Function Symbols in ASP: Overview and Perspectives

Mario Alviano Francesco Calimeri Wolfgang Faber
Giovambattista Ianni Nicola Leone
Department of Mathematics, University of Calabria
P.te P. Bucci, Cubo 30B, I-87036 Rende, Italy
{alviano,calimeri,faber,ianni,leone}@mat.unical.it

Abstract

Answer Set Programming (ASP) is a highly expressive language that is widely used for knowledge representation and reasoning in many application scenarios. Thanks to disjunction and negation, the language allows the use of nondeterministic definitions for modeling complex problems in computer science, in particular in Artificial Intelligence. Traditionally, ASP has often been used as a first-order language without function symbols, similar to Datalog, in order to deal with finite structures only. More recently, also uninterpreted function symbols have been frequently considered in the setting of ASP, enabling a natural representation of recursive structures. Function symbols can be used for encoding strings, lists, stacks, trees and many other common data structures. However, the common reasoning tasks are undecidable for programs with no restrictions on the usage of function symbols. Therefore, identifying relevant classes of programs with decidable reasoning is important for practical applications, and many authors have addressed this issue in the past decade.

This article provides a survey of the decidability results for ASP programs with functions. We classify the decidable ASP programs in three main groups: programs allowing for finite bottom-up evaluations; programs suitable for finite top-down evaluations; programs characterized by finite representations of stable models. We focus on the decidability of ground reasoning and computability of non-ground reasoning. Moreover, we consider decidability of coherence checking and of class membership; expressiveness issues are briefly discussed as well.

1 Introduction

Logic Programming (LP) under the answer set or stable model semantics, often called Answer Set Programming (ASP), is a convenient and effective method for declarative knowledge representation and reasoning [Baral, 2003, Gelfond and Lifschitz, 1991]. The success of ASP in many practical applications has been encouraged by the availability of some efficient inference engines, such as DLV
ASP allows for disjunction in rule heads and nonmonotonic negation in rule bodies. Over finite structures, the language allows for expressing all properties in the second level of the polynomial hierarchy. Restricting terms to constants and variables guarantees structures to be finite. ASP with this restriction has been successfully used for knowledge representation and reasoning in numerous applications. If uninterpreted function symbols of positive arities are permitted, instead, the expressive power of the language increases considerably, up to the first level of the analytical hierarchy if disjunction or recursive negation are allowed [Cadoli and Schaerf, 1993]. However, this high expressive power implies that the common reasoning tasks are undecidable for programs with function symbols. This is a consequence of the fact that even for Horn programs these tasks are undecidable [Tärnlund, 1977]. In the past decade, several efforts have been made to identify large classes of ASP programs with function symbols for which some important reasoning tasks are still decidable. A couple of interesting classes have already been discovered, and the research in this area is quite active; the reader may refer, for instance, to Baselice, Bonatti, and Criscuolo [2009], Bonatti [2002, 2004], Cabalar [2008], Calimeri, Cozza, Ianni, and Leone [2008, 2009], Eiter and Simkus [2009b], Gebser, Schaub, and Thiele [2007b], Lierler and Lifschitz [2009], Lin and Wang [2008], Simkus and Eiter [2007], Syrjänen [2001].

In this work, a survey of the main results achieved in this research area are given. In particular, three groups of ASP fragments with decidable reasoning are identified: bottom-up computable, top-down computable and finitely representable stable models.

- Programs in the first group are characterized by the existence of a finite ground program, which is equivalent to the (infinite) program instantiation and can be obtained by a bottom-up computation. This allows for stable model computation and query answering by means of the standard stable model search techniques over these finite ground programs. Therefore, all of the programs in this first group are characterized by decidable ground and non-ground reasoning as well as a decidable coherence check. Classes of programs in this group are \(\omega\)-restricted [Syrjänen, 2001], \(\lambda\)-restricted [Gebser, Schaub, and Thiele, 2007b], finite domain [Calimeri, Cozza, Ianni, and Leone, 2008], argument restricted [Lierler and Lifschitz, 2009] and finitely ground programs [Calimeri, Cozza, Ianni, and Leone, 2008].

- Classes in the second group have been defined having query answering and top-down computation in mind. In contrast to the first group, programs in these classes are usually characterized by an infinite number of stable models, each one possibly comprising an infinite number of atoms. To guarantee decidability of reasoning, it is necessary that a finite number of atoms is sufficient for answering an input query. Moreover, these atoms need to be effectively identifiable. Classes in this group are FP2 [Baselice
and Bonatti, 2010], positive/stratified finitely recursive [Calimeri et al., 2009][Alviano, Faber, and Leone, 2010] and finitary programs [Bonatti, 2002, 2004].

• Programs in the third group are characterized by finitely representable sets of stable models, where a set can potentially comprise infinitely many stable models of possibly infinite size. Typically, these stable models have the shape of a forest of trees. Classes in this group are FDNC [Simkus and Eiter, 2007] and bidirectional programs [Eiter and Simkus, 2009a].

For each of these classes we highlight whether decidability of ground reasoning and/or computability of non-ground reasoning are guaranteed. Ground reasoning consists of checking the presence of a specific ground atom among the consequences of a program, while non-ground reasoning means computing all answers for a given non-ground query. Decidability of coherence checks and of class membership are also considered. A coherence check consists of establishing whether a given program has at least one stable model. By class membership, instead, we refer to establishing whether a given program (possibly associated with a query) belongs to the class of programs at hand. Expressiveness is briefly discussed as well, highlighting whether a class still allows for representing every recursive relation or not. Furthermore, we give a description of the efforts for endowing the ASP system DLV with function symbols. In fact, DLV (and its branches) can compute stable models, answers to queries and decide class membership for a number of the surveyed classes.

The remainder of this article is organized as follows. First, syntax and semantics of ASP are introduced in Section 2. After that, classes of ASP programs characterized by decidable reasoning tasks are presented in Section 3. Finally, DLV—a system supporting a powerful class of function symbols—is presented in Section 4 and few concluding comments are given in Section 5.

2 ASP with Function Symbols

In this section, we recall the basics of ASP with uninterpreted function symbols. We start by introducing syntax and semantics of the language. Then, we briefly discuss undecidability of reasoning for arbitrary programs with function symbols.

2.1 Syntax

A term is either a variable or a functional term. A functional term is of the form \( f(t_1, \ldots, t_k) \), where \( f \) is a function symbol (functor) of arity \( k \geq 0 \), and \( t_1, \ldots, t_k \) are terms. A functional term with arity 0 is a constant. If \( p \) is a predicate of arity \( k \geq 0 \), and \( t_1, \ldots, t_k \) are terms, then \( p(t_1, \ldots, t_k) \) is an atom. A literal is either an atom \( \alpha \) (a positive literal), or an atom preceded by the negation as failure symbol not \( \alpha \) (a negative literal). A rule \( r \) is of the form

\[
\alpha_1 \vee \cdots \vee \alpha_k : - \beta_1, \ldots, \beta_n, \not\beta_{n+1}, \ldots, \not\beta_m.
\]
where $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_m$ are atoms and $k \geq 1$, $m \geq n \geq 0$. The disjunction $\alpha_1 \vee \cdots \vee \alpha_k$ is the head of $r$, while the conjunction $\beta_1, \ldots, \beta_n$, $\text{not } \beta_{n+1}, \ldots, \text{not } \beta_m$ is the body of $r$. The set of head atoms is denoted by $H(r)$, while $B(r)$ is used for denoting the set of body literals. We also use $B^+(r)$ and $B^-(r)$ for denoting the set of atoms appearing in positive and negative body literals, respectively. A rule $r$ is normal (or disjunction-free) if $|H(r)| = 1$, positive (or negation-free) if $B^-(r) = \emptyset$, a fact if both $B(r) = \emptyset$, $|H(r)| = 1$ and no variable appears in $H(r)$.

A program $\mathcal{P}$ is a finite set of rules. If all the rules of a program $\mathcal{P}$ are positive (resp. normal), $\mathcal{P}$ is a positive (resp. normal) program. If all functional terms appearing in a program $\mathcal{P}$ are constants, $\mathcal{P}$ is function-free. Stratified and odd-cycle-free programs constitute other interesting classes of programs. Intuitively, for a program $\mathcal{P}$, a predicate $p$ occurring in the head of a rule $r \in \mathcal{P}$ depends on each predicate $q$ occurring in $B(r)$; if $q$ occurs in $B^+(r)$, $p$ depends on $q$ positively, otherwise negatively.\footnote{A similar notion of dependency among ground atoms can be given for ground programs.} If no cycle of dependencies involves negative dependencies (i.e., there are no “negative cycles”), the program is stratified. If no cycle of dependencies contains an odd number of negative dependencies (i.e., there are no “odd cycles”), the program is odd-cycle-free.

2.2 Semantics

The set of terms constructible by combining function symbols and constants appearing in a program $\mathcal{P}$ is the universe of $\mathcal{P}$ and is denoted by $U_\mathcal{P}$, while the set of ground atoms constructible from predicates in $\mathcal{P}$ with elements of $U_\mathcal{P}$ is the base of $\mathcal{P}$, denoted by $B_\mathcal{P}$. We call a term (atom, rule, or program) ground if it does not contain any variable. A ground atom $\alpha'$ (resp. a ground rule $r'$) is an instance of an atom $\alpha$ (resp. of a rule $r$) if there is a substitution $\theta$ from the variables in $\alpha$ (resp. in $r$) to $U_\mathcal{P}$ such that $\alpha' = \alpha\theta$ (resp. $r' = r\theta$). Given a program $\mathcal{P}$, let $\text{grnd}(\mathcal{P})$ denote the set of instances of all the rules in $\mathcal{P}$.

An interpretation $I$ for a program $\mathcal{P}$ is a subset of $B_\mathcal{P}$. A positive ground literal $\alpha$ is true with respect to an interpretation $I$ if $\alpha \in I$; otherwise, it is false. A negative ground literal $\text{not } \alpha$ is true with respect to $I$ if and only if $\alpha$ is false with respect to $I$. The body of a ground rule $r$ is true with respect to $I$ if and only if all the body literals of $r$ are true with respect to $I$, that is, if and only if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. An interpretation $I$ satisfies a ground rule $r \in \text{grnd}(\mathcal{P})$ if either (i) at least one atom in $H(r)$ is true with respect to $I$, or (ii) the body of $r$ is false with respect to $I$. An interpretation $I$ is a model of a program $\mathcal{P}$ if $I$ satisfies all the rules in $\text{grnd}(\mathcal{P})$.

Given an interpretation $I$ for a program $\mathcal{P}$, the reduct of $\mathcal{P}$ with respect to $I$, denoted by $\text{grnd}(\mathcal{P})^I$, is obtained by deleting from $\text{grnd}(\mathcal{P})$ all the rules $r$ such that some body literal in $B(r)$ is false with respect to $I$. The semantics of a program $\mathcal{P}$ is then given by the set $SM(\mathcal{P})$ of the stable models of $\mathcal{P}$, where an interpretation $M$ is a stable model for $\mathcal{P}$ if and only if $M$ is a subset-minimal model of $\text{grnd}(\mathcal{P})^M$.

Given a ground atom $\alpha$ and a program $\mathcal{P}$, $\alpha$ is a cautious (resp. brave)
consequence of $\mathcal{P}$, denoted by $\mathcal{P} \models_c \alpha$ (resp. $\mathcal{P} \models_b \alpha$), if $\alpha \in M$ for each (resp. some) $M \in SM(\mathcal{P})$. The cautious (resp. brave) semantics of a query $\mathcal{Q} = \alpha$ for a program $\mathcal{P}$, where $\alpha$ is an atom, is given by the set $Ans_c(\mathcal{Q}, \mathcal{P})$ (resp. $Ans_b(\mathcal{Q}, \mathcal{P})$) of substitutions $\vartheta$ for the variables of $\alpha$ such that $\mathcal{P} \models_c \alpha \vartheta$ (resp. $\mathcal{P} \models_b \alpha \vartheta$) holds.

2.3 Undecidability

A well-known result about logic programming with uninterpreted function symbols is the undecidability of the reasoning. Indeed, Horn clauses (under the classic first-order semantics) can represent any partial recursive function [Tämnelund, 1977], and this result can be adapted to ASP (even without using disjunction and negation) [Alviano et al., 2010]. However, in the past decade, relevant classes of ASP programs with function symbols guaranteeing decidability of the common reasoning tasks have been identified. Some of them are presented in the next section.

3 Decidable Reasoning: Class Overview

In this section the most important classes of ASP programs with uninterpreted function symbols that guarantee decidability of reasoning tasks are surveyed. More specifically, for each class we discuss decidability of ground reasoning and computability of non-ground reasoning. We recall that by ground reasoning we mean checking the presence of specific ground atoms among the consequences of a program, while by non-ground reasoning we mean computing all answers to non-ground queries. For each class we also consider decidability of coherence checks and of class membership. By coherence check we refer to verifying whether the set of stable models of a given program is empty or not. By class membership, instead, we refer to establishing whether a given program (possibly associated with a query) belongs to the class in question. Expressiveness is briefly discussed and, in particular, we highlight whether a class allows for representing every recursive relation or not.

3.1 Bottom-Up Computable Classes

Programs in this group can be finitely instantiated by means of bottom-up procedures. Typically, these procedures are obtained by extending existing ASP grounding techniques. Since all of these bottom-up procedures yield a finite ground program, the stable models can be computed from this ground program by adopting standard techniques developed for ASP without function symbols. Classes in this group, which include $\omega$-restricted, $\lambda$-restricted, finite domain, argument restricted and finitely-ground programs, are represented in Fig. 1; their properties and relationships are discussed throughout the current section.

\footnote{Note that more complex queries can still be expressed using appropriate rules. We assume that each functor appearing in $\mathcal{Q}$ also appears in $\mathcal{P}$.}
3.1.1 \(\omega\)-Restricted Programs [Syrjänen, 2001]

The definition of \(\omega\)-restricted programs is based on the concept of \(\omega\)-stratification, a stratification on predicates (see Section 2.1) in which an extra stratum, defined to be the uppermost stratum, is considered; this extra stratum is called \(\omega\)-stratum, and comprises all predicates involved in cycles with negative dependencies. A normal program is \(\omega\)-restricted if there exists an \(\omega\)-stratification satisfying the following condition: for each rule \(r\), and for each variable \(X\) appearing in \(r\), there is a positive body literal \(\beta \in B^+(r)\) such that \(X\) occurs in \(\beta\), and the predicate of \(\beta\) belongs to a strictly lower stratum than the predicate in the head atom. Extending the concept of \(\omega\)-restriction to general, disjunctive ASP programs is straightforward: the predicate of \(\beta\) must belong to a strictly lower stratum than the predicates of all head atoms.

**Example 3.1.** Let us consider the program \(\mathcal{P}_1\) consisting of

\[
    r_1 : \quad p(f(X)) :- q(X), \text{not } p(X).
\]

and some facts for predicate \(q\) (that are omitted for simplicity). It is easy to see that program \(\mathcal{P}_1\) can be finitely instantiated. Intuitively, the only variable in \(r_1\) is bound by \(q(X)\), which is only defined by facts. We will now show that \(\mathcal{P}_1\) is \(\omega\)-restricted. We start by observing that, in all \(\omega\)-stratifications for \(\mathcal{P}_1\), \(p\) must belong to the \(\omega\)-stratum because it is involved in a negative cycle (actually, a self-loop). Moreover, \(q\) can be assigned to any stratum, in particular to a stratum different from the \(\omega\)-stratum. Thus, \(\mathcal{P}_1\) is \(\omega\)-restricted because \(X\) is bound by \(q(X)\), and \(q\) belongs to a strictly lower stratum than \(p\).

The above restriction ensures that reasoning on \(\omega\)-restricted programs can be performed on a finite instantiation thereof, thus guaranteeing decidability.
of ground reasoning and computability of non-ground reasoning. For the same reason, the coherence check is decidable as well. Concerning class membership, it amounts to determining the existence of an \( \omega \)-stratification for the program at hand, which is easily seen to be decidable as well. However, the restrictions imposed by \( \omega \)-stratifications are fairly strong and cause a loss of expressive power: there are recursive relations that cannot be expressed by \( \omega \)-restricted programs. Reasoning for the class of \( \omega \)-restricted programs has been implemented in SMODELS [Simons et al., 2002] — one of the most popular ASP systems.

### 3.1.2 \( \lambda \)-Restricted Programs [Gebser et al., 2007b]

The notion of \( \lambda \)-restricted program is based on a level mapping that assigns an integer \( \lambda(p) \) to each predicate \( p \) occurring in the program. A normal program is \( \lambda \)-restricted if for any rule \( r \) defining predicate \( p \) (i.e., where \( p \) occurs in the head), each variable occurring in \( r \) is bounded by means of an occurrence of a predicate \( q \) in \( B^+(r) \) such that \( \lambda(q) < \lambda(p) \). Intuitively, this means that the feasible ground instances of \( r \) are completely determined by predicates from levels lower than the one of \( p \).

**Example 3.2.** Let us consider the following program \( P_2 \):

\[
\begin{align*}
r_2 : & \quad q(X) :\neg r(X), p(X). \\
r_3 : & \quad p(f(X)) :\neg q(X).
\end{align*}
\]

where \( r \) is assumed to be defined by facts. Program \( P_2 \) above can be finitely instantiated. Intuitively, the instantiation of \( r_2 \) is finite because \( X \) is bound by \( r(X) \), which trivially has a finite extension. Thus, also the instantiation of \( r_3 \) is finite because of \( q(X) \) in the body. In fact, \( P_2 \) is \( \lambda \)-restricted: a level mapping \( \lambda \) could be \( \lambda(r) = 1, \lambda(q) = 2, \lambda(p) = 3 \).

Program \( P_1 \) in Example 3.1 is \( \lambda \)-restricted as well; a level mapping \( \lambda \) could be: \( \lambda(q) = 1, \lambda(p) = 2 \). On the other hand, \( P_2 \) is not \( \omega \)-restricted: in all possible \( \omega \)-stratifications for \( P_2 \), predicates \( p \) and \( q \) must belong to the same stratum (because of a cyclic, positive dependency). Hence, there is no \( \omega \)-stratification for \( P_2 \) such that variable \( X \) in \( r_3 \) satisfies the condition required by \( \omega \)-restricted programs.

The restriction above ensures a finite instantiation of \( \lambda \)-restricted programs, which in turn ensures decidability of ground reasoning, computability of non-ground reasoning, and decidability of the coherence check. For deciding class membership, the existence of a suitable \( \lambda \) has to be determined, which is easy to see to be decidable as well.

As in the case of \( \omega \)-restricted programs, there are recursive relations that cannot be expressed by \( \lambda \)-restricted programs. Nevertheless, we note that the class of \( \lambda \)-restricted programs strictly contains the class of \( \omega \)-restricted programs. Reasoning and a class membership test for \( \lambda \)-restricted programs have been implemented in the ASP grounder GRINGO [Gebser et al., 2007b].
3.1.3 Finite Domain Programs [Calimeri et al., 2008]

The notion of finite domain program has been introduced for general ASP and is based on syntactic restrictions on the arguments of head atoms. Basically, for an ASP program $\mathcal{P}$, a special dependency graph is defined such that there is a node for each argument of all predicates appearing in $\mathcal{P}$, and there are arcs according to dependencies as described in Section 2.1 — the only difference is that here arguments are considered instead of predicates, and a dependency is introduced only when arguments share a variable within a given rule. A program $\mathcal{P}$ is finite domain if, for each atom $p(t_1, \ldots, t_n)$ in the head of a rule $r \in \mathcal{P}$, and for each argument $p[i]$ of $p$, at least one of the following conditions is satisfied: (i) $t_i$ is variable-free; (ii) $t_i$ appears as a (sub)term of an atom in $B^+ (r)$; (iii) all variables appearing in $t_i$ are bound by argument terms in $B^+ (r)$ which are not recursive with $p[i]$.

Example 3.3. Let us consider the following program $\mathcal{P}_3$:

$$r_4: \quad s(X) :- s(f(X)).$$

Program $\mathcal{P}_3$ above can be finitely instantiated. Intuitively, for each instance of $s(f(X))$, only a finite number of instances of $s(X)$ can be derived by means of $r_4$. In fact, $\mathcal{P}_3$ is a finite domain program: the term $X$ in $s[1]$ appears as a subterm of $s(f(X))$, which belongs to $B^+ (r_4)$ (i.e., condition (ii) holds).

Program $\mathcal{P}_1$ in Example 3.1 is a finite domain program as well because variable $X$ in $p[1]$ is bound by a positive body literal, $q(X)$, which is not recursive with $p$. On the other hand, $\mathcal{P}_2$ in Example 3.2 is a $\lambda$-restricted program which is not finite domain. Indeed, $p[1]$ in $r_3$ does not satisfy any of the required conditions: $f(X)$ contains variables, $f(X)$ does not occur as a (sub)term in $B^+ (r_3)$, and $X$ is only bound by $q[1]$, which is however recursive with $p[1]$. Finally, we observe that $\mathcal{P}_3$ is neither $\omega$-restricted nor $\lambda$-restricted: $s$ depends on itself because of $r_4$.

Finite domain programs can be finitely instantiated. Therefore, for these programs, ground reasoning is decidable and non-ground reasoning is computable, and also the coherence check is decidable. Class membership can be done by checking conditions (i)–(iii) for each argument of head atoms and therefore is clearly decidable.

As for the classes discussed earlier, not all recursive relations can be expressed by finite domain programs. Comparing with the previously introduced classes, the class of finite-domain programs strictly contains the class of $\omega$-restricted programs, while it is incomparable with the class of $\lambda$-restricted programs. Reasoning for finite-domain programs has been implemented in DLV [Calimeri et al., 2008], see also Section 4.

3.1.4 Argument Restricted Programs [Lierler and Lifschitz, 2009]

Similarly to $\lambda$-restricted programs, the definition of argument restricted programs relies on a level mapping $\gamma$, but defined for arguments rather than predicates. A program $\mathcal{P}$ is argument restricted if there is a $\gamma$ such that, for each
atom $p(t_1, \ldots, t_n)$ in the head of a rule $r \in P$, and for each variable $X$ occurring in some argument term $t_i$, there is an atom $q(s_1, \ldots, s_n)$ in $B^+(r)$ such that $X$ occurs in some argument term $s_j$ satisfying the following inequality:

$$\gamma(p[1]) - \gamma(q[j]) \geq d(X, t_i) - d(X, s_j),$$

where $d(X, t)$ is the maximum depth level of $X$ in the term $t$.

**Example 3.4.** Let us consider the following program $P_4$:

$$r_2 : \quad q(X) \leftarrow r(X), \ p(X).$$

$$r_3 : \quad p(f(X)) \leftarrow q(X).$$

$$r_4 : \quad s(X) \leftarrow s(f(X)).$$

Note that $P_4$ is the union of $P_2$ and $P_3$ from Examples 3.2 and 3.3, which have finite instantiations and disjoint predicate symbols. Thus, $P_4$ also possesses a finite instantiation. In fact, we can show that $P_4$ is argument restricted, for instance, by considering the level mapping $\gamma(q[1]) = \gamma(r[1]) = \gamma(s[1]) = 1$, $\gamma(p[1]) = 2$. Indeed, in this case, the following inequalities would be satisfied:

$$r_2 : \quad 0 = 1 - 1 = \gamma(q[1]) - \gamma(r[1]) \geq d(X, X) - d(X, X) = 0 - 0 = 0;$$

$$r_3 : \quad 1 = 2 - 1 = \gamma(p[1]) - \gamma(q[1]) \geq d(X, f(X)) - d(X, X) = 1 - 0 = 1;$$

$$r_4 : \quad 0 = 1 - 1 = \gamma(s[1]) - \gamma(s[1]) \geq d(X, X) - d(X, f(X)) = 0 - 1 = -1.$$

Clearly, $\gamma$ and the above inequalities witness that $P_2$ and $P_3$ are argument restricted as well. Also $P_1$ from Example 3.1, which is argument restricted. For instance, for $\gamma(q[1]) = 1$ and $\gamma(p[1]) = 2$, we obtain

$$r_1 : \quad 1 = 2 - 1 = \gamma(p[1]) - \gamma(q[1]) \geq d(X, f(X)) - d(X, X) = 1 - 0 = 1.$$

On the other hand, it can be shown that $P_4$ is not $\omega$-restricted, $\lambda$-restricted nor finite domain (by the observation made in Examples 3.2 and 3.3 for the rules of $P_2$ and $P_3$).

As for the classes discussed before, argument restricted programs can be finitely instantiated, which implies that ground reasoning is decidable and non-ground reasoning is computable, and that also the coherence check is decidable. Class membership amounts to determining the existence of a suitable level mapping and is decidable as well. As in the previous cases, not all recursive relations can be expressed by programs in this class. However, the class of argument restricted programs include all $\omega$-restricted, $\lambda$-restricted and finite domain programs. At the time of writing, no system implements a class membership check for argument restricted programs. Nonetheless, GRINGO [Gebser et al., 2007b] and DLV are able to compute the finite instantiation of an argument restricted program as a special case of finitely ground program, a broader class discussed next.

### 3.1.5 Finitely Ground Programs [Calimeri et al., 2008]

The definition of finitely ground program relies on the notion of “intelligent instantiation,” obtained by means of an operator which is iteratively applied
on program submodules. In order to properly split a given program $P$ into modules, the \textit{dependency graph} and the \textit{component graph} are considered. The former connects predicate names, and represents dependencies in rules (see Section 2.1), the latter connects strongly connected components (SCC) of the dependency graph. Each module is constituted by all rules defining predicates in a corresponding SCC. An ordering relation is then defined among modules/components: a \textit{component ordering} $\gamma$ for $P$ is a total ordering such that the intelligent instantiation $P_\gamma$, obtained by following the sequence given by $\gamma$, has the same stable models of \textit{grnd}(P). If $P_\gamma$ is finite with respect to each possible component ordering $\gamma$, then $P$ is finitely ground.

\textbf{Example 3.5.} Let us consider the following program $P_5$:

$$r_5 : \quad p(f(g(X))) \leftarrow p(g(X)).$$

Program $P_5$ is finitely ground and can be finitely instantiated. Intuitively, even if $p$ is recursive in $r_5$, instances of $p(f(g(X)))$ (the head atom) and instances of $p(g(X))$ (the body atom) are disjoint, so that the recursion is only apparent. Also $P_4$ from Example 3.4 is finitely ground. In fact, $P_4$ has two independent submodules, $\{r_2, r_3\}$ and $\{r_4\}$: the instantiation of the first submodule is limited by $r(X)$, while for the second submodule a bound is provided by the depth nesting level of functions in $s[1]$. By similar arguments, it can be shown that $P_1$, $P_2$ and $P_3$ are finitely ground. On the other hand, we observe that $P_5$ above is not argument restricted because $X$ appears solely in $p[1]$ and with a greater nesting depth in the head than in the body. Thus, for all possible level mappings $\gamma$, it holds that

$$\gamma\left(p[1]\right) - \gamma\left(p[1]\right) \geq d(X, f(g(X))) - d(X, g(X)) = 2 - 1 = 1.$$

Since $P_5$ is not argument restricted, it does not belong to any of the previously presented classes. In particular, $P_5$ is neither $\omega$-restricted nor $\lambda$-restricted: $p$ depends on itself because of $r_5$; it is not finite domain because $p[1]$ in $r_5$ does not satisfy any of the required conditions: $f(g(X))$ contains variables, $f(g(X))$ does not occur as a (sub)term in $B^+(r_5)$, and $X$ is only bound by $p[1]$ in $B^+(r_5)$, which however recursively depends on itself. \hfill $\square$

As Example 3.5 suggests, the class of finitely ground programs is the most general known class of ASP programs allowing for finite program instantiations, strictly including $\omega$-restricted, $\lambda$-restricted, finite domain and argument restricted programs. Since the instantiation is guaranteed to be finite, ground reasoning is decidable and non-ground reasoning computable, and also the coherence check is decidable. In addition, it has been proved that all recursive relations can be expressed by finitely-ground programs [Calimeri et al., 2008]. However, this comes at a price: checking whether a program is finitely ground is semi-decidable in general. Finitely ground programs have been effectively implemented in DLV [Calimeri et al., 2008], which is now able to finitely compute all answer sets of any such program.\footnote{We recall here that a strongly connected component of a directed graph is a maximal subset $C$ of its vertices, such that each vertex in $C$ is reachable from all other vertices in $C$.}

\footnote{As a recent addition, GRINGO 3.0.X introduced a semi-decidable grounding procedure.}
3.2 Top-Down Computable Classes

The scientific community has also proposed some classes of programs specifically designed for query answering, and thus typically characterized by top-down computation schemata. Programs in these classes usually feature an infinite number of answer sets, each of which may contain an infinite number of atoms. In order to guarantee decidability of reasoning, a finite number of atoms must be sufficient for answering a query. Moreover, these atoms have to be effectively identified. Classes in this group, namely FP2 programs, positive and stratified finitely recursive programs, and finitary programs, are depicted in Fig. 2; their properties and relationships are discussed throughout the current section.

3.2.1 FP2 Programs [Baselice and Bonatti, 2010]

The class of FP2 programs has been defined for normal programs only. The definition relies on two key concepts: recursion patterns and call-safeness. A recursion pattern $\pi$ is a function mapping each predicate $p$ to a subset of the arguments of $p$. Essentially, a recursion pattern for a program $P$ ensures that the dependency graph of $\text{grnd}(P)$ is such that: (i) no cycle of dependencies is an odd cycle; and (ii) every path contains finitely many different atoms. A program is call-safe with respect to a recursion pattern $\pi$ if, in a top-down computation, variables appearing in negative subgoals or in an argument of a subgoal selected by $\pi$ are bound by previous resolved subgoals. Hence, a normal program $P$ belongs to FP2 if there is a recursion pattern $\pi$ such that $P$ is call-safe with respect to $\pi$. 
Example 3.6. Consider the following program $P_6$:

\begin{align*}
  r_6 : & \quad p(X) :- \neg q(X). \\
  r_7 : & \quad q(X) :- \neg p(X). \\
  r_8 : & \quad p(f(X)) :- p(X).
\end{align*}

The program is FP2. Indeed, a recursion pattern can be obtained by mapping $p$ and $q$ on their first, and unique, arguments; in addition, all rules are call-safe because, in each of them, the unique variable $X$ appears in a selected argument of the head (see Definition 5.4 of Baselice and Bonatti [2010]). Hence, ground and non-ground reasoning over $P_6$ is computable with respect to any query atom. \Halmos

FP2 programs have decidable ground reasoning and class membership, but cannot express all computable sets or relations. The coherence check is trivial (FP2 programs are odd-cycle-free). Non-ground reasoning is uncomputable.

3.2.2 Positive and Stratified Finitely Recursive Programs [Calimeri et al., 2009, Alviano et al., 2010]

Finitely recursive programs are based on a notion of relevant atoms (or sub-queries). The relevant atoms for a ground query $Q$ with respect to a program $P$ are defined as follows: (i) $Q$ itself is relevant; (ii) if a ground atom $\alpha$ is relevant and $r$ is a rule in $\text{grnd}(P)$ having $\alpha$ in its head, all atoms appearing in $r$ are relevant. A ground query $Q$ is finitely recursive on a program $P$ if the number of relevant atoms for $Q$ with respect to $P$ is finite. A program $P$ is finitely recursive if every ground query $Q$ is finitely recursive on $P$.

Example 3.7. Consider the query atom $p(f(f(0)))$ for the following program $P_7$:

\begin{align*}
  r_8 : & \quad p(f(X)) :- p(X). \\
  r_9 : & \quad p(X) \lor q(X).
\end{align*}

Both the query and the program are finitely recursive. In fact, it can be observed that a bound to the depth nesting level of functors of all relevant atoms is implicitly provided by queries. The program is not FP2 because $r_8$ contains disjunction. On the other hand, $P_6$ from Example 3.6 contains recursive negation (see $r_6$ and $r_7$), and so it is not a positive or stratified finitely recursive program. \Halmos

Decidability of ground reasoning has been proved for disjunctive positive programs and generalized to the case of disjunctive programs with stratified negation, while non-ground reasoning remains undecidable in general because an infinite number of queries would have to be considered in this case. In proving these results, an interesting link between top-down and bottom-up computable classes has been established: positive and stratified finitely recursive programs can be mapped to equivalent finitely ground programs by applying a Magic Set rewriting [Alviano et al., 2010].

Recognizing whether a query belongs to one of these classes is a semi-decidable task, while coherence checking is trivially decidable (positive and
stratified programs are coherent). Concerning expressiveness, finitely recursive programs are sufficient to express all computable relations, even if disjunction and negation are forbidden. Positive finitely recursive programs are clearly contained in the class of stratified finitely recursive programs, while the class of FP2 programs is incomparable with respect to subset-inclusion.

3.2.3 Finitary Programs [Bonatti, 2002, 2004]

Finitary programs are a subset of finitely recursive programs. The class has been originally defined for normal programs and subsequently enlarged for allowing disjunctive heads. A program $P$ is finitary if the following conditions are satisfied: (i) $P$ is finitely recursive; (ii) only a finite number of odd cycles are present in the dependency graph of $grnd(P)$; (iii) only a finite number of head cycles are present in the dependency graph of $grnd(P)$.

Example 3.8. Programs $P_6$ and $P_7$ from Examples 3.6 and 3.7 are finitary. Consider now the following program $P_8$:

$r_8 : p(f(X)) :- p(X)$.

$r_9 : p(X) \lor q(X)$.

$r_{10} : p(X) :- q(X)$.

$r_{11} : q(X) :- p(X)$.

Note that $P_8$ is a positive finitely recursive program. Note also that the program is not finitary. Indeed, infinitely many head cycles are present in $grnd(P_8)$ because of $r_9$, $r_{10}$, and $r_{11}$ (even if only a finite number of these cycles are relevant for answering a given ground query).

Strictly speaking, knowing that a program is finitary does not guarantee decidability of ground reasoning. In fact, in order to ensure decidability, odd cycles and head cycles have to be provided as an additional input together with the program. Also class membership and the coherence check are undecidable for finitary programs, and non-ground reasoning is uncomputable. Finitary programs include FP2 programs, while they are incomparable with respect to positive and stratified finitely recursive programs.

3.3 Classes with Finitely Representable Stable Models

Classes in this group are FDNC and bidirectional programs, which are characterized by infinite stable models having a finite representation in the shape of a forest of trees.

3.3.1 FDNC Programs [Simkus and Eiter, 2007]

The class of FDNC programs allows for function symbols, disjunction, non-monotonic negation, and constraints. In order to retain the decidability of ground reasoning, rules have to be of the shape of one of seven predefined rules.

---

$^6$A head cycle is a cycle of dependencies involving a pair of ground atoms occurring in the head of the same rule.
schemata. In particular, FDNC constraints function symbols to be unary (or constants), and predicates to be unary or binary. These syntactic restrictions ensure that programs have a forest-shaped model property, which means that stable models of FDNC programs are in general infinite, but have a finite representation that can be exploited for knowledge compilation and fast query answering.

Ground reasoning, coherence check and class membership are decidable for FDNC programs. Non-ground reasoning is also computable, but there are recursive relations that cannot be expressed by programs in this class. We also note that the restrictions imposed on the syntax of FDNC programs require that atoms occurring in rule heads have to be structurally simpler than atoms in rule bodies. These limitations have considerable impact on practical domains like reasoning about actions. In this context, indeed, rules of FDNC programs do not allow to refer naturally to the past.

3.3.2 Bidirectional Programs [Eiter and Simkus, 2009a]

Bidirectional programs are a close relative to FDNC programs, but allow for referring to both past and future events when reasoning about actions. The restrictions ensuring decidability of ground reasoning for bidirectional programs are based on the notion of t-atoms. An atom $\alpha$ is a t-atom if the first argument of $\alpha$ is $t$ and each other argument of $\alpha$ is either a constant, or a variable not occurring in $t$. Hence, a safe program $\mathcal{P}$ is bidirectional if, for each $r \in \mathcal{P}$, there is a variable $X$ such that each atom in $r$ is either an $X$-atom, or an $f(X)$-atom, or a c-atom, where $f$ is a unary functor and $c$ a constant (fixed for all rules). Note that the definition of bidirectional programs does not limit the arity of predicates, while function symbols are constrained to be unary (or constants).

Ground reasoning, coherence check and class membership are decidable for bidirectional programs. Non-ground reasoning is also computable, but there are recursive relations that cannot be expressed by programs in this class. Note that FDNC and bidirectional programs are incomparable as there are FDNC programs that are not bidirectional and vice versa (see the example below).

Example 3.9. The following is an FDNC program that is not bidirectional:

$$
p(X, f(X)) :- q(X).
$$

Indeed, the structure of the rule above is allowed in FDNC, while bidirectional programs allow for using function symbols of positive arity only in the first argument of atoms. The following, instead, is a bidirectional program not belonging to FDNC:

$$
p(X) :- q(f(X)).
$$

Indeed, the rule above is allowed in bidirectional programs because $p(X)$ is an $X$-atom and $q(f(X))$ is an $f(X)$-atom, while it is forbidden in FDNC (in FDNC programs, depth levels of variables must be bound by body atoms).
3.4 Discussion

The main properties of each group of program classes are summarized in this section. Classes in the first group, graphically represented in Figure 1, are characterized by finite program instantiation, which in turn implies decidability of ground reasoning and coherence check, as well as computability of non-ground reasoning. These classes are particularly relevant for ASP because current systems can be easily adapted for computing stable models of programs in this group. Containment relationships between classes in this group are highlighted in Figure 1. The smallest class are $\omega$-restricted programs, which is strictly contained in the intersection of $\lambda$-restricted and finite domain programs. The most general known class that can be syntactically recognized are argument restricted programs, a strict superset of $\lambda$-restricted and finite domain programs. Argument restricted programs are strictly contained in finitely ground programs, the most general known fragment of ASP in this group. A strength of finitely ground programs is its expressive power: all recursive relations can be expressed by finitely ground programs. However, this class cannot be recursively separated from the full ASP language. For this reason, the boundary between finitely ground and ASP programs is dashed in Figure 1.

Programs in the second group are suitable for finite top-down evaluations of queries and allow for reasoning about infinite stable models. Containment relationships between classes in this group are represented in Figure 2. We note that FP2 is strictly contained in finitary programs, while positive finitely recursive programs are clearly contained in stratified finitely recursive programs. Note also that FP2 is the only syntactically recognizable class in this group (represented by a solid boundary in Figure 2). The other classes, instead, allow for expressing all recursive relations, even if only positive and stratified finitely recursive programs guarantee decidability of ground reasoning. In fact, the evaluation of finitary programs requires knowing the set of all odd and head cycles, which is not known to be computable. Moreover, class membership is semi-decidable for positive and stratified finitely recursive programs (visualized by dashed boundaries in Figure 2), while it is undecidable for finitary programs (dotted boundary in Figure 2). We recall that an interesting link between top-down and bottom-up computable classes has been established in the literature [Calimeri et al., 2009, Alviano et al., 2010]: positive and stratified finitely recursive programs can be mapped to equivalent finitely ground programs by means of a Magic Set rewriting.

The third group contains FDNC and bidirectional programs, characterized by forest-shaped stable models guaranteeing a finite representation. Ground reasoning, coherence check and class membership are decidable for these programs, and non-ground reasoning is computable. However, FDNC and bidirectional programs do not allow for representing all recursive relations.

Table 3.4 summarizes the most relevant properties discussed in this section. All classes guarantee decidability of reasoning for ground queries (ground reasoning), even if odd and head cycles have to be provided in input for ensuring decidability of ground reasoning for finitary programs. For non-ground queries (non-ground reasoning), instead, the set of answer substitutions can be effectively computed only by programs in the first and third group, which are
Table 1: Classification of ASP fragments with decidable reasoning

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ω-Restricted</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>λ-Restricted</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>Finite Domain</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>Argument Restr.</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>Finitely Ground</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>yes</td>
<td>semi-decid.</td>
</tr>
<tr>
<td>FP2</td>
<td>decid.</td>
<td>uncomp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>Finitary</td>
<td>decid.*</td>
<td>uncomp.</td>
<td>undecid.</td>
<td>yes</td>
<td>undecid.</td>
</tr>
<tr>
<td>FDNC</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>decid.</td>
<td>comp.</td>
<td>decid.</td>
<td>no</td>
<td>decid.</td>
</tr>
</tbody>
</table>

* If odd cycles and head cycles are provided as input.

also characterized by a decidable coherence check. Coherence is also decidable (actually, trivial) for FP2 and positive/stratified finitely recursive programs. The possibility of expressing all recursive relations (recursive completeness) is preserved by finitary, finitely ground and positive/stratified finitely recursive programs. However, while class membership (i.e., establishing whether a given program belongs to the class in question) is undecidable for finitary programs, it is semi-decidable for finitely ground and positive/stratified finitely recursive programs.

4 An ASP System Supporting Uninterpreted Function Symbols: DLV

The DLV system [Leone et al., 2006] is widely considered one of the state-of-the-art implementations of ASP. Since the first stable release, dated back in 1997, the DLV system has been significantly improved over and over, both thanks to the implementation of various optimization techniques and to the enrichment of its language that supports the most important ASP extensions such as aggregates and weak-constraints. DLV has been widely used in many practical application scenarios, including data integration [Leone, Gottlob, Rosati, Eiter, Faber, Fink, Greco, Ianni, Kalka, Lenbo, Lenzerini, Lio, Nowicki, Ruzzi, StaniszkiS, and Terracina, 2005], semantic-based information extraction [Manna, Ruffolo, Oro, Alviano, and Leone, 2011a, Manna, Scarcello, and Leone, 2011b], e-tourism [Ricca, Alviano, Dimasi, Grasso, Ielpa, Iiritano, Manna, and Leone, 2010], workforce management [Ricca, Grasso, Al-
viano, Manna, Lio, Iiritano, and Leone, 2011], and many more.

Beginning with version 2010–10–14, DLV supports a powerful (possibly recursive) use of function symbols, and list and set terms [Calimeri et al., 2008].

Within DLV, all recursive relations are expressible in a rich and fully declarative language. Termination is guaranteed on all finitely ground programs (which include all other program classes discussed in Section 3.1), or for all programs passing some syntactic check that can be performed on demand.

4.1 Language Overview

By supporting function symbols, DLV allows for aggregating atomic data, manipulating complex data structures and generating new symbols (value invention — see Calimeri, Cozza, and Ianni [2007] for a discussion). Strings, lists, sets, trees, and many other common data structures are representable by means of functions. In particular, for list and set terms, DLV offers an explicit notation. This and all the other features of DLV, such as aggregates and weak constraints, provide a solid basis for natural knowledge representation.

4.1.1 Function Symbols

Function symbols are supported by DLV with the syntax reported in this article. Strings, stacks, trees, and many other common data structures can be represented by means of function symbols. For instance, binary trees might be encoded by a function symbol \( bt \) of arity 3, where the first argument is associated with the root, and the other two stand for the left and the right subtrees. As an example, under such assumptions, the functional term \( bt(eq, bt(su, bt(x, ⊥, ⊥), bt(y, ⊥, ⊥)), bt(su, bt(y, ⊥, ⊥), bt(x, ⊥, ⊥))) \) can be seen as a binary tree representing the structure of the formula \( x + y = y + x \) (here, \( ⊥ \) is a constant used for denoting the empty tree).

Example 4.1. Let \( F \) be a set of binary trees. Suppose we are interested in determining all subsets \( S \) of \( F \) such that \( S \) does not contain trees \( t_1, t_2 \) such that \( t_1 \) is a subtree of \( t_2 \). Assuming that the input set \( F \) is encoded by instances of a unary predicate \( tree \), all such subsets can be determined by means of the following program:

\[
\begin{align*}
  r_{12} & : \text{in}(T) \lor \text{out}(T) :- \text{tree}(T). \\
  r_{13} & : \text{in}(T_1), \text{in}(T_2), \text{subtree}(T_1, T_2), T_1 \neq T_2. \\
  r_{14} & : \text{subtree}(T, T) :- \text{tree}(T). \\
  r_{15} & : \text{subtree}(L, T) :- \text{subtree}(bt(X, L, R), T). \\
  r_{16} & : \text{subtree}(R, T) :- \text{subtree}(bt(X, L, R), T).
\end{align*}
\]

In particular, possible subsets \( S \) of \( F \) are guessed by \( r_{12} \), and those subsets not fulfilling the required condition are discarded by \( r_{13} \). Pairs of subtrees are discarded by \( r_{15} \) and \( r_{16} \).

---

7 Note that the notation for lists and sets are currently only available in the DLV-Complex branch. This functionality is scheduled to be taken over as soon as possible to the main distribution.

8 Note that rules with empty heads (i.e., constraints) are not allowed by the syntax presented in Section 2.1. Intuitively, \( r_{13} \) is equivalent to the following rule: \( \text{co} :- \text{in}(T_1), \text{in}(T_2), \text{subtree}(T_1, T_2), T_1 \neq T_2, \text{not co} \), where \( \text{co} \) is a fresh symbol.
determined by means of the rules $r_{14}$, $r_{15}$ and $r_{16}$. Assuming that $F$ contains two trees, $t_1 = bt(a, bt(b, \perp, \perp), bt(c, \perp, \perp))$ and $t_2 = bt(b, \perp, \perp)$, the following subsets are determined: $\emptyset \{t_1\} \{t_2\}$. Thus, the only subset of $F$ not fulfilling the required condition is $F$ itself. Indeed, $t_2$ is a subtree of $t_1$. □

4.1.2 Lists

Lists can be profitably applied in order to model collections of objects in which position matters and repetitions are allowed. Some examples of list terms are $[\text{jan}, \text{feb}, \text{mar}]$ and $[1, 1, i, s, t]$. Lists can be nested arbitrarily, for instance, the following are valid list terms: $[[\text{jan}, 31], [\text{feb}, 28], [\text{mar}, 30]]$. Lists can be profitably applied in order to model collections of objects in which position matters and repetitions are allowed. Some examples of list terms are

determined by means of the rules $r_{14}$, $r_{15}$ and $r_{16}$. Assuming that $F$ contains two trees, $t_1 = bt(a, bt(b, \perp, \perp), bt(c, \perp, \perp))$ and $t_2 = bt(b, \perp, \perp)$, the following subsets are determined: $\emptyset \{t_1\} \{t_2\}$. Thus, the only subset of $F$ not fulfilling the required condition is $F$ itself. Indeed, $t_2$ is a subtree of $t_1$. □

Lists can be profitably applied in order to model collections of objects in which position matters and repetitions are allowed. Some examples of list terms are $[\text{jan}, \text{feb}, \text{mar}]$ and $[1, 1, i, s, t]$. Lists can be nested arbitrarily, for instance, the following are valid list terms: $[[\text{jan}, 31], [\text{feb}, 28], [\text{mar}, 30]]$; $[[t, h, i, s], [i, s], [a], [1, i, s, t]]$. An element $t$ can be appended to the front of a list $l$ by using the “à la Prolog” syntax, i.e., $[t]l$. For example, the list $[\text{jan}, \text{feb}, \text{mar}]$ can be equivalently written as $[\text{jan}][\text{feb}][\text{mar}]$.[3]

**Example 4.2.** A palindrome is a phrase that can be read the same way in either direction (ignoring punctuation and spaces). Examples of palindromes are the following phrases: “Was it a rat I saw?” and “Ai lati d’Italia.” Assume that a set of phrases is given by means of a predicate phrase, where each phrase is encoded by a list of characters (for simplicity, all characters in these lists are lowercase letters); for example, let us assume that the input comprises the following phases: phrase($[w, a, s, i, t, a, r, a, t, i, s, a, w]$), phrase($[a, i, l, a, t, i, d, i, t, a, l, i, a]$) and phrase($[n, o, t, a, p, a, l, i, n, d, r, o, m, e]$). The following program determines palindromes among input phrases:

\[
\text{palindrome}(X) \leftarrow \text{phrase}(X), \text{reverse}(X) = X.
\]

Note that #reverse is an interpreted built-in function. Intuitively, for a list $l$, the term #reverse($l$) is a list comprising the elements of $l$ in reversed order. In our example, the first two phrases are palindromes, while the last one is not. Thus, the unique stable model of the program above contains palindrome($[w, a, s, i, t, a, r, a, t, i, s, a, w]$) and palindrome($[a, i, l, a, t, i, d, i, t, a, l, i, a]$). □

**Example 4.3.** Let $G$ be an undirected graph. A path is a sequence of nodes $x_1, \ldots, x_n$ such that for each $i = 1, \ldots, n - 1$ there is an edge connecting $x_i$ and $x_{i+1}$ in $G$. An simple path is a path without repetition of nodes. Assuming that edges of $G$ are represented by instances of edge, the following program derives all simple paths in $G$:

\[
\text{path}([X, Y]) \leftarrow \text{edge}(X, Y).
\]

\[
\text{path}([X][Y][W]) \leftarrow \text{edge}(X, Y), \text{path}([Y][W]), \text{not} \text{member}(X, [Y][W]).
\]

Note that #member is an interpreted built-in predicate. Intuitively, for a term $t$ and a list $l$, the atom #member($t, l$) is true if and only if $t$ is an element in $l$. □

4.1.3 Sets

Set terms are used to model unordered collections of data, in which no duplicates are allowed. Examples of set terms are $\{a, b, c\}$ and $\{\text{alice}, \text{bob}, \text{alice}, \text{bob}, \text{alice}, \text{bob}\}$. For example, the list $[\text{jan}, \text{feb}, \text{mar}]$ can be equivalently written as $[\text{jan}][\text{feb}][\text{mar}]$.[3]
Figure 3: An undirected graph

Example 4.4. Let \( G = (V, E) \) be an undirected graph, where \( V \) is a set of vertices and \( E \) a set of edges. A dominating set for a graph \( G \) is a subset \( D \) of \( V \) such that every vertex in \( V \setminus D \) is joined to a vertex of \( D \) by an edge in \( E \).

Assume that the input graph \( G \) is represented by instances of \texttt{adjacent}(v,a), where \( v \) is a vertex and \( a \) the set of its adjacent vertices; for example, let us assume that the following set of facts, representing the graph in Figure 3, is given:

\[
\begin{align*}
\text{adjacent}(a, \{b, c\}), \text{adjacent}(b, \{a, c, d\}), \text{adjacent}(c, \{a, b, e\}), \\
\text{adjacent}(d, \{b, e, f\}), \text{adjacent}(e, \{c, d\}), \text{adjacent}(f, \{d\})
\end{align*}
\]

All dominating sets of \( G \) can be determined by means of the following program:

\[
\begin{align*}
r_{17} : & \quad \text{in}(X) \lor \text{out}(X) : \leftarrow \text{adjacent}(X, A). \\
r_{18} : & \quad \text{dominated}(X) : \leftarrow \text{out}(X), \text{in}(Y), \text{adjacent}(Y, A), \#\text{member}(X, A). \\
r_{19} : & \quad : \leftarrow \text{out}(X), \text{not dominated}(X).
\end{align*}
\]

Intuitively, \( r_{17} \) guesses a subset \( D \) of vertices, \( r_{18} \) determines dominated nodes, and \( r_{19} \) discards \( D \) if not a dominating set. In our example, \( \{a, e, f\}, \{c, d\}, \{b, d\} \) and all their supersets are dominating sets. \( \square \)

4.2 Implementation Issues

4.2.1 Overview

The system architecture of DLV is shown in Figure 4. The \textit{Rewriter} module is in charge of substituting all (non-constant) functional terms occurring in the input program, by introducing appropriate predefined built-in predicates (details are reported below in this Section). The rewritten program is then passed to the \textit{Finite Checker} module in order to verify the membership to a
class for which termination is guaranteed (see Section 3.1). In fact, these two
modules are essentially the only relevant ones for enabling DLV to deal with
function symbols. For descriptions of the remaining modules we refer to Leone
et al. [2006] for basic DLV and to Calimeri et al. [2007] for its extension by
external built-ins.

Currently, main-line DLV checks for “strong safety,” which is slightly more
restrictive than the class of finite domain programs, while the DLV-Complex
branch has a check for membership in the class of finite domain programs. It
is planned to provide a check for argument restriction [Lierler and Lifschitz,
2009] in the near future. If the program is guaranteed to have a finite in-
stantiation, answer sets can be computed as usual by processing the resulting
ground program. The Finite Checker module can be skipped by specifying the
command-line option -nofinitecheck. In this case, termination is only guar-
anteed for finitely ground programs (see Section 3.1.5), which however cannot
be characterized in terms of a syntactic restriction. It is also possible to limit
the nesting level of functional terms in the generated ground program, and
in turn, ensure the termination of the instantiation process, by means of the
command-line option -MAXNL=<N>.

Figure 4: System Architecture
4.2.2 Rewriter

For each rule \( r \) in \( \mathcal{P} \), each occurrence of a functional term of the form \( f(X_1,\ldots,X_k) \) in \( r \), where \( X_1,\ldots,X_k \) are variables, is replaced by a fresh variable \( F \). Moreover, depending on whether \( f(X_1,\ldots,X_k) \) appears in the head or in the body of \( r \), one of the following atoms is added to \( B^+(r) \):

- \( \texttt{function.pack}(f,X_1,\ldots,X_k,F) \) if \( f(X_1,\ldots,X_k) \) appears in \( H(r) \);
- \( \texttt{function.unpack}(F,f,X_1,\ldots,X_k) \) if \( f(X_1,\ldots,X_k) \) appears in \( B(r) \).

This process is repeated until no functional terms of the form \( f(X_1,\ldots,X_k) \) appear in \( r \). Note that \texttt{function.pack} and \texttt{function.unpack} are built-in predicates [Calimeri and Ianni, 2005].

Example 4.5. Let us consider the following rule:

\[
p(f(f(X))) := q(X, g(X,Y)).
\]

According to the process described earlier, the rule above is rewritten as follows:

\[
p(F_2) := q(X,F_1), \texttt{function.unpack}(F_1,g,X,Y), \texttt{function.pack}(F_2,f,X), \texttt{function.pack}(F_3,f,F_2).
\]

Note that two \texttt{function.pack} atoms have been introduced in order to represent the nested functional term \( f(f(X)) \): \texttt{function.pack}(F_2,f,X) uses the variable \( F_2 \) to refer to the functional term \( f(X) \); \texttt{function.pack}(F_3,f,F_2), instead, refers to \( f(F_2) = f(f(X)) \) by means of \( F_3 \).

4.2.3 Intelligent Grounding

Rewritten programs are instantiated by means of standard techniques (the reader may refer, for instance, to Leone et al. [2006]). In this section, we only describe how the built-in predicates \texttt{function.pack} and \texttt{function.unpack}, introduced by the Rewriter module, are interpreted during the instantiation phase.

Predicate \texttt{function.pack} is used to build ground functional terms. In particular, for a function symbol \( f \) of arity \( k \), \( k \) ground functional terms \( t_1,\ldots,t_k \) and a variable \( F \), the built-in atom \texttt{function.pack}(f,t_1,\ldots,t_k,F) causes the creation of a ground functional term \( f(t_1,\ldots,t_k) \), which is assigned to the variable \( F \). The second predicate, instead, is used to disassemble functional terms. In particular, for a function symbol \( f \) of arity \( k \), \( k \) ground functional terms \( t_1,\ldots,t_k \) and \( k \) variables \( X_1,\ldots,X_k \), the atom \texttt{function.unpack}( f(t_1,\ldots,t_k), f,X_1,\ldots,X_k) assigns the ground functional term \( t_i \) to the variable \( X_i \), for each \( i \in \{1,\ldots,k\} \). If some variable \( X_i \) has been previously bound to a ground functional term \( t \), \texttt{function.unpack} simply checks that \( t \) is equal to \( t_i \).

\[\text{List and set terms are treated by means of pack and unpack built-in predicates, which act analogously to function.pack and function.unpack.}\]

21
5 Conclusion

ASP is a highly expressive language that has found many application scenarios. However, until fairly recently, practical ASP was limited to express properties over finite structures. This major deficiency of the language prevented the representation of recursive structures, which are common in computer science. The introduction of uninterpreted function symbols allows for circumventing this problem. However, the common reasoning tasks become undecidable if functions are permitted without any restrictions. Many relevant classes of programs with function symbols for which decidability of reasoning is guaranteed were identified in the past decade. The most popular of these have been discussed in this paper. In particular, three groups of classes have been considered: programs allowing for finite bottom-up evaluations (\(\omega\)-restricted [Syrjänen, 2001], \(\lambda\)-restricted [Gebser et al., 2007b], finite domain [Calimeri et al., 2008], argument restricted [Lierler and Lifschitz, 2009] and finitely ground programs [Calimeri et al., 2008]), programs suitable for finite top-down evaluations (\(FP_2\) [Baselice and Bonatti, 2010], positive/stratified finitely recursive [Calimeri et al., 2009, Alviano et al., 2010] and finitary programs [Bonatti, 2002, 2004]) and programs characterized by finitely representable stable models (\(FDNC\) [Simkus and Eiter, 2007] and bidirectional programs [Eiter and Simkus, 2009a]). The main properties of these program classes and their interrelationships have been analyzed and summarized. Finally, a system that can deal with programs containing function symbols has been presented, which allows for terminating reasoning over programs of several of the presented program classes.

References


