Grounding HEX-Programs with Expanding Domains*

Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,redl}@kr.tuwien.ac.at

Abstract. Recently, liberally domain-expansion safe HEX-programs have been presented as a generalization of strongly safe HEX-programs that enlarges the scope of effective applicability. While such programs can be finitely grounded, a concrete grounding algorithm that is practically useful remained open. In this paper, we present such an algorithm and show how to integrate it into the model-building framework for HEX-programs, which is extended for this purpose. While traditional HEX-evaluation relies on program decomposition for grounding, our new algorithm can directly ground any liberally domain-expansion safe program without decomposition. However, as splitting is still sometimes useful for performance reasons, we develop a new decomposition heuristics that aims at maximizing efficiency. An experimental evaluation confirms the practicability of our approach.

1 Introduction

HEX-programs [6] are declarative logic programs that enrich Answer Set Programming (ASP) by so-called external atoms. They provide a means to couple any external computation or data source with a logic program: intuitively, information from the program, given by predicate extensions, is passed to an external source which returns output values of an (abstract) function that it computes. This extension has been motivated by emerging needs such as accessing distributed information, context awareness, complex or specific data structures, etc. Widening the application range of ASP with its systems like SMODELS, DLV and CLASP, HEX programs and the DLVHEX solver have enabled challenging applications such as querying data and ontologies on the Web, multi-context reasoning, e-government, and more (cf. [3]). In that, external atoms proved to be a valuable construct of high expressivity, which enables recursive data exchange between the program and external sources and, via a modular software plugin architecture, is convenient to realize customized data access, adding built-ins or to process specific datatypes.

A characteristic feature of HEX-programs is that new values (not occurring in the program) might arise by external source access. For example, an atom $\&concat[ab, c](Y)$ that intuitively appends c to ab, returns in Y the string abc. However, admitting such a behavior, so called *value invention*, poses severe challenges to grounding a respective HEX-program prior to solving, as common in ASP systems. It is intuitively clear that in

* 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.

general it is impossible to predetermine the "relevant" domain, like in the example above when $concat$ bears no meaning, and may be practically infeasible (even if it is finite). Imposing *strong safety* [7] amounts to disallowing value invention, which prevents the natural usage of even simple external atoms (like $concat$ above). On the other hand, to adopt standard safety conditions and to either perform a pre-evaluation of external atoms (as, e.g., for lua [9]), or to request the user to provide domain predicates, is likewise an unsatisfactory treatment of value invention.

To remedy the situation, strong safety has been recently relaxed [5] to yield *liberally domain-expansion safe* HEX-programs, which are still finitely groundable and more general than various other safety notions in the literature, e.g., VI-programs [1] and ω-restricted programs [14]. However, two important issues remained unresolved. First, to provide a concrete grounding algorithm for liberally domain-expansion safe HEX-programs that can efficiently produce an (equivalent) finite ground program. Second, to suitably integrate this algorithm into the existing HEX evaluation framework, in which a program is decomposed (exploiting a generalized splitting theorem [3]) into *evaluation units* that are grounded and solved separately; this needs to be respected by the grounder.

In this work we tackle these issues providing the following contributions:

- We introduce a grounding algorithm for the recently defined class of *(liberally) domain-expansion safe (de-safe)* HEX*-programs* [5]. Roughly speaking, the algorithm is based on (optimized) iterative grounding using a guess-and-check approach, which first computes a partial grounding and then checks its sufficiency for answer set computation.
- We integrate the new grounding algorithm into the existing evaluation framework for HEX-programs. To this end, we generalize the modular decomposition underlying the model-building process to arbitrary liberally de-safe HEX programs as evaluation units.
- A new evaluation heuristics for the framework aims at dealing with the opposing goals of, on the one hand, larger units to exploit learning during evaluation (cf. [4]), and on the other hand, splitting units for more efficient grounding. It greedily merges evaluation units unless efficient grounding requires a split, in preference of learning.
- We present an experimental evaluation of an implementation our algorithm on synthetic and application-driven benchmarks, which witnesses significant improvements.

An extended version of the paper, which includes proofs, is available at http://www.kr.tuwien.ac.at/staff/redl/grounding/groundingext.pdf.

2 Preliminaries

HEX-programs are built over mutually disjoint sets $\mathcal{P}$ of ordinary predicates, $\mathcal{X}$ of external predicates, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. In accordance with [10, 4], a *(signed) ground literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground atom of form $p(c_1, \ldots, c_\ell)$, with predicate $p \in \mathcal{P}$ and constants $c_1, \ldots, c_\ell \in \mathcal{C}$, abbreviated $p(\mathbf{c})$. For a ground literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e., $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* $\mathbf{A}$ is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that a is false. An *interpretation* is a complete (maximal) assignment $\mathbf{A}$, also identified by the set of true atoms $\mathbf{TA} = \{a \mid \mathbf{T}a \in \mathbf{A}\}$.

HEX-Program Syntax. HEX-programs are a generalization of (disjunctive) logic programs under the answer set semantics [11]; for details and background see [6].

An *external atom* is of the form $\&g[\mathbf{Y}](\mathbf{X})$, where $\mathbf{Y} = Y_1, \ldots, Y_\ell$ are input parameters with $Y_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, and $\mathbf{X} = X_1, \ldots, X_m$ are output terms with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq m$. Moreover, we assume that the input parameters of every external predicate $\&g \in \mathcal{X}$ are typed such that $type(\&g, i) \in \{\mathbf{const}, \mathbf{pred}\}$ for every $1 \leq i \leq \ell$. We make also the restriction that $Y_i \in \mathcal{P}$ if $type(\&g, i) = \mathbf{pred}$ and $X_i \in \mathcal{C} \cup \mathcal{V}$ otherwise. For an ordinary predicate $p \in \mathcal{P}$, let $ar(p)$ denote the arity of p and for an external predicate $\&g \in \mathcal{X}$, let $iar(\&g)$ denote the input arity and $oar(\&g)$ the output arity of $\&g$.

A HEX-program consists of rules

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

where each a_i is an (ordinary) atom $p(X_1, \ldots, X_\ell)$ with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The *head* of a rule r is $H(r) = \{a_1, \ldots, a_n\}$ and the *body* is $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$. We call b or $\text{not}\, b$ in a rule body a *default literal*; $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ is the *negative body*. For a program Π (a rule r), let $A(\Pi)$ ($A(r)$) be the set of all ordinary atoms and $EA(\Pi)$ ($EA(r)$) be the set of all external atoms occurring in Π (in r).

HEX**-Program Semantics**. The semantics of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an interpretation, i.e., a complete assignment, $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary Boolean-valued *oracle function*, denoted by $f_{\&g}$, that is defined for all possible values of $\mathbf{A}$, $\mathbf{p}$ and $\mathbf{c}$. We make the restriction that for given $\mathbf{A}$, $\mathbf{p}$, set $\{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{x}) = 1\}$ is computable. An input predicate p of an external predicate with input list $\&g[\mathbf{p}]$ is *monotonic* (*antimonotonic*), iff $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ implies $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = 1$ ($f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 0$ implies $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = 0$) for all $\mathbf{A}'$ s.t. $ext(p, \mathbf{A}') \supseteq ext(p, \mathbf{A})$ and $ext(q, \mathbf{A}') = ext(q, \mathbf{A})$ for $q \in \mathbf{p}$ and $q \neq p$. It is *nonmonotonic* iff it is neither monotonic nor antimonotonic. We denote this by $\mathbf{Y}_m$, $\mathbf{Y}_a$, $\mathbf{Y}_n$ the predicates from $\mathbf{Y}$ which are monotonic, antimonotonic and nonmonotonic, respectively. Satisfaction of ground ordinary rules and ASP programs [11] is then extended to HEX-rules and programs in the obvious way.

Non-ground programs are handled by grounding as usual. The *grounding* $grnd_C(r)$ of a rule r wrt. $C \subseteq \mathcal{C}$ is the set of all rules $\{\sigma(r) \mid \sigma : \mathcal{V} \mapsto C\}$, where σ is a *grounding substitution* mapping each variable to a constant, and $\sigma(r)$ denotes the rule which results if each variable X in r is replaced by $\sigma(X)$. The *grounding of a program* Π wrt. C (respectively $\mathcal{C}$ if not mentioned explicitly) is defined as $grnd_C(\Pi) = \bigcup_{r \in \Pi} grnd_C(r)$. The set of constants appearing in a program Π is denoted C_Π.

An answer set of a program Π is a model $\mathbf{A}$ of the FLP-reduct [8] $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}} = \{r \in grnd_{\mathcal{C}}(\Pi) \mid \mathbf{A} \models B(r)\}$ such that $\mathbf{TA}$ is subset-minimal, i.e., no $\mathbf{A}'$ with $\mathbf{TA}' \subsetneq \mathbf{TA}$ is a model. We denote the set of all answer sets of a program Π by $\mathcal{AS}(\Pi)$, and write $\Pi \equiv \Pi'$ if $\{\mathbf{TA} \mid \mathbf{A} \in \mathcal{AS}(\Pi)\} = \{\mathbf{TA} \mid \mathbf{A} \in \mathcal{AS}(\Pi')\}$.

Example 1. Consider program $\Pi = \{p \leftarrow \&id[p]()\}$, where $\&id[p]()$ is true iff p is true. Π has the unique answer set $\mathbf{A}_1 = \emptyset$, which is a subset-minimal model of $fgrnd_{\mathcal{C}}(\Pi)^{\mathbf{A}_1} = \emptyset$.

Safety. In general, the set $\mathcal{C}$ contains constants that do not occur in the program Π and $\mathcal{C}$ can even be infinite (e.g., the set of all strings). Therefore, safety criteria are adopted which guarantee the existence of a finite portion $\Pi' \subseteq grnd_{\mathcal{C}}(\Pi)$ (also called *finite*

grounding of Π; usually by restricting to a finite $C \subseteq \mathcal{C}$) that has the same answer sets as Π. A program is *safe*, if all rules r are *safe*, i.e., every variable in r is *safe* in the sense that it occurs either in an ordinary atom in $B^+(r)$, or in the output list $\mathbf{X}$ of an external atom $\&g[\mathbf{Y}](\mathbf{X})$ in $B^+(r)$ where all variables in $\mathbf{Y}$ are safe. However, this notion is not sufficient, as the following example shows.

Example 2. Let $\Pi = \{s(a);\ t(Y) \leftarrow s(X), \&concat[X, a](Y); s(X) \leftarrow t(X), d(X)\}$, where $\&concat[X, a](Y)$ is true iff Y is the string concatenation of X and a. Then Π is safe but $\&concat[X, a](Y)$ can introduce infinitely many constants. □

Therefore, *strong safety* was introduced in [7], which ensures that the output of cyclic external atoms is limited. This notion was recently relaxed to *(liberal) domain-expansion safety (de-safety)* [5], on which we focus here. It is based on *term bounding functions*, which intuitively declare terms in rules as *bounded*, if there are only finitely many substitutions for this term in a *canonical grounding* $CG(\Pi)$ of Π.[1] This grounding is infinite in general and serves to define liberal de-safe HEX-programs; for such programs, however, $CG(\Pi)$ is finite. In this paper, we present an algorithm for efficient construction of a concrete finite ground program that is equivalent to $CG(\Pi)$ (and thus to Π).

Definition 1 (Term Bounding Function (TBF)). *A TBF $b(\Pi, r, S, B)$ maps a program Π, a rule $r \in \Pi$, a set S of already safe attributes, and a set B of already bounded terms in r to an enlarged set $b(\Pi, r, S, B) \supseteq B$ of bounded terms, s.t. every $t \in b(\Pi, r, S, B)$ has finitely many substitutions in $CG(\Pi)$ if (i) the attributes S have a finite range in $CG(\Pi)$ and (ii) each term in $terms(r) \cap B$ has finitely many substitutions in $CG(\Pi)$.*

Liberal domain-expansion safety of programs is then parameterized with a term bounding function, such that concrete syntactic and/or semantic properties can be plugged in; concrete term bounding functions are described in [5]. The concept is defined in terms of domain-expansion safe attributes $S_\infty(\Pi)$, which are stepwise identified as $S_n(\Pi)$ in mutual recursion with bounded terms $B_n(r, \Pi, b)$ of rules r in Π.

Definition 2 ((Liberal) Domain-expansion Safety). *Given a TBF b, the set of* bounded terms $B_n(r, \Pi, b)$ *in step $n \geq 1$ in a rule $r \in \Pi$ is $B_n(r, \Pi, b) = \bigcup_{j \geq 0} B_{n,j}(r, \Pi, b)$ where $B_{n,0}(r, \Pi, b) = \emptyset$ and for $j \geq 0$, $B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j})$.*

The set of domain-expansion safe attributes $S_\infty(\Pi) = \bigcup_{n \geq 0} S_n(\Pi)$ *of a program Π is iteratively constructed with $S_0(\Pi) = \emptyset$ and for $n \geq 0$:*

- $p\restriction i \in S_{n+1}(\Pi)$ *if for each $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$, it holds that $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is* bounded*;*
- $\&g[\mathbf{Y}]_r\restriction_{\text{I}} i \in S_{n+1}(\Pi)$ *if each $\mathbf{Y}_i$ is a* bounded *variable, or $\mathbf{Y}_i$ is a predicate input parameter p and $p\restriction 1, \ldots, p\restriction ar(p) \in S_n(\Pi)$;*
- $\&g[\mathbf{Y}]_r\restriction_{\text{O}} i \in S_{n+1}(\Pi)$ *if and only if r contains an external atom $\&g[\mathbf{Y}](\mathbf{Y})$ such that $\mathbf{Y}_i$ is bounded, or $\&g[\mathbf{Y}]_r\restriction_{\text{I}} 1, \ldots, \&g[\mathbf{Y}]_r\restriction_{\text{I}} ar_{\text{I}}(\&g) \in S_n(\Pi)$.*

A program Π is (liberally) de-safe*, if it is safe and all its attributes are de-safe.*

[1] $CG(\Pi)$ is least fixed point $G_\Pi^\infty(\emptyset)$ of a monotone operator $G_\Pi(\Pi') = \bigcup_{r \in \Pi}\{r' \mid r' \in grnd_{\mathcal{C}}(r), \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r')\}$ on programs Π' [5], where $\mathcal{A}(\Pi')$ denotes the set of all atoms in Π'.

Example 3. The program Π from Example 2 is liberally de-safe as infinitely many constants are prevented by domain predicate $d(X)$ in the last rule. □

As shown in [5], every de-safe HEX-program has a finite grounding with the same answer sets as the original program. This result holds for *every* TBF, because the preconditions of a TBF force it to be sufficiently strong.

For further explanation and discussion of liberal de-safety, and for an analysis showing that the concept subsumes a number of other notions of safety we refer to [5].

3 Grounding Liberally Domain-expansion Safe HEX-Programs

In this section we present a grounding algorithm for *liberally domain-expansion safe* HEX*-programs* as introduced in [5]. It is based on the following idea. Iteratively ground the input program and then check if the grounding contains all relevant ground rules. The check works by evaluating external sources under relevant interpretations and testing if they introduce any new values which were not respected in the grounding. If this is the case, then the set of constants is expanded and the program is grounded again. If the check does not identify additional constants which must be respected in the grounding, then it is guaranteed that the unrespected constants from $\mathcal{C}$ are irrelevant in order to ensure that the grounding has the same answer sets as the original program. For liberally domain-expansion safe programs, this procedure will eventually reach a fixpoint, i.e., all relevant constants are respected in the grounding.

We start with some basic concepts which are all demonstrated in Example 4. We assume that rules are standardized apart (i.e., have no variables in common). Let R be a set of external atoms and let r be a rule. By $r|_R$ we denote the rule obtained by removing external atoms not in R, i.e., such that $H(r|_R) = H(r)$ and $B^s(r|_R) = ((B^s(r) \cap A(r)) \cup (B^s(r) \cap R))$ for $s \in \{+, -\}$. Similarly, $\Pi|_R = \bigcup_{r \in \Pi} r|_R$, for a program Π. Furthermore, let $var(r)$ be the set of variables from $\mathcal{V}$ appearing in a rule r.

Definition 3 (Liberal Domain-expansion Safety Relevance). *A set R of external atoms is* relevant for liberal de-safety *of a program Π, if $\Pi|_R$ is liberally de-safe and $var(r) = var(r|_R)$, for all $r \in \Pi$.*

Intuitively, if an external atom is not relevant, then it cannot introduce new constants. Note that for a program, the set of de-safe relevant external atoms is not necessarily unique, leaving room for heuristics. In the following definitions we choose a specific set.

We further need the concepts of input auxiliary and external atom guessing rules. We say that an external atom $\&g[\mathbf{Y}](\mathbf{X})$ *joins* an atom b, if some variable from $\mathbf{Y}$ occurs in b, where in case b is an external atom the occurrence is in the output list of b.

Definition 4 (Input Auxiliary Rule). *Let Π be a* HEX*-program, and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom with input list $\mathbf{Y}$ occurring in a rule $r \in \Pi$. Then, for each such atom, a rule $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *The head is $H(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{g_{inp}(\mathbf{Y})\}$, where g_{inp} is a fresh predicate; and*
- *The body $B(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains each $b \in B^+(r) \backslash \{\&g[\mathbf{Y}](\mathbf{X})\}$ such that $\&g[\mathbf{Y}](\mathbf{X})$ joins b, and b is de-safety-relevant if it is an external atom.*

Intuitively, input auxiliary rules are used to derive all ground tuples $\mathbf{y}$ under which the external atom needs to be evaluated. Next, we need *external atom guessing rules*.

Definition 5 (External Atom Guessing Rule). *Let Π be a* HEX*-program, and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom. Then a rule $r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *The head is $H(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{e_{r,\&g[\mathbf{Y}]}(\mathbf{X}), ne_{r,\&g[\mathbf{Y}]}(\mathbf{X})\}$*
- *The body $B(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains*
  - *(i) each $b \in B^+(r) \setminus \{\&g[\mathbf{Y}](\mathbf{X})\}$ such that $\&g[\mathbf{Y}](\mathbf{X})$ joins b and b is de-safety-relevant if it is an external atom; and*
  - *(ii) $g_{inp}(\mathbf{Y})$.*

Intuitively, they guess the truth value of external atoms using a choice between the *external replacement atom* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$, and fresh atom $ne_{r,\&g[\mathbf{Y}]}(\mathbf{X})$.

Our approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_{\mathcal{C}}(\Pi)$, which substitutes all constants for all variables in all possible ways, we allow the ASP grounder GroundASP to optimize rules such that, intuitively, rules may be eliminated if their body is always false, and ordinary body literals may be removed from the grounding if they are always true, as long as this does not change the answer sets.

Definition 6. *We call rule r' an* o-strengthening *of r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains only ordinary literals, i.e., no external atom replacements.*

Definition 7. *An algorithm GroundASP is a* faithful ASP grounder *for a safe ordinary program Π, if it outputs an equivalent ground program Π' such that*

- *Π' consists of o-strengthenings of rules in $grnd_{C_\Pi}(\Pi)$;*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has no o-strengthening in Π', then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has some o-strengthening $r' \in \Pi'$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies $B(r) \setminus B(r')$.*

The formalization of the algorithm is shown in Algorithm GroundHEX. Our naming convention is as follows. Program Π is the non-ground input program. Program Π_p is the non-ground ordinary ASP *prototype program*, which is an iteratively updated extension of Π with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} converges against a fixpoint from which the final *ground* HEX*-program* Π_g is extracted.

The algorithm first chooses a set of de-safety relevant external atoms, e.g., all external atoms as a naive and conservative approach or following a greedy approach as in our implementation, and then introduces input auxiliary rules $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ for every external atom $\&g[\mathbf{Y}](\mathbf{X})$ in a rule r in Π in Part (a). For all non-relevant external atoms, we introduce external atom guessing rules which ensure that the ground instances of these external atoms are introduced in the grounding, even if we do not explicitly add them. Then, all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p are replaced by ordinary *replacement atoms* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$. This allows the algorithm to use a faithful ASP grounder GroundASP in the main loop at (b). After the grounding step, the algorithm checks

Algorithm GroundHEX

Input: A liberally de-safe HEX-program Π
Output: A ground HEX-program Π_g s.t. $\Pi_g \equiv \Pi$

(a) Choose a set R of *de-safety-relevant* external atoms in Π
$\Pi_p := \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\} \cup \{r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \notin R\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
(b) **repeat**
  $\Pi_{pg} :=$ GroundASP(Π_p) `/* partial grounding */`
  `/* evaluate all de-safety-relevant external atoms */`
(c) **for** $\&g[\mathbf{Y}](\mathbf{X}) \in R$ *in a rule* $r \in \Pi$ **do**
    $\mathbf{A}_{ma} := \{\mathbf{T}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
    `/* do this under all relevant assignments */`
(d) **for** $\mathbf{A}_{nm} \subseteq \{\mathbf{T}p(\mathbf{c}), \mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_n\}$ *s.t.* $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ **do**
      $\mathbf{A} := (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow\in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow\in \Pi_{pg}\}$
(e) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})\}$ **do**
(f) Let $O = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
        `/* add the respective ground guessing rules */`
        $\Pi_p := \Pi_p \cup \{e_{r,\&g[\mathbf{Y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{Y}]}(\mathbf{x}) \leftarrow \mid \mathbf{x} \in O\}$
until Π_{pg} *did not change*
(g) Remove input auxiliary rules and external atom guessing rules from Π_{pg}
Replace all $e_{\&g[\mathbf{Y}]}(\mathbf{x})$ in Π by $\&g[\mathbf{Y}](\mathbf{x})$
return Π_{pg}

if the grounding is large enough, i.e., if it contains all relevant constants. For this, it traverses all relevant external atoms at (c) and all relevant input tuples at (d) and at (e). Then, constants returned by external sources are added to Π_p at (f); if the constants were already respected, then this will have no effect. Thereafter the main loop starts over again. The algorithm will find a program which respects all relevant constants. It then removes auxiliary input rules and translates replacement atoms to external atoms at (g).

We illustrate our grounding algorithm with the following example.

Example 4. Let Π be the following program:

$$f_1: \ d(a). \quad f_2: \ d(b). \quad f_3: \ d(c). \qquad r_1: \ s(Y) \leftarrow \&diff[d,n](Y), d(Y).$$
$$r_2: \ n(Y) \leftarrow \&diff[d,s](Y), d(Y).$$
$$r_3: \ c(Z) \leftarrow \&count[s](Z).$$

Here, $\&diff[s_1, s_2](x)$ is true for all elements x, which are in the extension of s_1 but not in that of s_2, and $\&count[s](i)$ is true for the integer i corresponding to the number of elements in s. The program partitions the domain (extension of d) into two sets (extensions of s and n) and computes the size of s. The external atoms $\&diff[d,n](Y)$ and $\&diff[d,s](Y)$ are not relevant for de-safety. Π_p at the beginning of the first iteration is as follows (neglecting input auxiliary rules, which are facts). Let $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ be shorthands for $e_{r_1,\&diff[d,n]}(Y)$, $e_{r_2,\&diff[d,s]}(Y)$. and $e_{r_3,\&count[s]}(Z)$, respectively.

$$f_1: \ d(a). \quad f_2: \ d(b). \quad f_3: \ d(c). \qquad r_1: \ s(Y) \leftarrow e_1(Y), d(Y).$$
$$g_1: \ e_1(Y) \vee ne_1(Y) \leftarrow d(Y). \qquad r_2: \ n(Y) \leftarrow e_2(Y), d(Y).$$
$$g_2: \ e_2(Y) \vee ne_2(Y) \leftarrow d(Y). \qquad r_3: \ c(Z) \leftarrow e_3(Z).$$

The ground program Π_{pg} contains no instances of r_3 because the optimizer recognizes that $e_{r_3,\&count[s]}(Z)$ occurs in no rule head and no ground instance can be true in any answer set. Then the algorithm comes to the checking phase. It does not evaluate the external atoms in r_1 and r_2, because they are not relevant for de-

safety because of the domain predicate $d(Y)$. But it evaluates $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because the external atom is nonmonotonic in s. Then the algorithm adds rules $\{e_3(Z) \vee ne_3(Z) \leftarrow \mid Z \in \{0,1,2,3\}\}$ to Π_p. After the second iteration, the algorithm terminates. □

One can show that this algorithm is sound and complete:

Proposition 1. *If Π is a liberally de-safe* HEX*-program, then* GroundHEX$(\Pi) \equiv \Pi$.

4 Integrating the Algorithm into the Model-building Framework

The answer sets of a HEX-program Π are determined using a modular decomposition based on the concept of an *evaluation graph* $\mathcal{E}(V, E)$, whose nodes V are *evaluation units*, i.e. subsets of Π, that are acyclically connected by edges $E = \rightarrow_m \cup \rightarrow_n$ that are inherited from an underlying *dependency graph* $G = \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$, where $\rightarrow_m$ captures monotonic and $\rightarrow_n$ nonmonotonic dependencies of the units resp. rules [3].

The evaluation proceeds then unit by unit along the structure of the evaluation graph bottom up. For a unit u, each union of answer sets of predecessor units of u, called an *input model of* u, is added as facts to the program at u. This extended program is grounded and solved; the resulting set of *output models of* u is sent to the successor units of u in the same way. The properties of evaluation graphs guarantee that the output models of a dedicated final unit correspond to the answer sets of the whole program.

In order to ground the units before evaluation using a grounding algorithm for ordinary ASP, each unit in the evaluation graph must be from the class of *extended pre-groundable* HEX*-programs*, which is a proper subset of all strongly safe HEX programs. It was shown in [13] that every strongly safe HEX-program possesses at least one evaluation graph, i.e., the program can be decomposed into extended pre-groundable HEX-programs.

The motivation for the evaluation framework in [3] was mainly performance enhancement. However, as not every strongly safe program is extended pre-groundable, program decomposition is in some cases *indispensable* for program evaluation. This is in contrast to the grounding algorithm introduced in this paper, which can directly ground any liberally de-safe, and thus strongly safe, program.

Example 5. Program Π from Example 4 cannot be grounded by the traditional HEX algorithms as it is not extended pre-groundable. Instead, it needs to be partitioned into two units $u_1 = \{f_1, f_2, f_3, r_1, r_2\}$ and $u_2 = \{r_3\}$ with $u_1 \rightarrow_n u_2$. Now u_1 and u_2 are extended pre-groundable HEX-programs. Then the answer sets of u_1 must be computed before u_2 can be grounded. Our algorithm can ground the whole program immediately.□

Therefore, in contrast to the previous algorithms one can keep the whole program as a single unit, but also still apply decomposition with liberally de-safe programs as units. To this end, we define a *generalized evaluation graph* like an evaluation graph in [3], but with de-safe instead of extended pre-groundable programs as nodes. We can then show that the algorithm BUILDANSWERSETS in [3] remains sound and complete for generalized evaluation graphs, if the grounding algorithm from above is applied:

Proposition 2. *For a generalized evaluation graph $\mathcal{E} = (V, E)$ of a de-safe* HEX*-program Π,* BUILDANSWERSETS *with* GroundHEX *for grounding returns $\mathcal{AS}(\Pi)$.*

Algorithm GreedyGEG

Input: A liberally de-safe HEX-program Π
Output: A generalized evaluation graph $\mathcal{E} = \langle V, E \rangle$ for Π
Let V be the set of (subset-maximal) strongly connected components of $G = \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$
Update E
while V *was modified* **do**
  for $u_1, u_2 \in V$ *such that* $u_1 \neq u_2$ **do**
(a) **if** *there is no indirect path from* u_1 *to* u_2 *(via some* $u' \neq u_1, u_2$*) or vice versa* **then**
(b) **if** *no de-relevant* $\&g[\mathbf{y}](\mathbf{x})$ *in some* u_2 *has a nonmonotonic predicate input from* u_1 **then**
        $V := (V \setminus \{u_1, u_2\}) \cup \{u_1 \cup u_2\}$
        Update E
return $\mathcal{E} = \langle V, E \rangle$

While program decomposition led to performance increase for the solving algorithms from [3], it is counterproductive for new learning-based algorithms [4] because learned knowledge cannot be effectively reused. In guess-and-check ASP programs, existing heuristics for evaluation graph generation frequently even split the guessing from the checking part, which is derogatory to the learning. Thus, from this perspective is advantageous to have few units. However, for the grounding algorithm a worst case is that a unit contains an external atom that is relevant for de-safety and receives nonmonotonic input from the same unit. In this case it needs to consider exponentially many assignments.

Example 6. Reconsider program Π from Example 4. Then the algorithm evaluates $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic and de-safety-relevant. Now assume that the program contains the additional constraint

$$c_1 : \quad \leftarrow s(X), s(Y), s(Z), X \neq Y, X \neq Z, Y \neq Z \ ,$$

i.e., no more than two elements can be in set s. Then the algorithm would still check all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$, but it is clear that the subset with three elements, which introduces the constant 3, is irrelevant because this interpretation will never occur in an answer set. If the program is split into units $u_1 = \{f, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \rightarrow_n u_1$, then $\{s(a), s(b), s(c)\}$ does not occur as an answer set of u_1. Thus, u_2 never receives this interpretation as input and never is evaluated under this interpretation. □

Algorithm GroundHEX evaluates the external sources under all interpretations such that the set of observed constants is maximized. While monotonic and antimonotonic input atoms are not problematic (the algorithm can simply set all to true resp. false), nonmonotonic parameters require an exponential number of evaluations. Thus, in such cases program decomposition is still useful as it restricts grounding to those interpretations which are actually relevant in some answer set. Program decomposition can be seen as a hybrid between traditional and lazy grounding [12], as program parts are instantiated which are larger than single rules but smaller than the whole program.

We thus introduce a heuristics in Algorithm GreedyGEG for generating a good generalized evaluation graph, which iteratively merges units. Condition (a) maintains acyclicity, while the condition at (b) deals with two opposing goals: (1) minimizing the number of units, and (2) splitting the program whenever a de-relevant nonmonotonic external atom would receive input from the same unit. It greedily gives preference to (1).

We illustrate the heuristics with an example.

Example 7. Reconsider program Π from Examples 4 and 6. Algorithm GreedyGEG creates a generalized evaluation graph with the two units $u_1 = \{f_1, f_2, f_3, r_1, r_2, c_1\}$ and $u_2 = \{r_3\}$ with $u_2 \rightarrow_n u_1$, which is as desired. □

It is not difficult to show that the heuristics yields a sound result.

Proposition 3. *For a liberally de-safe program Π, Algorithm GreedyGEG returns a suitable generalized evaluation graph of Π.*

5 Implementation and Evaluation

For implementing our technique, we integrated GRINGO as grounder GroundASP and CLASP into our prototype system DLVHEX[2]. We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM. For this we use five benchmarks and present the total wall clock runtime (wt), the grounding time (gt) and the solving time (st). We possibly have $wt \neq gt + st$ because wt includes also computations other than grounding and solving (e.g., passing models through the evaluation graph). For determining de-safety relevant external atoms, our implementation follows a greedy strategy and tries to identify as many external atoms as irrelevant as possible. Detailed benchmark results are available at http://www.kr.tuwien.ac.at/staff/redl/grounding/benchmarks.ods.

Reachability. We consider reachability, where the edge relation is provided as an external atom $\&out[X](Y)$ delivering all nodes Y that are directly reached from a node X. The traditional implementation imports all nodes into the program and then uses domain predicates. An alternative is to query outgoing edges of nodes on-the-fly, which needs no domain predicates. This benchmark is motivated by route planning applications, where importing the full map might be infeasible due to the amount of data.

The results are shown in Table 1a. We use random graphs with a node count from 5 to 70 and an edge probability of 0.25. For each count, we average over 10 instances. Here we can observe that the encoding without domain predicates is more efficient in all cases because only a small part of the map is active in the logic program, which does not only lead to a smaller grounding, but also to a smaller search space during solving.

Set Partitioning. In this benchmark we consider a program similar to Example 4, which implements for each domain element x a choice from $sel(x)$ and $nsel(x)$ by an external atom, i.e., a partitioning of the domain into two subsets.

The domain predicate $domain$ is not necessary with de-safety because $\&diff$ does not introduce new constants. The effect of removing it is presented in Table 1b. Since $\&diff$ is monotonic in the first parameter and antimonotonic in the second, the measured overhead is small in the grounding step. Although the ground programs of the strongly safe and the liberally safe variants of the program are identical, the solving step is slower in the latter case; we explain this with caching effects. Grounding liberally de-safe programs needs more memory than grounding strongly safe programs, which might have negative effects on the later solving step. However, the total slowdown is moderate.

[2] http://www.kr.tuwien.ac.at/research/systems/dlvhex

Table 1: Benchmark Results (in secs; "—" means timeout, set to 300 secs)

(a) Reachability

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
15	0.59	0.28	0.08	0.49	0.23	0.06
25	5.78	4.67	0.33	2.94	1.90	0.35
35	36.99	33.99	1.00	14.02	11.30	0.95
45	161.91	155.40	2.18	53.09	47.19	2.22
55	—	—	n/a	171.46	158.58	5.74
65	—	—	n/a	—	—	n/a

(b) Set Partitioning

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
10	0.49	0.01	0.39	0.52	0.02	0.41
20	3.90	0.05	3.62	4.67	0.10	4.23
30	16.12	0.18	15.32	19.59	0.36	18.32
40	48.47	0.48	46.71	51.55	0.90	48.74
50	115.56	1.00	112.14	119.40	1.79	114.11
60	254.66	1.84	248.88	257.78	3.35	248.51

(c) Bird-penguin

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5	0.06	<0.005	0.01	0.08	0.02	0.01
10	0.14	<0.005	0.08	1.32	1.12	0.10
11	0.27	<0.005	0.19	2.85	2.43	0.27
12	0.32	<0.005	0.23	6.05	5.53	0.26
13	0.69	0.01	0.60	12.70	11.76	0.61
14	0.66	<0.005	0.57	28.17	26.70	0.73
15	1.66	0.01	1.49	59.73	57.14	1.46
16	1.69	0.01	1.53	139.47	131.87	1.92
17	3.83	0.01	3.57	—	—	n/a
18	4.34	0.01	4.08	—	—	n/a
19	10.07	0.01	9.56	—	—	n/a
20	11.36	0.01	10.87	—	—	n/a
24	95.60	0.01	93.35	—	—	n/a
25	—	0.01	—	—	—	n/a

(d) Merge Sort

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5	0.22	0.04	0.10	0.10	0.01	0.04
6	1.11	0.33	0.54	0.10	0.01	0.04
7	9.84	4.02	4.42	0.11	0.01	0.05
8	115.69	61.97	42.30	0.12	0.01	0.05
9	—	—	n/a	0.14	0.01	0.07
10	—	—	n/a	0.15	0.08	0.01
15	—	—	n/a	0.23	0.14	0.01
20	—	—	n/a	0.47	0.35	0.02
25	—	—	n/a	1.90	1.58	0.06
30	—	—	n/a	4.11	3.50	0.12
35	—	—	n/a	20.98	18.45	0.51
40	—	—	n/a	61.94	54.62	1.46
45	—	—	n/a	144.22	133.99	2.26
50	—	—	n/a	—	—	n/a

(e) Argumentation

#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve
4	0.57	0.11	0.38	0.25	0.01	0.18
5	2.12	0.67	1.26	0.44	0.01	0.37
6	18.93	7.45	10.86	0.88	0.01	0.80
7	237.09	170.12	65.12	1.65	0.01	1.57
8	—	—	n/a	3.13	0.01	3.05
9	—	—	n/a	7.41	0.02	7.31

#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve
10	—	—	n/a	15.92	0.02	15.81
11	—	—	n/a	31.19	0.02	31.05
12	—	—	n/a	63.16	0.02	62.95
13	—	—	n/a	172.75	0.03	172.38
14	—	—	n/a	256.60	0.01	256.44
15	—	—	n/a	290.01	<0.005	290.00

Bird-Penguin. We consider now a scenario using the DL-plugin for DLVHEX, which integrates description logics (DL) knowledge bases and nonmonotonic logic programs. The DL-plugin allows to access an ontology. We consider the ontology on the right, which encodes that penguins are birds and do not fly, and the logic program on the left which implements the rule that birds fly unless the contrary is derivable.

DL-Program:

$$birds(X) \leftarrow DL[Bird](X).$$
$$\mathit{flies}(X) \leftarrow birds(X), \text{not } neg\_\mathit{flies}(X).$$
$$neg\_\mathit{flies}(X) \leftarrow birds(X), DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X).$$

Ontology:

$$\mathit{Flier} \sqsubseteq \neg \mathit{NonFlier}$$
$$\mathit{Penguin} \sqsubseteq \mathit{Bird}$$
$$\mathit{Penguin} \sqsubseteq \mathit{NonFlier}$$

Intuitively, $DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}]$ requests all individuals in $\neg \mathit{Flier}$ under the assumption that Flier is extended by the elements in the extension of flies.

The third rule uses the domain predicate $birds(X)$, which is necessary under strong safety conditions, but with liberal de-safety, we might drop it because finiteness is guaranteed by finiteness of the ontology. The results are shown in Table 1c. The DL-

atom in the third rule is nonmonotonic and appears in a cycle, which is the worst case which cannot be avoided by the greedy heuristics. The results show a slowdown for the encoding without domain predicates. It is mainly caused by the grounding, but also solving becomes slightly slower without domain predicates due to caching effects.

Recursive Processing of Data Structures. This benchmark shows how data structures can be recursively processed. As an example we implement the merge sort algorithm using external atoms for *splitting a list in half* and *merging two sorted lists*, where lists are encoded as constants consisting of elements and delimiters. However, this is only a showcase and performance cannot be compared to native merge sort implementations.

In order to implement the application with strong safety, one must manually add a domain predicate with the set of all instances of the data structures at hand as extension, e.g., the set of all permutations of the input list. This number is factorial in the input size and thus already unmanagable for very small instances. The problems are both due to grounding and solving. Similar problems arise with other recursive data structures when strong safety is required (e.g., trees, for the pushdown automaton from [5], where the domain is the set of all strings up to a certain length). However, only a small part of the domain will ever be relevant during computation, hence the new grounding algorithm for liberally de-safe programs performs quite well, as shown in Table 1d.

Argumentation. This benchmark demonstrates the advantage of our new *greedy heuristics*, which is compared to the evaluation without splitting (*monolithic*). We compute ideal set extensions for randomized instances of abstract argumentation frameworks [2] of different sizes. External atoms are used for checking candidate extensions. Additionally, we perform a processing of the arguments in each extension, e.g., by using an external atom for generating LaTeX code for the visualization of the framework and its extensions. Without program decomposition, this is the worst case for our grounding algorithm because the code generating atom is nonmonotonic and receives input from the same component. But then our grounding algorithm calls it for exponentially many extensions, although only few of them are actually extensions of the framework.

We use random instances with an argument count from 1 to 20, and an edge probability from $\{0.30, 0.45, 0.60\}$; we use 10 instances for each combination. We can observe that grounding the whole program in a single pass causes large programs wrt. grounding time and size. Since the grounding is larger, also the solving step takes much more time than with our new decomposition heuristics, which avoids the worst case, cf. Table 1e.

Summary. Our new grounding algorithm allows for grounding liberally de-safe programs. Instances that can be grounded by the traditional algorithm as well, usually require domain predicates to be manually added (often cumbersome and infeasible in practice, as for recursive data structures). Our algorithm does not only relieve the user from writing domain predicates, but in many cases also has a significantly better performance. Nonmonotonic external atoms might be problematic for our new algorithm. However, the worst case can mostly be avoided by our new decomposition heuristics.

6 Conclusion

In this paper we presented a new grounding algorithm for the recent class of liberally domain-expansion safe HEX-programs [5]. In contrast to previous grounding techniques

for HEX-programs, it can handle all such programs directly and does not rely on a program decomposition. This is an advantage, as splitting has negative effects for learning techniques introduced in [4]. However, in the worst case the new algorithm requires exponentially many calls to external sources to determine the relevant constants for grounding. We thus developed a novel heuristics for program evaluation that aims at avoiding this worst case while retaining the positive features of the new algorithm, and we have extended the current HEX evaluation framework for its use. An experimental evaluation of our implementation on synthetic and real applications shows a clear benefit.

Future work includes *refinements* of our algorithm and the heuristics. In particular, we plan to exploit *meta-information* about external sources to identify classes of programs that allow for a better grounding, and in particular reduce worst case inputs for our algorithm. Furthermore, ongoing work investigates logic programs with *existentially quantified variables* in rule heads, which might be realized by a variant of our algorithm.

References

1. Calimeri, F., Cozza, S., Ianni, G.: External Sources of Knowledge and Value Invention in Logic Programming. Ann. Math. Artif. Intell. 50(3–4), 333–361 (2007)
2. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321–357 (1995)
3. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing efficient evaluation of HEX programs by modular decomposition. In: LPNMR'11. pp. 93–106. Springer (2011)
4. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Conflict-driven ASP solving with external sources. Theor. Pract. Log. Prog. 12(4-5), 659-679 (2012)
5. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Liberal Safety Criteria for HEX-Programs. In: AAAI'13. AAAI Press (2013), http://www.kr.tuwien.ac.at/staff/tkren/pub/2013/aaai2013-liberalsafety.pdf, to appear
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: IJCAI'05. pp. 90–96. Professional Book Center (2005)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In: ESWC'06. pp. 273–287. Springer (2006)
8. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan & Claypool Publishers (2012)
10. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187–188, 52–89 (2012)
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generat. Comput. 9(3–4), 365–386 (1991)
12. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. Fund. Inform. 96(3), 297–322 (2009)
13. Schüller, P.: Inconsistency in Multi-Context Systems: Analysis and Efficient Evaluation. Dissertation, Vienna University of Technology, Vienna, Austria (August 2012)
14. Syrjänen, T.: Omega-restricted logic programs. In: LPNMR'01. pp. 267–279 (2001)