Decidable fragments of Logic Programming with Value Invention

Francesco Calimeri and Susanna Cozza and Giovambattista Ianni*

Abstract. The issue of value invention in logic programming embraces many scenarios, such as logic programming with function symbols, object oriented logic languages, inter-operability with external sources of knowledge, set unification. This paper introduces a framework embedding value invention in a general context. The class of programs having a suitable (but, in general, not decidable) 'finite grounding property' is identified, and the class of 'value invention restricted' programs is introduced. Value invention restricted programs have the finite grounding property and can be decided in polynomial time. They are, in a sense, the broadest polynomially decidable class having this property, whenever no assumption can be made about the nature of invented values (while this latter is the case in the specific literature about logic programming with function symbols). Relationships with existing formalisms are eventually discussed; in particular, value invention restricted programs subsume ω-restricted programs and are incomparable with finitary programs.

1 Introduction

The notion of 'value invention' has been formerly adopted in the database field (see e.g. [1, 2]) for denoting those mechanisms aimed at allowing to introduce new domain elements in a logic based query language. Indeed, applications of logic programming often need to deal with a universe of symbols which is not a priori known. We can divide these demands in two main categories: *(i)* 'Constructivist' demands: the majority of logic programming languages has indeed the inherent possibility to build new symbols from pre-existing ones, e.g. by means of traditional constructs like functional terms. Manipulating and creating complex data structures other than simple constant symbols, such as sets, lists, is also a source of value invention. Also, controlled value invention constructs have been proposed in order to deal with the creation of new object identifiers in object oriented deductive databases [3]. *(ii)* 'Externalist' demands: in this setting, non-predictable external sources of knowledge have to be dealt with. For instance, in the Semantic Web area, rule based languages must explicitly embrace the case where ontologies and the universe of individuals is external and not a priori known [4], or is explicitly assumed to be open [5].

Whatever popular semantics is chosen for a rule based logic program (*well-founded, answer set, first order*, etc.), both of the above settings are a source of undecidability difficult to cope with.

Top down solvers (such as SLD solvers), do not usually address this issue, and leave to the programmer the burden of ensuring termination. Also, the

* Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy. e-mail:{calimeri,cozza,ianni}@mat.unical.it.

programmer needs a good knowledge of the evaluation strategy implemented in her adopted system, since termination is often algorithm dependent. *Bottom up solvers* (such as DLV or Smodels for the Answer Set Semantics [6, 7]), and in general, languages derived from Datalog, are instead conceived for ensuring algorithm independent decidability and full declarativity.

To this aim, the implementation of such languages relies on the explicit choice of computing a ground version of a given program. In a context where value invention is explicitly allowed, grounding a program against an infinite set of symbols leads to an infinite ground program which cannot be built in practice.

The paper adopts the notion of `VI` programs, which are logic programs enriched with the notion of *external predicate* [8]. External predicates model the mechanism of value invention by taking input from a given set of values and returning (possibly newly invented) values. These latter are computed by means of an associated evaluation function (called *oracle*).

In [8] we proved that, although assuming as decidable the external functions defining oracles, the consistency check of `VI` programs is, in general, undecidable.

Thus, it is important to investigate on nontrivial sub-classes of decidable programs. This problem is not addressed satisfactorily in the above paper, which is mainly focused on the operational and declarative properties of the framework and its technical realizability. Indeed, a very strict safety condition for granting decidability of `VI` programs is therein given.

The contributions of the paper are overviewed next:
– We introduce a *safety* condition defining the class of 'value invention restricted' (`VI`-restricted, in the following) programs. This class enjoys the *finite grounding property*, characterizing those programs that can be computed with a finite ground program. Decidability of consistency checking is thus ensured (Section 4).
– The `VI`-restrictedness condition is less restraining than previously introduced syntactic restrictions (such as ω-restricted programs [9] or semi-safe programs [8]). The programmer is thus relieved from the burden of introducing explicit syntactic modifications. However, `VI`-restrictedness can be checked in time polynomial in the size of the non-ground program (Section 5).
– The above condition is generic: no assumption is made on the structure of new invented symbols. Indeed, `VI` programs embed settings such as programs with function symbols, programs with sets (in general logic languages with a generalized notion of unification), or with external constructs (Section 6).
– `VI`-restricted programs subsume the class of ω-restricted programs [9]. Finitary programs [10], a class of programs with answer set semantics and function symbols, are not directly comparable with `VI`-restricted programs. Also, our former definition of semi-safe programs [8] is subsumed (Section 7).
– Our framework relies on the traditional notion of ground program. Thus, results about `VI`-restricted programs can be be adapted to semantics other than Answer Set Programming, such as the Well-Founded Semantics.

2 Motivating example

The Friend of a Friend (FOAF) [11] project is about creating a Web of machine-readable homepages describing people, the links between them and the things

they create and do. It is an RDF/XML Semantic Web vocabulary. Each person P stores its FOAF ontology portion at some url U.

In order to reason on this vocabulary, a rule based logic language would need some special construct for importing this external knowledge. The aim of this language is anyway to keep decidability and declarativity. So it is important not to rely on an operational semantics for the language. In this spirit, [4] introduces a form of *external predicates*, very similar to ours.

Imagine we want to perform the transitive closure of the relation of knowledge among people, starting from the homepage of a given person. Let's suppose to have an external predicate called "*#rdf*" which allows us to access a FOAF ontology located at URL:

$$\#rdf(URL, Object_1, Relation, Object_2).$$

We first collect a set of homepages. In order to avoid wrong information we can accept only a restricted subset of somehow *trusted* urls. Then we simply encode the transitive closure as usual, exploiting the knowledge provided by the collected pages. Let the starting homepage be "myurl"; thus, the following program implements what described above.

$$trusted(X,U) \leftarrow \#rdf(\text{“}myurl\text{”}, X, \text{“}trusts\text{”}, U). \quad (1)$$
$$url(X,U) \leftarrow \#rdf(\text{“}myurl\text{”}, X, \text{“}seealso\text{”}, U),\ trusted(X,U). \quad (2)$$
$$url(X,U) \leftarrow url(\_, U1), \#rdf(U1, X, \text{“}seealso\text{”}, U),\ trusted(X,U). \quad (3)$$
$$connected(X,Y) \leftarrow url(X,U),\ \#rdf(U, X, \text{“}knows\text{”}, Y). \quad (4)$$
$$connected(X,Y) \leftarrow connected(X,Z),\ url(Z,U),\ \#rdf(U, Z, \text{“}knows\text{”}, Y). \quad (5)$$

The above program has two sources of new values: trusted urls, and persons. For instance, in particular the fifth rule may induce a loop, leading to the invention of an infinite number of new symbols. The above program is anyway `VI`-restricted and can be solved over a finite ground version of it. Intuitively, the number of URLs is finite. Although not explicitly bounded, new persons (coming from the value of Y in the fifth rule) can be extracted only from a finite set of URLs. Observe that rule 1 *invents* new values, but these do not ever propagate through a loop involving an external atom, while this is the case of the Y variable in the fifth rule. The intuition of `VI`-restricted programs is to investigate how new information propagates in a program, and whether it is bounded in some way. Note that a programmer is not explicitly forced (in order to ensure decidability) to bound variables explicitly such as in this modified version of the fifth rule: $\{\ connected(X,Y) \leftarrow known(Y), connected(X,Z), url(Z,U), \#rdf(U, Z, \text{“}knows\text{”}, Y).\ \}$.

3 Preliminaries

In this section we briefly recall some notions which we introduced in [8]. Our framework coincides basically with Answer Set Programming, extended with the notion of *external atom*.

Let $\mathcal{U}$, $\mathcal{X}$, $\mathcal{E}$ and $\mathcal{P}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, *external predicate names*, and *ordinary predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{U}$) are denoted with first letter in upper case (resp., lower case); elements from $\mathcal{E}$ are

usually prefixed with '#'. $\mathcal{U}$ constitutes the default *Herbrand Universe*. We assume that any constant appearing in a program or generated by external computation is taken from $\mathcal{U}$, which is possibly infinite[1].

Elements from $\mathcal{U} \cup \mathcal{X}$ are called *terms*. An *atom* is a structure $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \mathcal{P} \cup \mathcal{E}$; $n \geq 0$ is the *arity* of the atom. p is *the predicate name*. The atom is *ordinary*, if $p \in \mathcal{P}$, otherwise we call it *external atom*. A list of terms $t_1, \ldots, t_n$ is succinctly represented by $\bar{t}$. A positive *literal* is an atom, whereas a *negative literal* is $\texttt{not}\, a$ where a is an atom.

Given a predicate p, $p[i]$ is its i-th argument. A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \texttt{not}\, \beta_{n+1}, \ldots, \texttt{not}\, \beta_m, \tag{6}$$

where $m \geq 0$, $k \geq 1$, $\alpha_1, \ldots, \alpha_k$, are ordinary atoms, and $\beta_1, \ldots, \beta_m$ are (ordinary or external) atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. $E(r)$ is the set of external atoms of r. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms. A *VI program* is a finite set P of rules; it is *ordinary*, if all rules are ordinary. We assume P has no constraints[2], only ground facts, and that each rule is *safe* with respect to negation, i.e. for each rule r, each variable appearing in some negated atom $a \in B^-(r)$ or in the head, appears also in some positive atom $b \in B^+(r)$. Given a set of atoms A and a predicate p, with small abuse of notation we say that $p \in A$ if there is some atom in A with predicate name p. An atom having p as predicate name is usually referred as a_p.

We denote as $Attr(P)$ the set of all arguments of all the predicates appearing in the program P. The *dependency graph* $G(P)$ of P is built in the standard way.

We give the semantics by generalizing the answer-set semantics [12].

In the sequel, we will assume P as a VI program. The *Herbrand base* of P with respect to $\mathcal{U}$, denoted $HB_{\mathcal{U}}(P)$, is the set of all possible ground versions of ordinary atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{U}$. The grounding of a rule r, $grnd_{\mathcal{U}}(r)$, is defined accordingly, and the grounding of program P by $grnd_{\mathcal{U}}(P) = \bigcup_{r \in P} grnd_{\mathcal{U}}(r)$. Note that this ground program can be of infinite size.

An *interpretation* I for P is a pair $\langle S, F \rangle$ where:
– $S \subseteq HB_{\mathcal{U}}(P)$ contains only ordinary atoms; I (or, by small abuse of notation, S) is a *model* of ordinary atom $a \in HB_{\mathcal{U}}(P)$, denoted $I \models a$ ($S \models a$), if $a \in S$.
– F is a mapping associating with every external predicate name $\#\mathrm{e} \in \mathcal{E}$, a decidable n-ary function (which we call *oracle*) $F(\#\mathrm{e})$ assigning each tuple $(x_1, \ldots, x_n)$ either 0 or 1, where n is the fixed arity of $\#\mathrm{e}$, and $x_i \in \mathcal{U}$. I (or, by small abuse of notation, F) is a *model* of a ground external atom $a = \#\mathrm{e}(x_1, \ldots, x_n)$, denoted $I \models a$ ($F \models a$), if $F(\#\mathrm{e})(x_1, \ldots, x_n) = 1$.

A positive literal is satisfied if its atom is satisfied, whereas a negated literal is satisfied if its corresponding atom is not satisfied.

[1] Also, we assume that constants are encoded using some finite alphabet Σ, i.e. they are finite elements of Σ^*.

[2] Under Answer Set Programming semantics, a constraint $\leftarrow B(r)$ can be easily simulated through the introduction of a corresponding standard rule fail $\leftarrow$ B(r), not fail, where fail is a fresh predicate not occurring elsewhere in the program.

Let r be a ground rule. We define:

i. $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$;
ii. $I \models B(r)$ iff $I \models a$ for each atom $a \in B^+(r)$ and $I \not\models a$ for each atom $a \in B^-(r)$;
iii. $I \models r$ (i.e., r is satisfied) iff $I \models H(r)$ whenever $I \models B(r)$.

We say that I is a *model* of a VI program P with respect to a universe $\mathcal{U}$, denoted $I \models_{\mathcal{U}} P$, iff $I \models r$ for all $r \in grnd_{\mathcal{U}}(P)$. For a fixed F, a model $M =< S, F >$ is minimal if there is no model $N =< T, F >$ such that $S \subset T$.

Given a general ground program P, its *Gelfond-Lifschitz reduct* [12] w.r.t. an interpretation I is the positive ground program P^I obtained from P by: *(i)* deleting all rules having a negated literal not satisfied by I; *(ii)* deleting all negated literals from the remaining rules. $I \subseteq HB_{\mathcal{U}}(P)$ is an *answer set* for a program P w.r.t. $\mathcal{U}$ iff I is a minimal model for the positive program $grnd_{\mathcal{U}}(P)^I$. Let $ans_{\mathcal{U}}(P)$ be the set of answer sets of $grnd_{\mathcal{U}}(P)$. We call P *F-satisfiable* if it has some answer set for a fixed function mapping F, i.e. if there is some interpretation $\langle S, F \rangle$ which is an answer set. We will assume in the following to deal with a fixed set F of functions mappings for external predicates. F-satisfiability is undecidable [8]. Given an external predicate name #p, of arity n and its oracle function $F(\#\mathrm{p})$, a *pattern* is a list of b's and u's, where a b represents a placeholder for a constant (or a bounded variable), and an u is a placeholder for a variable. Given a list of terms, the corresponding pattern is given by replacing each constant with a b, and each variable with a u. Positions where u appears are called *output* positions whereas those denoted with b are called *input* positions. For instance, the pattern related to the list of terms (X, a, Y) is (u, b, u).

Let *pat* be a pattern of length n having k placeholders b (*input positions*), and $n-k$ placeholders of u type (*output positions*). A *functional oracle* $F(\#\mathrm{p})[pat]$ for the pattern *pat*, associated with the external predicate #p, is a partial function taking k constant arguments from $\mathcal{U}$ and returning a finite relation of arity $n-k$, and such that $d_1, ..., d_{n-k} \in F(\#\mathrm{p})[pat](c_1, ..., c_k)$ iff $F(\#\mathrm{p})(h_1, \ldots, h_n) = 1$, where for each $i (1 \leq i \leq n)$, $h_i = c_j$ if the j-th b value occurs in position i in *pat*, otherwise $h_i = d_j$ if the j-th u value occurs in position i in *pat*.

An external predicate #p might be associated to one or more functional oracles 'consistent' with the originating 2-valued one. For instance, consider a binary external predicate #sqr, intuitively associating a natural number to its square value. We can have two functional oracles, $\mathrm{F}(\#\mathrm{sqr})[b, u]$ and $\mathrm{F}(\#\mathrm{sqr})[u, b]$. The two functional oracles are such that, e.g. $\mathrm{F}(\#\mathrm{sqr})[b, u](3) = 9$ and $\mathrm{F}(\#\mathrm{sqr})[u, b](16) = 4$, consistently with the fact that $\mathrm{F}(\#\mathrm{sqr})(3, 9) = \mathrm{F}(\#\mathrm{sqr})(4, 16) = 1$[3].

In the sequel, given an external predicate #e, we will assume that it comes equipped with its oracle $F(\#\mathrm{e})$ (called also *base oracle*) and one functional oracle $F(\#\mathrm{e})[pat_{\#\mathrm{e}}]$, having pattern $pat_{\#\mathrm{e}}$ [4].

We recall now a first condition of safety, which unfortunately does not guarantee finiteness and decidability, but will be exploited in the next Section. Given a rule r, a variable X is *weakly safe in* r if either *(i)* X is safe (i.e. it appears

[3] Unlike this example, note that in the general case functional oracles might return a set of tuples and are not restricted to single output values.
[4] In [8] we address explicitly the issue of external predicates equipped with multiple functional oracles.

in some positive atom of $B^+(r) \setminus E(r)$; or *(ii)* X appears in some external atom $\#\mathrm{e}(\overline{T}) \in E(r)$, the functional oracle of #e is $F(\#\mathrm{e})[pat]$, X appears in output position with respect to *pat* and each variable Y appearing in input position in the same atom is weakly safe. A weakly safe variable X is *free* if it appears in $B^+(r)$ only in output position of some external atom. A rule r is weakly safe if each variable X appearing in some atom $a \in B(r)$ is weakly safe. A program P is weakly safe if each rule $r \in P$ is weakly safe.

Example 1. Assume that #sqr is associated to the functional oracle $F(\#\mathrm{sqr})[b, u]$ defined above. The program { square(Y) ← number(X), #sqr(X,Y) } is weakly safe (intuitively the value of Y can be computed once the value of X is known). The same rule is not weakly safe if we consider the functional oracle $F(\#\mathrm{sqr})[u, b]$. □

Definition 1. Let $A = \langle I, F \rangle$ an interpretation. We call $ins(r, A)$ the set of ground instances r_θ of r for which $A \models B^+(r_\theta)$, and such that $A \models E(r_\theta)$. □

Proposition 1. [8] Given an interpretation A and a weakly safe rule r, $ins(r, A)$ is finite. □

Weakly safe rules have the important property of producing a finite set of *relevant* ground instances provided that we know a priori the domain of positive ordinary body atoms. Although desirable, weak safety is intuitively not sufficient in order to guarantee finiteness of answer sets and decidability. For instance, it is easy to see that the program { square(2) ←; square(Y) ← square(X), #sqr(X,Y); } has answer set $\{\mathrm{square}(2), \mathrm{square}(4), \dots\}$.

4 Decidable VI programs

The introduction of new symbols in a logic program by means of external atoms is a clear source of undecidability. As illustrated in Section 6, value invention is nonetheless desirable in a variety of contexts.

Our approach investigates which programs can be solved by means of a finite ground program having a finite set of models of finite size. This class of programs (having the *finite grounding property*) is unluckily not recognizable in finite time. We assume to deal with functional oracles that might have an infinite co-domain. Nonetheless, we will assume also to deal with weakly safe programs and with functional oracles associating to each fixed combination of the values in input always a finite number of combination of values in output.

Definition 2. A class of VI programs $\mathcal{C}$ has the *finite grounding* property if, for each $P \in \mathcal{C}$ there exists a finite set $U \subset \mathcal{U}$ such that $ans_U(P) = ans_{\mathcal{U}}(P)$. □

Theorem 1. Recognizing the class of all the VI programs having the finite grounding property is undecidable.

Proof. (Sketch). Positive logic programs with function symbols can simulate Turing machines. Also weakly safe VI programs can mimic (see section 6 and [8]) programs with function symbols. Given a Turing machine $\mathcal{T}$ and an input string x we can thus build a suitable VI program $P_{\mathcal{T},x}$ encoding $\mathcal{T}$ and x. $\mathcal{T}(x)$ terminates iff $P_{\mathcal{T},x}$ has the finite grounding property. Indeed, if $\mathcal{T}(x)$ terminates,

the content of U can be inferred from the finite number of transitions of $\mathcal{T}(x)$. Viceversa, if U is given, the evolution of $\mathcal{T}(x)$ until its termination can be mimicked by looking at the answer sets of $grnd_U(P_{\mathcal{T},x})$. □

4.1 VI-restricted programs

The intuition leading to our definition of VI-restrictedness, is based on the idea of controlled propagation of new values throughout a given program. Assume the following VI program is given ($\#b$ has a functional oracle with pattern $[b, u]$): { a(k,c) ←; p(X,Y) ← a(X,Y); p(X,Y) ← s(X,Y), a(Z,Y); s(X,Y) ← p(Z,X), #b(X,Y).}. The last rule of the program generates new symbols by means of the Y variable, which appears in the second attribute of $s(X, Y)$ and in output position of $\#b(X, Y)$. This situation is per se not a problem, but we observe that values of $s[2]$ are propagated to $p[2]$ by means of the last but one rule, and $p[2]$ feeds input values to $\#b(X, Y)$ in the last rule. This occurs by means of the binding given by the X variable. The number of ground instances to be considered for the above program is thus in principle infinite, due to the presence of this kind of cycles between attributes.

We introduce the notion of *dangerous* rule for those rules that propagate new values in recursive cycles, and of *dangerous* attributes for those attributes (e.g. $s[2]$) that carry new information in a cycle.

Actually, the above program can be reconducted to an equivalent finite ground program: we can observe that $p[2]$ takes values from the second and third rule above. In both cases, values are given by bindings to $a[2]$ which has, clearly, a finite domain. So, the number of input values to #b(X,Y) is bounded as well. In some sense, the 'poisoning' effect of the last (dangerous) rule, is canceled by the fact that $p[2]$ limits the number of symbols that can be created.

In order to formalize this type of scenarios we introduce the notion of *savior* and *blocked* attributes. $p[2]$ is *savior* since all the rules where p appears in the head can be proven to bring values to $p[2]$ from blocked attributes, or from constant values, or from other savior attributes. Also, $s[2]$ is dangerous but *blocked* with respect to the last rule, because of the indirect binding with $p[2]$, which is savior. Note that an attribute is considered blocked with respect to a given rule. Indeed, $s[2]$ might not be blocked in other rules where s appears in the head.

We define an *attribute dependency graph* useful to track how new symbols propagate from an attribute to another by means of bindings of equal variables.

Definition 3. The *attribute dependency graph* $AG(P)$ associated to a weakly safe program P is defined as follows. For each predicate $p \in P$ of arity n, there is a node for each predicate attribute $p[i](1 \leq i \leq n)$, and, looking at each rule $r \in P$, there are the following edges:
– $(q[j], p[i])$, if p appears in some atom $a_p \in H(r)$, q appears in some atom $a_q \in B^+(r) \setminus E(r)$ and $q[j]$ and $p[i]$ share the same variable.
– $(q[j], \#p[i])$, if q appears in some atom $a_q \in B^+(r) \setminus E(r)$, $\#p$ appears in some atom $a_{\#p} \in E(r)$, $q[j]$ and $\#p[i]$ share the same variable, and i is an input position for the functional oracle of $\#p$;
– $(\#q[j], \#p[i])$, if $\#q$ appears in some atom $a_{\#q} \in E(r)$, $\#p$ in some $a_{\#p} \in E(r)$, $\#q[j]$ and $\#p[i]$ share the same variable, j is an output position for the functional oracle of $\#q$, i is an input position for the functional oracle of $\#p$;

– $(\#p[j], \#p[i])$, if $\#p$ appears in some atom $a_{\#p} \in E(r)$, $\#p[j]$ and $\#p[i]$ both have a variable, j is an input position for the functional oracle of $\#p$ and i is an output position for the functional oracle of $\#p$;
– $(\#q[j], p[i])$, if p appears in some atom $a_p \in H(r)$, $\#q$ appears in some atom $a_{\#q} \in E(r)$ and $\#q[j]$ and $p[i]$ share the same variable, and j is an output position for the functional oracle of $\#q$; □

Example 2. The *attribute dependency graph* induced by the *first three* rules of the motivating example in Section 2 is depicted in Figure 1. □

Definition 4. It is given a weakly safe program P. The following definitions are given (all examples refer to the Motivating Example, Section 2, and we assume #rdf has functional oracle with pattern $[b, u, u, u]$):
– A rule r *poisons* an attribute $p[i]$ if some atom $a_p \in H(r)$ has a free variable X in position i. $p[i]$ is said to be *poisoned* by r. For instance, *connected*[2] is poisoned by rule (5).
– A rule r is *dangerous* if it poisons an attribute $p[i]$ ($p \in H(r)$) appearing in a cycle in $AG(P)$. Also, we say that $p[i]$ is *dangerous*. For instance, rule (5) is dangerous since connected[2] is poisoned and appears in a cycle.
– Given a dangerous rule r, a dangerous attribute $p[i]$ (bounded in $H(r)$ to a variable name X), is *blocked* in r if for each atom $a_{\#e} \in E(r)$ where X appears in output position, each variable Y appearing in input position in the same atom is *savior*. Y is *savior* if it appears in some predicate $q \in B^+(r)$ in position i, and $q[i]$ is *savior*.
– An attribute $p[i]$ is *savior* if at least one of the following conditions holds for each rule $r \in P$ where $p \in H(r)$.

- $p[i]$ is bound to a ground value in $H(r)$;
- there is some savior attribute $q[j]$, $q \in B^+(r)$ and $p[i]$ and $q[j]$ are bound to the same variable in r;
- $p[i]$ is blocked in r.

For instance, the dangerous attribute *connected*[2] of rule (5) is blocked since the input variable U is savior (indeed it appears in $url[2]$).
– A rule is `VI`-restricted if all its dangerous attributes are blocked. P is said to be *`VI`-restricted* if all its *dangerous* rules are *`VI`-restricted*. □

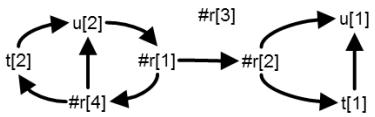

Fig. 1. Attributes Dependency Graph *(Predicate names are shortened to the first letter)*

Theorem 2. `VI`-restricted programs have the finite grounding property.

Proof. (Sketch). Given a `VI`-restricted program P, we show how to compute a finite ground program gr_P such that $ans_{\mathcal{U}}(P) = ans_U(gr_P)$, where U is the set of constants appearing in gr_P.

Let's call A the set of *active ground atoms*, initially containing all atoms appearing in some fact of P. gr_P can be constructed by an algorithm $\mathcal{A}$ that repeatedly updates gr_P (initially empty) with the output of $ins(r, I)$ (Definition 1) for each rule $r \in P$, where $I = \langle A, F \rangle$; all atoms belonging to the head of some rule appearing in gr_P are then added to A. The iterative process stops when A is not updated anymore. That is, gr_P is the least fixed point of the operator

$$T_P(Q) = \{\bigcup_{r \in P} ins(r, I) \mid I = \langle A, F \rangle, \text{ and } A = atoms(Q)\}$$

where $atoms(Q)$ is the set of ordinary atoms appearing in Q. $T_P^\infty(\emptyset)$ is finite in case P is VI-restricted. Indeed, gr_P might not cease to grow only in case an infinite number of new constants is generated by the presence of external atoms. This may happen only because of some *dangerous* rule having some 'poisoned' attributes. However, in a *VI-restricted* program all poisoned attributes are *blocked* in dangerous rules where they appear, i.e. they depend from savior attributes. It can be shown that, for a given savior attribute $p[i]$, the number of symbols that appear in position i in an atom a_p such that $a_p \in T_P^\infty(\emptyset)$ is finite. This means that only a finite number of calls to functional oracles is made by $\mathcal{A}$, each of which producing a finite output.

Because of the way it has been constructed, it is easy to see that the set $A = atoms(gr_P)$ is a splitting set [13], for $grnd_{\mathcal{U}}(P)$. Based on this, it is possible to observe that no atom $a \notin A$ can be in any answer set, and to conclude that $ans_U(P) = ans_{\mathcal{U}}(P)$, where U is the set of constants appearing in A. □

5 Recognizing VI-restricted Programs

An algorithm recognizing VI-restricted programs is depicted in Figure 2. The idea is to iterate through all *dangerous rules* trying to prove that all of them are *VI-restricted*. In order to prove VI-restriction for rules, we incrementally build the set of all *savior attributes*; this set is initially filled with all attributes which can be proven to be savior (i.e. they do not depend from any dangerous attribute). This set is updated with a further attribute $p[i]$ as soon it is proved that each dangerous attribute which $p[i]$ depends on is blocked. The set RTBC of rules to be checked initially consists of all dangerous rules, then the rules which are proven to be VI-restricted are gradually removed from RTBC. If an iteration ends and nothing new can be proved the algorithm stops. The program is VI-restricted if RTBC is empty at the last iteration.

The algorithm execution takes polynomial time in the size of a program P: let m be the total number of rules in P, n the number of different predicates, k the maximum number of attributes over all predicates, and l the maximum number of atoms in a single rule. $O(n * k)$ is an upper bound to the total number of different attributes, while $O(l * k)$ is an upper bound to the number of variables in a rule. A naive implementation of the *isBlocked* function has complexity $O(n * l * k^2)$. The *recognizer* function (Figure 2) iterates $O(n * k)$ times over an inner cycle which performs at most $O(m * k * l)$ steps: each inner step iterates over all rules in $RTBC$, which are at most m; and for each rule all free variables must be checked (this requires $O(k * l)$ checks, in the worst case).

```
Bool Function recognizer ( var SA: Set{ Attr };
                               % SA is initialized with provable savior attributes
                               % (i.e. attributes that do not depend from dangerous attributes.
                           var NSA: Set{ pair⟨ Attr, Set{ Attr } ⟩ };
                               % NSA is initialized with attributes which cannot be proven to be
                               % savior, each of which is associated with the set of dangerous
                               % attributes that prevent them to be savior
                           var RTBC : Set{ Rule } ) % Set of dangerous rules to be checked.
    Bool NSA_Updated = true;
    While ( NSA_Updated ) do % Try to prove VI-restriction as far as some change occurs.
        NSA_Updated = false;
        For each Rule r ∈ RTBC do % free(r)=the set of free variables appearing in the rule r.
                Set{Var} varsTBC = free( r );
                Bool allBlocked = true;
                For each Var v ∈ varsTBC do
                    % isBlocked tells if v is blocked in r by means of attributes currently in SA.
                    If ( isBlocked( v, r, SA ) ) then
                        % headAttr returns reference to the head attribute of r containing v
                        Attr p[i] = headAttr( v, r );
                        % update processes the NSA set, deleting p[i] from each set S
                        % such that p[i] ∈ S and ⟨q[j], S⟩ ∈ NSA.
                        % Then each attribute q[j] such that ⟨q[j], S⟩ ∈ NSA
                        % and S = ∅ is moved from NSA to SA.
                        update( NSA, SA, p[i] );
                        % A change occurred, so we have to continue cycling.
                        NSA_Updated = true;
                    Else % At least one free variable can't be proved as blocked.
                        allBlocked = false;
                    EndIf
                EndFor
                If ( allBlocked ) then
                    RTBC.delete(r); %.The rule is VI-restricted, can be deleted from RTBC.
                EndIf
        EndFor
    EndWhile
    If ( RTBC == ∅ ) then
        Return true
    Else % Display the set of rules that can't be proved as VI-restricted.
        printINSAne( RTBC )
        Return false
    EndIf
EndFunction
```

Fig. 2. *The VI-Restricted Recognizer Algorithm*

6 Modeling semantic extensions by VI programs

Several semantic extensions contemplating value invention can be mapped to VI programs. We show next how programs with function symbols and with sets can be translated to weakly safe VI programs. When the resulting translation is VI-restricted as well, these semantic extension can be thus evaluated by an answer set solver extended with external predicates.

Functional terms. We consider rule based languages allowing functional terms whose variables appearing in the head appear also in the positive body. A functional term is either a constant, a variable, or $f(X_1, \ldots, X_n)$, where f is a function symbol and $X_1, \ldots, X_n$ are terms.

For each natural number k, we introduce two external predicates $\#function_k$ and $\#function'_k$ of arity $k+2$; they are such that $f_{\#function_k}(F, f, X_1, \ldots, X_k) = f_{\#function'_k}(F, f, X_1, \ldots, X_k) = true$ if and only if the term F is $f(X_1, \ldots, X_k)$. Each $\#function_k$ ($\#function'_k$) predicate is associated to a functional oracle $F(\#function_k)[u, b, b, \ldots, b]$ ($F(\#function'_k)[b, u, u, \ldots, u]$, respectively).

The two families of external predicates are respectively intended in order to construct a functional term if all of its arguments are bounded ($\#function_k$

predicates) or if the whole functional term is grounded and we want to take its arguments ($\#function'_k$ predicates).

Basically, this transformation flattens each rule $r \in P$ containing some functional term $t = f(X_1, \ldots, X_n)$, by replacing it with a fresh variable F, and adding an appropriate atom $\#function_k(F, X_1, \ldots, X_n)$ or $\#function'_k(F, X_1, \ldots, X_n)$ to the body of r. The transformation is continued until a functional term is still in r. We choose $\#function'_k$ if t appears in the body of r, whereas an atom using $\#function_k$ is used if t appears in the head of r.

Example 3. The rule { p(s(X)) ← a(X, f(Y, Z)). } contains two function symbols, *s* and *f*. The rewritten rule is { p(F1) ← a(X, F2), $\#function_1(F1, s, X)$, $\#function'_2(F2, f, Y, Z)$. } □

Proposition 2. Given a logic program with functional terms P, $\mathcal{F}(P)$ is the program obtained by applying the above transformation; it is weakly safe. Also, there is a 1-to-1 mapping between the answer sets of P and $ans_{\mathcal{U}}(\mathcal{F}(P))$. □

Set unification. The accommodation of sets in logic programming, often attempted, obliges to reconsider the classic notion of term unification to a generalized one. For instance, the set term $\{X, a, b, c\}$ can be grounded to $\{a, d, b, c\}$. It is possible to embody set constructors and set unification in the context of `VI` programs. Roughly speaking, a logic program with sets replaces the classical notion of term with the notion of *set term*. A set term is either a *(i)* classical term or, *(ii)* $\{X_1, \ldots, X_n\}$ where $X_1, \ldots, X_n$ are set terms, or *(iii)* $X \cup Y$ where X and Y are set terms. Two ground set terms $\{a_1, \ldots, a_n\}$ are equal if they contain the same set of ground terms. For space reasons, we only outline here a method, and refer the reader to [14] for a survey on sets in logic programming and on set unification methods and algorithms.

Remarking that the special symbol $\{\}$ represents the empty set, the following set of external predicates are introduced: *(i)* A pair of external predicates $\#set_k$, $\#set'_k$ for each natural number k; each of them has $k+1$ arguments such that $f_{\#set_k}(X, Y_1, \ldots, Y_k) = f_{\#set'_k}(X, Y_1, \ldots, Y_k) =$ true if X is the set $\{Y_1, \ldots, Y_k\}$. $\#set_k$ has the functional oracle $F(\#set_k)[u, b, \ldots, b]$ while $\#set'_k$ has the functional oracle $F(\#set_k)[b, u, \ldots, u]$; *(ii)* Two ternary external predicate $\#union$ and $\#union'$; they are such that $f_{\#union}(X, Y, Z) = f_{\#union'}(X, Y, Z) = true$ either if $X = Y \cup Z$, or if X and Y are classical terms, $Z = \emptyset$ and $X = Y$. $\#union$ has the functional oracle $F(\#union)[u, b, b]$ while $\#union'$ has the functional oracle $F(\#union')[b, u, u]$.

A logic program with set terms P is replaced by an equivalent `VI` program by modifying each rule $r \in P$ this way:

– Replacing each set term $\{X_1, \ldots, X_n\}$ appearing in r with a fresh variable T, and adding in the body of r the external atom $\#set_n(T, X_1, \ldots, X_n)$ if the set term appears in the head of r , $\#set'_n(T, X_1, \ldots, X_n)$ otherwise;

– Replacing each set term $X \cup Y$ appearing in r with a fresh variable U, and adding in the body of r the external atom $\#union(U, X, Y)$ if the set term appears in the head of r, $\#union'(U, X, Y)$ otherwise. This and step 6 are applied to r until it contains any set term;

– If a variable X appears in r for m times ($m > 1$), then each occurrence of X is replaced with a fresh variable $X_i (1 \leq i \leq n)$, and for each pair $(X_i, X_j), 1 \leq i < j < m$ the atom $\#union(X_i, X_j, \{\})$ is added to r.

Example 4. Let's consider the rule: { p(X ∪ Y) ← a({a, X}), b({Y}). }; the analogous `VI` rule is: { p(S1) ← a(S2), b(S3), #union(S1, X1, Y1), $\#set'_2(S2, a, X2)$, $\#set'_1(S3, Y2)$, #union(X1,X2,{}), #union(Y1,Y2,{}). } □

Proposition 3. Given a logic program with set terms P, we call $\mathcal{S}(P)$ the `VI` program obtained applying the above transformation. $\mathcal{S}(P)$ is weakly safe. There is a 1-to-1 mapping between the answer sets of P and $ans_{\mathcal{U}}(\mathcal{S}(P))$. □

7 Relationships with other classes of programs

ω-restricted programs. In the same spirit of this paper are ω-stratified programs [9], that allow function symbols under answer set semantics. The introduced restrictions aim at controlling the creation of new functional terms.

Definition 5. [9] An ω-stratification is a traditional stratification extended by the ω-stratum, which contains all predicates depending negatively on each other. ω is conventionally assumed to be uppermost layer of the program. A rule r is ω-restricted iff all variables appearing in r also occur in a positive body literal belonging to a *strictly* lower stratum than the head. A program P is ω-restricted iff all the rules are ω-restricted. □

ω-stratified programs have the finite grounding property: only a finite amount of functional terms can be created since each variable appearing in a rule's head must be bounded to a predicate belonging to a lower layer. `VI`-restricted programs do not introduce special restrictions for non-stratified cycles. Also, it is not necessary to bound each variable to a previous layer explicitly. The class of `VI`-restricted programs contains, in a sense, the class of ω-restricted ones.

Theorem 3. Given an ω-restricted program P, $\mathcal{F}(P)$ is `VI`-restricted.

Proof. We are given an ω-restricted program P. We observe that:
–Attributes belonging to predicates which are not in the ω-stratum can be proven to be savior: the relevant instantiation of these predicates is computable starting from the lowermost layer, and is finite.
–The rewritten rules in $\mathcal{F}(P)$ corresponding to function-free rules cannot be dangerous, since there is no value invention at all.
–Rules with functional terms are rewritten using external atoms; then, all variables occurring in these new external atoms already occur in the original rules, except fresh variables used for substituting functional terms (that we call FTRs, *functional term representations*). Thus, the variables appearing in the poisoned attributes must necessarily appear also in a predicate belonging to a strictly lower stratum than the head (ω-restrictedness). Let's consider an FTR appearing in an external atom $\#function'_k(F1, X_1, \ldots, X_k)$ in first position. If $F1$ is already bound to a positive atom, then there is no value invention; otherwise, it can be shown that all terms $X_1, \ldots, X_k$ are bound either to a positive atom or to another external atom in output position (see Section 6). As stated before, the attributes where $X_1, \ldots, X_k$ appear are savior, and so the FTR $F1$ as well. □
On the other hand, the opposite does not hold.

Theorem 4. It is possible to find non-ω-restricted programs whose transformation $\mathcal{F}$ outputs a `VI`-restricted program.

Proof. The program $P_{n\omega r} = \{p(f(X)) \leftarrow q(X), t(X); q(X) \leftarrow p(X); p(1); t(1)\}$ is not ω-restricted, while $\mathcal{F}(P_{n\omega r}) = \{p(F1) \leftarrow q(X), t(X), \#function_2(F1, f, X); q(X) \leftarrow p(X); p(1); t(1)\}$ is VI-restricted. □

Finitary programs. Finitary programs allow function symbols under answer set semantics [10]. Although they don't have the finite grounding property, brave and cautious ground querying is decidable. A ground program P is finitary iff its dependency graph $G(P)$ is such that *(i)* any atom p appearing as node in $G(P)$ depends only on a finite set of atoms (through head-body dependency), and *(ii)* $G(P)$ has only a finite number of cycles with an odd number of negated arcs.

Theorem 5. The class of finitary programs is not comparable with the class of VI-restricted programs.

Proof. (sketch) A program having rules with free variables is not finitary (eg. $p(X) \leftarrow q(X, Y)$): a ground instance $p(a)$ may depend on infinite ground instances of $q(X, Y)$ $e.g.(q(a, f(a)), q(a, f(f(a)))...)$. In general, the same kind of rules are allowed in VI-restricted programs. Vice versa, the class of programs $\{\mathcal{F}(P) \mid P \text{ is finitary}\}$ is not VI-restricted: for instance the translation of the finitary program $\{p(0); p(s(X)) \leftarrow p(X)\}$ is not VI-restricted. □

Other literature. In the above cited literature, infinite domains are obtained through the introduction of compound functional terms. Thus, the studied theoretical insights are often specialized to this notion of term, and take advantage e.g., of the common unification rules of formal logics over infinite domains. It is, in this setting, possible to study ascending and descending chains of functional terms in order to prove decidability. Similar to our approach is the work on open logic programs, and conceptual logic programs [15]. Such paper addresses the possibility of grounding a logic program, under Answer Set Semantics, over an infinite domain, in a way similar to classical logics and/or description logics. Each constant symbol has no predefined compound structure however. Also similar are [3] and [16], where a special construct, aimed at creating new tuple identifiers in relational databases is introduced.

In [17] and [4] the authors address the issue of implementing generalized quantifiers under Answer Set Semantics, in order to enable Answer Set Solvers to communicate, both directions, with external reasoners. This approach is different from the one considered in this paper since the former is inspired from second order logics and allows bidirectional flow of relational data (to and from external atoms), whereas, in our setting, the information flow is strictly value (first order) based, and allows relational output only in one direction. HEX programs, as defined in [4], do not address explicitly the issue of value invention (although semantics is given in terms of an infinite set of symbols). VI programs can simulate external predicates of [4] when relational input is not allowed.

An external predicate *à* la [4] (HEX predicate) is of the form $\#g[Y_1, \ldots, Y_m](X_1, \ldots, X_n)$, where $Y_1, \ldots, Y_n$ are input terms and $X_1, \ldots, X_n$ are output terms. Semantics of these atoms is given by means of a base oracle $f_{\#g}(I, Y_1, \ldots, Y_m, X_1, \ldots, X_m)$ where I is an interpretation. Note that HEX predicates depend on a current interpretation, thus enabling to quantify over predicate extensions.

Assuming that for each HEX predicate $f_{\#g}$ do not depend on the current interpretation, and that *higher order atoms* (another special construct featured by HEX programs) are not allowed we can state the following equivalence theorem.

Theorem 6. An HEX program without higher order atoms is equivalent to a `VI` program where each HEX atom $\#g[Y_1, \ldots, Y_m](X_1, \ldots, X_n)$ is replaced by an atom $\#g'(Y_1, \ldots, Y_m, X_1, \ldots, X_n)$, provided that each evaluation function $f_{\#g'}$ is such that for each I we have that $f_{\#g'}(Y_1, \ldots, Y_m, X_1, \ldots, X_m) = f_{\#g}(I, Y_1, \ldots, Y_m, X_1, \ldots, X_m)$. □

`VI`-restricted programs overcome the notion of semi-safe programs [8]. These programs have the finite grounding property: a weakly safe program P is *semi-safe* if each cycle in $G(P)$ contains only edges whose label corresponds to a safe rule. Semi-safe programs are strictly contained in the class of `VI`-restricted programs.

8 Conclusions

`VI` programs herein presented accommodate several cases where value invention is involved in logic programming. `VI`-restrictions allow to actually evaluate by means of a finite ground program a variety of programs (such as those with function symbols or set constructors) in many nontrivial cases.

A topic for future work is to investigate to what extent the notion of `VI`-restrictedness can be relaxed although keeping the complexity of recognizing the class in polynomial time. Intuitively, local analysis techniques can enlarge the class of programs whose finite grounding property is decidable, but this would force to renounce to polynomial complexity. Nonetheless, the spirit of restriction checkers is to keep evaluation times greatly smaller than the overall solving times.

`VI` programs have been implemented in the DLV system as well as a `VI`-restriction checker. Further details on the implementation can be found in [8]. A complete toolkit for developing custom external predicates is provided. Specific extensions of the DLV system with function symbols and sets, using `VI` as underlying framework, are in advanced stage of development and will be dealt with in appropriate papers. The system prototype, examples, manuals and benchmark results are available at `http://www.mat.unical.it/ianni/wiki/dlvex`.

References

1. Abiteboul, S., Vianu, V.: Datalog Extensions for Database Queries and Updates. JCSS **43**(1) (1991) 62–124
2. Cabibbo, L.: Expressiveness of Semipositive Logic Programs with Value Invention. Logic in Databases. (1996) 457–474.
3. Hull, R., Yoshikawa, M.: ILOG: Declarative Creation and Manipulation of Object Identifiers. VLDB 1990. 455–468.
4. Eiter, T., et al.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. IJCAI 2005, 90–96.
5. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. ESWC 2005. 392–407.

6. Leone, N., et al.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2006) To appear. http://www.arxiv.org/ps/cs.AI/0211004.
7. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234.
8. Calimeri, F., Ianni, G.: External sources of computation for Answer Set Solvers. LPNMR 2005, LNCS 3662. 105–118.
9. Syrjänen, T.: Omega-restricted logic programs. LPNMR 2001. 267-279.
10. Bonatti, P.A.: Reasoning with Infinite Stable Models. IJCAI 2001. 603–610.
11. The Friend of a Friend (FOAF) Project. http://www.foaf-project.org/.
12. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385.
13. Lifschitz, V., Turner, H.: Splitting a Logic Program. ICLP 1994. 23–37.
14. Dovier, A., Pontelli, E., Rossi, G.: Set unification. TPLP (2006) To appear.
15. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Semantic web reasoning with conceptual logic programs. RuleML 2004. 113–127.
16. Cabibbo, L.: The Expressive Power of Stratified Logic Programs with Value Invention. Inf. and Comp. **147**(1) (1998) 22–56.
17. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Nonmonotonic description logic programs: Implementation and experiments. LPAR 2004. 511–527.