Noname manuscript No.
(will be inserted by the editor)

Francesco Calimeri · Susanna Cozza · Giovambattista Ianni

External Sources of Knowledge and Value Invention in Logic Programming

Received: date / Accepted: date

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, or set unification. This work 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 a very large polynomially decidable class having this property, when 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, and the implementation of a system supporting the class of such programs is described.

Keywords Artificial Intelligence · Logic Programming · Answer Set Semantics · Value Invention

1 Introduction

The notion of 'value invention' has been formerly adopted in the database field (see e.g. [1; 2]) for denoting mechanisms aimed at allowing the introduction of new domain elements in a logic based query language. Indeed,

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni
Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy.
Tel.: +39 0984 49 {6430, 6480}
Fax: +39 0984 49 6410
E-mail: {calimeri,cozza,ianni}@mat.unical.it

applications of logic programming often need to deal with a universe of symbols which is not known a priori. We can divide these demands in two main categories:

(*i*) 'Constructivist' demands: the majority of logic programming languages has the inherent capability to build new symbols from pre-existing ones, e.g. by means of traditional constructs like functional terms. Furthermore, manipulating and creating complex data structures other than simple constant symbols, such as sets, or lists, is 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 known a priori [4], or is explicitly assumed to be open [5].

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

Top down solvers (such as SLD solvers) do not usually address this issue: the programmer is requested to carry the burden of ensuring termination. In order to achieve this, she has to have a good knowledge of the evaluation strategy implemented in her specific adopted system, since termination is often algorithm-dependent. On the other hand, *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. Unfortunately, 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 obviously cannot be built in practice.

Throughout this work we adopt the notion of `VI` programs, which are logic programs enriched with the notion of *external predicates*. External predicates model the mechanism of value invention by taking input from a given set of values and returning (possibly newly invented) values. These are computed by means of an associated evaluation function (called *oracle*). We prove that, although assuming as decidable the external functions defining oracles, the consistency check of `VI` programs is, in general, undecidable. Therefore, it is important to investigate on (nontrivial) sub-classes of decidable programs. We address this problem identifying a *safety* condition for granting decidability of `VI` programs.

The contributions of the work are overviewed next:

– We introduce a formal framework for accommodating external source of computation in the context of Answer Set Programming. In this framework, logic programs (called `VI` programs), include the explicit possibility of invention of new values from external sources (Section 3).

- We investigate the consequences of allowing value invention, in terms of undecidability of consistency check for VI programs (Section 4).
- We show how to cope with value invention, providing a characterization of those programs that can be computed even if external sources of computation are exploited (Section 5).
- 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 6).
- We show that VI-restrictedness can be checked in polynomial time in the size of the non-ground program (Section 7).
- We show that VI programs embed settings such as programs with function symbols, programs with sets (or, in general, logic languages with a generalized notion of unification), or with external constructs (Section 8). Indeed, the condition of VI-restrictedness is generic: no assumption is made on the structure of new invented symbols.
- We prove that the VI-restrictedness condition is less restraining than previously introduced syntactic restrictions (such as ω-restricted programs [8]). Programmers are thus relieved from the burden of introducing explicit syntactic modifications. We prove also that finitary programs [9], a class of programs with answer set semantics and function symbols, are not directly comparable with VI-restricted programs (Section 9).
- We present a full prototype that integrates the support for VI programs in the DLV system [10]. A static check for VI restrictedness is executed on each DLV program using external sources of computation. Several libraries of external predicates are available. We carried out some experiments, confirming that the accommodation of external predicates does not cause any relevant computational overhead (Section 10).

Although we take *Answer Set* as the reference semantics, 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 [11], or else.

2 A Motivating Example

The Friend of a Friend (FOAF) [12] 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 her FOAF ontology portion at some url U.

In order to deal with such a vocabulary, a rule based logic language needs some special construct for importing external knowledge. But this makes the set of constant values unknown a priori, as it is always enriched at reasoning time. It is not possible to establish where and when new information is brought into play, since the aim of the proposed formalism is to keep declarativity (i.e. the order of evaluation for external sources of knowledge can not be established).

Let's imagine that we want to specify the transitive closure of the relation of knowledge among people, starting from the homepage of a given person. Each homepage contains a person description listing other linked people. Let's suppose also to have an external predicate, called "#rdf", which allows us to access a FOAF ontology located at URL:

$$\#\text{rdf}(\ \text{URL}, \text{Object}_1, \text{Relation}, \text{Object}_2). \tag{1}$$

Note that, for a given URL, the predicate *#rdf* induces a ternary relation whose extension is not known a priori. It might 'create', in a sense explained later on in the work, new information (symbols).

We say that external predicates allow 'value invention'; the intuition of `VI`-restricted programs is to investigate how new information propagates in a program, and whether its size is finite.

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 of the graph representing people, exploiting the knowledge provided by the collected pages. Let the starting homepage be "myurl"; thus, the following program implements what is described above.

trusted(X,U) ← #rdf("myurl", X, "trusts", U). (2)
url(X,U) ← #rdf("myurl", X, "seealso", U), trusted(X,U). (3)
url(X,U) ← url(_, U1), #rdf(U1, X, "seealso", U), trusted(X,U). (4)
connected(X, Y) ← url(X, U), #rdf(U, X, "knows", Y). (5)
connected(X, Y) ← connected(X, Z), url(Z, U), #rdf(U, Z, "knows", Y). (6)

The above program has two sources of new values: trusted URLs, and persons. For instance, in particular, rule (6) may induce a loop, leading to the invention of an infinite number of new symbols. As will be seen later, the above program is anyway `VI`-restricted and can be solved using a finite ground version of it: intuitively, the number of URLs to be taken into account is finite. Although not explicitly bounded, new persons (coming from the value of Y in the sixth rule) can be extracted only from a finite set of URLs. Observe that rule (2) *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 sixth rule.

It is worth noting that the programmer is not forced (in order to ensure decidability) to bound the domain of variables explicitly such as in this modified version of rule (6): { connected(X, Y) ← known(Y), connected(X, Z), url(Z, U), #rdf(U,Z,"knows",Y). }, where the predicate known is supposed to restrict the range of values for the Y variable.

3 Preliminaries

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

case). We assume that constants are encoded using some finite alphabet Σ, i.e. they are finite elements of Σ^*. External predicate names, that is elements from $\mathcal{E}$ are prefixed with ' # '.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called (simple) *terms*. A term may also be functional, and in this case is of the form $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are either simple terms or functional terms, and f is a function symbol from $\mathcal{F}$. A list of terms $t_1, \ldots, t_n$ is succinctly represented by $\bar{t}$.

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*. Given an *ordinary* atom a, $\neg a$ is said to be its *complementary* atom, where "$\neg$" is intended as the *classical* negation. Let A be a set of ordinary atoms. A is said to be *consistent* if $\forall a \in A$ we have that $\neg a \notin A$.

For instance, node(X), and #succ(a,Y) are atoms; the first is ordinary, whereas the second is an external atom.

A *literal* l is of the form a or `not` a, where a is an atom; in the former case l is *positive*, and in the latter case *negative*.

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

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \texttt{not}\, \beta_{n+1}, \ldots, \texttt{not}\, \beta_m. \tag{7}$$

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 $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ (the *head* of r) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of r), where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ (the *positive body* of r) and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$ (the *negative body* of r). $E(r)$ 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. Let A be a set of atoms and p be a predicate. 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 by inserting a node n_p for each predicate name p appearing in P and a directed edge (p_1, p_2), labelled r, for each rule r such that $p_2 \in B(r)$ and $p_1 \in H(r)$.

The following is a short VI program:

$$\begin{aligned} \text{mustChangePasswd(Usr)} \leftarrow\ & \text{passwd(Usr,Pass)}, \\ & \#\text{strlen(Pass,Len)}, \#<\text{(Len,8)}. \end{aligned} \tag{8}$$

We define the semantics of VI programs by generalizing the answer-set semantics, proposed by [13] as an extension of the stable model semantics of normal logic programs [14].

We denote with $\mathcal{U}$ the default *Herbrand Universe*. If $\mathcal{F}$ is empty then $\mathcal{C} = \mathcal{U}$. In case functional symbols are allowed, $\mathcal{U}$ consists of the set $\mathcal{C}$ plus all possible functional term that can be built with constants and function symbols from $\mathcal{C}$ and $\mathcal{U}$.

From now on, we will assume to deal with VI programs without functional terms, unless otherwise specified. We assume also that VI programs have no constraints,[1] 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 $b \in B^+(r)$[2].

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 elements 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 even when $\mathcal{C} = \mathcal{U}$.

An *interpretation* I for P is a pair $\langle S, F \rangle$ where:

- $S \subseteq HB_{\mathcal{U}}(P)$ is a *consistent* set of ordinary atoms; we say that 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 #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.

Example 1 We give an interpretation $I = \langle S, F \rangle$ such that the external predicate #strlen is associated to the oracle $F(\#\mathrm{strlen})$, and $F(\#<)$ to $\#<$. Intuitively these oracles are defined such that #strlen(pat4dat,7) and #<(7,8) are satisfied by I, whereas #strlen(mypet,8) and #<(10,8) are not.

The following is a ground version of rule 8:

$$\begin{aligned} \text{mustChangePasswd(frank)} \leftarrow\ & \text{passwd(frank,pat4dat)}, \\ & \#\text{strlen(pat4dat,7)}, \#<\text{(7,8)}. \end{aligned} \tag{9}$$

□

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 $S \subset T$.

[1] Under Answer Set 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.

[2] See section 9 for a discussion about the different notions of safety adopted in this paper.

Let P be a ground program. The *Gelfond-Lifschitz reduct* [13] of P, w.r.t. an interpretation I, is the positive ground program P^I obtained from P by:

- deleting all rules having a negated literal not satisfied by I;
- 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.

Example 2 Consider the following program P:

$$\begin{array}{l} \mathrm{r} \leftarrow \mathrm{p}. \\ \mathrm{r} \leftarrow \mathrm{q}. \\ \mathrm{p} \leftarrow \#\mathrm{e(a,b)}, \mathit{not}\ \mathrm{q}. \\ \mathrm{q} \leftarrow \mathit{not}\ \mathrm{p}. \end{array} \tag{10}$$

and the interpretation $I = \langle \{p, r\}, F\rangle$, where the oracle $F(\#\mathrm{e})$, associated to the external predicate $\#\mathrm{e}$, is defined such that $\#\mathrm{e(a,b)}$ is satisfied by I. Thus, the *Gelfond-Lifschitz reduct* w.r.t. I is:

$$\begin{array}{l} \mathrm{r} \leftarrow \mathrm{p}. \\ \mathrm{r} \leftarrow \mathrm{q}. \\ \mathrm{p} \leftarrow \#\mathrm{e(a,b)}. \end{array} \tag{11}$$

I is a minimal model for the reduct; thus, it's also an answer set for P. □

4 Properties of `VI` Programs

Although simple in its definition, the above semantics does not give any hint on how to actually compute the answer sets of a given program P. In general, given an infinite domain of constants $\mathcal{U}$, and a program P, $HB_{\mathcal{U}}(P)$ is infinite.

Theorem 1 Let P be a `VI` program, $\mathcal{U}$ be a domain of constants, F be a function mapping such that its co-domain contains only 2-valued functions decidable in polynomial time in the size of their arguments. Deciding whether P is F-satisfiable in the domain $\mathcal{U}$ is undecidable.

Proof The proof is easily entailed by Proposition 3 (appearing in Section 8, and stating that the Answer Set Semantics of an ordinary program $\overline{P}$ with function symbols can be reduced to the Answer Set Semantics of a `VI` program P), and by what we know about positive Horn programs with function symbols (indeed, they are undecidable: see e.g. [15]). □

4.1 Splitting sets

It is of interest to tailor cases where a finite portion of $\mathcal{U}$ is enough to evaluate the semantics of a given program. In the following we reformulate some

results regarding splitting sets. The notion of splitting set has been introduced in [16] in order to provide a technique for decomposing a given ground program P, so that its answer sets can be computed from the answer sets of two separate programs. This technique is commonly adopted for enabling modular computation of the answer sets. In this paper splitting sets are exploited as a tool for decomposing P into a finite part P' and an infinite part P''. Then we identify classes of programs for which P'' is provable to be always consistent, since it has a single, empty, answer set. Thus, computing the answer sets of P can be reduced to computing the answer sets of P'.

Definition 1 Let P be a VI program. A *splitting set* is a set of atoms $A \in HB_{\mathcal{U}}(P)$ such that for each atom $a \in A$, if $a \in H(r)$ for some $r \in grnd_{\mathcal{U}}(P)$, then $B(r) \cup H(r) \subseteq A$. The *bottom* $b_A(P)$ is the set of rules $\{r \mid r \in grnd_{\mathcal{U}}(P) \text{ and } H(r) \subseteq A\}$. A literal whose atom belongs to A is said A-literal. Given an interpretation I, the *residual* $r_A(P, I)$ is a program obtained from $grnd_{\mathcal{U}}(P)$ by deleting all the rules whose body contains an A-literal not satisfied by I, and removing from the remaining rules all the A-literals. □

Example 3 Consider the program 10 of example 2. $A = \{p,\ q,\ \#e(a,b)\}$ is a *splitting set* for P. The *bottom* $b_A(P)$ is the set containing the last two rules of P. Consider the interpretation $I = \langle\{p, r\}, F\rangle$, where the oracle $F(\#\text{e})$, associated to the external predicate #e, is defined such that #e(a,b) is satisfied by I. The *residual* $r_A(P \backslash b_A(P), I)$ is the program consisting of the single rule:

$$r \leftarrow \ .$$

□

We reformulate here the splitting theorem as given in [17].

Theorem 2 *(Splitting theorem [16; 17])* Let P be a program and A be a splitting set. Then, $M \in ans_{\mathcal{U}}(P)$ iff M can be split in two disjoint sets I and J, such that $I \in ans_{\mathcal{U}}(b_A(P))$ and $J \in ans_{\mathcal{U}}(r_A(P) \setminus b_A(P)), I)$.

4.2 Safety

We consider now an interesting subclass of VI-programs, namely *vi-safe* programs; we will exploit the above theorem in order to prove that each vi-safe program can be evaluated simply taking into account only the constants originally appearing in the program itself.

Definition 2 Let r be a rule. A variable X is *vi-safe* in r if it appears in some *ordinary* atom $a \in B^+(r)$. A rule r is *vi-safe* if each variable X appearing in r is vi-safe. A program P is *vi-safe* if each rule $r \in P$ is vi-safe.

Note that the notion of *vi-safety* makes distinction between ordinary and external atoms, while *safety* as defined in Section 3, does not. That is, for making a variable safe, it is necessary its appearance in a positive literal, while vi-safety requires a variable to appear explicitly in a ordinary (non-external) atom. Both conditions are syntactic. As an important semantic consequence, vi-safety prevents completely the appearance of new symbols in a given program, as next theorem shows.

Theorem 3 Let P be a vi-safe `VI` program. Let $U \subset \mathcal{U}$ be the set of constants appearing in P. Then $ans_U(P) = ans_{\mathcal{U}}(P)$.

Proof Let's denote with A the set of ground atoms appearing in $grnd_U(P)$. Assuming P as vi-safe, it is easy to see that A is a finite splitting set for P. Furthermore, $grnd_U(P) = b_A(P)$. For each $M \in ans_U(P)$, we have that $r_A(grnd_{\mathcal{U}}(P) \setminus b_A(P), M)$ is consistent and its only answer set is the empty set (indeed, no rule can be ever satisfied unless the variables are bound to constants appearing in U). Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$ (Theorem 2). Viceversa, assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments easily lead to conclude that $M \in ans_U(P)$. □

In case a vi-safe program is given, the above theorem allows to consider as the set of 'relevant' constants only those values explicitly appearing in the program at hand. The semantics of a vi-safe program P can be evaluated by means of the following algorithm:

1. compute $grnd_U(P)$, where U is defined as in Theorem 3;
2. remove from $grnd_U(P)$ all the rules containing at least one external literal e such that $F \not\models e$, and remove from each rule all the remaining external literals, obtaining a reduced ground program that we will call $\overline{grnd}_U(P)$.
3. evaluate the answer sets of $\overline{grnd}_U(P)$ by means of a standard Answer Set solver.

It is worth pointing out that, assuming the complexity of computing oracles is polynomial in the size of their arguments, this algorithm has the same complexity as computing $grnd_U(P)$[3].

Example 4 Consider the rule 8 and the following two facts:

$$\begin{array}{r} \text{passwd(jack,short).} \\ \text{passwd(bill,longpasswd).} \end{array} \tag{12}$$

The resulting ground program after step 1 contains, among the others, the rules:

$$\begin{array}{rl} \text{mustChangePasswd(jack)} \leftarrow & \text{passwd(jack,short),} \\ & \#\text{strlen(short,5)}, \#<(5,8). \\ \text{mustChangePasswd(bill)} \leftarrow & \text{passwd(bill,longpasswd),} \\ & \#\text{strlen(longpasswd,10)}, \#<(10,8). \end{array} \tag{13}$$

Step 2 is such that only one of the two above rules is kept (after modification):

$$\text{mustChangePasswd(jack)} \leftarrow \text{passwd(jack,short).} \tag{14}$$

that no longer contains external atoms. □

[3] Assuming rules can have unbounded length, grounding a disjunctive logic program is in the worst case exponential in the size of the Herbrand base (see e.g. [18]).

5 Dealing with Value Invention

Although 2-valued oracles are important for clarifying the given semantics, we aim at introducing the possibility to specify *functional oracles*, keeping anyway the simple reference semantics given previously.

For instance, assume $\mathcal{U}$ contains encoded values that can be interpreted as natural numbers and that the external predicate #sqr is defined such that the atom #sqr(X,Y) is satisfied whenever Y encodes a natural number representing the square of the natural number X; we want to extract a series of squared values from this predicate; consider the short program

$$\begin{array}{l} \text{number}(2) \leftarrow . \\ \text{square}(\text{Y}) \leftarrow \text{number}(\text{X}), \#\text{sqr}(\text{X},\text{Y}). \end{array} \tag{15}$$

In the presence of unsafe rules as in the above example, Theorem 3 ceases to hold: it is indeed unclear whether there is a finite set of constants which the program can be grounded on. In the above example, we can intuitively conclude that the set of meaningful constants is $\{2, 4\}$. Nonetheless, it is in general undecidable, given a computable oracle f, to establish whether a given set S contains all and only those tuples $\bar{t}$ such that $f(\bar{t}) = 1$.

In the new setting we are going to introduce, it is also very important that an external atom brings knowledge from external sources of computation, in terms of new symbols added to a given program. We extend our framework with the possibility of explicitly computing missing values on demand. Although restrictive, this setting is not far from a realistic scenario where external predicates are defined by means of generic partial functions.

Definition 3 Let #p be an external predicate name of arity n, and let $F(\#\text{p})$ be its oracle function. A *pattern* is a list of i's and o's, where a i represents a placeholder for a constant (or a bounded variable), and an o is a placeholder for a variable. Given a list of terms, the corresponding pattern is given by replacing each constant with a i, and each variable with a o. Positions where o appears are called *output* positions whereas those denoted with i are called *input* positions. For instance, the pattern related to the list of terms (X, a, Y) is (o, i, o). □

Let *pat* be a pattern of length n having k placeholders i (*input positions*), and $n - k$ placeholders of o type (*output positions*). A *functional oracle* $F(\#\text{p})[pat]$ 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 $d_1, ..., d_{n-k} \in F(\#\text{p})[pat](c_1, ..., c_k)$ iff $F(\#\text{p})(h_1, \ldots, h_n) = 1$, where for each $l(1 \leq l \leq n)$, $h_l = c_j$ if the j-th i value occurs in position l in *pat*, otherwise $h_l = d_j$ if the j-th o value occurs in position l in *pat*. Let $pat[j]$ be the j-th element of a pattern *pat*. Let $output_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = o$ for each $X_j \in \overline{X}$, and $input_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = i$ for each $X_j \in \overline{X}$.

An external predicate #p might be associated to one or more functional oracles 'consistent' with the originating 2-valued one. For instance, consider the #sqr external predicate, defined as mentioned above. We can have two

functional oracles, $\mathrm{F}(\#\mathrm{sqr})[i,o]$ and $\mathrm{F}(\#\mathrm{sqr})[o,i]$. The two functional oracles are such that, e.g. $\mathrm{F}(\#\mathrm{sqr})[i,o](3) = 9$ and $\mathrm{F}(\#\mathrm{sqr})[o,i]\ (16) = 4$, consistently with the fact that $\mathrm{F}(\#\mathrm{sqr})(3,9) = \mathrm{F}(\#\mathrm{sqr})(4,16) = 1$, whereas $\mathrm{F}(\#\mathrm{sqr})[o,i](5)$ is set as undefined since $\mathrm{F}(\#\mathrm{sqr})(X,5) = 0$ for any natural number X.[4]

For the sake of simplicity, 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 exactly one functional oracle $F(\#\mathrm{e})[pat_{\#\mathrm{e}}]$, having pattern $pat_{\#\mathrm{e}}$. It is worth noting that this does not cause any loss of generality: indeed, having an external predicate with two (or more) different functional oracles is equivalent to having two (or more) different external predicates with one functional oracle each, and using the proper one every time a particular oracle is desired.

Once functional oracles are given, it is important to investigate which are the cases where they can be used for computing the actual set of ground instances of a given rule.

To this end, we introduce the notion of weakly safe variable and of weakly safe rule. Intuitively, a variable is weakly safe if its domain, although not explicitly bound to the domain of an ordinary atom, can be computed indirectly through a functional oracle.

For instance, the second rule of Program 15 is not vi-safe (since Y is not vi-safe), but is such that, intuitively, the domain of Y can be computed once the domain of X is known, provided a proper oracle $F(\#\mathrm{sqr})[i,o]$ is given for #sqr. The following definition captures this intuition.

Definition 4 Let r be a rule. A variable X is *weakly safe in* r if either

- X is vi-safe (i.e. it appears in some positive atom of $B^+(r) \setminus E(r)$); or
- 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 5 Assume that #sqr is associated to the functional oracle $F(\#\mathrm{sqr})[i,o]$ defined above. The second rule of Program 15 is weakly safe (X is vi-safe, while Y appears in output position in the atom $\#\mathrm{sqr}(X,Y)$). The same rule is not weakly safe if we consider the functional oracle $F(\#\mathrm{sqr})[o,i]$. □

Proposition 1 Let $\mathcal{E}$ be a set of external predicates, and $\mathcal{L}$ be the list of functional oracles associated to elements of $\mathcal{E}$. It can be checked in polynomial time whether a program P is weakly safe.

Proof Simply observe that for each rule $r \in P$ it can be checked in time linear in the number of atoms of r whether the patterns of the functional oracle associated to each external atom make the rule vi-safe or not. □

[4] 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.

Weakly safe rules can be grounded with respect to functional oracles as follows.

Definition 5 Let $I = \langle S, F \rangle$ an interpretation. We call $ins(r, I)$ the set of ground instances r_θ of r for which $I \models B^+(r_\theta)$, and such that $I \models E(r_\theta)$. □

Proposition 2 Let I be a finite interpretation, and r be a weakly safe rule. $ins(r, I)$ is finite.

Proof Indeed, given a weakly safe rule r, the set of functional oracles associated to each external atom, and a set of ordinary ground atoms I, any ground rule r' which is member of $ins(r, I)$ can be generated by the following algorithm:

1. replace positive literals of r with a consistent nondeterministic choice of matching ground atoms from I; let θ the resulting variable substitution;
2. until θ instantiates all the variables of r:
   - pick from $r\theta$ an external atom $\#\mathrm{e}(\overline{X})\theta$ such that θ instantiates all the variables $X \in input_{pat}(\overline{X})$.
   - choose nondeterministically a tuple $\langle a_1, \ldots, a_k \rangle \in F(input_{pat}(\overline{X}\theta))$, then update θ by assigning $a_1, \ldots, a_k$ to $output_{pat}(\overline{X}\theta)$;
3. return $r' = r\theta$. □

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:

$$\begin{array}{l} \mathrm{square(2)} \leftarrow . \\ \mathrm{square(Y)} \leftarrow \mathrm{square(X)}, \#\mathrm{sqr(X,Y)}. \end{array} \tag{16}$$

has the infinite set of atoms $\{\mathrm{square(2)}, \mathrm{square(4)}, \ldots\}$ as answer set.

6 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 8 below, value invention is nonetheless desirable in a variety of contexts.

Our approach investigates which programs, allowing value invention, can be solved by means of a finite ground program having a finite set of models of finite size.

Definition 6 A class of VI programs $\mathcal{V}$ has the *finite grounding* property if, for each $P \in \mathcal{V}$ there exists a finite set $U \subset \mathcal{U}$ such that $ans_U(P) = ans_{\mathcal{U}}(P)$. □

This class of programs (having the *finite grounding property*) is unluckily not recognizable in finite time.

Theorem 4 Recognizing the class of all the `VI` programs having the finite grounding property is undecidable.

Proof Positive logic programs with function symbols can simulate Turing machines. Also weakly safe `VI` programs can mimic (see section 8.1) 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 a finite set of symbols U, such that Definition 6 is applicable, 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 $grnd_U(P_{\mathcal{T},x})$. Hence the result follows. □

Note that te above theorem holds under the assumption that functional oracles might have an infinite co-domain, although functional oracles are supposed to associate, to each fixed combination of input values, a finite number of combination of values in output. Also, it is assumed to deal with weakly safe programs.

6.1 `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 $[i, o]$):

$$\begin{array}{r} \text{a(k,c)} \leftarrow . \\ \text{p(X,Y)} \leftarrow \text{a(X,Y)}. \\ \text{p(X,Y)} \leftarrow \text{s(X,Y)}, \text{a(Z,Y)}. \\ \text{s(X,Y)} \leftarrow \text{p(Z,X)}, \#\text{b(X,Y)}. \end{array} \tag{17}$$

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 7 The *attribute dependency graph* $AG(P)$ associated to a 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:

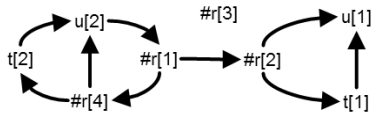
- $(q[j], p[i])$, if p appears in some atom $a_p \in H(r)$, q in some atom $a_q \in B^+(r) \setminus E(r)$ and $q[j]$ and $p[i]$ share the same variable in a_q and a_p respectively.
- $(q[j], \#p[i])$, if q appears in some atom $a_q \in B^+(r) \setminus E(r)$, $\#p$ in some atom $a_{\#p} \in E(r)$, $q[j]$ and $\#p[i]$ share the same variable in a_q and $a_{\#p}$ respectively, 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 in $a_{\#q}$ and $a_{\#p}$ respectively, 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 in $a_{\#q}$ and $a_{\#p}$ respectively, 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 $a_p \in H(r)$, $\#q$ in some atom $a_{\#q} \in E(r)$, $\#q[j]$ and $p[i]$ share the same variable in $a_{\#q}$ and a_p respectively, and j is an output position for the functional oracle of $\#q$; □

Example 6 The *attribute dependency graph* induced by the *first three* rules of the motivating example in Section 2 is depicted in Figure 1. □

Definition 8 Let P be a weakly safe program. Then[5]:

- 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 (6).
- 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 (6) is dangerous since connected[2] is poisoned and appears in a cycle.
- Let r be a dangerous rule. 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

[5] All examples refer to the Motivating Example, Section 2. Also, we assume that #rdf has functional oracle with pattern $[i, o, o, o]$



trusted(X,U) ← #rdf("myurl", X, "trusts", U).
url(X,U) ← #rdf("myurl", X, "seealso", U), trusted(X,U).
url(X,U) ← url(_, U1), #rdf(U1, X, "seealso", U), trusted(X,U).

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

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 (6) 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*. □

Theorem 5 `VI`-restricted programs have the finite grounding property.

Proof Let P be a `VI`-restricted program. 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 5) 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

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

where X is a set of ground rules and $atoms(X)$ is the set of ordinary atoms appearing in X. $T_P^\infty(\emptyset)$ 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. Now, 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 one producing a finite output.

Because of the way it has been constructed, it is easy to see that $A = atoms(gr_P)$ is a splitting set [16], 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. □

7 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 proven the algorithm stops. The program is VI-restricted if RTBC is empty at the last iteration.

The algorithm consumes 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* (Appendix A) 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 (when all attributes are initially in NSA and only one attributes can be stated as savior at each step); 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).

8 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.

8.1 Functional terms

It is worth noting that we consider here rule based languages allowing functional terms such that the variables appearing in the head appear also in the positive body. A program P with functional terms is reduced to a VI program $\mathsf{F}(P)$ as follows.

```
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 check.
    Bool NSA_Updated = true;
    While ( NSA_Updated ) do % Try to prove VI-restriction when some change occurs.
        NSA_Updated = false;
        For each Rule r ∈ RTBC do % free(r) = 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); % r 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*

For each natural number k, we introduce two distinct external predicates: $\#function_k$ and $\#function'_k$, of arity $k+2$ each; they are such that: $F(\#function_k)(Ft, f, X_1, \ldots, X_k) = F(\#function'_k)(Ft, f, X_1, \ldots, X_k) = 1$ if and only if the term Ft is $f(X_1, \ldots, X_k)$. Each $\#function_k$ ($\#function'_k$, respectively) predicate is associated to a functional oracle $F(\#function_k)[o, i, i, \ldots, i]$ ($F(\#function'_k)$ $[i, o, o, \ldots, o]$, respectively).

The family of $\#function_k$ external predicates are intended to construct a functional term if all of its arguments are bound, whereas the family of $\#function'_k$ predicates are exploited when the whole functional term is known and we want to extract its arguments.

The transformation F is such that, any functional term $t = f(X_1, \ldots, X_n)$, appearing in some rule $r \in P$, is replaced by a fresh variable Ft. A proper atom $\#function_k(Ft, f, X_1, \ldots, X_n)$ or $\#function'_k(Ft, f, X_1, \ldots, X_n)$ is then added to the body of r. This kind of transformation is performed until a

functional term is still in r. We choose $\#function'_k$ if t appears in the positive body of r, whereas an atom using $\#function_k$ is used if t appears in the negative body or in the head of r.

Example 7 The rule { p(s(X)) ← a(X, f(Y, Z)). } contains two function symbols, *s* and *f*. It can be rewritten as { p(Ft1) ← a(X, Ft2), $\#function_1$(Ft1, s, X), $\#function'_2$(Ft2, f, Y, Z). } □

Proposition 3 Let P be a weakly safe logic program with functional terms P. Then: (i) $\mathsf{F}(P)$ is weakly safe, and (ii) there is a 1-to-1 mapping between $ans_{\mathcal{U}}(P)$ and $ans_{\mathcal{U}}(\mathsf{F}(P))$. □

Proof (i) Each variable in $\mathsf{F}(P)$ already appears in P, and thus is weakly safe by hypothesis. The only exceptions are constituted by the fresh variables Fts. Whatever the rule, such a variable will always appear in some external atom (even if appears also in a regular atom); by definition of F, this must be either of type $\#function_k()$ or of type $\#function'_k()$. In the first case, Ft is the only output term (all the others are input ones): this means that Ft is weakly safe by Definition 4, second point. In the latter case, Ft is an input term, and thus weakly safe by Definition 4, first point.

(ii) It is easy to see that semantics of a `VI` program P is defined such that the answer sets of $\mathsf{F}(P)$ are the answer sets of $\overline{grnd}_{\mathcal{U}}(\mathsf{F}(P))$, which does not contain any external atom. Because of the way the families of $\#function_k$ and $\#function'_k$ external predicates are defined, this ground program is the same as $\overline{grnd}_{\mathcal{U}}(P)$.

8.2 Set unification and set terms

The accommodation of sets in logic programming has often been attempted; the reader may refer to [19] for a survey on sets in logic programming and on set unification methods and algorithms.

Set terms are usually encoded as lists of elements. But, unlike lists, set terms do not have to carry information about the position of a given element inside the set itself. Thus the classic notion of term unification has to be generalized.

For instance, if a set term is encoded as $\{X, a, b, c\}$ then it has to unify with the encoding $\{d, a, b, c\}$, but also with $\{a, d, b, c\}$.

It is possible to embody set constructors and set unification in the context of `VI` programs by means of a proper reduction. Roughly speaking, a logic program with sets replaces the classical notion of term with the notion of *set term*. A set term is either *(i)* a classical term, or *(ii)* a term of the form $\{X_1, \ldots, X_n\}$ where $X_1, \ldots, X_n$ are set terms, or *(iii)* a term of the form $X \cup Y$ where X and Y are set terms. Two ground set terms are equal if they contain the same set of ground terms.

A program with set terms P is reduced to a `VI` program $\mathsf{S}(P)$ as follows. 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, having exactly $k+1$ arguments each such that $F(\#set_k)(X, Y_1, \ldots, Y_k) = F(\#set'_k)(X, Y_1, \ldots, Y_k) = 1$ if X encodes the set $\{Y_1, \ldots, Y_k\}$. Predicates $\#set_k$ have the functional oracle $F(\#set_k)[o, i, \ldots, i]$, while predicates $\#set'_k$ has the functional oracle $F(\#set'_k)[i, o, \ldots, o]$;

(ii) two ternary external predicates $\#union$ and $\#union'$; they are such that $F(\#union)(X, Y, Z) = F(\#union')(X, Y, Z) = 1$ either if $X = Y \cup Z$, or if X and Y are classical terms, $Z = \{\}$ and $X = Y$. The predicate $\#union$ has the functional oracle $F(\#union)[o, i, i]$, while the predicate $\#union'$ has the functional oracle $F(\#union')[i, o, o]$.

Then each rule $r \in P$ is modified, until it has no set terms, by:

- choosing a set term $t = \{X_1, \ldots, X_n\}$ appearing in r, replacing it with a fresh variable St, and adding to the body of r
  - the external atom $\#set_n(St, X_1, \ldots, X_n)$, if the set term appears in the negative body or in the head of r;
  - the external atom $\#set'_n(St, X_1, \ldots, X_n)$ otherwise;
- replacing each set term $X \cup Y$ appearing in r with a fresh variable Ut, and adding in the body of r the external atom $\#union(Ut, X, Y)$ if the set term appears in the negative body or in the head of r, $\#union'(Ut, X, Y)$ otherwise. This and the previous step are applied to r for all set terms;
- if a variable X appears in r for m times ($m > 1$), then each occurrence of X is replaced with a fresh variable $X_i (1 \leq i \leq m)$, and for each pair $(X_i, X_j), 1 \leq i < j \leq m$, the atom $\#union(X_i, X_j, \{\})$ is added to r. This last step is due since we have to force occurrences of the same variable in a rule to be unified by means of set unification instead of the classic unification.

Example 8 If we consider the rule: { p(X ∪ Y) ← a({a, X}), b({Y}). }, then the analogous `VI` rule is: { p(St1) ← a(St2), b(St3), $\#union$(St1, X1, Y1), $\#set'_2$(St2, a, X2), $\#set'_1$(St3,Y2), $\#union$(X1,X2,{}), $\#union$(Y1,Y2,{}). } □

Proposition 4 Let P be a logic program with set terms. Then $\mathsf{S}(P)$ is, by construction, weakly safe. Also, there is a 1-to-1 mapping between $ans_{\mathcal{U}}(P)$ and $ans_{\mathcal{U}}(\mathsf{S}(P))$. □

Proof Fully analogue to the proof of Proposition 3.

9 Relationships with other Classes of Programs allowing Value Invention

9.1 ω-restricted programs

In the same spirit of this paper are ω-restricted programs [8], that allow function symbols under answer set semantics. The introduced restrictions aim at controlling the creation of new functional terms.

Definition 9 [8] Let P be a program with function symbols and no external predicates. An ω-stratification is a traditional stratification (i.e. a function

mapping each predicate name to a *level* number) 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. Given an ω-stratification for P, 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-restricted programs have the finite grounding property: only a finite amount of functional terms can be created since each variable appearing in the head of a rule must be bound to a predicate belonging to a lower layer. VI-restricted programs do not introduce special restrictions for non-stratified cycles, instead. Also, it is not necessary to bound the domain of each variable to a previous layer explicitly. The class of VI-restricted programs contains, in a sense, the class of ω-restricted ones. That is, any ω-restricted program P is such that the counterpart VI program $\mathsf{F}(P)$ is VI-restricted.

Theorem 6 Let P be an ω-restricted program. $\mathsf{F}(P)$ is VI-restricted.

Proof Given an ω-restricted program P, we observe that:

- Attributes belonging to predicates which are not in the ω-stratum are obviously savior: the relevant instantiation of these predicates is computable starting from the lowermost layer, and is finite.
- The rewritten rules in $\mathsf{F}(P)$ 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 for fresh variables substituting functional terms. 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 a fresh variable $Ft1$ appearing in $\#function'_k(Ft1, X_1, \ldots, X_k)$. If $Ft1$ is already bound to a positive atom, then there is no value invention; otherwise, all terms $X_1, \ldots, X_k$ are bound either to a positive atom or to another external atom in output position (see Section 8.1). As stated before, the attributes where $X_1, \ldots, X_k$ appear are savior, and so $Ft1$ is as well. □

On the other hand, the opposite does not hold.

Theorem 7 It is possible to find non-ω-restricted programs whose transformation F outputs a VI-restricted program.

Proof The program $P_{n\omega r}$:

$$p(f(X)) \leftarrow q(X), t(X).$$
$$q(X) \leftarrow p(X). \quad p(1). \quad t(1).$$

is easily recognizable as not ω-restricted; nevertheless, the transformation $\mathsf{F}(P_{n\omega r})$ is VI-restricted:

$$p(F1) \leftarrow q(X), t(X), \#function_2(F1, f, X).$$
$$q(X) \leftarrow p(X). \quad p(1). \quad t(1).$$

□

9.2 Finitary programs

Finitary programs allow function symbols under answer set semantics [9]. 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 8 The class of finitary programs is not comparable with the class of VI-restricted programs.

Proof 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, programs which are in the class $\{\mathsf{F}(P) \mid P \text{ is finitary}\}$ are not necessarily VI-restricted: for instance the translation of the finitary program $\{p(0); p(s(X)) \leftarrow p(X)\}$ is not VI-restricted. □

9.3 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. In this setting, it is possible to study ascending and descending chains of functional terms in order to prove decidability. Similar to our approach is the work on open and conceptual logic programs [20], that 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 [1; 3] and [21], where special constructs, aimed at creating new tuple identifiers in relational databases is introduced.

In [22] 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. The approach is different from the one considered in this work, as it 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). A simple safety condition for HEX programs is given in [23]. 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 9 If an HEX program P does not contain higher order atoms, but only HEX predicates, then it is equivalent to a `VI` program.

Proof By construction: it is easy to obtain an equivalent `VI` program P' by simply replacing each HEX atom of the form $\#g[Y_1, \ldots, Y_m](X_1, \ldots, X_n)$ by an external atom of the form $\#g'(Y_1, \ldots, Y_m, X_1, \ldots, X_n)$; then defining each evaluation function $F(\#g')$ such that for each I we have that $f_{\#g'}(Y_1, \ldots, Y_m, X_1, \ldots, X_m) = f_{\#g}(I, Y_1, \ldots, Y_m, X_1, \ldots, X_m)$. □

9.4 Other notions of safety

The adoption of a terminology like *weakly* (*resp. strong*) *safe rule*, *safe program*, has been used in the past literature with several intended meanings. Our notion of safe rule mimics the traditional syntactic notion, carrying the same name, adopted for Datalog [24] and sometimes also referred as *range-restrictedness* in the database field. It is worth noting that there is some literature where the notion of *safety* is semantic more than syntactic. In such acception, a logic program P is considered *safe* if its iterative fixpoint semantics terminates giving a finite result like in [25; 26]. This idea resembles our idea of finite grounding. Nonetheless, this notion of safety is usually related to the termination of a specific algorithm (namely, the traditional iterative application of the immediate consequence operator), on a given positive Datalog program, while ours is independent from a given operational semantics.

Also, in [1], value invention is considered in the context of queries and update languages for databases. Abiteboul and Vianu allow non range-restricted rules in a program. Non range-restricted variables appearing in rules are given a special semantics such that they can be associated with freshly created symbols. Their notion of *strong safety* resembles our notion of *vi-safety*, in the sense that both notions disallow at all the appearance of variables that may carry invented values. In the same work, the word *safe* is referred to a query (or update program) whose output contains only symbols appearing in the input database.

The notion of *weak safety* has been also overridden many times in literature. Just to cite some, it refers to programs whose invented values may play a role, but only within intermediate predicates of a given query [21; 1], and not within output predicates. Also, *weak safety* refers to logic programs allowing input relations of infinite size. Weakly safe programs assure that each single step of their iterative computation gives finite output [26], although they might not be safe in the sense of [25] (i.e. their overall evaluation does not terminate).

In our acception, a weakly safe rule is such that available oracles ensure that it can be found an order of evaluation for the external atoms appearing

in a rule. Nonetheless, by Proposition 2, weakly safe rules have a property that might be considered close to the notion of *weak safety* in the sense of [26].

10 Implementation and Experiments

The proposed language has been integrated into the ASP system DLV [6]. The resulting system is called DLV-EX [27].

10.1 Design of the system

From a practical point of view, the external atoms are dealt with in the following steps (see Figure 3):

1. at design time: a developer provides a library of external atoms, each of them associated with a set of functional oracles.[6] Each functional oracle has a corresponding pattern. Although useful in practice, it is not compulsory to provide functional oracles other than the base oracle. However, the absence of specific functional oracles limits *de facto* the possibility to exploit an external atom in weakly safe rules. A testing environment helps checking the correctness of the oracles by means of automatically generated test programs.
2. at run-time (pre-processing stage): first, each rule is checked to be weakly safe, and at the same time the system associate to each external atom a proper oracle (the user is not in charge of specifying the one to be chosen); then, the program is checked to be VI restricted;
3. at run-time (rule instantiation stage): the optimized grounder of the DLV system has been extended in order to compute $ins(r, A)$ for a given rule r and a set of 'active' atoms A, in the presence of external atoms. For each external atom in r, the chosen functional oracle is repeatedly invoked according to Definition 5.

10.2 Integration into DLV

We briefly recall the instantiation algorithm of the DLV system [18]. Given a rule r, this algorithm exploits an intelligent backjumping algorithm, where a given atom $a \in B(r)$ is picked at each stage and it is tried to be instantiated with respect to currently allowed values. The picking order is crucial in order to tailor the search space to the smallest extent. Each atom has a set of possible instances, which will be known only at the end of the computation; nevertheless, its size can be estimated, and used as a guideline for the choice of the atom to be picked. In principle, it is preferred to pick first those atoms

[6] The real implementation allows the association of more than one functional oracle to each external predicate, since this is far more comfortable from the user's viewpoint.

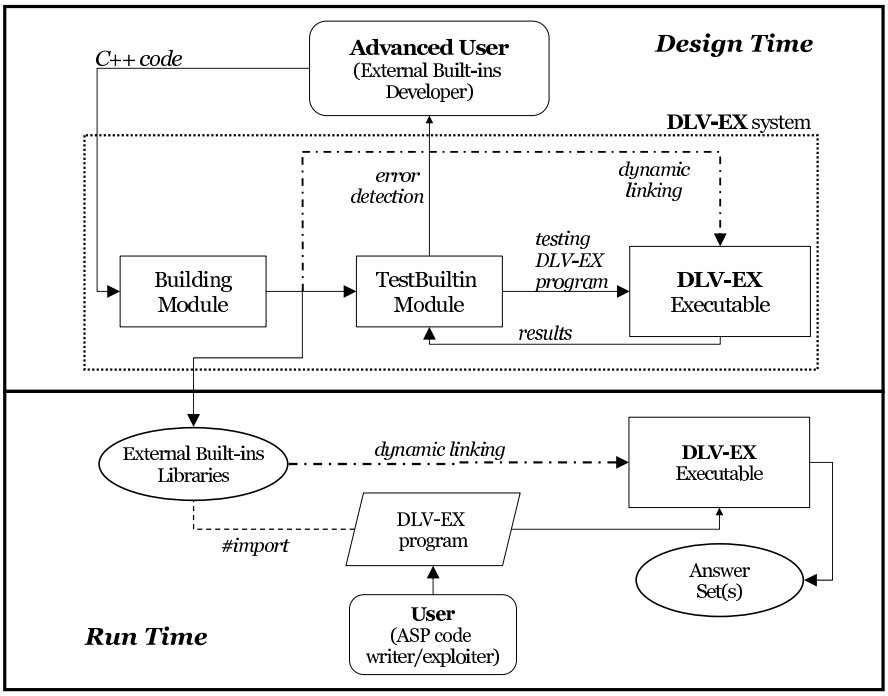


Fig. 3 *System Architecture*

whose estimated number of instances is smaller. Of course, this estimate might not be accurate in some border cases.

Point 2 and 3 above have been integrated in the grounding algorithm.

Point 2 required the implementation of the recognition algorithm depicted in Figure 2 which is invoked *before* the actual grounding process. The algorithm has been implemented exploiting what already provided by the dependency graph computed by `DLV`, i.e., starting from the information on components of a program (presence of cycles, rules involved, etc.), and the relationships among variables; a check on dangerous attributes to verify if all of them are blocked is performed. This stage is performed only once, analyzing statically all rules in the program. When a program is recognized as `VI`-restricted the usual evaluation starts; otherwise, an error message listing the dangerous rules is shown. However, it is always possible to relax this check, either for a specific external built-in predicate, or for all of them.[7]

Given a rule r, among possible choices of functional oracles, the algorithm prefers patterns with bigger numbers of unbounded variables. This intuitively allows to reduce the space of possible instantiations for a given external atom. For instance, given the atom #sqr(X,Y), the choosing algorithm prefers, whenever possible, to choose the oracle with pattern $[i, o]$ instead of the base oracle (pattern $[i, i]$), since this way it is searched only the space of values where Y is equal to the square of X. In the second case, an oracle with

[7] Indeed, it is worth to remember that there may be some programs having the finite grounding property even if they can not be recognized as being `VI`-restricted.

Problem	a	b	c
knowledge-discovery	6.88	6.78	6.72
constraint-3col[n=30,e=40]	0.00	0.00	0.01
scheduling	0.85	0.85	0.89
scheduling-alternate	0.88	0.88	1.01
cristal	8.45	8.62	10.33
2qbf1	0.39	0.40	0.38
ancestor	109.65	109.90	108.45
3col-simplex1	2.14	2.20	2.11
3col-n-ladder	2.10	2.20	2.01
3col-random1	0.31	0.31	0.31
hp-random1	1.60	1.71	1.92
decomp2	12.78	11.93	12.32
hanoi-towers	0.30	0.31	0.38
ramsey(3,7) != 22	22.01	21.71	21.98
21-queens	0.99	0.99	1.60
school_timetabling	93.95	99.60	103.66

Table 1 Average Real Grounding Times.

pattern $[i, i]$ forces in principle to check all the possible couples of values for X and Y. It is worth noting that the choice of the oracle does not affect the result of the computation, since each oracle must be coherent to the base one.

Thus, the atom pick-up ordering strategy has an impact in performance of point 3. This strategy relies on the assumption, often true in practice, that the computation of a functional oracle is less time consuming than several computations of the corresponding base oracle.

10.3 Testing

First of all, it is worth stating that the implementation of such extension into the system has not been a matter of performance: what is relevant, here, is the possibility to widely customize the system capabilities and to deal with value invention. Nevertheless, it is still important to ensure that the system does not perform too bad on standard `DLV` programs (i.e., `VI`-free). Thus, some benchmarks[8] have been carried out; all the pre-existing built-in atoms available in `DLV` (such as arithmetic and relational operators) have been rewritten using the new general framework. In order to appreciate the impact and the possible overhead of the new construct, we built three different versions of the `DLV` system:

a. the `DLV` system "as it is", i.e., without any support to `VI` programs;
b. the `DLV` system modified in order to support `VI` programs;
c. the `DLV` system as the latter, but with the pre-existing built-in predicates (arithmetic predicates, comparison predicates), replaced by external atoms equipped with proper oracles.

The three versions have been tested on a set of problems coming from a suite used to be exploited by the `DLV` team for internal purposes. The full set

[8] For some notes on benchmarking have a look at `http://www.dbai.tuwien.ac.at/proj/dlv/bench/`.

of problem, instances and executables is available at `http://www.gibbi.com/test_amai2007.tar.gz`; results are reported in Table 1. At a glance, grounding times are basically equivalent. In some cases, system *c* is affected by a small loss of performances: this was an expected payload, as calls to built-in predicates were replaced by indirect invocation to external modules. Other swings in performance among all the systems are due to the modification of the atom pick-up order, induced by the presence of external atoms whose estimated domain size might be different; as already discussed, this may affect (heavily, sometimes) the instantiation performances (see again [18]).

Some usage experiments have been carried out as well; several users have started implementing some customized external built-ins, and the feedbacks are very positive, so that some relevant scientific works have taken advantage from the new system [28; 29].

10.4 Libraries

Defining a new external predicate entails the implementation of one or more oracles written in the C++ language. External predicate definitions can be grouped in one or more libraries and have to be compiled such that they can be dynamically linked to the `DLV-EX` executable. Proper tools have been designed and realized in order to help the user in compiling and collecting oracles. A special directive inside `DLV-EX` programs tells the system which libraries have to be linked at runtime.

Since the first release of the system prototype, we realized a number of external predicates of practical interest. These, as well as some of those developed by other users for their own purposes, have been collected in libraries which are now publicly available. Currently, there are three main libraries. Two of them include a very rich set of manipulation functions for natural and real number, respectively. It is worth noting that, at present, numeric data different from natural numbers are not natively supported by the `DLV` system. The third library includes a set of predicates for string manipulation. Libraries are continuously enriched and improved, and others are coming.

The system itself, as well as examples, manuals, and external built-in libraries, can be found on the system website [27].

11 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 (all those whose corresponding translation is `VI`-restricted).

A topic for future work is to investigate to what extent the notion of `VI`-restrictedness can be relaxed, yet 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 of checking. Nonetheless, the spirit of restriction checkers is to keep evaluation times greatly smaller than the overall solving times.

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.

Acknowledgements We thank anonymous referees for their useful comments and suggestions. We also acknowledge Nicola Leone for his fruitful comments on the former versions of this paper.

The work has been supported by the Italian Research Ministry (MIUR) under the projects 'Logic based Knowledge representation languages: extensions and optimization techniques' (Interlink II04CG8AGG), 'Application and enhancement of Disjunctive Logic Programming' (PRIN 2006019157), and by the Austrian Science Funds (FWF), under the projects 'Answer Set Programming for the Semantic Web' (FWF P17212).

References

[1] Abiteboul, S., Vianu, V.: Datalog Extensions for Database Queries and Updates. Journal of Computer and System Sciences **43**(1) (1991) 62–124

[2] Cabibbo, L.: Expressiveness of Semipositive Logic Programs with Value Invention. In: Logic in Databases. (1996) 457–474

[3] Hull, R., Yoshikawa, M.: ILOG: Declarative Creation and Manipulation of Object Identifiers. In McLeod, D., Sacks-Davis, R., Schek, H.J., eds.: 16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings, Morgan Kaufmann (1990) 455–468

[4] Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: International Joint Conference on Artificial Intelligence (IJCAI) 2005, Edinburgh, UK (2005) 90–96

[5] Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In: Proceedings of the Second European Semantic Web Conference, ESWC 2005. Volume 3532 of Lecture Notes in Computer Science. (2005) 392–407

[6] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562

[7] Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234

[8] Syrjänen, T.: Omega-restricted logic programs. In: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Vienna, Austria, Springer-Verlag (2001)

[9] Bonatti, P.A.: Reasoning with Infinite Stable Models. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, Seattle, WA, USA, Morgan Kaufmann Publishers (2001) 603–610

[10] Faber, W., Pfeifer, G.: `DLV` homepage (since 1996) `http://www.dlvsystem.com/`.

[11] Ross, K.: The Well-Founded Semantics for Disjunctive Logic Programs. In Kim, W., Nicolas, J.M., Nishio, S., eds.: Deductive and Object-Oriented Databases. Elsevier Science Publishers B. V. (1990) 385–402

[12] Miller, L., Brickley, D.: The Friend of a Friend (FOAF) Project (since 2000) `http://www.foaf-project.org/`.

[13] Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385

[14] Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080

[15] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425

[16] Lifschitz, V., Turner, H.: Splitting a Logic Program. In Van Hentenryck, P., ed.: Proceedings of the 11th International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy, MIT Press (1994) 23–37

[17] Bonatti, P.A.: Reasoning with infinite stable models. Artificial Intelligence **156**(1) (2004) 75–111

[18] Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In Eiter, T., Faber, W., Truszczyński, M., eds.: Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings. Volume 2173 of Lecture Notes in AI (LNAI)., Springer Verlag (2001)

[19] Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory and Practice of Logic Programming (2006) To appear.

[20] Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Semantic web reasoning with conceptual logic programs. In: Rules and Rule Markup Languages for the Semantic Web: Third International Workshop, RuleML 2004, Hiroshima, Japan, November 8, 2004. (2004) 113–127

[21] Cabibbo, L.: The Expressive Power of Stratified Logic Programs with Value Invention. Information and Computation **147**(1) (1998) 22–56

[22] Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Nonmonotonic description logic programs: Implementation and experiments. In: Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004. (2004) 511–527

[23] Eiter, T., Ianni, G., Tompits, H., Schindlauer, R.: Effective Integration of Declarative Rules with External Evaluations for Semantic Web Reasoning. In: Proceedings of the 3rd European Semantic Web Conference (ESWC 2006). (2006) 273–287

[24] Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)

[25] Ramakrishnan, R., Bancilhon, F., Silberschatz, A.: Safety of recursive horn clauses with infinite relations. In: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, ACM (1987) 328–339

[26] Sagiv, Y., Vardi, M.Y.: Safety of datalog queries over infinite databases. In: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, ACM Press (1989) 160–171

[27] Calimeri, F., Ianni, G.: `DLVEX` homepage (since 2004) `http://www.mat.unical.it/ianni/wiki/dlvex`.

[28] Palopoli, L., Rombo, S., Terracina, G.: Flexible Pattern Discovery with (Extended) Disjunctive Logic Programming. In: International Symposium on Methodologies for Intelligent Systems (ISMIS 2005). Volume 3448 of Lecture Notes in AI (LNAI)., Saratoga Springs, New York, USA, Springer-Verlag (2005) 504–513

[29] Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387

A Details on VI-restrictedness Recognizing Algorithm

Briefly, recalling Definition 8, it is enough to find at least an external atom in the body of the given rule such that the variable appears in output position and all the variables in input position are blocked.

```
Bool Function isBlocked ( v: Var; % The variable to be checked as blocked.
                          r: Rule; % Current rule.
                          SA: Set{ Attr } ) % Savior attributes.
    % The set of all external predicate atoms in the positive body,
    % including the free variable v.
    Set{External_Atom} EAS = externalAtomsWithFreeVar( r, v );
    Bool isB = false;
    External_Atom #b = EAS.first();
    % At least one external predicate must have all of its input variables blocked.
    While ( !isB  &&  #b ≠ EAS.end() ) do
        % The set of input variables for the current external predicate.
        Set{ Var } inputVarsTBC = inputPatternVars( #b );
        Bool allVarsBlocked = true;
        Var currInputVar = inputVarsTBC.first();
        % Check savior property for all variables included in inputVarsTBC.
        While ( allVarsBlocked  &&  currInputVar ≠ inputVarsTBC.end() ) do
            % All the attributes of standard positive atom in the rule,
            % having as variable currInputVar.
            Set{ Attr } potentiallySavior =
                attrsWithVar( currInputVar, r );
            Bool saviorAttrFound = false;
            Attr  currAttr = potentiallySavior.first();
            % One savior attribute is sufficient.
            While ( !saviorAttrFound  &&
                currAttr ≠ potentiallySavior.end() ) do
                If ( currAttr ∈ SA ) then
                    saviorAttrFound = true;
                Else
                    currAttr = potentiallySavior.next();
                EndIf
            EndWhile
            If ( saviorAttrFound ) then
                % Check the next input variable.
                currInputVar = inputVarsTBC.next();
            Else
                % This input variable is currently not blocked.
                allVarsBlocked = false;
            EndIf
        EndWhile
        If ( allVarsBlocked ) then
            % An external predicate atom having
            % all its input variables blocked has been found.
            isB = true;
        Else
            % Try the next external atom including the free variable v.
            #b = EAS.next();
        EndIf
    EndWhile
    return isB;
EndFunction
```

Fig. 4 *The isBlocked Algorithm*