TLP **10** (4–6): 481–496, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000232

A decidable subclass of finitary programs

SABRINA BASELICE and PIERO A. BONATTI
Università di Napoli "Federico II", Italy

submitted 7 February 2010; revised 29 April 2010; accepted 14 May 2010

Abstract

Answer set programming—the most popular problem solving paradigm based on logic programs—has been recently extended to support uninterpreted function symbols (Syrjänen 2001; Bonatti 2004; Simkus and Eiter 2007; Gebser *et al.* 2007; Baselice *et al.* 2009; Calimeri *et al.* 2008). All of these approaches have some limitation. In this paper we propose a class of programs called FP2 that enjoys a different trade-off between expressiveness and complexity. FP2 is inspired by the extension of finitary normal programs with local variables introduced in (Bonatti 2004, Section 5). FP2 programs enjoy the following unique combination of properties: (i) the ability of expressing predicates with infinite extensions; (ii) full support for predicates with arbitrary arity; (iii) decidability of FP2 membership checking; (iv) decidability of skeptical and credulous stable model reasoning for call-safe queries. Odd cycles are supported by composing FP2 programs with argument restricted programs.

KEYWORDS: answer set programming with function symbols, infinite stable models, norms

1 Introduction

Answer set programming has become the most popular problem solving paradigm based on logic programs. It is founded on the stable model semantics (Gelfond and Lifschitz 1991) and supported by well-engineered implementations such as SMODELS (Niemelä and Simons 1997) and DLV (Eiter *et al.* 1997), just to name a few. Recent developments of the paradigm and its implementations include support for uninterpreted function symbols, pioneered by the work on finitary programs (Bonatti 2004; Bonatti 2008). These works gave rise to further developments, including argument restricted programs (Lierler and Lifschitz 2009) and FDNC programs (Simkus and Eiter 2007), that address three limitations of finitary programs: a restriction on the number of odd-cycles in the dependency graph; the undecidability of the class of finitary programs; the dependency of reasoning on the set of odd-cycles, for which there is currently no general algorithm (Bonatti 2008). The drawback of these approaches, in turn, is that either they cannot express predicates with infinite extensions such as the standard list and tree manipulation predicates (Calimeri *et al.* 2008), or they have to restrict predicate arity and rule structure in such a way that - roughly speaking - only models shaped like labelled trees can be characterized (Simkus and Eiter 2007).

In this paper we propose a class of programs called FP2 that enjoys a different trade-off between expressiveness and complexity. FP2 is inspired by U-bounded programs (the extension of finitary normal programs with bounded local variables

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

introduced in (Bonatti 2004, Section 5)). FP2 programs retain the ability of expressing predicates with infinite extensions, and fully support predicates with arbitrary arity; moreover, deciding whether a program belongs to FP2 is decidable, as well as skeptical and credulous stable model reasoning, provided that the query is *call-safe*. Odd cycles are supported by composing FP2 programs with argument restricted programs. q The paper is organized as follows. After some preliminaries on logic programming, in Section 3 we introduce *term comparison* relations based on a measure of term size called norm, and show how to compute those relations. In Section 4 we apply the term comparison relations to define *recursion patterns*, that is, distinguished sets of arguments whose size almost never increases during recursion; we prove that if a recursion pattern exists then acyclic recursion depth is bounded, and there can be no odd-cycles. Then, in Section 6 we define FP2 and show that a form of SLD resolution with loop checking called *acyclic derivations* suffice to compute a representative set of supports for each subgoal relevant to a given query. The output of this phase is a finite ground program that can be fed to an ASP solver to answer credulous and skeptical queries, in the same spirit as finitary programs. To re-introduce odd-cycles (and hence the ability to express constraints) in FP2, we show in Section 7 how to compose FP2 programs with argument restricted programs. Two sections, on related work and a final discussion, conclude the paper. Many proofs are omitted due to space limitations.

2 Preliminaries and notation

We assume the reader to be familiar with classical logic programming (Lloyd 1984). *(Normal) logic programs* are finite sets of rules $A \leftarrow L_1, ..., L_n$ ($n \geqslant 0$), where A is a logical atom and each L_i ($i = 1, ..., n$) is a *literal*, that is, either a logical atom B or a negated atom $\texttt{not}\ B$. If R is a rule with the above structure, then let $head(R) = A$ and $body(R) = \{L_1, ..., L_n\}$. Moreover, let $body^+(R)$ (respectively $body^-(R)$) be the set of all atoms B s.t. B (respectively $\texttt{not}\ B$) belongs to $body(R)$. For all predicate symbols p, a p-atom A is an atom whose predicate, denoted by $\mathsf{pred}(A)$, is p. Similarly a p-literal L is a literal whose predicate, denoted by $\mathsf{pred}(L)$, is p. The ground instantiation of a program P is denoted by $\mathsf{Ground}(P)$. A Herbrand model M of P is a *stable model* of P iff M is the least Herbrand model of P^M and P^M is the *Gelfond-Lifschitz transformation* of P (Gelfond and Lifschitz 1991), obtained from $\mathsf{Ground}(P)$ by (i) removing all rules R such that $body^-(R) \cap M \neq \emptyset$, and (ii) removing all negative literals from the body of the remaining rules. A *skeptical* consequence of a program P is a formula satisfied by all the stable models of P. A *credulous* consequence of P is a formula satisfied by at least one stable model of P.

The *atom dependency graph of a program P* is a labelled directed graph, denoted by $\mathsf{DG}_a(P)$, whose vertices are the ground atoms of P's language. Moreover,

(i) there exists an edge labelled '+' (called positive edge) from A to B iff for some rule $R \in \mathsf{Ground}(P)$, $A = \mathsf{head}(R)$ and $B \in body(R)$;
(ii) there exists an edge labelled '-' (called negative edge) from A to B iff for some rule $R \in \mathsf{Ground}(P)$, $A = \mathsf{head}(R)$ and $\texttt{not}\ B \in body(R)$.

An atom A *depends* on B if there is a directed path from A to B in $\mathsf{DG}_a(P)$. Similarly, the *predicate dependency graph of a program P* is a labelled directed graph, denoted

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

by $\mathsf{DG}_p(P)$, whose vertices are the predicate symbols of P's language. Edges are defined by analogy with the atom dependency graph. An *odd-cycle* is a cycle in an atom (resp. predicate) dependency graph with an odd number of negative edges. A ground atom (resp. a predicate symbol) is *odd-cyclic* if it occurs in an odd-cycle. Given a graph G, we denote by $\mathsf{SCC}(G)$ the set of all strongly connected components in G. We say that a rule R *is in a component* C of a predicate dependency graph if $\mathsf{pred}(\mathsf{head}(R))$ is a vertex in C.

3 Norms and term comparisons

Norms have been introduced for the static termination analysis of logic programs, see for example (Bossi *et al.* 1994; Genaim *et al.* 2002). Termination proofs require certain predicate arguments to decrease strictly during recursion; we admit cyclic programs, instead, and consider non-strict orderings. For all sets of (possibly nonground) terms t, let $|t|$ (the *norm* of t) be the number of variables and function symbols occurring in t (constants are regarded as 0-ary functions). Norms are extended to term sequences $\vec{t} = t_1, \ldots, t_n$ in the natural way: By $|t_1, \ldots, t_n|$ we denote $|t_1| + \cdots + |t_n|$. For all vectors of terms $\vec{t}$ and $\vec{u}$, define $\vec{t} \preccurlyeq \vec{u}$ (resp. $\vec{t} \prec \vec{u}$) iff for all grounding substitutions σ, $|\vec{t}\sigma| \leqslant |\vec{u}\sigma|$ (resp. $|\vec{t}\sigma| < |\vec{u}\sigma|$). Moreover, we write $\vec{t} \precsim \vec{u}$ iff $\vec{t}$ is *almost never larger than* $\vec{u}$, that is, there exist only finitely many (possibly no) grounding substitutions σ such that $|\vec{t}\sigma| > |\vec{u}\sigma|$. Note that $\vec{t} \prec \vec{u} \Rightarrow \vec{t} \preccurlyeq \vec{u}$ and $\vec{t} \preccurlyeq \vec{u} \Rightarrow \vec{t} \precsim \vec{u}$. Note also that the norm over term sequences and the three comparison relations are insensitive to permutations. More precisely, for all permutations $\vec{t}_1$ of $\vec{t}$, we have $|\vec{t}_1| = |\vec{t}|$; therefore, if $\prec$ is any of the relations $\preccurlyeq$, $\prec$, and $\precsim$, then for all $\vec{u}$, $\vec{t}_1 \prec \vec{u} \Leftrightarrow \vec{t} \prec \vec{u}$ and $\vec{u} \prec \vec{t}_1 \Leftrightarrow \vec{u} \prec \vec{t}$. All of these relations can be computed via simple variable occurrence counting.

Theorem 3.1
Let $\mathsf{NOcc}(s, \vec{t})$ denote the number of occurrences of symbol s in $\vec{t}$. For all (possibly nonground) term vectors $\vec{t}$ and $\vec{u}$,

1. $\vec{t} \prec \vec{u}$ iff $|\vec{t}| < |\vec{u}|$ and for all variables x, $\mathsf{NOcc}(x, \vec{t}) \leqslant \mathsf{NOcc}(x, \vec{u})$;
2. $\vec{t} \preccurlyeq \vec{u}$ iff $|\vec{t}| \leqslant |\vec{u}|$ and for all variables x, $\mathsf{NOcc}(x, \vec{t}) \leqslant \mathsf{NOcc}(x, \vec{u})$;
3. $\vec{t} \precsim \vec{u}$ iff either $\vec{t} \preccurlyeq \vec{u}$ or for all variables x, $\mathsf{NOcc}(x, \vec{t}) < \mathsf{NOcc}(x, \vec{u})$ [1].

Proof. It is easier to prove the contrapositive (which is equivalent).

1) First suppose that $|\vec{t}| \geqslant |\vec{u}|$. Then for all σ mapping all variables onto constants, we have $|\vec{t}| = |\vec{t}\sigma|$ and $|\vec{u}| = |\vec{u}\sigma|$. This implies $|\vec{t}\sigma| \geqslant |\vec{u}\sigma|$, and hence $\vec{t} \prec \vec{u}$ does not hold. Second, if for some variable x, $\mathsf{NOcc}(x, \vec{t}) > \mathsf{NOcc}(x, \vec{u})$, then there exists a σ mapping all variables but x onto constants, and mapping x on a term with size $> |u|$. It is not hard to see that $|\vec{t}\sigma| > |\vec{u}\sigma|$, and hence $\vec{t} \prec \vec{u}$ does not hold. This concludes the proof of point 1. The proof of points 2 and 3 is based on analogous arguments. □

[1] We cannot relax this condition. Indeed, let $\vec{t} = [X, Y, f(a)]$ and $\vec{u} = [X, Y, Y]$. It holds that $\mathsf{NOcc}(X, \vec{t}) \leqslant \mathsf{NOcc}(X, \vec{u})$ and $\mathsf{NOcc}(Y, \vec{t}) < \mathsf{NOcc}(Y, \vec{u})$. However, if we set $Y = a$, for infinitely many substitutions σ for X we have that $|[X, a, f(a)]\sigma| > |[X, a, a]\sigma|$.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Example 3.2
Clearly $X \prec f(X) \preccurlyeq g(X)$. Moreover, $f(X, g(a)) \precsim f(X, Y)$, because $|f(X, g(a))\sigma| < |f(X, Y)\sigma|$ holds whenever $|Y\sigma| > 2$ (for a finite program P, the set of terms with norm 1 or 2 is finite). Finally, $f(X)$ and $f(Y)$ are incomparable.

4 Restricting recursion and odd-cycles

In FP2 programs recursion and odd-cycles are restricted, by analogy with finitary programs. This is partly achieved by requiring that for some groups of predicate arguments, norms should not increase "too much" during recursion. Such groups of arguments are formalized via a suitable notion of argument *selection indexes*.

An *n2k-selection index* is a set of distinct integers $a = \{a_1, \ldots, a_k\}$ such that $1 \leqslant a_1 < a_2 < \ldots < a_k \leqslant n$. An *n-selection index* is any *n2k*-selection index. By $-a$ we denote the *complement* of an *n*-selection index a, that is, the set of integers between 1 and n that do not occur in a. A selection index can be *applied* to an atom to extract the corresponding arguments: for all atoms $A = p(t_1, \ldots, t_n)$ and *n2k*-selection indexes a, define $A[a] = t_{a_1}, \ldots, t_{a_k}$. Similarly, for all literals $L = \texttt{not}\ p(t_1, \ldots, t_n)$ and *n2k*-selection indexes a, define $L[a] = t_{a_1}, \ldots, t_{a_k}$.

In FP2 programs each predicate is associated by a selection index to a group of arguments whose size almost never increases during recursion. Formally, a *selection index mapping* for a program P is a function μ mapping each n-ary predicate symbol p in P on an n-selection index μ_p. With a slight abuse of notation, if p is the predicate occurring in an atom B then we abbreviate $B[\mu_p]$ with $B[\mu]$. Similarly, if L is a p-literal then $L[\mu]$ abbreviates $L[\mu_p]$.

We are only left to formalize two requirements: (i) the selected arguments should almost never increase during top-down evaluations, and (ii) there should be no odd-cycles, that in this context might be a symptom of the presence of infinitely many odd-cycles, thereby violating one of the essential properties of finitary programs. A preliminary notion is needed first: a selection index mapping π is *complete* for a n-ary predicate symbol p if π maps p on an *n2n*-selection index π_p.

Definition 4.1

- A rule R in a program P is *decreasing* w.r.t. a selection index mapping π for P iff for all literals $L \in \mathsf{body}(R)$ such that $\mathsf{pred}(L)$ and $\mathsf{pred}(\mathsf{head}(R))$ occur in the same strongly connected component of $\mathsf{DG}_p(P)$, $L[\pi] \prec \mathsf{head}(R)[\pi]$.
- A rule R in a program P is *almost never increasing* w.r.t. a selection index mapping π if, for all literals $L \in \mathsf{body}(R)$ s.t. $\mathsf{pred}(L)$ and $\mathsf{pred}(\mathsf{head}(R))$ occur in the same strongly connected component of $\mathsf{DG}_p(P)$, the following conditions hold:
  1. $L[\pi] \precsim \mathsf{head}(R)[\pi]$, and
  2. π is complete for $\mathsf{pred}(L)$ and $\mathsf{pred}(\mathsf{head}(R))$.

Definition 4.2
A *recursion pattern* π for a program P is a selection index mapping for P s.t. for each strongly connected component $C \in \mathsf{SCC}(\mathsf{DG}_p(P))$ at least one of the following conditions holds:

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

1. all $R \in C$ are decreasing w.r.t. π;
2. all $R \in C$ are almost never increasing w.r.t. π and C does not contain any odd-cycles.

As we anticipated, the existence of recursion patterns implies bounds on recursion depth and odd-cycle freedom.

Lemma 4.3
If a program P has a recursion pattern then all paths in $\mathsf{DG}_a(P)$ contain finitely many different atoms[2].

Lemma 4.4
If a normal program P has a recursion pattern then $\mathsf{DG}_a(P)$ is odd-cycle-free.

Example 4.5
Consider the classical program for appending lists:

$$append([\,], L, L). \qquad append([X|X_s], L, [X|Y_s]) \leftarrow append(X_s, L, Y_s).$$

Let μ be a selection index mapping for this program. If $\mu_{append} = \{1\}$ then μ is a recursion pattern; indeed, the first rule is vacuously decreasing because it has an empty body, and the second rule is decreasing because the selected argument is decreasing: $X_s \prec [X|X_s]$. Similarly, if $\mu_{append} = \{3\}$ then μ is a recursion pattern. On the contrary $\mu_{append} = \{2\}$ does not yield a recursion pattern; we have $L \precsim L$, but μ_{append} is not complete. Finally, $\mu_{append} = \{1, 2, 3\}$ yields a recursion pattern (both rules are decreasing w.r.t. μ).

5 Acyclic derivations, supports, and stable models

By analogy with the theory of finitary programs, a query G over an FP2 program P is answered by computing in a top-down fashion a representative, partially evaluated fragment of $\mathsf{Ground}(P)$ that suffices to answer G. FP2 programs will be defined so that such top-down computations are finite and finitely many, therefore a complete enumeration thereof is possible. Note that Lemma 4.3 is not enough for this purpose for two reasons. First, a loop-checking mechanism should be set up to avoid infinite cyclic derivations. Second, there could still be infinitely many bounded, acyclic derivations (a situation that commonly arises in the presence of local variables, that make $\mathsf{DG}_p(P)$ infinitely branching). We shall constrain queries and rule bodies to be *call safe* (see below) so that the selected, almost never increasing arguments of each predicate are bound whenever the predicate is called; we shall prove that, as a consequence, every subgoal yields finitely many answer substitutions that are all grounding. In this section, we set up the technical machinery for the acyclic top-down computations which is based on annotating each goal with the history of previously resolved atoms in order to check for loops.

[2] It is not hard to see that if π were not required to be complete for almost never increasing rules (cf. point 2 in Def. 4.1) then this lemma would not be valid.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

5.1 Annotated and acyclic derivations

An *annotated literal* (*a-literal* for short) is a pair $L\alpha$ where L is a literal and α is an *annotation*, that is, a sequence of atoms. The empty annotation will be denoted with ε. $L\alpha$ is *positive* (resp. *negative*) if L is positive (resp. negative). An *annotated goal* (*a-goal* for short) is a finite sequence $G = L_1\alpha_1, \ldots, L_n\alpha_n$ of annotated literals. An annotated goal is *cyclic* if some positive L_i occurs in α_i, *acyclic* otherwise.

Given an a-goal $G = L_1\alpha_1, \ldots, L_n\alpha_n$, a positive $L_i\alpha_i$ in G ($1 \leqslant i \leqslant n$), and a rule $R = A \leftarrow L'_1, \ldots, L'_m$ such that L_i and A are unifiable and $\mathsf{mgu}(L_i, A) = \theta$, the goal

$$\big(L_1\alpha_1, \ldots, L_{i-1}\alpha_{i-1}, L'_1\alpha', \ldots, L'_m\alpha', L_{i+1}\alpha_{i+1}, \ldots, L_n\alpha_n\big)\theta$$

where $\alpha' = L_i \cdot \alpha_i$ is called the *annotated resolvent* of G, L_i, and R with mgu θ. The atom L_i is called *selected atom*, and in this paper it will always be the *leftmost positive literal* of G. Accordingly, the selected literal will frequently be omitted.

An *annotated derivation* (*a-derivation* for short) of G_0 from a program P with rules $R_1, \ldots, R_i, \ldots$ and mgu's $\theta_1, \ldots, \theta_i, \ldots$ is a (possibly infinite) sequence of a-goals $G_0, \ldots, G_i, \ldots$ such that each G_j in the sequence with $j > 0$ is the annotated resolvent of G_{j-1} and R_j with mgu θ_j, for some standardized apart variant R_j of a rule in P. An a-derivation is *acyclic* if all of its a-goals are, possibly with the exception of the last goal if the derivation is finite. Intuitively, an acyclic derivation *fails* as soon as a cycle is detected.

An a-derivation is *successful* if it is finite and its last element contains no positive a-literals. If $G_0, \ldots, G_n$ is a successful a-derivation with mgu's $\theta_1, \ldots, \theta_n$, then we call the composition $\theta^g = \theta_1 \circ \cdots \circ \theta_n$ a *global answer* to G_0, and the restriction of θ^g to the variables of G_0 an *answer substitution* to G_0.

Example 5.1
Consider the program P consisting of the rules $p(X) \leftarrow q(X)$, $q(X) \leftarrow p(X)$, and $p(a)$. The goal $p(a)$ has both a successful acyclic a-derivation $p(a)\varepsilon, \Box$ (where $\Box$ denotes the empty goal) where $p(a)$ is resolved with the third rule, and a failed acyclic derivation using the first two rules: $p(a)\varepsilon$, $q(a)p(a)$, $\underline{p(a)}(q(a) \cdot \underline{p(a)})$. The underlined literals show that the last goal is cyclic.

The main results of the paper will need the following technical definitions and lemmata.

Let G_0 be an a-goal with at least k positive a-literals. An *embedded a-derivation of degree k* for G_0 is an a-derivation $\Delta = G_0, G_1, \ldots$ such that for some suffix G'' of G_0:

- for all $i = 0, 1, \ldots$, it holds $G_i = G'_i G''$, for some G'_i;
- the number of positive a-literals in G'_0 is k;
- if Δ is finite and G_n is its last goal, either G'_n has no positive a-literals or G'_n is failed; in the former case, the embedded a-derivation is *successful*, otherwise it is *failed*.

Intuitively, an embedded derivation of degree k, if successful, resolves the first k positive literals of the initial goal. In the following sections, we will sometimes split derivations into multiple embedded derivations to apply the induction hypotheses. The following two lemmata help.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Lemma 5.2 (*Decomposition 1*)
Δ is an embedded a-derivation of G_0 from P with degree 1 iff either (i) G_0 is failed and $\Delta = G_0$, or (ii) G_0 has an annotated resolvent G_1 with rule R and mgu θ, and $\Delta = G_0 \cdot \Delta'$, where Δ' is an embedded a-derivation of G_1 with degree k and k is the number of positive literals in the body of $R\theta$.

Given two a-derivations $\Delta = G_0, \ldots, G_m$ and $\Delta' = G'_0, \ldots, G'_n$ such that $G_m = G'_0$, the *join* of Δ and Δ' is $G_0, \ldots, G_m, G'_1, \ldots, G'_n$.

Lemma 5.3 (*Decomposition 2*)
Δ is an embedded a-derivation of G_0 from P with degree k iff either (i) Δ is the join of an embedded a-derivation Δ' of G_0 from P with degree 1 and an embedded a-derivation of G_n from P with degree $k-1$, where G_n is the last a-goal of Δ'; or (ii) Δ is an infinite embedded a-derivation for G_0 of degree 1.

5.2 Finiteness and groundness properties of call-safe, acyclic a-derivations

The good finiteness and termination properties we need acyclic derivations to enjoy in order to prove the termination of our algorithms can be enforced by a "call safeness" property that ensures that the arguments selected by recursion pattern are always bound when a predicate is called.

Definition 5.4 (*Call-safeness*)
An a-goal $L_1\alpha_1, ..., L_n\alpha_n$ is *call-safe* w.r.t. a selection index mapping μ iff for all variables X occurring in some $L_i[\mu]$ or in a negative literal L_i $(1 \leqslant i \leqslant n)$, X occurs also in a positive literal L_j, with $1 \leqslant j < i$. Similarly, a rule $R : A \leftarrow L_1, L_2, ..., L_n$ is *call-safe* w.r.t. μ iff for each variable X occurring in R, some of the following conditions hold:

1. X occurs in $A[\mu]$;
2. X occurs in $\mathsf{body}(R)$; moreover, if X occurs in $L_i[\mu]$ or in a negative literal L_i, for some $i = 1, \ldots, n$, then X occurs also in a positive literal L_j, with $1 \leqslant j < i$.

Finally, a program P is *call-safe* w.r.t. μ iff for all $R \in P$, R is call-safe w.r.t. μ.

Example 5.5
Consider the following program for reversing a list:

$$reverse([\,],[\,]). \qquad reverse([X|Y],Z) \leftarrow reverse(Y,W), append(W,[X],Z).$$

The first rule is trivially call safe w.r.t. any selection index mapping. If $\mu_{reverse} = \{1\}$ and $\mu_{append} = \{1\}$ then the second rule is call-safe w.r.t. μ. To see this, note that: (i) X and Y satisfy condition (1); (ii) Z satisfies (2) because it occurs in the body but not in any selected argument nor in any negative literal; (iii) W, the selected argument of the second subgoal, satisfies condition (2) because it occurs also in the first subgoal.

If both P and G are call-safe, then call-safeness is preserved along all the steps of a derivation:

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Lemma 5.6
Let P be a normal logic program and G an a-goal. If P and G are call-safe w.r.t. a selection index mapping μ, then all resolvents of G and a rule $R \in P$ are call-safe w.r.t. μ, too.

Proof. Let G' be an annotated resolvent of the first positive a-literal $L\alpha$ in G and a rule $R = A \leftarrow L_1, ..., L_k$ in P with a substitution $\theta = \mathsf{mgu}(L, A)$. Since G is call-safe, $L[\mu]$ is ground, and so must be $A\theta[\mu]$. It follows—since P is call-safe—that the a-goal $L_1(L \cdot \alpha), ..., L_k(L \cdot \alpha)$ must be call-safe w.r.t. μ. Then G' is call-safe w.r.t. μ, too. □

Furthermore, the binding propagation schema imposed by call-safeness ensures that global answers are grounding:

Lemma 5.7
Let P be a normal logic program and G_0 an a-goal. Assume that P and G_0 are call-safe w.r.t. a selection index mapping μ. If G_0 has a successful a-derivation $\Delta = G_0, ..., G_n$ from P with global answer θ, then for all $i = 0, ..., n$, the a-goal $G_i\theta$ is ground.

Proof. By induction on the length of Δ.

Base case (the length of Δ is 0): Let $\Delta = G_0$. Since G_0 is call-safe, all its a-literals have to be negative and ground. Then, $G_0\theta$ is ground.

Inductive step (the length of Δ is $n+1$): Let $\Delta = G_0, G_1, ..., G_{n+1}$. By Lemma 5.6, G_1 is call-safe w.r.t. μ. Moreover, $\Delta' = G_1, ..., G_{n+1}$ is a successful a-derivation of length n for G_1 with global answer θ' more general than θ. By inductive hypohesis, for all $i = 1, ..., n+1$, the a-goal $G_i\theta'$ is ground. Since G_1 is a resolvent of G_0 and both goals are call-safe, all variables in $G_0\theta$ must be also in $G_1\theta'$. Consequently, $G_0\theta$ is ground. □

The proof of the main theorem—that we need to prove the termination of our reasoning algorithm—will be based on inductions over the three indices defined below.

Let the height of a predicate q be the cardinality of the set of predicates reachable from q in $DG_p(P)$. The *height* of an atom A is the height of $\mathsf{pred}(A)$. Note that (i) $\mathsf{height}(A) \geqslant 1$; (ii) if $\mathsf{pred}(A)$ depends on $\mathsf{pred}(A')$ but not viceversa, then $\mathsf{height}(A) > \mathsf{height}(A')$; (iii) if $\mathsf{pred}(A)$ and $\mathsf{pred}(A')$ belong to the same strongly connected component of $DG_p(P)$, then $\mathsf{height}(A) = \mathsf{height}(A')$. By convention, the height of a negative literal is 0.

Let π be a recursion pattern for P, and A be an atom such that $A[\pi]$ is ground. The *call size* of A is $|A[\pi]|$ (the norm of $A[\pi]$).

For the strongly connected components C of $DG_p(P)$ in which the call size does not decrease during recursion, we adopt a "loop saturation" index. Let a *C-atom* be an atom A with $\mathsf{pred}(A) \in C$. Given a ground C-atom A, let $\max_A$ be the number of ground C-atoms B such that A depends on B. Such an integer $\max_A$ exists due to the following lemma (that shows why π should be complete over such C):

Lemma 5.8
Let P be a program with a recursion pattern π and C be a strongly connected component of $DG_p(P)$ s.t. π is complete for predicates in C. Every ground C-atom A depends on finitely many C-atoms in P.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Clearly $\max_A$ is an upper bound to the number of consecutive ground C-atoms occurring in an acyclic annotation. The *loop saturation* index of an a-literal $A\alpha$ is $\max_A$ minus the length of the longest prefix of α consisting of C-atoms only. If the loop saturation index of $A\alpha$ is 0, then every resolvent of $A\alpha$ that contains a C-atom is cyclic; if the loop saturation index is $\ell > 0$, then all the C-atoms in the resolvents of $A\alpha$ have loop saturation index $\ell - 1$.

Theorem 5.9 (*Strong finiteness*)
Let P be a program with a recursion pattern π. Let G_0 be an a-goal with k or more positive a-literals. Assume that P and G_0 are call-safe w.r.t. π. Then G_0 has finitely many acyclic embedded a-derivations of degree k from P. Moreover, they are all finite.

Proof. By induction on the maximum height of the literals in G_0. The base case is trivial. Now assume that the theorem holds for all heights $\leqslant n$; let $A_1\alpha_1, \ldots, A_k\alpha_k$ be the first k positive a-literals of G_0, and assume that the maximum height of $A_1, \ldots, A_k$ is $n+1$.

We first prove the theorem for "homogeneous" cases where the atoms with maximum height belong to a same strongly connected component C of $DG_p(P)$, that is, the members of $\{A_1, \ldots, A_k\}$ with height $n+1$ are all C-atoms. This case is further divided in two subcases:

SC1 all rules $R \in C$ are decreasing w.r.t. the recursion pattern π;
SC2 all rules $R \in C$ are almost never increasing w.r.t. π.

Proof of SC1. By induction on the maximum call size of the members of $\{A_1, \ldots, A_k\}$ with height $n+1$.

Base case for SC1 (the maximum call size is 0). By induction on k. If $k = 1$, then for all resolvents G_1 of G_0 with rule R and mgu θ, consider the positive literals $A'_1 \ldots A'_j$ in the body of $R\theta$. Since the call size of A_1 is 0, R is decreasing w.r.t. π, and the call size is non-negative, it follows that the height of $A'_1 \ldots A'_j$ must be smaller than $n+1$. By the induction hypothesis relative to height, the embedded a-derivations for G_1 of degree j are finite and finitely many. Then the same property holds for the embedded a-derivations for G_0 of degree 1, by Lemma 5.2. This completes the proof for $k = 1$.

Now assume $k > 1$. By Lemma 5.3, every embedded a-derivations for G_0 of degree k is the join of two embedded a-derivations of degree 1 and $k-1$, respectively. Then the theorem easily follows from the induction hypothesis for degrees $k' < k$.

Induction step for SC1 (the maximum call size is $c > 0$). The proof is similar to the proof of the base case. The only difference is that the the positive literals $A'_1 \ldots A'_j$ in the body of $R\theta$ may belong to C and have degree $n+1$, however their maximum call size must be smaller than c because R is decreasing w.r.t. π. Then it suffices to apply the induction hypothesis relative to the maximum call size instead of the one relative to height. This completes the proof of SC1.

Proof of SC2. Analogous to the proof of SC1. In this case the induction is on the maximum loop saturation index ℓ of the first positive a-literals $A_1\alpha_1, \ldots, A_k\alpha_k$. Call safeness, Lemma 5.6, and the completeness of τ over C ensure that all C-atoms occurring in the derivation are ground, so that the loop saturation index is

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

well-defined. Details are omitted due to space limitations. This completes the proof of the homogeneous case.

Finally, we are left to prove the theorem for "non homogeneous" goals where the atoms with maximum height among $A_1, \ldots, A_k$ may belong to different strongly connected components. The proof is by induction on k; the induction step relies on Lemma 5.3. □

Corollary 5.10
Let P be a program with a recursion pattern π and G_0 be an a-goal. Assume that P and G_0 are call-safe w.r.t. π. Then G_0 has finitely many acyclic a-derivations from P. Moreover, they are all finite.

5.3 Acyclic supports and stable models

It is well-known that the stable models of a program P are completely characterized by the supports of P's ground atoms. In our setting, a *support* for an a-goal G with answer substitution θ is a set of negative literals $\{L_1, \ldots, L_n\}$ such that G has a successful a-derivation from $\mathsf{Ground}(P)$ with answer substitution θ and last goal $L_1\alpha_1, \ldots, L_n\alpha_n$. A support for an atom A is a support for the a-goal $A\varepsilon$. The set of (negative) literals occurring in the last a-goal of a successful a-derivation Δ is called the *support of* Δ. By *acyclic support* we mean a support generated by an acyclic derivation. The first result tells that by adopting acyclic a-derivations, only redundant supports can be lost:

Theorem 5.11 (*Completeness of acyclic derivations w.r.t. supports*)
If G_0 has a successful a-derivation Δ from P with global answer θ, then G_0 has a successful acyclic a-derivation Δ^a from P with global answer θ^a such that θ^a is more general than θ and the support of Δ^a is more general than a subset of the support of Δ.

A-derivations and the related notion of support are in close correspondence with the P-proofs of (Marek and Remmel 2008) and the corresponding supports. By exploiting these relationships and the previous lemma, one can easily prove the following characterization of stable models in terms of the supports of acyclic a-derivations.

Theorem 5.12
Let P be a normal program. A set M of ground atoms is a stable model of P iff M is the set of all ground atoms that have a ground acyclic support in $\mathsf{Ground}(P)$ satisfied by M.

6 The class FP2

We are finally ready to introduce the class of FP2 programs. If π and τ are two selection index mappings, we say that τ *contains* π (in symbols, $\tau \supseteq \pi$) iff, for each predicate symbol p, it holds that $\tau_p \supseteq \pi_p$.

Definition 6.1 (*Call patterns*)
A selection index mapping τ for a normal program P is a *call pattern* for P iff (i) τ contains a recursion pattern of P, and (ii) for each rule $R \in P$ there exists a permutation $L_1, L_2, ..., L_n$ of $\mathsf{body}(R)$ such that $\mathsf{head}(R) \leftarrow L_1, L_2, ..., L_n$ is call-safe w.r.t. τ.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Definition 6.2 (FP2)
A normal logic program belongs to the class FP2 iff it has a call pattern.

Example 6.3
The append program of Example 4.5 is in FP2. It is easy to verify that if $\tau_{append} = \{3\}$ then τ is not only a recursion pattern (see Ex. 4.5), but also a call pattern. On the contrary, the recursion pattern yielded by $\tau_{append} = \{1\}$ is not a call pattern because the variable L in the first rule occurs neither in the selected argument (the first one) nor in the body. However this recursion pattern is contained in two call patterns, defined by $\tau_{append} = \{1, 2\}$ and $\tau_{append} = \{1, 3\}$.

For an example of a cyclic FP2 program with negation see the blocks world program in (Bonatti 2004, Fig.4). To make it an FP2 program, uniformly replace $T + 1$ with T in the second arguments of predicate *ab*. Then the (unique) selection index that is complete for all predicates is a call pattern for the program.

In general, it may be necessary to use different call patterns for different initial goals, in order to satisfy call safeness. In the above example a goal $append(t_1, t_2, t_3)$ is call safe w.r.t. some call pattern iff either t_3 is ground or at least two arguments are ground; different situations require different call patterns.

6.1 Inference in FP2

The ground skeptical and credulous consequences of finitary programs can be computed by using a ground "relevant" fragment of their ground instantiation (Bonatti 2004). Similarly, we can reason over FP2 programs by answering queries over finite and ground programs called support subprograms.

We start by defining a function $\mathsf{Ssup}(G, P)$ that, for all call-safe a-goals G and FP2 programs P, returns a representative set of supports for G w.r.t. P. More precisely, let $\mathsf{Ssup}(G, P)$ be the set of all pairs (θ, s) such that s is an acyclic support of G in P with global answer θ.

Proposition 6.4
Let $P \in$ FP2 be a program with a call pattern τ. The restriction of $\mathsf{Ssup}(G, P)$ to all G that are call-safe w.r.t. τ is computable.

Proof. (Sketch) By Corollary 5.10, acyclic a-derivations of G from P are finite and finitely many. Then, it suffices to enumerate all acyclic a-derivations of G from P. □

Definition 6.5
Let P be a program and Q be an atom. The *support subprogram* $\mathsf{S}(P, Q)$ for Q w.r.t. P is the set computed by the algorithm supportSubProgram(P, Q) below.

For all FP2 programs and suitably instantiated atoms Q, the support subprogram $\mathsf{S}(P, Q)$ is finite, ground, and computable.

Theorem 6.6
Let $P \in$ FP2 be a program with a call pattern τ. For all atoms Q such that $Q[\tau]$ is ground, the algorithm supportSubProgram(P, Q) terminates and returns a ground program.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Algorithm SUPPORTSUBPROGRAM (P, Q)
1: $SSUP = \mathsf{Ssup}(Q\varepsilon, P)$;
2: $T = \{(Q\theta, G) \mid (\theta, G) \in SSUP\}$;
3: $\bar{T} = \emptyset$;
4: $S = \emptyset$;
5: **while** $T \neq \emptyset$ **do**
6: CHOOSE $(A, G) \in T$;
7: $T = T \setminus \{(A, G)\}$;
8: $\bar{T} = \bar{T} \cup \{(A, G)\}$;
9: $S = S \cup \{A \leftarrow G\}$;
10: **for all** not $B \in G$ **do**
11: $SSUP = \mathsf{Ssup}(B\varepsilon, P)$;
12: $T = T \cup (\{(B, G') \mid (\theta, G') \in SSUP\} \setminus \bar{T})$;
13: **return** S;

Proof. (Sketch) It can be proved by simultaneous induction that the contents of T are always ground, the atom A selected at step 6 is always call-safe, and the input goal of Ssup is call-safe therefore Ssup is computable. The induction argument relies on the following observations: (i) the initial a-goal $Q\varepsilon$ is call-safe w.r.t. τ by hypothesis, (ii) by Lemma 5.7, all the supports returned by Ssup are ground if the input goal is call-safe, and (iii) ground atoms are vacuously call-safe. We are left to show that the loop at lines 5-12 terminates. Observe that the atoms occurring in T during the computation belong to the following forest: The roots are the finitely many instances $Q\theta$ inserted at step 2; the children of each (ground) node A are atoms B occurring in the acyclic supports of A and different from A and its ancestors. By Corollary 5.10, this tree is finitely branching; by Theorem 4.3 all the paths are finite. Then the tree must be finite. It follows that the algorithm cannot produce infinitely many different atoms B, and hence after a finite number of steps all the pairs (B, G') at line 12 shall be already contained in $\bar{T}$ and the while statement terminates. □

We are only left to show that $\mathsf{S}(P, Q)$ can be used to answer Q. By using the properties of relevant subprograms (Bonatti 2004) and Theorem 5.12, we can prove that:

Theorem 6.7
Let $P \in$ FP2 be a program with a call pattern τ and let Q be an atom s.t. $Q[\tau]$ is ground. For all grounding substitutions θ, $Q\theta$ is a credulous/skeptical consequence of P iff it is a credulous/skeptical consequence of $\mathsf{S}(P, Q)$.

It follows that call-safe queries are computable over FP2 programs. In general, this property does not hold if call safeness does not hold, as proved by the following theorem that is based on an FP2 encoding of a Turing machine similar to those used in (Bonatti 2004):

Theorem 6.8
The problems of deciding whether an FP2 program P credulously/skeptically entails an existentially quantified goal $\exists G$ are both r.e.-complete.

Moreover, the class of FP2 programs is decidable because the space of call patterns and recursion patterns for every given program P is finite, and a simple generate and test algorithm can be used for FP2 membership checking. Then we get:

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

Proposition 6.9
Deciding whether a program P is in FP2 is decidable.

7 Extending FP2 with odd-cycles

By Lemma 4.4, FP2 programs cannot be inconsistent nor express denials (that require odd-cycles). This restriction can be relaxed simply by composing FP2 programs with argument restricted programs (Lierler and Lifschitz 2009), that are currently the largest known decidable class of programs with the *persistent CFSP property* and have no restriction on odd-cycles.

Definition 7.1
(Baselice and Bonatti 2008) A class of programs $\mathscr{C}$ has the *computable finite semantics property* (CFSP for short) iff (i) for all P in $\mathscr{C}$, P has finitely many stable models each of which is finite, and (ii) there exists a computable function f mapping each member of $\mathscr{C}$ onto its set of stable models. Moreover, the CFSP property is *persistent* iff $\mathscr{C}$ is closed under language extensions (i.e., adding more constants or function symbols to the language of a program $P \in \mathscr{C}$ yields another program in $\mathscr{C}$).

The CFSP property abstracts a number of program classes with function symbols: ω-restricted programs (Syrjänen 2001), λ-restricted programs (Gebser *et al.* 2007), argument restricted programs, and more generally the semidecidable class of finitely-ground programs (Calimeri *et al.* 2008). The persistent CFSP property is important because, under suitable hypotheses, programs with this property can be composed with finitary programs without affecting the decidability of inference (Baselice and Bonatti 2008). We need a preliminary result:

Proposition 7.2
Argument restricted programs have the persistent $CFSP$.

The forms of composition studied in (Baselice and Bonatti 2008) are the following. Let $\mathsf{Def}(P)$ denote the set of predicates *defined in* P, that is, the set of all predicate symbols occurring in the head of some rule in P. Let $\mathsf{Called}(P)$ be the set of predicates *called by* P, that is, the set of all predicate symbols occurring in the body of some rule in P. Then we say that P_1 *depends on* P_2, in symbols $P_1 \rhd P_2$, if and only if

$$\mathsf{Def}(P_1) \cap \mathsf{Def}(P_2) = \emptyset,\ \mathsf{Def}(P_1) \cap \mathsf{Called}(P_2) = \emptyset,\ \mathsf{Called}(P_1) \cap \mathsf{Def}(P_2) \neq \emptyset.$$

Moreover, P_1 *and* P_2 *are independent*, in symbols $P_1 \| P_2$, if and only if

$$\mathsf{Def}(P_1) \cap \mathsf{Def}(P_2) = \emptyset,\ \mathsf{Def}(P_1) \cap \mathsf{Called}(P_2) = \emptyset, \mathsf{Called}(P_1) \cap \mathsf{Def}(P_2) = \emptyset.$$

Now the techniques of (Baselice and Bonatti 2008), based on the splitting theorem, can be easily adapted to prove the following result:

Theorem 7.3
For all programs P and Q such that P is in FP2 and Q has the persistent CFSP, if $P \rhd Q$ or $P \| Q$, then both credulous and skeptical consequences from $P \cup Q$ are decidable.

In particular, this result shows that it is theoretically possible to add the expressiveness of FP2 programs to argument restricted (actually, all finitely ground) programs.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

$$member(X,[X|Y]).$$
$$member(X,[Y|Z]) \leftarrow member(X,Z).$$
$$reverse(L,R) \leftarrow reverse(L,[\,],R).$$
$$reverse([\,],R,R).$$
$$reverse([X|X_s],A,R) \leftarrow reverse(X_s,[X|A],R).$$

Fig. 1. List processing.

$$s(and(X,Y)) \leftarrow s(X), s(Y).$$
$$s(or(X,Y)) \leftarrow s(X).$$
$$s(or(X,Y)) \leftarrow s(Y).$$
$$s(not(X)) \leftarrow \texttt{not}\ s(X).$$
$$s(A) \leftarrow member(A,[p,q,r]), \texttt{not}\ ns(A).$$
$$ns(A) \leftarrow member(A,[p,q,r]), \texttt{not}\ s(A).$$

Fig. 2. SAT problem.

8 Related work

ASP programs with function symbols are able to encode infinite domains and recursive data structures, such as lists, trees, XML/HTML documents, time. However, some restrictions are needed to keep inference decidable and, to this end, ASP researchers have recently made several proposals (Syrjänen 2001; Bonatti 2004; Simkus and Eiter 2007; Gebser *et al.* 2007; Baselice *et al.* 2009; Calimeri *et al.* 2008; Calimeri *et al.* 2009). We will discuss finitely-ground and FDNC programs, as they include all the other classes mentioned above.

Finitely-ground programs are DLP programs with function symbols introduced in (Calimeri *et al.* 2008). If we compare this class with FP2 programs, we note that:

- Ground and nonground queries are computable for finitely-ground programs; call-safe queries are computable for FP2 programs while, in general, nonground queries are r.e.-complete.
- The answer sets of finitely-ground programs are computable because their semantics is finite. Infinite stable models are ruled out. On the contrary, FP2 programs may have infinite and infinitely many answer sets.
- Finitely-ground programs are safe, while FP2 programs admit unsafe rules.
- Odd-cycles may occur in finitely-ground programs but not in FP2 programs. The latter can be extended with odd-cyclic predicates through composition with CFSP programs as shown in Section 7.
- Deciding whether a program is finitely-ground is semidecidable, while the class of FP2 programs is decidable.
- Finitely-ground programs are disjunctive, while FP2 is currently restricted to normal programs.

Finitely-ground and FP2 programs are not comparable due to the different recursion modes that they admit and that make finitely-ground programs suitable for a bottom-up evaluation and FP2 programs suitable for a top-down evaluation. Figures 1, 2 and 3 and Example 4.5 illustrate some programs that are FP2 but not finitely-ground.

FDNC programs (Simkus and Eiter 2007) achieve inference decidability by exploiting a tree-model property, by analogy with decidable fragments of first-order logic such as description logics and the guarded fragment. The tree-model property derives from syntactic restrictions on predicate arity and on the occurrences of function symbols (modelled around the skolemization of guarded formulae). FDNC programs can be applied to encode ontologies expressed in description logics, and are suitable to model a wide class of planning problems. Summarizing:

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

$qbf(A, I) \leftarrow atomic(A), curr_value(A/t, I).$
$qbf(or(F, G), I) \leftarrow qbf(F, I).$
$qbf(or(F, G), I) \leftarrow qbf(G, I).$
$qbf(and(F, G), I) \leftarrow qbf(F, I), qbf(G, I).$
$qbf(not(F), I) \leftarrow \texttt{not}\ qbf(F, I).$
$qbf(exists(X, F), I) \leftarrow qbf(F, [X/t|I]).$
$qbf(exists(X, F), I) \leftarrow qbf(F, [X/f|I]).$
$qbf(forall(X, F), I) \leftarrow qbf(F, [X/t|I]), qbf(F, [X/f|I]).$
$curr_value(B, [B|L]).$
$curr_value(X/V, [Y/W|L]) \leftarrow \texttt{not}\ X = Y, curr_value(X/V, L).$

Fig. 3. Satisfiability check for Quantified Boolean Formulas.

- Both FDNC and FP2 programs may have infinite and infinitely many answer sets.
- Unlike FP2 programs, the answer sets of FDNC programs can be finitely represented.
- Ground and nonground queries over FDNC programs are always computable; only call-safe queries are computable for FP2 programs.
- FDNC programs are safe, while FP2 programs admit unsafe rules.
- Odd-cycles may occur in FDNC programs but not in FP2 programs.
- FDNC programs are disjunctive.

Therefore, FDNC and FP2 programs are incomparable. The programs in Figures 1, 2 and 3 and in Example 4.5 are examples of FP2 programs that are not FDNC.

9 Summary and conclusions

We have introduced FP2, a decidable class of well-behaved normal programs whose properties are orthogonal to those of the other decidable classes of ASP programs with function symbols.

Inference is decidable, too. We have shown a method based on a partial evaluation of the program w.r.t. a query Q (algorithm SUPPORTSUBPROGRAM) that produces a ground program $\mathsf{S}(P, Q)$ that can be fed to any ASP reasoner in order to answer Q. The query Q needs not be ground: it can be call-safe, and it is not hard to see that the method can produce answer substitutions by unifying Q with the stable models of $\mathsf{S}(P, Q)$.

Note that currently this mixed top-down/ASP solving method is not intended to be an efficient implementation; it is only a proof method for decidability results. In future work the potential of top-down computations as an implementation technique should be evaluated and compared with the magic-set approach adopted in (Calimeri *et al.* 2008).

The norm-based definition of FP2 programs is actually a simplification of the (approximate) static analysis method for recognizing U-bounded finitary programs described in (Bonatti 2001; Bonatti 2004). The binding propagation analysis of the old recognizer is more powerful, and we are planning to improve FP2 to cover more programs accepted with the old method. Further interesting issues for future work comprise: a precise complexity analysis of FP2 membership checking and inference; support for disjunctive programs; more general forms of composition with persistently CFSP programs; integration with FDNC programs for more general support to odd-cycles.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press

References

Baselice, S. and Bonatti, P. A. 2008. Composing normal programs with function symbols. See de la Banda and Pontelli (2008), 425–439.

Baselice, S., Bonatti, P. A., and Criscuolo, G. 2009. On finitely recursive programs. *Theory and Practice of Logic Programming 9,* 2, 213–238.

Bonatti, P. 2001. Prototypes for reasoning with infinite stable models and function symbols. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001*. LNCS, vol. 2173. Springer, 416–419.

Bonatti, P. A. 2004. Reasoning with infinite stable models. *Artificial Intelligence 156,* 1, 75–111.

Bonatti, P. A. 2008. Erratum to: Reasoning with infinite stable models [Artificial Intelligence 156 (1) (2004) 75–111]. *Artificial Intelligence 172,* 15, 1833–1835.

Bossi, A., Cocco, N., and Fabris, M. 1994. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science 124,* 2, 297–328.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2008. Computable functions in ASP: Theory and implementation. See de la Banda and Pontelli (2008), 407–424.

Calimeri, F., Cozza, S., Ianni, G., and Leone, N. 2009. Magic sets for the bottom-up evaluation of finitely recursive programs. In *LPNMR*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer, 71–86.

de la Banda, M. G. and Pontelli, E., Eds. 2008. *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9–13, 2008, Proceedings*. Lecture Notes in Computer Science, vol. 5366. Springer.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. 1997. A deductive system for non-monotonic reasoning. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Proceedings*. Lecture Notes in Computer Science, vol. 1265. Springer, 364–375.

Gebser, M., Schaub, T., and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 266–271.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3–4, 365–386.

Genaim, S., Codish, M., Gallagher, J., and Lagoon, V. 2002. Combining norms to prove termination. In *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002*. Lecture Notes in Computer Science, vol. 2294. Springer, 126–138.

Lierler, Y. and Lifschitz, V. 2009. One more decidable class of finitely ground programs. In *ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 489–493.

Lloyd, J. W. 1984. *Foundations of Logic Programming, 1st ed*. Springer.

Marek, V. W. and Remmel, J. B. 2008. On the continuity of Gelfond-Lifschitz operator and other applications of proof-theory in ASP. See de la Banda and Pontelli (2008), 223–237.

Niemelä, I. and Simons, P. 1997. Smodels—an implementation of the stable model and well-founded semantics for normal LP. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Proceedings*. Lecture Notes in Computer Science, vol. 1265. Springer, 421–430.

Simkus, M. and Eiter, T. 2007. FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. In *14th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2007*. Lecture Notes in Computer Science, vol. 4790. Springer, 514–530.

Syrjänen, T. 2001. Omega-restricted logic programs. In *LPNMR*, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 267–279.

https://doi.org/10.1017/S1471068410000232 Published online by Cambridge University Press