External sources of computation for Answer Set Solvers*

Francesco Calimeri and Giovambattista Ianni

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

Abstract. The paper introduces Answer Set Programming with External Predicates (`ASP-EX`), a framework aimed at enabling ASP to deal with external sources of computation. This feature is realized by the introduction of "parametric" *external predicates*, whose extension is not specified by means of a logic program but computed through external code. With respect to existing approaches it is explicitly addressed the issue of invention of new information coming from external predicates, in form of new, and possibly infinite, constant symbols. Several decidable restrictions of the language are identified as well as suitable algorithms for evaluating Answer Set Programs with external predicates. The framework paves the way to Answer Set Programming in several directions such as pattern manipulation applications, as well as the possibility to exploit function symbols. `ASP-EX` has been successfully implemented in the `DLV` system, which is now enabled to make external program calls.

1 Introduction

Among nonmonotonic semantics, Answer Set Programming (ASP) is nowadays taking a preeminent role, witnessed by the availability of efficient answer-set solvers, like ASSAT [Lin and Zhao, 2002], Cmodels [Babovich and Maratea, 2003], `DLV` [Leone *et al.*, 2005b], and Smodels [Simons *et al.*, 2002], and various extensions of the basic language with features such as classical negation, weak constraints, aggregates, cardinality and weight constraints. ASP has become an important knowledge representation formalism for declaratively solving AI problems in areas including planning [Eiter *et al.*, 2003], diagnosis and information integration [Leone *et al.*, 2005a], and more.

Despite these good results, state-of-the-art ASP systems hardly deal with data types such as strings, natural and real numbers. Although simple, this data types bring two kinds of technical problems: first, they range over infinite domains; second, they need to be manipulated with primitive constructs which can be encoded in logic programming at the cost of compromising efficiency and declarativity. Furthermore, interoperability with other software is nowadays important, especially in the context of those Semantic Web applications aimed at managing external knowledge.

The contributions of the paper are the following:

* Work supported by the EC under projects INFOMIX (IST-2001-3357) and WASP (IST-2001-37004), and by FWF under project "Answer Set Programming for the Semantic Web" (P17212-N04)

– we introduce a formal framework, named ASP-EX, for accommodating *external predicates* in the context of Answer Set Programming;
– ASP-EX includes the explicit possibility of invention of new values from external sources: since this setting could lead to non-termination of any conceivable evaluation algorithm, we tailor specific cases where decidability is preserved.
– we show that ASP-EX enhances the applicability of Answer Set Programming to a variety of problems such as string and algebraic manipulation. Also the framework paves the way for simulating function symbols in a setting where the notion of term is kept simple (Skolem terms are not necessary).
– we discuss implementation issues, and show how we have integrated ASP-EX in the DLV system, which is, this way, enabled with the possibility of using external sources of computation.
– we carry out some experiments, confirming that the accommodation of external predicates does not cause any relevant computational overhead.

2 Motivating example

The introduction of external sources of computation in tight synergy with Answer Set Solvers opens a variety of possible applications. We show next an example of these successful experiences.

The discovery of complex pattern repetitions in string databases plays an important role in genomic studies, and in general in the areas of knowledge discovery. Genome databases mainly consist of sets of strings representing DNA or protein sequences (biosequences) and most of these strings still require to be interpreted. In this context, discovering common patterns in sets of biologically related sequences is very important.

It turns out that specifying pattern search strategies by means of Answer Set Programming and its extensions is an appealing idea: constructs like strong and weak constraints, disjunction, aggregates may help an algorithm designer to fast prototype search algorithms for a variety of pattern classes.

Unfortunately, state-of-the-art Answer Set Solvers lack the possibility to deal in a satisfactory way with infinite domains such as strings or natural numbers. Furthermore, although very simple, such data types need of ad hoc manipulation constructs, which are typically difficult to be encoded and cannot be efficiently evaluated in logic programming.

So, in order to cope with these needs, one may conceive to properly extend answer set programming with the possibility of introducing *external* predicates. The extension of an external predicate can be efficiently computed by means of an intensional definition expressed using a traditional imperative language.

Thus, we might allow a pattern search algorithm designer to take advantage of Answer Set Programming facilities, but extended with special atoms such as e.g. *#inverse(S1,S2)* (true if *S1* is the inverse string of *S2*), *#strcat(S1,S2,S3)* (true if *S3* is equal to the concatenation of *S1* and *S2*), or *#hammingDistance(S1,S2,N)* (true if *S1* has N differences with respect to *S2*). Note that it is desirable that these predicates introduce new values in the domain of a program whenever necessary. For instance, the semantics of *#strcat(a,b,X)* should be such that X matches with the new symbol *ab*.

Provided with a suitable mechanism for defining external predicates, the authors of [Palopoli *et al.*, 2005] have been able to define and implement a framework allowing to specify and resolve genomic pattern search problems; the framework is based on automatically generating logic programs starting from user-defined extraction problems, and exploits disjunctive logic programming properly extended in order to enable the possibility of dealing with a large variety of pattern problems. The external built-in framework implemented into the `DLV` system is essential in order to deal with strings and patterns. We provide next syntax and semantics of the proposed framework.

3 Syntax and semantics

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 "$\#$". $\mathcal{U}$ will constitute the default *Herbrand Universe*. We will 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. Intuitively, 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 $\overline{t}$. A positive *literal* is an atom, whereas a *negative literal* is `not` a where a is an atom.

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

A *rule* r is of the form

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

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 *`ASP-EX` program* is a finite set P of rules; it is *ordinary*, if all rules are ordinary. Without loss of generality, we will assume P has no constraints[2] and only ground facts.

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 `ASP-EX` program:

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

[2] 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.

$$\begin{aligned} \textit{mustChangePasswd(Usr)} \leftarrow\ & \textit{passwd(Usr,Pass)}, \\ & \#\textit{strlen(Pass,Len)}, \#{<}\textit{(Len,8)}. \end{aligned} \tag{2}$$

We define the semantics of ASP-EX by generalizing the answer-set semantics, proposed by Gelfond and Lifschitz [1991] as an extension of the stable model semantics of normal logic programs [Gelfond and Lifschitz, 1988]. In the sequel, we will assume P is a ASP-EX 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 $grnd_{\mathcal{U}}(P) = \bigcup_{r \in P} grnd_{\mathcal{U}}(r)$.

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

- $S \subseteq HB_{\mathcal{U}}(P)$ contains only 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 $\#e \in \mathcal{E}$, a decidable n-ary Boolean function (which we will call *oracle*) $F(\#e)$ 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 $a = \#e(x_1, \ldots, x_n)$, denoted $I \models a$ ($F \models a$), if $F(\#e)(x_1, \ldots, x_n) = 1$.

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

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

The following is a ground version of rule 2:

$$\begin{aligned} \textit{mustChangePasswd(frank)} \leftarrow\ & \textit{passwd(frank,pat4dat)}, \\ & \#\textit{strlen(pat4dat,7)}, \#{<}\textit{(7,8)}. \end{aligned} \tag{3}$$

□

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 ASP-EX program P with respect to a universe $\mathcal{U}$, denoted $I \models_{\mathcal{U}} P$, iff $I \models r$ for all $r \in grnd_{\mathcal{U}}(P)$. A model M is minimal if there is no model N such that $N \subset M$.

Given a general ground program P, its *GL reduct* w.r.t. an interpretation I is the positive ground program P^I, obtained from P by:

- deleting all rules having a negated literal which is not modeled by I;
- deleting all the 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. In the following we will assume the semantics associated to each external predicate is defined a priori, i.e. F is fixed.

4 Properties of ASP-EX programs

Although simple in its definition, the above semantics does not give any hint on how to actually compute 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 indeed infinite.

Theorem 1. *It is given a ASP-EX program P, a domain of constants $\mathcal{U}$, and a function mapping F where the co-domain of F contains only boolean 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. (Sketch) The proof is carried out by showing that the Answer Set Semantics of a ordinary program $\overline{P}$ with function symbols[3] can be reduced to the Answer Set Semantics of a ASP-EX program P. We take advantage of a family of external predicates $\{\#function_i\}$. In a given interpretation $\langle S, F\rangle$, F will be such that $\#function_i(C, f, x_1, \ldots, x_i)$ is modeled if C unifies with the compound term $f(x_1, \ldots, x_i)$.

This allows to rewrite a logic program $\overline{P}$ with function symbols by means of external predicates. For instance, given the rule

$$p(s(X)) \leftarrow a(X, f(Y, h(Z))).$$

This can be rewritten in an equivalent ASP-EX rule:

$$p(S) \leftarrow a(X, F), \quad \#function_1(S, s, X), \quad \#function_2(F, f, Y, H), \\ \#function_1(H, h, Z).$$

□

Tailoring cases where a finite portion of $\mathcal{U}$ is enough to evaluate the semantics of a given program is thus of interest. In the following we reformulate some results regarding splitting sets [Lifschitz and Turner, 1994].

Definition 1. Given a ASP-EX program P, 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 $\{r \mid r \in grnd_{\mathcal{U}}(P)$ and $H(r) \subseteq A\}$. The *residual* $r_{\mathcal{U}}(P, I)$ is a program obtained from $grnd_{\mathcal{U}}(P)$ by deleting all the rules which are not modeled by I, and removing from the remaining rules all the $a \in A$ modeled by I. □

We take advantage here of the formulation of the splitting theorem as given in [Bonatti, 2004].

[3] Positive Horn programs with function symbols are undecidable, see e.g. [Dantsin *et al.*, 2001]

Theorem 2. *(Splitting theorem [Lifschitz and Turner, 1994; Bonatti, 2004]) Given a program P and a splitting set A, $M \in ans_{\mathcal{U}}(P)$ iff M can be split in two disjoint sets I and J, such that $I \in ans_{\mathcal{U}}(b_A(P))$ and $J \in ans_{\mathcal{U}}(r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_A(P)), I)$.*

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

Theorem 3. *Given a safe `ASP-EX` program P, let $U \subset \mathcal{U}$ be the set of constants appearing in P. Then $ans_U(P) = ans_{\mathcal{U}}(P)$.*

Proof. (Sketch) The line of reasoning of the theorem is proving that, assuming P is safe, $grnd_U(P)$ is a finite splitting set for P. Furthermore, $grnd_U(P) = b_U(P)$. For each $M \in ans_U(P)$, we can prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_U(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Viceversa, assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

In case a 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. Intuitively, the semantics of a safe program P can be evaluated by means of the following steps:

- compute $grnd_U(P)$;
- 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.
- compute the remaining ordinary program 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 as same complexity as computing $grnd_U(P)$[4].

5 Dealing with values invention

Although important for clarifying the given semantics, it is an actual practice to specify external sources of computation not in terms of boolean oracles. So we aim at introducing the possibility to specify *functional oracles*, keeping anyway the simple reference semantics given previously. 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.

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 true whenever Y encodes a natural number representing the

[4] 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. [Leone *et al.*, 2001]).

square of the natural number X; we want to extract a series of squared values from this predicate; consider the short program

$$\begin{array}{l} \mathit{number(2)} \leftarrow \\ \mathit{square(Y)} \leftarrow \mathit{number(X)}, \#\mathit{sqr(X,Y)}. \end{array} \tag{4}$$

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 $\{2, 4\}$. It is however undecidable, given a computable boolean oracle f to establish whether a given set S contains all and only all those tuples $\overline{t}$ such that $f(\overline{t}) = 1$.

In order to overcome these limits, 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 instead of boolean ones.

Definition 3. It is given an external predicate name $\#p$, having arity n and its oracle function $F(\#p)$. A *pattern* is a list of b's and u's. A b will represent a placeholder for a constant (or a bounded variable), whereas an u will be a placeholder for a variable. Given a list of terms, the corresponding pattern will be given by replacing each constant with a b, and each variable with a u. □

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 (which we will call input positions), and $n-k$ placeholders of u type (which we will call output positions). A *functional oracle* $F(\#p)[pat]$ for the pattern pat, associated to the external predicate $\#p$, is a partial function taking k constant arguments from $\mathcal{U}$ and returning a tuple of arity $n-k$, and such that $F(\#p)[pat](a_1, ..., a_k) = b_1, ..., b_{n-k}$ iff $F(\#p)(a_1, \ldots, a_k, b_1, \ldots, b_{n-k}) = 1$. Let $pat[j]$ be the j-th element of a pattern pat. Let $unbound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = u$ for each $X_j \in \overline{X}$, and $bound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = b$ for each $X_j \in \overline{X}$.

An external predicate $\#p$ might be associated to one or more functional oracles "consistent" with the originating boolean oracle. For instance, consider the $\#sqr$ external predicate, defined as mentioned above. We associate to it two functional oracles, $F(\#sqr)[b, u]$ and $F(\#sqr)[u, b]$. The two functional oracles are such that, e.g.

$$F(\#sqr)[b, u](3) = 9 \tag{5}$$

$$F(\#sqr)[u, b](16) = 4 \tag{6}$$

consistently with the fact that $F(\#sqr)(3, 9) = F(\#sqr)(4, 16) = 1$, whereas $F(\#sqr)[u, b](5)$ is set as undefined since $F(\#sqr)(X, 5) = 0$ for any natural X.

In the sequel, given an external predicate $\#e$, we will assume it comes equipped with its oracle $F(\#e)$ (called also *base oracle*) and a list of consistent functional oracles $\{F(\#e)[pat_1], \ldots, F(\#e)[pat_m]\}$, having different patterns $pat_1, \ldots, pat_m$[5].

[5] Note that functional oracles prevent, to some extent, to define multivalued functions and/or generic relations. We consider anyway this setting acceptable for a variety of applications.

Adopting functional oracles in the context of safe programs is however to big a restriction. We thus aim at enlarging the class of programs that can be evaluated against a finite Herbrand universe. To this end, we introduce a relaxed notion of safety. Intuitively, a variable is weakly safe if its value, although not explicitly appearing in a program, can be computed through a functional oracle.

Definition 4. Given a rule r, let $E(r)$ its set of external atoms. A *choice C of functional oracles* is a mapping $C : E(r) \mapsto \mathbf{N}$ associating each external atom of r with the index of one of its functional oracles. Given a choice C, let $F_C(\#e)$ a shortcut for the functional oracle $F(\#e)[pat_{C(\#e)}]$.

Given a rule r and a choice C, a variable X is *weakly safe in r w.r.t. to C* if either

- X is safe; or
- X appears in some external atom $\#e(\overline{X}) \in B^+(r)$, $X \in unbound_{pat_{C(\#e)}}$ and each variable $Y \in bound_{pat_{C(\#e)}}$ is weakly safe. □

A rule r is weakly safe if there is a choice C_r such that each variable X appearing in some atom $a \in B(r)$ is weakly safe with respect to C_r. A program P is weakly safe if each rule $r \in P$ is weakly safe. □

Example 2 Assume that $\#sqr$ is associated to the list of functional oracles $\{F(\#sqr)[b, u],\ F(\#sqr)[u, b]\}$ defined above. Given a choice of oracles C such that $C(\#sqr(X, Y)) = 2$, the second rule of Program 4 is not weakly safe (intuitively there is no way for computing the value of the variable Y with the oracle $F(\#sqr)[u, b]$. The same rule is weakly safe if we set $C(\#sqr(X, Y)) = 1$. □

It turns out that deciding whether a given rule is weakly safe or not depends on a given choice, but also from the set of available functional oracles. It is assumed indeed that an external predicate does not come with all its possible functional oracles.

Proposition 1. *Given a set of external predicates $\mathcal{E}$, and a list of functional oracles for each $\#e \in \mathcal{E}$, it can be checked in polynomial time whether a program P is weakly safe.*

Proof. (Sketch) Simply observe that for each rule $r \in P$ it can be checked in time linear in the number of atoms of r whether a choice making r weakly safe exists. □

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

Definition 5. Given a weakly safe rule r, a choice C for it, and a set of ordinary ground atoms A, a ground rule r' is member of $ins(r, A)$ if r can be grounded to r' by the following algorithm:

1. replace positive literals of r with a consistent nondeterministic choice of matching ground atoms from A; let θ the resulting variable substitution;
2. until θ instantiates all the variables of r:

– pick from $r\theta$ an external atom $\#e(\overline{X})\theta$ such that θ instantiates all the variables $X \in bound_{pat_{C(\#e)}}(\overline{X})$.
– If $F_C(\#e)(bound_{pat_{C(\#e)}}(\overline{X}\theta)) = a_1, \ldots, a_k$, then update θ by assigning $a_1, \ldots, a_k$ to $unbound_{pat_{C(\#e)}}(\overline{X}\theta)$; else fail;

3. return $r' = r\theta$. □

Example 3 *Let's consider the second rule of Program 4; then,* $ins(r, \{number(1), number(2)\})$ *contains the two rules:*

$$square(1) \leftarrow number(1), \#sqr(1,1).$$
$$square(4) \leftarrow number(2), \#sqr(2,4).$$

□

Although desirable, weak safety is not sufficient in order to intuitively guarantee finiteness of answer sets and decidability. For instance, the program

$$\begin{array}{l} square(2) \leftarrow \\ square(Y) \leftarrow square(X), \#sqr(X,Y). \end{array} \tag{7}$$

is modeled by the infinite set of atoms $\{square(2), square(4), \ldots\}$.

We thus introduce the notion of *semi-safe* program. Intuitively a semi-safe program is such that external atoms cannot create infinite chains of new values to be taken in account.

Definition 6. A weakly safe program P is *semi-safe* if each cycle in $G(P)$ contains only edges corresponding to safe rules. □

Example 4 *For instance, the program*

$$square(Y) \leftarrow square(X), number(Y), \#sqr(X,Y).$$
$$square(Y) \leftarrow number(X), \#sqr(X,Y).$$

is semi-safe. □

We extend next Theorem 3 to the case of semi-safe programs.

Theorem 4. *It is given a semi-safe program P. Then there is a finite set of constants U such that $ans_U(P) = ans_{\mathcal{U}}(P)$.*

Proof. (Sketch) The set U is defined as all the constant symbols appearing in the set of atoms $T_P^\infty(\emptyset)$ where the operator T_P is defined as follows.

$$T_P(A) = A \cup \{a \in H(r') | r' \in ins(r, A) \text{ for some } r \in P\}$$

It is provable that $T_P^\infty(\emptyset) = T_P^n(\emptyset)$ for some n in case P is semi-safe; $T_P^\infty(\emptyset)$ is a splitting set, and U is finite; as in Theorem 3 for each $M \in ans_U(P)$, we can prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_{T_P^\infty}(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

The above theorem allows to compute semantics of a semi-safe program P by means of a traditional answer set solver, following the steps:

– compute the ground program $T_P^\infty(\emptyset)$. This computation involves a number of evaluation of $ins(r, A)$ that trigger evaluation of functional oracles whenever needed;
– eliminate external literals as in the case of safe programs;
– evaluate the remaining ordinary program by means of a traditional solver;

We observe that, assuming F contains polynomial-time functional oracles, the complexity of the above algorithm is not greater than the complexity of computing grounding for an ordinary program.

6 Implementation and experiments

The proposed language has been integrated into the ASP system `DLV` [Leone *et al.*, 2005b]. We called this prototype `DLV-EX`. From a practical point of view, the external atoms are dealt with in the following steps (see Figure 1):

1. at design time: a developer provides a library of external atoms, each of them associated with a set of functional oracles. 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 in a pre-processing stage: each rule is checked to be weakly safe, and a suitable choice of functional oracles is made. Then the overall program is checked to be semi-safe. It is anyway possible to relax this second condition, provided that termination of grounding algorithms is not guaranteed in this case. It is worth pointing out that an user developing a logic program is not in charge of specifying a choice of oracles, since the system itself will choose the best functional oracles among a variety of possibilities.
3. at run-time during the 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. For each external atom in r,the chosen functional oracles are repeatedly invoked according to Definition 5.

Point 2 and 3 above are integrated in the existing grounding algorithm of the system. We briefly recall the rule instantiation algorithm of the `DLV` system [Leone *et al.*, 2001]. Given a rule r, this algorithm exploits an intelligent backtracking 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: in principle, it is preferred to pick first those atoms whose estimated set of possible values is smaller.

The presence of external atoms impacts within such algorithm in a two-fold way: for what point 2 above is concerned, given a rule r, among possible choices of functional oracles, our algorithm prefers those patterns whose number of unbounded variables is bigger. 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 (b, u) instead of the base oracle (which can be seen has having

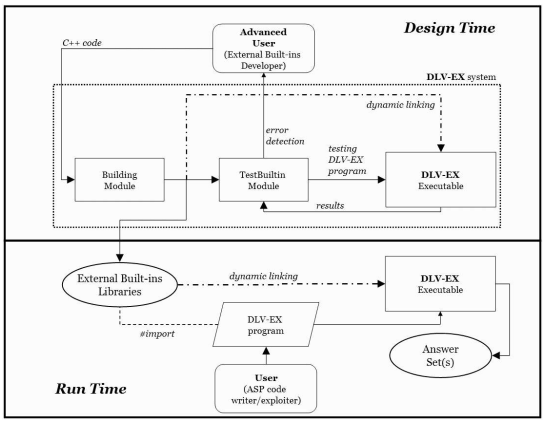

Fig. 1. *System Architecture*

the pattern (b, b)), 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 pattern (b, b) forces in principle to check all the possible couples of values for X and Y.

Point 3 impacts on the atom pick-up ordering strategy. For the same reasons above, it is preferred to pick up external atoms, with pattern having many unbounded variables, as earlier as possible. 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.

All the pre-existing built-in atoms available in the `DLV` system (such as arithmetic and relational operators) have been rewritten using the new general framework. We carried out some experiment in order to appreciate the impact and the possible overhead of the new construct. Results are encouraging: grounding times are in most cases equivalent, and the slowdown reported in few cases is never above 6-7%.

External predicate definitions can be grouped in one or more libraries. Libraries have to be compiled such that they can be dynamically linked to the `DLV-EX` executable; oracles are written in the C++ language. A special directive inside `DLV-EX` programs tells the system which libraries have to be linked at runtime. Also, built-in developers are enabled to redefine predefined operators in order to deal with new data types, e.g. real numbers.

Some usage experiments have been carried out as well; few users have been requested to start implementing some customized libraries [Palopoli *et al.*, 2005; Cumbo *et al.*, 2004], and early feedbacks are positive both from the correctness and the ease of use points of view.

7 Related works

For what the possibility of calling external modules in a logic program is concerned, it is worth to mention the foundational work of Eiter *et al.* [1997]. This paper takes the notion of generalized quantifier, known in formal logics, and adapts it in the context of modular logic programming. A generalized quantifier indeed, can be seen as a way for delegating the truth value of a formula

to an external source of computation. Based on this work, the same authors are addressing the issue of implementing generalized quantifiers under Answer Set Semantics, in order to enable Answer Set Solvers to communicate, in both directions, with external reasoners [Eiter *et al.*, 2004; 2005]. 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 an external atom), whereas, in our setting, the information flow is strictly value based. Nonetheless, HEX programs, as defined in [Eiter *et al.*, 2005], do not deal with infinite domains explicitly.

Although this know-how has not been explicitly divulgated yet, other Answer Set Solvers introduced the possibility to deal with externally computed functions [Syrjänen, 2002; Osorio and Corona, 2003].

Furthermore, there are several works aiming at bringing in Answer Set Programming a restricted capability of dealing with infinite domains. Among these, it is worth citing the notion of ω-restricted programs [Syrjänen, 2001]. ω-restricted programs allow to keep decidability of Answer Set Semantics in the presence of functions symbols, and constitute a subclass of finitary programs. It is indeed important to recall the work of Bonatti [2004], aimed at tailoring the class of finitary programs. Although, in general, recognizing this class of programs is undecidable, finitary programs allow function symbols but are decidable under brave/skeptical reasoning with ground queries. As shown in Theorem 1, external functions might be exploited in order to simulate function symbols. It is a matter of future search to extend the notion of semi-safe program to a larger class and investigate equivalence conditions with the notion of finitary program.

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. Similar in spirit to our approach is the work on open logic programs, and conceptual logic programs [Heymans *et al.*, 2004]. 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 is the work of Cabibbo [1998], which extend the work of Hull and Yoshikawa [1998]. The latter authors introduce a language (ILOG) with a special construct aimed at introducing new invented values in a logic program, for the purpose of creating new tuple identifiers in relational databases. Based on this work, Cabibbo investigates about decidable fragments of the language. Despite some crucial semantic differences, the presented notion of weak safety is similar to the one herein presented, and describes conditions such that new values do not propagate in infinite chains.

8 Conclusions

We presented a framework where external atoms with value invention are taken in account. The purpose of this work is in the direction of closing the gap between Answer Set Programming and practical applications. Also, we believe this works paves the way to an actual implementation of finitary programs with function symbols. The system prototype, examples, manuals and benchmark results are available at `http://www.mat.unical.it/kali/dlv-ex`.

References

[Babovich and Maratea] Y. Babovich and M. Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. `http://www.cs.utexas.edu/users/tag/cmodels.html`, 2003.

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

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

[Cumbo *et al.*] C. Cumbo, S. Iiritano, and P. Rullo. Reasoning-based knowledge extraction for text classification. In *Discovery Science*, pp. 380–387, 2004.

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

[Eiter *et al.*] T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In *LPNMR-97*, LNCS 1265.

[Eiter *et al.*] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artif. Intell.*, 144(1–2):157–211, 2003.

[Eiter *et al.*] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Nonmonotonic description logic programs: Implementation and experiments. In *LPAR 2004*, pp. 511–527, 2004.

[Eiter *et al.*] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *IJCAI 2005*, to appear, Edinburgh, UK, 2005.

[Gelfond and Lifschitz] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP 1988*, pp. 1070–1080, MIT Press.

[Gelfond and Lifschitz] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.

[Heymans *et al.*] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Semantic web reasoning with conceptual logic programs. In *RuleML 2004*, pp. 113–127, 2004.

[Hull and Yoshikawa] R. Hull and M. Yoshikawa. On the equivalence of database restructurings involving object identifiers. In *PODS 1991*, pp. 328–340. ACM Press.

[Leone *et al.*] N. Leone, S. Perri, and F. Scarcello. Improving ASP Instantiators by Join-Ordering Methods. *LPNMR'01, Vienna, Austria, 2001*, LNCS 2173.

[Leone *et al.*] N. Leone, G. Gottlob, R. Rosati, T. Eiter et al. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *SIGMOD 2005*, ACM, to appear.

[Leone *et al.*] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 2005. To appear.

[Lifschitz and Turner] V. Lifschitz and H. Turner. Splitting a Logic Program. In *ICLP'94*, pp. 23–37, MIT Press.

[Lin and Zhao] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI-2002*, AAAI Press / MIT Press.

[Osorio and Corona] M. Osorio and Enrique Corona. The A-Pol system. In *ASP*,2003.

[Palopoli *et al.*] L. Palopoli, S. Rombo, and G. Terracina. Flexible Pattern Discovery with (Extended) Disjunctive Logic Programming. In *International Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, pp. 504–513, Verlag.

[Simons *et al.*] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artif. Intell.*, 138:181–234, 2002.

[Syrjänen] T. Syrjänen. Omega-restricted logic programs. In *LPNMR-6*, Vienna, Austria, 2001. Verlag.

[Syrjänen] T. Syrjänen. Lparse 1.0 User's Manual, 2002.