Exploiting Support Sets for Answer Set Programs with External Evaluations*

Thomas Eiter and **Michael Fink** and **Christoph Redl** and **Daria Stepanova**
Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,redl,dasha}@kr.tuwien.ac.at

Abstract

Answer set programs (ASP) with external evaluations are a declarative means to capture advanced applications. However, their evaluation can be expensive due to external source accesses. In this paper we consider HEX-programs that provide external atoms as a bidirectional interface to external sources and present a novel evaluation method based on *support sets*, which informally are portions of the input to an external atom that will determine its output for any completion of the partial input. Support sets allow one to shortcut the external source access, which can be completely eliminated. This is particularly attractive if a compact representation of suitable support sets is efficiently constructible. We discuss some applications with this property, among them description logic programs over DL-Lite ontologies, and present experimental results showing that support sets can significantly improve efficiency.

1 Introduction

Answer Set Programming (ASP) has been gaining popularity as a tool for declarative problem solving (Brewka, Eiter, and Truszczyński 2011). The need for accessing external information in them has led to HEX-programs (Eiter et al. 2005), which feature external atoms as a generic bidirectional interface to arbitrary sources of computation. Thanks to such atoms (which can be tailored and plugged in by the user), a powerful and flexible formalism for advanced reasoning is available; this has been exploited in many different application areas, including Semantic Web, multi-context systems, or argumentation, to mention a few. A prominent application are DL-programs (Eiter et al. 2008), which amount to HEX-programs with access to an external ontology.

For instance, consider the following DL-program rule

$$cust(X) \leftarrow isIn(X,Y), \\ not\ \mathrm{DL}[worksIn \uplus goTo; \neg Cust](X)$$

taken from a taxi-driver assignment program (see Section 3 for the full example). An external ontology $\mathcal{O}$ is acessed by a DL-atom (i.e., external atom of a particular HEX-program) $a(X) = \mathrm{DL}[worksIn \uplus goTo; \neg Cust](X)$. The latter specifies an update of $\mathcal{O}$, via the operator $\uplus$, prior to querying it: i.e. additional assertions $worksIn(c, c')$ are considered for each pair of individuals (c, c') such that $goTo(c, c')$ is true under the rules part, before all instances X of $\neg Cust$ (i.e., non-customers) are retrieved from the ontology.

The semantics of HEX-programs is given in terms of answer sets, whose computation involves evaluating external atoms. The number of external source accesses for that can be large, especially if cyclic dependencies or nondeterminism (choices) occur in the program, which are typical for the use of ASP in many applications–and a bottleneck of the existing evaluation approach.

To counter this, we present a new evaluation approach based on *support sets*. Intuitively, a support set for an external atom is a portion of its input whose presence settles a concrete output value. For instance, given that $\mathcal{O} \models \exists.worksIn \sqsubseteq \neg Cust$ as in our example, $S = \{\mathbf{T}goTo(d_3, r_1)\}$ is a support set of the DL-atom $a(X)$ above for output $X = d_3$, since truth of $goTo(d_3, r_1)$ guarantees that $a(d_3)$ will evaluate to true. Such support sets can be effectively used to reduce the number of external source accesses in answer set checking, but also to prune the candidate space of answer sets. In particular, if a sufficiently rich collection of support sets is available, the external source access can be *entirely* eliminated. The latter is highly attractive if a suitable representation of such a *complete set of support sets* is efficiently computable, and promises high performance gains.

In developing this approach, we proceed as follows.

- We formalize the notion of *support set* and introduce a non-ground form for compact representation. The latter includes optional source information to increase usability (Section 3).
- We show how to use support sets effectively for optimizing HEX-program evaluation (Section 4).
- As applications, we consider DL-programs over $DL\text{-}Lite_{\mathcal{A}}$ ontologies (Calvanese et al. 2007), a premier formalisms for ontology based data access, for which small representations of complete sets of support sets can be efficiently obtained, and query answering over external ASP programs (Section 5).
- We compare an implementation of the novel against the traditional approach. The results show its effectiveness,

*This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

with significant gains (up to two orders of magnitude; Section 6).

In conclusion, the support sets approach opens a new and fruitful perspective for increasing scalability of HEX-programs, which is crucial for serving applications.

2 Preliminaries

We assume a vocabulary of a set $\mathcal{C}$ of constants, a set $\mathcal{V}$ of variables, a set $\mathcal{P}$ of predicates, and a set $\mathcal{X}$ of external predicates (all pairwise disjoint).

Following Gebser et al. (2012), a *(signed) literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is an atom of form $p(\mathbf{X}) = p(X_1, \ldots, X_\ell)$, with predicate p and terms $X_1, \ldots, X_\ell \in \mathcal{C} \cup \mathcal{V}$; a signed literal or atom is *ground*, if all terms in $\mathbf{X}$ are constants, and *nonground* otherwise.

An *assignment* $\mathbf{A}$ over a (finite) set A of ground atoms is a consistent set of ground signed literals $\mathbf{T}a$ and $\mathbf{F}a$, $a \in A$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that a is false. An *interpretation* is any assignment $\mathbf{A}$ that is *complete*, i.e., no strictly larger assignment $\mathbf{A}' \supset \mathbf{A}$ over A exists.

HEX-Programs: Syntax. HEX-programs generalize (disjunctive) extended logic programs under the answer set semantics (Gelfond and Lifschitz 1991) with *external atoms* $a(\mathbf{Z})$ of the form $\&g[\mathbf{Y}](\mathbf{X})$, where $\&g \in \mathcal{X}$, $\mathbf{Y} = Y_1, \ldots, Y_\ell$, and $\mathbf{X} = X_1, \ldots, X_m$, such that $Y_i, X_j \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$, for $1 \leq i \leq \ell$ and $1 \leq j \leq m$; and $\mathbf{Z}$ is the restriction of $\mathbf{Y}$ and $\mathbf{X}$ to elements from $\mathcal{V}$.

A HEX-*program* (or *program*) consists of rules r of form

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\ , \quad (1)$$

where each a_i is an (ordinary) atom, each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The *head* of r is $H(r) = \{a_1, \ldots, a_k\}$ and the *body* is $B(r) = B^+(r) \cup \text{not } B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*, and $\text{not } S = \{\text{not } b \mid b \in S\}$. For any rule, set of rules, etc. O, let $A(O)$ and $EA(O)$ be the set of all ordinary and external atoms occurring in O, respectively.

A program is *ground*, if it contains no variables. We will also consider non-ground programs in our examples, for which suitable safety conditions allow to use a *grounding procedure* that transforms the program to a variable-free program with the same answer sets. We thus confine our formal investigations here to ground programs.

Semantics. The semantics of a HEX-program is defined via interpretations $\mathbf{A}$ over the *Herbrand base* $HB(\mathcal{P}, \mathcal{C})$ i.e., all atoms constructible from $\mathcal{P}$ and $\mathcal{C}$; throughout the rest, we assume that $A = HB(\mathcal{P}, \mathcal{C})$ is fixed. Ordinary atoms are evaluated w.r.t. an interpretation $\mathbf{A}$ while the value of a ground external atom $\&g[\mathbf{y}](\mathbf{x})$ is given by the value $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x})$ of a $1+|\mathbf{y}|+|\mathbf{x}|$-ary Boolean *oracle function* $f_{\&g}$ (Eiter et al. 2005). Satisfaction of (sets of) literals, rules, etc. O w.r.t. $\mathbf{A}$ ($\mathbf{A}$ is a model of O), denoted $\mathbf{A} \models O$, extends naturally from ordinary (Gelfond and Lifschitz 1991) to HEX-programs as follows. An ordinary atom b is satisfied w.r.t. $\mathbf{A}$ if $\mathbf{T}b \in \mathbf{A}$, an (ordinary or external) literal $\text{not } b$ is satisfied if $\mathbf{A} \not\models b$, and a rule of form (1) is satisfied w.r.t. $\mathbf{A}$ if $\mathbf{A} \models a_i$ for some $1 \leq i \leq k$ or $\mathbf{A} \not\models b_i$ for some $1 \leq i \leq m$ or $\mathbf{A} \models b_i$ for some $m < i \leq n$.

An *answer set* of Π is any model $\mathbf{A}$ of the program $\Pi^{\mathbf{A}} = \{r \in \Pi \mid \mathbf{A} \models B(r)\}$ (the *FLP-reduct* of Π w.r.t. $\mathbf{A}$, cf. (Faber et al. 2011)) whose positive part (i.e., $\{\mathbf{T}a \in \mathbf{A}\}$) is subset-minimal; $\mathcal{AS}(\Pi)$ is the set of all answer sets of Π.

Example 1 (Set Partitioning) *Consider the program* Π

$$\begin{aligned} d(c) \leftarrow; \qquad q(c) &\leftarrow d(c), \&diff[d, p](c) \\ p(c) &\leftarrow d(c), \&diff[d, q](c) \end{aligned}$$

where $\&diff[p, q](X)$ *computes the set of all elements* X *which are in the extension of* p *but not that of* q*. Informally, this program implements a choice from* $p(c)$ *and* $q(c)$*.*

Evaluation. The usual way to compute the answer sets of a HEX-program Π is via a transformation to an ordinary ASP program $\hat{\Pi}$. Each external atom $a = \&g[\mathbf{y}](\mathbf{x})$ in a rule $r \in \Pi$ is replaced by an ordinary *replacement atom* $\hat{a} = e_{\&g[\mathbf{y}]}(\mathbf{x})$ (resulting in a rule $\hat{r}$), and a rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ is added to the program. The answer sets of the resulting *guessing program* $\hat{\Pi}$ are computed by an ASP solver and projected to non-replacement atoms. However, each answer set $\hat{\mathbf{A}}$ of $\hat{\Pi}$ merely gives rise to a *candidate answer set* of Π, as the guess for $e_{\&g[\mathbf{p}]}(\mathbf{c})$ must be checked against the actual value of $\&g[\mathbf{p}]$. If no discrepancy is found, the model candidate is a *compatible set* of Π. More precisely,

Definition 1 (Compatible Set) *A* compatible set *of a program* Π *is an assignment* $\hat{\mathbf{A}}$ *s.t. (i)* $\hat{\mathbf{A}} \in \mathcal{AS}(\hat{\Pi})$ *and (ii) for all* $\&g[\mathbf{y}](\mathbf{x})$ *in* Π*,* $f_{\&g}(\hat{\mathbf{A}}, \mathbf{y}, \mathbf{x}) = 1$ *iff* $\mathbf{T}e_{\&g[\mathbf{y}]}(\mathbf{x}) \in \hat{\mathbf{A}}$*.*

As each answer set of Π is a projection of a (unique) compatible set that fulfills an additional minimality check, computing compatible sets is essential for evaluation (and a focus here).

External Behaviour Learning. To improve efficiency, Eiter et al. (2012) extended the ASP solving approach of Gebser et al. (2012), which uses nogoods in a program representation, to HEX-programs. A *nogood* is a set $\{L_1, \ldots, L_n\}$ of ground signed literals $L_i, 1 \leq i \leq n$ that must not be all true; formally, an interpretation $\mathbf{A}$ is a *solution* to a nogood δ (a set Δ of nogoods), iff $\delta \not\subseteq \mathbf{A}$ ($\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$). Moreover, by external behavior learning, nogoods are learned from evaluations of external atoms during the computation; they are added to $\hat{\Pi}$'s representation to prune model candidates that violate the known semantics of external atoms.

Formally, learned nogoods are specified with a *learning function* Λ for Π, which assigns to each pair $(\&g[\mathbf{y}], \hat{\mathbf{A}})$ of an external predicate with input list $\&g[\mathbf{y}]$ that occurs in Π and assignment $\hat{\mathbf{A}}$ over $A(\hat{\Pi})$ a set $\Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$ of nogoods over $A(\hat{\Pi})$. These nogoods must not eliminate answer sets, which is ensured by the following *correctness* condition.

Definition 2 (Correctness) *Given a program* Π*, a nogood* δ *is* correct *for* Π*, if each compatible set of* Π *is a solution to* δ*; a learning function* Λ *is* correct *for* Π*, iff each* $\delta \in \Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$ *is correct for* Π*, for all possible* $\&g[\mathbf{y}]$ *and* $\hat{\mathbf{A}}$*.*

For a complete procedure of computing answer sets of HEX-programs see Eiter et al. (2013).

$$\mathcal{O} = \left\{ \begin{array}{ll} (1)\ Driver \sqsubseteq \neg Cust & (4)\ worksIn(d1, r1) \\ (2)\ \exists.worksIn \sqsubseteq Driver & (5)\ worksIn(d2, r3) \\ (3)\ EDriver \sqsubseteq Driver & (6)\ EDriver(d2) \end{array} \right\}$$

$$P = \left\{ \begin{array}{l} (7)\ isIn(c2, r3);\ \ (8)\ isIn(d3, r3);\ \ (9)\ isIn(d2, r3); \\ (10)\ needsTo(c2, r4);\ \ (11)\ goTo(d3, r2); \\ (12)\ cust(X) \leftarrow isIn(X, Y), \\ \qquad\qquad not\ \mathrm{DL}[worksIn \uplus goTo; \neg Cust](X); \\ (13)\ driver(X) \leftarrow not\ cust(X), isIn(X, Y); \\ (14)\ drives(X, Y) \leftarrow cust(Y), isIn(Y, Z), isIn(X, Z), \\ \qquad\qquad driver(X), not\ omit(X, Y); \\ (15)\ omit(X, Y) \leftarrow needsTo(Y, Z), drives(X, Y), \\ \qquad\qquad \mathrm{DL}[Driver \uplus driver; EDriver](X), \\ \qquad\qquad not\ \mathrm{DL}[; worksIn](X, Z) \end{array} \right\}$$

Figure 1: DL-program $\Pi = (\mathcal{O}, P)$ over a driver ontology

3 Support Sets

In this section we present the notion of a support set for an external atom, which intuitively encodes a (partial) behavior of its boolean oracle function.

Before defining support sets formally, let us introduce our running example, which has been briefly mentioned in the introduction and is used throughout the paper, in more detail.

Example 2 *Imagine a system that assigns customers to taxi drivers under constraints, using (in a simplistic form) the DL-program $\Pi = (\mathcal{O}, P)$ in Figure 1. The (external) ontology $\mathcal{O}$ has a taxonomy $\mathcal{T}$ in (1)-(3) and a data part $\mathcal{A}$ about drivers and their working regions in (4)-(6). The logic program P has facts about current positions and needs in (7)-(11) and the following rules: (12) and (13) single out customers resp. taxi drivers; (14) assigns taxi drivers to customers in the same region; and (15) forbids drivers of electro-cars to serve needs going outside their working region (see Section 5 for formal details on DL-programs).*

Intuitively, support sets are consistent sets of signed literals that as (part of the) input of an external atom completely determine its output values. Formally, ground support sets are defined as follows:

Definition 3 (Ground Support Set) *Let $a = \&g[\mathbf{y}](\mathbf{x})$ be a ground external atom of a program Π.*

- *A* support set *for a is a consistent set $S \subseteq \{\mathbf{T}p(\mathbf{c}), \mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in HB(\mathcal{P}, \mathcal{C}),\ p \in \mathbf{y}\}$ of ground signed literals s.t. for all assignments $\mathbf{A}$, $\mathbf{A}' \supseteq S$, it holds that $\mathbf{A} \models a$ iff $\mathbf{A}' \models a$.*
- *A support set S is* positive *(resp.* negative*), iff for all assignments $\mathbf{A} \supseteq S$ it holds that $\mathbf{A} \models a$ (resp. $\mathbf{A} \not\models a$).*

We denote by $\mathcal{S}(a)$ the set of all support sets for a; for any $\mathcal{S} \subseteq \mathcal{S}(a)$, we denote by $\mathcal{S}^+$ (resp. $\mathcal{S}^-$) the set of all positive (resp. negative) support sets $S \in \mathcal{S}$ for a.

Example 3 *$S = \{\mathbf{T}goTo(d_3, r_2)\}$ is a ground positive support set for $a(d_3) = \mathrm{DL}[worksIn \uplus goTo; \neg Cust](d_3)$ from Example 2, i.e. $S \in \mathcal{S}^+(a(d_3))$. For $\&diff[d, p](c)$ in Example 1, $\{\mathbf{T}d(c), \mathbf{F}p(c)\}$ is a positive ground support set and $\{\mathbf{T}p(c)\}$ is a negative one.*

Unlike learning functions, the machinery of support sets allows to conveniently encode the full external source behavior through the notion of completeness for a *support family*, i.e., a set of support sets. A complete support family for a ground external atom is a set of its support sets that is sufficient for determining the value of the external atom under all possible assignments. We now state this formally:

Definition 4 (Complete Support Family) *A support family $\mathcal{S} \subseteq \mathcal{S}(a)$ for a ground external atom a is* complete *iff for each assignment $\mathbf{A}$ there exists some $S \in \mathcal{S}$ s.t. $\mathbf{A} \supseteq S$.*

Given a complete support family $\mathcal{S} = \mathcal{S}^+ \cup \mathcal{S}^-$ for a, we can decide on the value of a w.r.t. an assignment $\mathbf{A}$ by simply checking if some $S \in \mathcal{S}$ "covers" $\mathbf{A}$, i.e., $S \subseteq \mathbf{A}$ holds. In fact, to decide this we need just one of $\mathcal{S}^+$ and $\mathcal{S}^-$; for space reasons, storing the smaller set is preferable. A support family is *$+$-complete* (resp. *$-$-complete*) for a, if it equals $\mathcal{S}^+$ (resp. $\mathcal{S}^-$) for some complete support family $\mathcal{S} \subseteq \mathcal{S}(a)$.

Different ground support sets might have similar structure. The atom $a(d_3)$ from Example 3 has $S = \{\mathbf{T}goTo(d_3, c)\}$ as support set, for each $c \in \mathcal{C}$. Moreover, $S = \{\mathbf{T}goTo(d, c)\}$ is a support set for any $a(d)$, $c, d \in \mathcal{C}$. This suggests that a nonground representation of support sets is desirable.

Given a set of nonground signed literals, in general, one can not decide on the value of the external source without any knowledge about the latter. Thus nonground support sets are useful only if they work on a conditional basis and take source information into account. In our framework this is served by so-called *guards* (γ), which (unlike the existing machinery of learning functions) allow to elegantly specify the support sets on a nonground level as follows:

Definition 5 (Nonground Support Set) *Let Π be a program and let $a(\mathbf{Z}) = \&g[\mathbf{Y}](\mathbf{X})$ be an external atom of Π. A* positive (negative) nonground support set *S for a is a pair $\langle N, \gamma \rangle$, where*

- *$N \subseteq \{\mathbf{T}p_i(X), \mathbf{F}p_i(X) \mid p_i \in \mathbf{Y} \cap \mathcal{P}\}$ is a set of nonground signed literals over predicates p_i;*
- *$\gamma : \mathcal{C}^{|\mathbf{Z}|} \times grnd_{\mathcal{C}}(N) \to \{0, 1\}$ is a Boolean function (called the* guard*), s.t. for all $\mathbf{z} \in \mathcal{C}^{|\mathbf{Z}|}$ and $N_{gr} \in grnd_{\mathcal{C}}(N)$ it holds that $\gamma(\mathbf{z}, N_{gr}) = 1$ only if N_{gr} is a positive (negative) ground support set for $a(\mathbf{z})$.*

Here, $grnd_{\mathcal{C}}(N)$ is the support family constructed from N by replacing all variables with constants from $\mathcal{C}$ in all ways. The guard γ selects instances from $grnd_{\mathcal{C}}(N)$ which are ground support sets for an external atom, i.e. every positive input $(\mathbf{z}, N_{gr})$ of γ yields a ground support set N_{gr} for $a(\mathbf{z})$.

In principle, the technical toolkit of guards is flexible and elaborate conditions can be defined using arbitrary functions as guards. However, clearly nonground support sets are only useful if their ground instances can be easily constructed. This should be taken into account when nonground support sets for an external atom are defined.

Frequently, an external source may be abstractly viewed as a data part with an algorithm on top; e.g. an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, has the ABox $\mathcal{A}$ as data part and a query answering algorithm (which respects also $\mathcal{T}$) on top. Here γ might check conditions on the data part, without invoking the algorithm.

Example 4 *For $a(X)$ in Example 3, the pair $S = \langle \emptyset, \gamma \rangle$, where $\gamma : \mathcal{C} \times \{\emptyset\} \to \{0,1\}$ is such that $\gamma(c, \emptyset) = 1$ iff $EDriver(c) \in \mathcal{A}$, is a nonground support set. Indeed, since $\mathcal{T} \cup \{EDriver(c)\} \models \neg Cust(c)$ for every $c \in \mathcal{C}$, any c such that $\gamma(c) = 1$ will be a's output w.r.t. any interpretation $\mathbf{A}$.*

$S = \langle \{\mathbf{T}goTo(X, X')\}, \top \rangle$ is another nonground support set for $a(X)$, where the guard $\top$ returns 1 for each grounding of $\mathbf{T}goTo(X, X')$; hence $\{\mathbf{T}goTo(c, d)\}$ is a ground support set of $a(c)$, for all $c, d \in \mathcal{C}$.

In the above example the guard γ checks whether a certain assertion occurs in the data part of an ontology. Due to guards, the notion of completeness for a set of ground support sets becomes strongly application dependent when lifted to the nonground case. Formally, it amounts to the following.

Definition 6 (Nonground Completeness) *A family $\mathcal{S}$ of nonground support sets for an external atom $a(\mathbf{Z})$ is* complete*, if for all $\mathbf{z} \in \mathcal{C}^{|\mathbf{Z}|}$, $\{N_{gr} \in grnd_{\mathcal{C}}(N) \mid \gamma(\mathbf{z}, N_{gr}){=}1\}$ is a complete support family for $a(\mathbf{z})$.*

Note that if $\mathcal{S}$ is complete, we may similarly as in the ground case only store $\mathcal{S}^+$ resp. $\mathcal{S}^-$ (defining them as usual).

4 Using Support Sets

In this section, we present two ways of using support sets to optimize answer set computation, and we briefly discuss support set generation.

Using Support Sets in the Search. Clearly, every $S \in \mathcal{S}^+(a)$ (resp. $S \in \mathcal{S}^-(a)$) yields a nogood $\delta = S \cup \{\mathbf{F}e_a\}$ (resp. $\delta = S \cup \{\mathbf{T}e_a\}$). Adding such nogoods to the nogood representation of $\hat{\Pi}$ eliminates interpretations that are known to fail the compatibility check. In this way, support sets can be fruitfully used to prune the candidate search space.

Proposition 1 *Let a be a ground external atom of Π. For every ground support set $S \in \mathcal{S}^+(a)$ (resp., $S \in \mathcal{S}^-(a)$), the nogood $S \cup \{\mathbf{F}e_a\}$ (resp. $S \cup \{\mathbf{T}e_a\}$) is correct for Π.*

Proof *We prove the claim for $S \in \mathcal{S}^+(a)$ (for $S \in \mathcal{S}^-(a)$ the proof is analogous). Since $S \in \mathcal{S}^+(a)$, it holds that $\mathbf{A} \models a$ for all $\mathbf{A} \supseteq S$. Towards a contradiction, suppose that $\delta = S \cup \{\mathbf{F}e_a\}$ is not correct for Π. Hence some compatible set $\hat{\mathbf{A}}' \supseteq S$ exists which is not a solution to δ, i.e. $\delta \subseteq \hat{\mathbf{A}}'$. As $\mathbf{F}e_a \in \delta$, by compatibility of $\hat{\mathbf{A}}'$, we conclude $\mathbf{A}' \not\models a$. Since $\mathbf{A}' \supseteq S$, this contradicts that $S \in \mathcal{S}^+(a)$.* □

Intuitively, the above property shows that we can safely eliminate answer sets of $\hat{\Pi}$ that contain nogoods generated from support sets; this does not prune any compatible set.

Example 5 *The support set $\{\mathbf{T}goTo(d_3, r_2)\} \in \mathcal{S}^+(a(d_3))$ for $a(d_3)$ from Example 3 yields a correct nogood $\delta = \{\mathbf{T}goTo(d_3, r_2), \mathbf{F}e_a(d_3)\}$. All answer sets $\hat{\mathbf{A}} \supseteq \delta$ of $\hat{\Pi}$ can be neglected without loss of compatible sets.*

Using Support Sets for Compatibility Checking. Given a complete support family, external source accesses can be avoided, which promises significant performance improvements. This beneficial property is now formally stated.

Proposition 2 *Let $\mathcal{S} \subseteq \mathcal{S}(a)$ be a complete (ground) support family for a ground external atom a of Π, let Δ be the set of nogoods constructed from $\mathcal{S}$, and let $\hat{\mathbf{A}} \in \mathcal{AS}(\hat{\Pi})$ be a solution to Δ. If $\mathbf{T}e_a \in \hat{\mathbf{A}}$ (resp. $\mathbf{F}e_a \in \hat{\mathbf{A}}$) then $\mathbf{A} \models a$ (resp. $\mathbf{A} \not\models a$).*

Proof *Assume $\hat{\mathbf{A}}' \in AS(\hat{\Pi})$ is a solution to Δ. Towards a contradiction, suppose:*

(1) $\mathbf{T}e_a \in \hat{\mathbf{A}}'$ and $\mathbf{A}' \not\models a$. Since $\mathcal{S}(a)$ is complete there exists a negative support set $S \in \mathcal{S}(a)$ such that $S \subseteq \mathbf{A}'$. The nogood $\delta \in \Delta$ constructed from S is of the form $S \cup \{\mathbf{T}e_a\}$. However, then it holds that $\delta \subseteq \hat{\mathbf{A}}'$ which contradicts $\hat{\mathbf{A}}'$ being a solution to Δ.

(2) $\mathbf{F}e_a \in \hat{\mathbf{A}}'$ and $\mathbf{A}' \models a$. This case is analogous. □

As a consequence of Propositions 1 and 2, considering all external atoms of a program rather than a single one, we get the following property of support sets.

Corollary 1 *Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be complete (ground) support families for all external atoms in Π, and let Δ be the set of nogoods constructed from $\mathcal{S}_1, \ldots, \mathcal{S}_n$. Then $\hat{\mathbf{A}} \in \mathcal{AS}(\hat{\Pi})$ is a compatible set for Π iff it is a solution to Δ.*

By exploiting Corollary 1, compatibility of answer sets $\hat{\mathbf{A}}$ of $\hat{\Pi}$ can be checked by testing whether each external atom guessed to be *true* (resp. *false*) is covered by some positive (negative) support set in a collection $\mathcal{S}$ of support sets. Given completeness of $\mathcal{S}$, only the positive (negative) nogoods of each external atom must be stored. This is in contrast to the approach of Eiter et al. (2012), which evaluates external atoms explicitly for verification of guesses.

Example 6 *Consider the ground DL-program $\Pi = \langle \mathcal{O}, P \rangle$, where $\mathcal{O}$ is as in Example 2 and the rule part P is as follows:*

$$isIn(d3, r3); \quad goTo(d3, r2);$$
$$cust(d3) \leftarrow isIn(d3, r2), \; not\ \mathrm{DL}[worksIn \uplus goTo; \neg Cust](d3).$$

For the DL-atom $a(d3) = \mathrm{DL}[worksIn \uplus goTo; \neg Cust](d3)$ the support family $\mathcal{S}(a(d3)) = \{\{\mathbf{T}goTo(d3, c)\}\}$, where $c \in \mathcal{C}$, is complete. Due to the fact there are no other DL-atoms in Π, the compatibility check of $\hat{\mathbf{A}} = \{\mathbf{T}isIn(d3, r3), \mathbf{T}goTo(d3, r2), \mathbf{T}e_a(d3)\}$ amounts to testing whether $\hat{\mathbf{A}}$ is a solution to the set Δ of nogoods constructed from $\mathcal{S}(a(d3))$. Therefore, $\hat{\mathbf{A}}$ is compatible, while $\hat{\mathbf{A}}' = \{\mathbf{T}isIn(d3, r3), \mathbf{T}goTo(d3, r2), \mathbf{F}e_a(d3)\}$ is not.

Support Set Construction. Clearly, support sets are only useful if there are procedures that effectively (and efficiently) construct them. For some applications, such procedures are already in place; e.g., for DL-programs over $DL\text{-}Lite_{\mathcal{A}}$ ontologies, $+$-complete support families can be efficiently obtained (in polynomial time), using algorithms for query answering. We consider this and other practically relevant applications with this property in the next section.

In general, the designer of an external atom may be aware of its semantic structure, and make this knowledge fruitfully available for the evaluation through support sets. In fact, for whole classes of external atoms (thus whole classes of HEX-programs) complete families of support sets can be efficiently obtained and automatically computed.

Another important aspect is that also partial (incomplete) families of support sets, in settings where it is difficult or computationally infeasible to use complete families, allow us to optimize external atom evaluation, as a code call is only needed if no support set applies. Clearly, choosing good partial support families is nontrivial, and many issues (e.g. a Pareto principle respectively power law for calls and resulting savings) may be investigated. However, the whole issue goes beyond this paper.

5 Some Applications

DL-programs. A well-known application of HEX-programs is DL-programs, which we informally considered in Example 2. They have a native, user-friendly syntax for external atoms (called DL-atoms) which is translated into HEX-syntax. Before discussing support sets of DL-atoms, we describe the translation.

A DL-program is a pair $\Pi = \langle \mathcal{O}, P \rangle$, where $\mathcal{O}$ is an ontology fixed as an external source and P is a set of rules allowing DL-atoms in the body. $\mathcal{O}$ is assumed to be consistent (otherwise each DL-atom is trivially true).

DL-atoms are of the form $\&DL[c^+, c^-, r^+, r^-, Q](\mathbf{x})$, where c^+, c^- (r^+, r^-) are binary (resp. ternary) predicates and Q is a string which encodes an ontology query. In this exposition we consider only instance queries, i.e. Q is a possibly negated ontology concept or a role name; other types of queries (e.g. subsumption, conjunctive queries) are possible in general. The oracle function of $\&DL$ is defined by

$$f_{\&DL}(\mathbf{A}, c^+, c^-, r^+, r^-, \mathbf{x}) = 1 \iff \mathcal{O} \cup U^{\mathbf{A}}(c^+, c^-, r^+, r^-) \models Q(\mathbf{x}),$$

where $U^{\mathbf{A}}(\cdots)$ is an update to $\mathcal{O}$, specified by the (extension of the) predicates c^+, c^-, r^+, r^-. More specifically, it contains for each $\mathbf{T}c^+(\text{“}C\text{”}, a) \in \mathbf{A}$ (resp. $\mathbf{T}c^-(\text{“}C\text{”}, a) \in \mathbf{A}$), a concept assertion $C(a)$ (resp. $\neg C(a)$). Updates of roles, generated by the predicates r^+ and r^- are analogous.

Example 7 *The DL-atom $DL[; worksIn](X)$ from Example 2 is translated to $\&DL[c^+, c^-, r^+, r^-, worksIn](X)$, s.t. none of the input predicates occurs anywhere else in P.*

The DL-atom $a = DL[worksIn \uplus goTo; \neg Cust](X)$ is translated to $\&DL[c^+, c^-, r^+, r^-, \text{“}\neg Cust\text{”}](X)$ by adding a rule $r^+(\text{“}worksIn\text{”}, X, Y) \leftarrow goTo(X, Y)$ to P. For $\mathbf{A} \supseteq \{\mathbf{T}r^+(\text{“}worksIn\text{”}, d3, r2)\}$ we get $\mathbf{A} \models a(d_3)$.

Informally, a DL-atom evaluates to true under $\mathbf{A}$ if either its query is entailed from a consistent updated ontology, or if the update makes the ontology inconsistent.

This observation together with the results by Calvanese et al. (2007), on the properties of $DL\text{-}Lite_{\mathcal{A}}$ yield the following characterization of ground support sets for DL-atoms accessing an $DL\text{-}Lite_{\mathcal{A}}$ ontology.

Proposition 3 *Let $a = \&DL[c^+, c^-, r^+, r^-, Q](\mathbf{x})$ be a ground DL-atom over a consistent $DL\text{-}Lite_{\mathcal{A}}$ ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, and let $\mathcal{S}$ be a complete (ground) support family for a. Then each $S \in \mathcal{S}^+$ is either $\emptyset$ or has one of the following forms ($P^{(\neg)}$ is P if $p \in \{c^+, r^+\}$, and is $\neg P$ otherwise):*

(1) $S \supseteq \{p(\text{“}P\text{”}, \mathbf{x})\}$, such that $P^{(\neg)}(\mathbf{x}) \cup \mathcal{T}$ is consistent and $P^{(\neg)}(\mathbf{x}) \cup \mathcal{T} \models Q(\mathbf{x})$;

(2) $S \supseteq \{p(\text{“}P\text{”}, \mathbf{x}')\}$ s.t. $P^{(\neg)}(\mathbf{x}') \cup \mathcal{T}$ is inconsistent;

(3) $S \supseteq \{p(\text{“}P\text{”}, \mathbf{c}'), q(\text{“}P'\text{”}, \mathbf{c}')\}$, such that $P^{(\neg)}(\mathbf{c}') \cup P'^{(\neg)}(\mathbf{c}') \cup \mathcal{T}$ is inconsistent, $p, q \in \{c^+, c^-, r^+, r^-\}$.

By results of Calvanese et al. (2007), at most one assertion α is needed to derive an instance query from a consistent $\mathcal{O}$. If α is in the update, then the support set encoding this knowledge is of form (1); $S = \emptyset$ encodes the case if $\alpha \in \mathcal{A}$. At most two ABox assertions are needed to make an $DL\text{-}Lite_{\mathcal{A}}$ ontology inconsistent. Given that $\mathcal{O}$ is consistent, we get support sets of forms (2) and (3).

Positive support sets for DL-atoms that query a $DL\text{-}Lite_{\mathcal{A}}$ ontology can not only be restricted to small size but are also easy to generate. One can construct suitable support sets by syntactic analysis of the (extensions of) DL-atoms input predicates and exploit TBox classification, which is efficiently doable for $DL\text{-}Lite_{\mathcal{A}}$; we have implemented this.

Query Answering Over Positive ASP Programs. As second application, we consider query answering over external positive ASP programs, using external atoms of form $\&query_n[c_\Pi, p, q](\mathbf{x})$. Here c_Π is a constant representing a positive ASP program Π (as a string or giving a filename), p is predicate providing input to Π, and q is the n-ary query predicate. We have that $f_{\&query_n}(\mathbf{A}, c_\Pi, p, q, \mathbf{x}) = 1$ iff $q(\mathbf{x})$ is true in the unique answer set of $\Pi \cup \{p(\mathbf{y}) \leftarrow \mid \mathbf{T}p(\mathbf{y}) \in \mathbf{A}\}$.

Example 8 *Suppose an external program Π_q checks whether a color assignment is* not *a valid 3-coloring of a graph G stored by facts Π_G over $node(X)$ and $edge(X, Y)$. If $\&query_0$ receives the edges by facts $inp(edge, x, y)$ and the color assignment by facts $inp(col, x, c)$, then $\Pi_q = \{inv \leftarrow inp(col, X, C), inp(col, Y, C), inp(edge, X, Y)\}$ derives inv iff the coloring is invalid. Any non 3-colorable G is then recognized in an answer set of the following program $\Pi = \Pi_G \cup \Pi_{col}$, where*

$$\Pi_{col} = \left\{ \begin{array}{r} col(V, r) \vee col(V, g) \vee col(V, b) \leftarrow node(V), \\ inp(p, X, Y) \leftarrow p(X, Y) \mid p \in \{col, edge\}, \\ inval \leftarrow \&query_0[c_{\Pi_q}, inp, inv](), \\ col(V, c) \leftarrow inval, node(V) \mid c \in \{r, g, b\} \end{array} \right\}$$

Intuitively, a guess for a coloring is checked using the external atom, which spoils the guess by assigning all colors to all nodes if it is invalid. If G is not 3-colorable, Π has a single answer set $\mathbf{A}_0$, which contains $\mathbf{T}inval$; otherwise, each valid 3-coloring induces an answer set $\mathbf{A}$ such that $\mathbf{F}inval \in \mathbf{A}$. Thus, Π makes $inval$ true iff G is not 3-colorable.

The basic structure of Π is reusable for related problems, resorting to appropriate checking programs Π_q; this gives rise to a *class of applications*, which we formalize next.

A *positive ASP program* Π_q is a set of rules of form (1) where $k = 1$ and $m = n$. For such a program, the clause set $cl(\Pi_q) = \{\{\mathbf{F}H_1(r)\} \cup \{\mathbf{T}b \mid b \in B(r)\} \mid r \in \Pi_q\}$ reflects the classical semantics of its rules. For two clauses C_1 and C_2 which share no variables, its *set of resolvents* is $res(C_1, C_2) = \{(C_1 \cup \rho(C_2)) \setminus \{\mathbf{T}x, \mathbf{F}\rho(x')\} \mid \mathbf{T}x \in C_1, \mathbf{F}x' \in C_2, x \sim_\rho x'\}$, where $\sim_\rho$ denotes unification of atoms x and x' using the most general unifier ρ: $\mathcal{V} \to \mathcal{V} \cup \mathcal{C}$; x is the *resolved atom*. For a set Γ of clauses, $res^\infty(\Gamma)$

denotes its (finite) closure under res, with clauses which are equivalent up to variable renaming being eliminated.

Definition 7 (Query Support) *Let Π_q be a positive ASP program and $a = \&query_n[c_{\Pi_q}, p, q](\mathbf{x})$ occur in program Π. Then $QS(a) = grnd_{\mathcal{C}}(\{C \subseteq (\{\mathbf{T}p(\mathbf{x}) \mid p(\mathbf{x}) \in A(\Pi)\} \mid C \cup \{\mathbf{F}q(\mathbf{x})\} \in res^{\infty}(cl(\Pi_q))\})$.*

This definition is motivated by the following idea. By soundness and completeness of the resolution calculus, $res^{\infty}(cl(\Pi_q))$ contains all and only clauses that are implied by the original clause set. Thus, the set materializes the implications encoded by the rules of the program. However, since support sets need to represent only the behavior of the external atom, it is sufficient to select those elements from $res^{\infty}(cl(\Pi_q))$ that relate an atom $q(\mathbf{x})$ over the query predicate q to the input atoms of $\&query_n$, i.e., atoms from $A(\Pi)$. Informally, each $S \in QS(a)$ is an instance of an unfolded rule deriving $q(\mathbf{x})$, projected to input facts. As one can show:

Proposition 4 *Let Π_q be a positive ASP program and $a = \&query_n[\Pi_q, p, q](\mathbf{x})$ occur in a program Π. Then $QS(a)$ is $+$-complete for a.*

Clearly, non-minimal sets can be pruned from $QS(a)$, and lifting $QS(a)$ to non-ground support sets is straightforward (but we omit it for simplicity). While in general deciding $S \in QS(a)$ is intractable and $QS(a)$ may be large, in practical settings (e.g. usage of small acyclic Π_q with low-arity predicates) $QS(a)$ may be fruitfully employed.

6 Implementation and Experiments

Both the traditional and the new algorithm are implemented in the DLVHEX system version 2.3.0, where a command-line switch allows to select the algorithm. The system is based on GRINGO and CLASP for either selection. External sources are supposed to provide a complete set of ground support sets (optionally, its positive resp. negative part), possibly in nonground form. Support sets are grounded before program evaluation, added in candidate search to the solver as nogoods (cf. Prop. 1), and used in compatibility checking.

Experimental Setup. We compared the novel approach against native HEX-program evaluation in a number of experiments[1]. They were run on a Linux server with two 12-core AMD 6176 SE CPUs/128GB RAM using a timeout of 300 secs per run. The results show clear benefits of the new approach.

Non 3-Colorability. In addition to the program Π in Example 8 for non 3-colorability, we consider one for *non checkered 3-colorings*, in which each pair of different colors must appear at adjacent vertices. To this end, we add to Π facts $inp(pair, c_1, c_2)$, $c_1 \neq c_2 \in \{r, g, b\}$, a rule

$$inv \leftarrow inp(pair, C_1, C_2), \\ \text{not } \&query_2[c_{\Pi_q}, inp, app](C_1, C_2),$$

and to Π_q a rule

$$app \leftarrow inp(pair, C_1, C_2), inp(col, X, C_1), \\ inp(col, Y, C_2), inp(edge, X, Y).$$

[1]Program instances used in the experiments are available at http://www.kr.tuwien.ac.at/research/projects/hexhex/supportsets/.

#nodes	Ordinary Graph Coloring			Complete Graph Coloring		
	-Sup.	+Sup.	ASP	-Sup.	+Sup.	ASP
5	59.73 (18)	0.05 (0)	0.04 (0)	116.20 (33)	0.09 (0)	0.04 (0)
10	256.48 (85)	0.07 (0)	0.05 (0)	268.35 (87)	0.12 (0)	0.05 (0)
15	289.17 (96)	0.10 (0)	0.05 (0)	294.76 (96)	0.17 (0)	0.05 (0)
20	300.00 (100)	0.13 (0)	0.06 (0)	300.00 (100)	0.24 (0)	0.06 (0)
40	300.00 (100)	0.46 (0)	0.10 (0)	300.00 (100)	0.67 (0)	0.13 (0)
60	300.00 (100)	1.35 (0)	0.18 (0)	300.00 (100)	1.72 (0)	0.24 (0)
80	300.00 (100)	3.30 (0)	0.29 (0)	300.00 (100)	3.86 (0)	0.38 (0)
100	300.00 (100)	7.12 (0)	0.43 (0)	300.00 (100)	8.03 (0)	0.58 (0)

Table 1: Graph Coloring Benchmark Results

n	$m = 10$			$m = 100$			$m = \infty$		
	-Sup.	+Sup.	ASP	-Sup.	+Sup.	ASP	-Sup.	+Sup.	ASP
5	0.66	0.32	0.05	8.67 (0)	0.45	0.07	108.55 (16)	3.89 (0)	0.39 (0)
6	0.92	0.55	0.07	13.37 (0)	0.71	0.09	236.29 (77)	10.04 (0)	0.87 (0)
7	1.28	0.85	0.07	14.74 (0)	1.05	0.09	244.01 (78)	28.53 (5)	3.80 (0)
8	1.93	1.47	0.08	21.60 (0)	1.70	0.10	269.70 (86)	62.10 (10)	14.76 (2)
9	2.56	2.30	0.08	31.04 (0)	2.56	0.10	277.36 (85)	94.20 (12)	27.44 (6)
10	3.39	3.21	0.10	45.16 (0)	3.50	0.12	295.44 (88)	114.47 (23)	31.85 (6)
11	3.93	4.88	0.12	49.04 (0)	5.21	0.14	300.00 (100)	134.21 (25)	64.69 (19)
12	5.23	7.78	0.17	65.70 (1)	8.15	0.20	300.00 (100)	193.03 (57)	103.96 (31)
13	8.57	8.98	0.16	81.83 (0)	9.35	0.18	300.00 (100)	161.61 (41)	77.68 (22)
14	6.60	11.94	0.16	99.22 (2)	12.31	0.19	300.00 (100)	208.21 (57)	103.80 (28)
15	19.70	17.67	0.27	133.76 (10)	18.20	0.30	300.00 (100)	211.30 (67)	128.18 (36)

Table 2: House Configuration Problem

The average runtimes on 100 randomly generated instances for graphs with n nodes are shown in Table 1, where *–Sup* is traditional and *+Sup* support set evaluation; numbers in parentheses are timeouts. The gain is obvious and increases scalability drastically. As expected, a native ASP encoding (*ASP*; no external source) is faster, but the gap is not huge.

House Problem. The *house problem* is an abstraction of configuration problems (Mayer et al. 2009). *Objects* owned by *persons* must be placed in *cabinets* located in *rooms* of a house. Instances consist of sets of persons, objects, cabinets and rooms, and an assignment of objects to persons. A solution assigns cabinets to persons, cabinets to rooms, and objects to cabinets, under the following constraints. Each room has $\leq$4 cabinets and each cabinet can store $\leq$5 objects. Objects of a person must be stored in her cabinets, and a room must contain cabinets of a single person. These constraints are expressible by positive ASP rules (with $\neq$). We consider a variant of the problem in which a partial assignment of objects to cabinets and cabinets to rooms must be completed; this is relevant e.g. for a shared flat if new persons move in. We use an ASP encoding[2] and outsource the checking part, obtaining a query answering problem as in Section 5.

Instances of size n have n persons, n+2 cabinets, n+1 rooms, and $2n$ objects randomly assigned to the persons; $2n-2$ objects are already stored. Table 2 shows the average runtime for 100 randomly generated instances per n; m bounds the number of answer sets to compute. This bench-

[2]Available at http://143.205.174.183/reconcile/tools.

p	all answer sets		first answer set	
	-Sup.	+Sup.	-Sup.	+Sup.
5 (100)	9.07 (1)	0.08 (0)	8.54 (1)	0.08 (0)
10 (100)	34.01 (6)	0.09 (0)	31.65 (6)	0.09 (0)
15 (100)	295.16 (98)	0.16 (0)	294.78 (97)	0.14 (0)
20 (100)	297.42 (99)	0.17 (0)	297.38 (99)	0.15 (0)
25 (100)	300.00 (100)	0.34 (0)	300.00 (100)	0.24 (0)
35 (100)	300.00 (100)	0.23 (0)	300.00 (100)	0.21 (0)
50 (100)	300.00 (100)	0.28 (0)	300.00 (100)	0.28 (0)
65 (100)	300.00 (100)	0.36 (0)	300.00 (100)	0.36 (0)
80 (100)	300.00 (100)	0.47 (0)	300.00 (100)	0.48 (0)
9 (100)	300.00 (100)	1.70 (0)	300.00 (100)	1.71 (0)
12 (100)	300.00 (100)	2.31 (0)	300.00 (100)	2.32 (0)
15 (100)	300.00 (100)	2.91 (0)	300.00 (100)	2.91 (0)

Table 3: Driver - Customer Assignment Problem

mark needs more support sets than non 3-colorability, which causes a higher initialization overhead and may be counterproductive if only few answer sets are computed. However, as the number of answer sets to be computed increases, there is again a significant gain and the gap to native ASP gets smaller.

Taxi Assignment. For the DL-program in Example 2, we fixed the ABox $\mathcal{A}$ of the ontology $\mathcal{O}$ to 50 customers, 20 drivers (among them 4 driving electro-cars), and 5 regions; every driver works in 2-4 regions. In the program P, facts $isIn(c, r)$, $needsTo(c, r)$, $goTo(d, r)$ for appropriate constants c, d, r from $\mathcal{A}$ are randomly added with probability $p/100$ under the following constraints: persons are in at most one region; customers need to go to at most one region, and their position is known in that case. Furthermore, drivers positions are added as facts $isIn(d, r)$ with fixed probabilities of 0.3, 0.7 and 1 growing discretely in accordance with p. The results are in the upper part of Table 3, where the first column shows in parentheses the number of instances generated per p. One can see dramatic improvements, even for small instances. Results for a larger ABox (500 customers and 200 drivers, including 40 driving electro-cars), shown in the bottom of Table 3, are similarly satisfactory: while the standard approach always times out, support sets scale well.

LUBM Diamond. Finally, we consider default reasoning over the famous LUBM ontology[3] in its $DL\text{-}Lite_{\mathcal{A}}$ form.The defaults–expressed in a simple DL-program–state that research assistants (RAs) are normally employees, while students are normally not employees; as the ontology entails that RAs are students, this instantiates the well-known Nixon diamond. Table 4 shows the results for a randomly generated ABox over 50 individuals. Here p is the percentile of the relevant domain (RAs, students, employees entailed by $\mathcal{O}$), which is generated as a set of facts in the program. Unsurprisingly, the combinatorial nature of the defaults, which is difficult to grasp for "blackbox" external atoms, affects scalability, but the support sets approach is clearly better; combined with domain independence properties (Eiter et al. 2009), it might be even further reduced drastically.

[3] http://swat.cse.lehigh.edu/projects/lubm/

p	all answer sets		first answer set	
	-Sup.	+Sup.	-Sup.	+Sup.
5 (100)	0.21 (0)	0.22 (0)	0.18 (0)	0.21 (0)
9 (100)	9.15 (2)	0.64 (0)	0.42 (0)	0.29 (0)
13 (100)	16.22 (2)	1.73 (0)	0.95 (0)	0.60 (0)
17 (100)	118.99 (32)	16.39 (2)	14.04 (1)	6.35 (1)
21 (100)	186.37 (53)	49.64 (10)	48.75 (11)	29.51 (7)
25 (100)	236.22 (66)	96.38 (22)	68.27 (13)	41.54 (7)
29 (100)	268.58 (85)	149.93 (37)	130.00 (34)	88.02 (21)
33 (100)	298.66 (99)	242.99 (68)	212.22 (62)	135.34 (36)
37 (100)	295.71 (98)	262.68 (81)	243.93 (75)	177.97 (49)

Table 4: Default Rules over LUBM in $DL\text{-}Lite_{\mathcal{A}}$

7 Conclusion

We introduced support sets as a means for optimizing HEX-program evaluation. While ground support sets are closely related to nogoods, our work is innovative in two aspects: 1. we exploit completeness properties to fully avoid external source access, and 2. we lift our approach to non-ground support sets that can include optional source information; to the best of our knowledge, this has no analogs in the literature. Applications such as DL-programs over $DL\text{-}Lite_{\mathcal{A}}$ and query answering over ASP, witness an effective exploitation with significant performance gains in experiments.

Support sets can be viewed as a form of knowledge compilation (Darwiche and Marquis 2002). They are loosely related to support clauses in clause management systems (Reiter and de Kleer 1987). While support clauses are geared towards deciding truth of a literal in a propositional knowledge base, our support sets are more general; they serve to answer abstract queries over arbitrary external sources. Drescher and Walsh (2012) characterized in constraint ASP external propagation using nogoods; this can be viewed as a special case of our approach, where external sources are constraint stores.

As for implementation, there is no comparable system apart from DLVHEX; there is room for further improvement, e.g. to develop sophisticated algorithms for nogood grounding and coverage checking. On the theoretical side, support sets may use information about a program at hand, such that not all assignments need to be considered; respective performance gains are traded for reusability, however.

Acknowledgements. We are grateful to the anonymous reviewers for their feedback, comments and suggestions, which helped to improve the paper.

References

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Calvanese, D.; Lembo, D.; Lenzerini, M.; and Rosati, R. 2007. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated Reasoning* 39(3):385–429.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *J. of Artificial Intelligence Research* 17:229–264.

Drescher, C., and Walsh, T. 2012. Answer set solving with lazy nogood generation. In Dovier, A., and Costa, V. S., eds., *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, 188–200. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, 90–96. Professional Book Center.

Eiter, T.; Ianni, G.; Lukasiewicz, T.; Schindlauer, R.; and Tompits, H. 2008. Combining answer set programming with description logics for the semantic web. *Artif. Intell.* 172(12-13):1495–1539.

Eiter, T.; Fink, M.; Krennwallner, T.; and Redl, C. 2012. Conflict-driven asp solving with external sources. *TPLP* 12(4-5):659–679.

Eiter, T.; Fink, M.; Krennwallner, T.; Redl, C.; and Schüller, P. 2014. Efficient HEX-program evaluation based on unfounded sets. *J. of Artificial Intelligence Research* 49:269–321.

Eiter, T.; Fink, M.; and Krennwallner, T. 2009. Decomposition of declarative knowledge bases with external functions. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, 752–758. AAAI Press/IJCAI.

Faber, W.; Leone, N.; and Pfeifer, G. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* 175(1):278–298.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 187–188:52–89.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9(3–4):365–386.

Mayer, W.; Bettex, M.; Stumptner, M.; and Falkner, A. 2009. On solving complex rack configuration problems using csp methods. In *IJCAI'09 Workshop on Configuration*.

Reiter, R., and de Kleer, J. 1987. Foundations of assumption-based truth maintenance systems: Preliminary report. In Forbus, K. D., and Shrobe, H. E., eds., *AAAI*, 183–189. Morgan Kaufmann.