.

The idea of $\mathcal{FG}$ programs is also related to termination studies of SLD-resolution for Prolog programs (see e.g. [18, 6, 7]). In this context, several notion of *norm* for complex terms were introduced. Intuitively, proving that norms of sub-goals are non-increasing during top-down evaluation ensures decidability of a given program. Note that such techniques can not be applied in a straightforward way to our setting, for a series of technical differences. First, propagation of norm information should be studied from rules bodies to heads while traditional termination analysis works the other way around. Also, top-down termination analysis often integrates right recursion avoidance techniques, which are not required in the context of ASP.

As for the the deductive database field, we recall that one of the first comprehensive proposals has been $\mathcal{LDL}$ [16], a declarative language featuring a non-disjunctive logic programming paradigm based on bottom-up model query evaluation. $\mathcal{LDL}$ provides a rich data model including the possibility to manage complex objects, lists and sets. The language allows for a stratified form of negation, while functional terms are managed by means of "infinite" base relations computed by external procedures; proper restrictions (called constraints) and checks based on structural properties of the program (interdependencies between arguments) ensure that a finite number of tuples are generated for each relation.

8 Conclusions

We have formally defined the class of $\mathcal{FG}$ programs, which allows for (possibly recursive) complex terms in the full ASP language (logic programs with disjunction and negation). We have proven that, for each program P in this class, there exists a finite subset P' of its instantiation having precisely the same answer sets as P. Importantly, such a subset P' is computable for $\mathcal{FG}$ programs. It turns out that: (i) both cautious and brave reasoning tasks are computable for finitely-ground, even if the query is not ground, (ii) the answer sets of the program are computable as well. We have also demonstrated that $\mathcal{FG}$ programs can express every computable function. We have singled out also a subclass of $\mathcal{FG}$ programs, called finite-domain programs, which are efficiently recognizable, while keeping the computability of the reasoning tasks. We have implemented all results in the the DLV system, further extending the language with list and

set terms, along with a rich library for their manipulation. The resulting system is very powerful: it combines the expressiveness of functions, sets, and lists, with the knowledge modeling features of ASP in a fully declarative framework. The system is available for downloading from [9], where the user can find also a manual and further examples. Many users already reported a very positive feedback.

Ongoing work focuses on the extensions of the classes of finitely-ground and finitely-domain programs and on their combinations with the notion of finitary and $\mathbb{FDNC}$ programs.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2003.
2. S. Baselice, P. A. Bonatti, and G. Criscuolo. On Finitely Recursive Programs. In *ICLP'07*, LNCS 4670, pp. 89–103. 2007.
3. P. A. Bonatti. Erratum to: "Reasoning with infinite stable models". *Artificial Intelligence*. Forthcoming.
4. P. A. Bonatti. Reasoning with infinite stable models II: Disjunctive programs. In *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, LNCS 2401, pp. 333–346. 2002.
5. P. A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.
6. A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124(2):297–328, 1994.
7. M. Bruynooghe, M. Codish, John P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM TOPLAS*, 29(2):10, 2007.
8. F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *AMAI*, 50(3–4):333–361, 2007.
9. F. Calimeri, S. Cozza, G. Ianni, and N. Leone. `DLV-Complex` homepage, since 2008. http://www.mat.unical.it/dlv-complex.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP/SLP 1988*, pp. 1070–1080, Cambridge, Mass., 1988. MIT Press.
11. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.
12. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, July 2006.
13. V. Lifschitz. Answer Set Planning. In *ICLP'99*, pp. 23–37, Las Cruces, New Mexico, USA, 1999.
14. F. Lin and Y. Wang. Answer Set Programming with Functions. In *KR 2008*, 2008. To appear.
15. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pp. 375–398. 1999.
16. S. Naqvi and S. Tsur. *A logical language for data and knowledge bases*. Computer Science Press, Inc., New York, USA, 1989.
17. C. Papadimitriou. *Computational Complexity*. 1994.
18. D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *JLP*, 19/20:199–260, 1994.
19. M. Simkus and T. Eiter. FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *LPAR 2007*, volume 4790 of *LNCS*, pp. 514–530. 2007.
20. P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, 2002.
21. T. Syrjänen. Omega-restricted logic programs. In *LPNMR 2001*, Vienna, Austria, 2001. Verlag.
22. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

A A DLP Program Simulating a Turing Machine

We next show how a Turing Machine can be encoded by a suitable DLP program simulating its computation. It is worth noting that this encoding is actually executable; it is available for download at [9], together with the system prototype. Let M be a Turing Machine ([17]) given by the 4-uple $\langle K, \Sigma, \delta, s_0 \rangle$, where K is a finite set of states, $s_0 \in K$ is the initial state, Σ is a finite set of symbols constituting the alphabet (with $\sqcup \notin \Sigma$ standing for the blank symbol), and $\delta : K \times \Sigma \rightarrow K \times \Sigma \times \{l, r, \lambda\}$ is the transition function describing the behavior of the machine. Given the current state and the current symbol, δ specifies the next state, the symbol to be overwritten on the current one, and the direction in which the cursor will move on the tape (l, r, λ standing for left, right, stay, respectively). Besides the initial state, there is another special state, which is called final state; the machine halts if the machine reaches this state at some point. Each configuration of M can be encoded in a program P_M by means of the following predicates.

- $tape(P, Sym, T)$: the tape position P stores the symbol Sym at time step T. For each time step, there is an instance of such predicate for every actually used position of the tape.
- $position(P, T)$: the head of M reads the position P on tape at time step T. $position$ has a single true ground instance for each time step.
- $state(St, T)$: at time step T M is in the state St. $state$ has a single true ground instance for each time step.

P_M encodes the transition function δ in the following way: For each St_c, Sym_c, St_n, Sym_n, D, such that $\delta(St_c, Sym_c) = (St_n, Sym_n, D)$ we add to PM a fact of the form $delta(St_c, Sym_c, St_n, Sym_n, D)$. The initial input is encoded by a proper set of facts describing all tape positions at the first time step (facts of the form $tape(P, Sym, 0)$), a fact of the form $state(s_0, 0)$, and a fact of the form $position(P, 0)$ where P is the initial position of the head. The rules defining the evolution of the machine configurations are reported next. For the sake of readability, we exploit some comparison built-ins, that could be easily simulated by means of suitable predicates.

(r_1) $position(P, s(T))$:- $position(s(P), T), state(St, T), tape(s(P), Sym, T), delta(St, Sym, \_, \_, l).$
(r_2) $position(s(P), s(T))$:- $position(P, T), state(St, T), tape(P, Sym, T), delta(St, Sym, \_, \_, r).$
(r_3) $position(P, s(T))$:- $position(P, T), state(St, T), tape(P, Sym, T), delta(St, Sym, \_, \_, \lambda).$
(r_4) $state(St1, s(T))$:- $position(P, T), state(St, T), tape(P, Sym, T), delta(St, Sym, St1, \_, \_).$
(r_5) $tape(P, Sym1, s(T))$:- $position(P, T), state(St, T), tape(P, Sym, T), delta(St, Sym, \_, Sym1, \_).$
(r_6) $tape(P, Sym, s(T))$:- $position(P1, T), tape(P, Sym, T), P \neq P1.$
(r_7) $tape(P, \sqcup, T)$:- $position(P, T), lastUsedPos(L, T), P > L.$
(r_8) $lastUsedPos(L, s(T))$:- $lastUsedPos(L, T), position(P, T), P \leq L.$
(r_9) $lastUsedPos(P, s(T))$:- $lastUsedPos(L, T), position(P, T), P > L.$

First three rules encode how the tape position changes according to the transition function; the fourth updates the state. Rule r_5 updates, for each time step, the current tape position with the new symbol to be stored, with rule r_6 stating that all other positions remain unchanged. Rules r_7, r_8, r_9 allow to manage the semi-infinite tape. Indeed, the whole tape is not explicitly encoded; rather, each tape position is initialized with a blank symbol when reached for the first time (moving right, the tape being limited at left).

Given a valid tape x encoded by means of a set X of facts of the form $tape(p, s, 0)$, one can show that the computation of $(P_M \cup X)^\gamma$ follows in one-to-one correspondence the computation of M on the tape x. γ is unique and contains a single component C having a corresponding module M. We have that $S_0 = EDB(P_M)$, and $S_1 = S_0 \cup \Phi^\infty_{M,S_0}(\emptyset)$. Let $\Phi(t) = \Phi^t_{M,S_0}(\emptyset)$. Then, the value of $\Phi(t)$ directly corresponds to the step t of M. It is easy to note that, at step $t+1$, $\Phi(t+1)$ can be larger than $\Phi(t)$ only if, at step t, $\Phi(t)$ contains an atom $state(st, t)$ for st not a final state. In such a case by means of rules r_1 through r_5, new atoms of form $position(p, sym, t+1), state(st, t+1), tape(p, sym, t+1)$ are added to $\Phi(t+1)$.

B Towers of Hanoi Example

We report next an $\mathcal{FG}$ program encoding the famous Towers of Hanoi puzzle. This program, as well as other examples, is available online at [9].

```
% ———————————————————— begin of logic program ————————————————————
#include ⟨ListAndSet⟩

%—— initial settings ——
number_of_moves(15).
largest_disc(4).
initial_state(towers([4,3,2,1],[ ],[ ])).
goal(towers([ ],[ ],[4,3,2,1])).
disc(X) :- largest_disc(X).
disc(X) :- disc(#succ(X)), X! = 0.
legalStack([ ]).
legalStack([T]) :- disc(T).
legalStack([T|[T1|S]]) :- legalStack([T1|S]), disc(T), T > T1.
% —— possible states ——
possible_state(0, towers(S1,S2,S3)) :- initial_state(towers(S1,S2,S3)).
possible_state(I, towers(S1,S2,S3)) :- possible_move(I,_,towers(S1,S2,S3)),
                                       legalStack(S1), legalStack(S2), legalStack(S3).
% —— possible moves ——
% from stack one to stack two.
possible_move(#succ(I), towers([X|S1],S2,S3), towers(S1,[X|S2],S3)) :-
                        possible_state(I, towers([X|S1],S2,S3)), legalMoveNumber(I), legalStack([X|S2]).
% from stack one to stack three.
possible_move(#succ(I), towers([X|S1],S2,S3), towers(S1,S2,[X|S3])) :-
                        possible_state(I, towers([X|S1],S2,S3)), legalMoveNumber(I), legalStack([X|S3]).
% from stack two to stack one.
possible_move(#succ(I), towers(S1,[X|S2],S3), towers([X|S1],S2,S3)) :-
                        possible_state(I, towers(S1,[X|S2],S3)), legalMoveNumber(I), legalStack([X|S1]).
% from stack two to stack three.
possible_move(#succ(I), towers(S1,[X|S2],S3), towers(S1,S2,[X|S3])) :-
                        possible_state(I, towers(S1,[X|S2],S3)), legalMoveNumber(I), legalStack([X|S3]).
% from stack three to stack one.
possible_move(#succ(I), towers(S1,S2,[X|S3]), towers([X|S1],S2,S3)) :-
                        possible_state(I, towers(S1,S2,[X|S3])), legalMoveNumber(I), legalStack([X|S1]).
% from stack three to stack two.
possible_move(#succ(I), towers(S1,S2,[X|S3]), towers(S1,[X|S2],S3)) :-
                        possible_state(I, towers(S1,S2,[X|S3])), legalMoveNumber(I), legalStack([X|S2]).
%—— actual moves ——
% a solution exists if and only if there is a "possible_move" leading to the goal.
% in this case, starting from the goal, we proceed backward to the initial state to single out the full set of moves.
move(I, towers(S1,S2,S3)) :- goal(towers(S1,S2,S3)), possible_state(I, towers(S1,S2,S3)).
move(I, towers(S1,S2,S3)) ∨ nomove(I, towers(S1,S2,S3)) :- move(#succ(I), towers(A1,A2,A3)),
                        possible_move(#succ(I), towers(S1,S2,S3), towers(A1,A2,A3)).

%—— precisely one move at each step ——
moveStepI(I) :- move(I,_).
:- legalMoveNumber(I), not moveStepI(I).
:- legalMoveNumber(I), move(I,T1), move(I,T2), T1! = T2.
legalMoveNumber(0).
legalMoveNumber(#succ(I)) :- legalMoveNumber(I), number_of_moves(J), I < J.
% ———————————————————— end of logic program ————————————————————
```

By invoking the system at the command line as follows:

$ ⟨DLV ∗ executable⟩ ⟨programfilename⟩ −fdnocheck −N = 15 −filter = move

the next (unique) answer set is output:

{move(15, towers([ ], [ ], [4, 3, 2, 1])), move(14, towers([ ], [4], [3, 2, 1])), move(13, towers([3], [4], [2, 1])),
move(12, towers([4, 3], [ ], [2, 1])), move(11, towers([4, 3], [2], [1])), move(10, towers([3], [2], [4, 1])),
move(9, towers([ ], [3, 2], [4, 1])), move(8, towers([ ], [4, 3, 2], [1])), move(7, towers([1], [4, 3, 2], [ ])),
move(6, towers([4, 1], [3, 2], [ ])), move(5, towers([4, 1], [2], [3])), move(4, towers([1], [2], [4, 3])),
move(3, towers([2, 1], [ ], [4, 3])), move(2, towers([2, 1], [4], [3])), move(1, towers([3, 2, 1], [4], [ ])),
move(0, towers([4, 3, 2, 1], [ ], [ ]))}