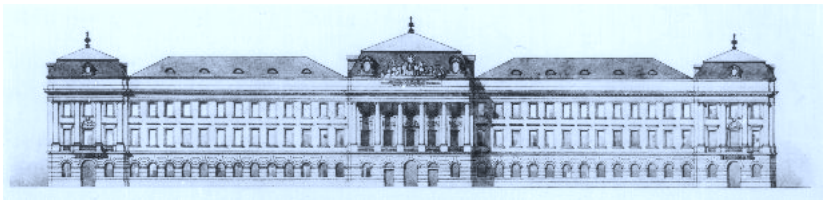
I N F S Y S

F O R S C H U N G S

B E R I C H T



INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

DOMAIN EXPANSION FOR ASP-PROGRAMS WITH EXTERNAL SOURCES

THOMAS EITER MICHAEL FINK

THOMAS KRENNWALLNER CHRISTOPH REDL

INFSYS FORSCHUNGSBERICHT 14-02

SEPTEMBER 2014

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TU WIEN

kbs Knowledge-Based Systems Group

INFSYS FORSCHUNGSBERICHT
INFSYS FORSCHUNGSBERICHT 14-02, SEPTEMBER 2014

DOMAIN EXPANSION FOR ASP-PROGRAMS WITH EXTERNAL SOURCES

Thomas Eiter[1] Michael Fink[1] Thomas Krennwallner[1] Christoph Redl[1]

Zusammenfassung. Answer set programming (ASP) is a popular approach to declarative problem solving which for broader usability has been equipped with external source access. The latter may introduce new constants to the program (known as *value invention*), which can lead to infinite answer sets and non-termination; to prevent this, syntactic safety conditions on programs are common which considerably limit expressiveness (in particular, recursion). We present *liberal domain-expansion (lde) safe programs*, a novel generic class of ASP programs with external source access and value invention that enjoy finite restrictability, i.e., equivalence to a finite ground version. They use *term bounding functions* as a parametric notion of safety, which can be instantiated with syntactic, semantic or combined safety criteria; this empowers us to generalize and integrate many other notions of safety from the literature, and modular composition of criteria makes future extensions easy. Furthermore, we devise a grounding algorithm for lde-safe programs which in contrast to traditional algorithms can ground any such program straight without the need for program decomposition. As the latter may still be useful for efficiency, we develop a new decomposition heuristics. Moreover, we generalize the grounding algorithm with application-specific termination criteria injected via *hooks*, such that *controlled grounding* is possible. This further increases usability and supports restricted model generation, which we exploit in showcases for answering $datalog$ queries with existential quantifiers on top of ASP programs with external sources and for answer set programs with function symbols. An experimental evaluation of lde-safety on various applications confirms the practicability of our approach.

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

Danksagungen: This research has been supported by the Austrian Science Fund (FWF) project P24090. Preliminary results of this work have been presented at AAAI 2013 [15], the 2nd GTTV workshop [14], and INAP 2013 [16].

Inhaltsverzeichnis

1 Introduction

Answer Set Programming (ASP) is a declarative programming approach which due to expressive and efficient systems like CLASP, DLV, and SMODELS, has been gaining popularity in several application areas, and in particular in artificial intelligence [6]. A problem at hand is represented by a set of rules (an ASP program) such that its models, called *answer sets*, encode the solutions to the problem. Compared to the similar approach of SAT solving, the rules might contain variables as a shortcut for all ground instances, transitive closure can be readily expressed as well as negation as failure; further extensions including optimization constructs, aggregates, preferences and many other features have turned ASP into an expressive and powerful problem solving tool.

Recent developments in computing require access from ASP programs to external sources, as information is increasingly stored in different sources and formats, or because complex, specific tasks can not be expressed directly or efficiently in the program itself. A prominent example are *DL-programs* [22], which integrate rules with description logic ontologies in a way such that queries to an ontology can be made in the rules; the formalism supports reasoning tasks which cannot be realized in ontologies alone, e.g., default classification. Another application with need for external access is planning in agent systems, which might require to import information from sensors and send commands back to agents, e.g. robots [39]; action or plan feasibility under physical or geometric constraints might be tested using special external libraries, etc. In other scenarios, the actions might be simple, but the planning domain is implicit in an external data structure; for example, in advanced route planning tasks for smart city applications [21], where Open Street Map data or some connection database may be used. Abstracting and accessing such data through an external interface is natural, as the data might not be fully accessible or too big to be simple added to the ASP program. Related to this is light-weight data access on the Web (e.g., XML, RDF, or other data repositories), which is getting more frequent and desired in complex applications, for instance in information integration; but like for a street map, a complete a priori data import is usually infeasible (in particular, in case of recursive data access). A concrete application scenario is, for instance, from the biomedical domain [23] where different online knowledge resources about genes, drugs and diseases are assessed in order to answer complex queries regarding their mutual relationships, e.g., for drugs that treat a certain disease while not targeting a particular gene. Finally, ASP is a popular host for experimental implementations of logic-based AI formalisms; however, the expressive capability of ordinary ASP may not be sufficient to cater a particular formalism, or a direct encoding in ASP may be cumbersome; in this case, it is convenient if some condition checks can be outsourced to external computation. For example, implementations of Dung-style semantics for abstract argumentation [10] (this will be more discussed below) or of multi-context systems [5], fall in this class.

To cater for the need of external source access, HEX-programs [18] extended ASP with so-called *external atoms*, through which the user can couple any external data source with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions, into an external source which returns output values of an (abstract) function that it computes. This convenient extension has been exploited for a growing range of applications, including those mentioned above; for a more detailed but not up to date account, see [12]. HEX-programs are highly expressive in general, as in addition to their high computational power, external atoms may introduce new constants that are not present in the program; this is commonly referred to as *value invention*. However, this feature makes the evaluation of ASP programs tricky and may lead to non-termination.

The predominant evaluation approach of current ASP solvers is grounding & search, which roughly speaking means that a ground (variable-free) version of the program is generated by substituting constants for variables, and thereafter an answer set of the resulting ground (propositional) program is searched; both

steps use quite sophisticated algorithms.

A naive support of value invention leads to infinite program groundings. For instance, the program

$$\Pi = \left\{ \begin{array}{ll} r_1 \colon p(a). & r_3 \colon s(Y) \leftarrow p(X), \&concat[X, a](Y). \\ r_2 \colon q(aa). & r_4 \colon p(X) \leftarrow s(X), q(X). \end{array} \right\}$$

where $\&concat[X, a](Y)$ returns in Y the string in X with a appended, has an infinite grounding assuming that the external source processes all finite strings over an alphabet. Yet it appears (and is easy to see for an ASP aficionado) that only ground rules using a, aa and aaa are relevant for program evaluation. However due to an API style interface, external sources are largely black boxes to an ASP solver. Thus while the relevant constants for grounding might be clear *intuitively*, they are not at the *formal level*, and predetermining them is in general not effectively possible (i.e., is undecidable). To ensure that a finite fragment of the program's grounding is faithful, i.e., has the same answer sets as the original program (referred to as *finite restrictability*), traditional approaches impose strong syntactic safety conditions on a program, such as strong domain-expansion safety [19] or VI-restrictedness [8]. However, they often limit expressiveness too much, i.e., programs may not fulfill the safety conditions while they are clearly finitely restrictable; the program Π above is a simple example.

In order to evaluate programs which violate safety, a common workaround is to use a *domain predicate* d, where each constant c from the external source is added by a fact $d(c)$ to the program and type literals $d(X)$ are added for "unsafe" variables X in rule bodies; that is, the constants which might be relevant for external source accesses are imported a priori into the program. E.g., for the realization of DL-programs [17] via HEX-programs, where the external atoms query a description logic ontology $\mathcal{O}$, the individuals (i.e., constants) occurring in $\mathcal{O}$ were added to the program in this way. However, this workaround is not only inconvenient, but also infeasible for large external domains.

This motivates us to introduce a *more liberal notion of safety* that still ensures finite restrictability of the program. However, rather than merely to generalize an existing notion of safety, we aim for a generic notion at a conceptual level which can incorporate besides syntactic also semantic information about sources, and which is flexible with regard to further generalizations and extensions. In combination with this notion, we present a *novel grounding algorithm* for the new class of programs. To enhance usability, we also present a generalized version of it which allows to customize the grounding, such that application specific properties can be exploited.

The contributions in this article are briefly summarized as follows:

- We introduce *liberal domain-expansion (lde) safety*, which is parameterized with *term bounding functions* (TBFs). Such functions embody criteria which ensure that only finitely many ground instances of a term expression in a program matter. The notion provides a generic framework in which TBFs can be modularly replaced and combined, which offers attractive flexibility and future extensibility. We provide sample TBFs which exploit like traditional approaches *syntactic structure*, but also TBFs that build on *semantic properties* of the program, hinging on cyclicity and meta-information; this allows us to cover the program Π above. Thanks to modularity, these TBFs can be fruitfully combined into a single, more powerful TBF. Notably, by resorting to lde-safety domain predicates may be dispensed.

- We present a new grounding algorithm for lde-safe programs. The algorithm is based on a grounder for ordinary ASP programs, which is iteratively called to enlarge the ground program until all relevant constants are respected; between the calls of the ordinary ASP grounder, external sources may be evaluated. The new algorithm is integrated into the model-building framework for HEX-programs, for which we also

develop a new evaluation heuristics that aims to balance between program decomposition and learning techniques.

• We consider some applications that take advantage of lde-safety and present an experimental evaluation. The applications include, among others, recursive processing of data structures, abstract argumentation frameworks, and route planning scenarios. The evaluation of our grounding algorithms will show that lde-safety not only relieves the user from writing domain predicates, but also leads to significantly better performance in many cases. In fact, the realization of some of our applications is impossible with the traditional notion of safety.

• We present a generalization of the grounding algorithm which features *hooks* that can be exploited to introduce application-specific stopping criteria, such that *controlled grounding* is possible. This provides the user with a means to generate groundings for computing *bounded models*, which can serve different purposes. One is to use a finite fragment of an infinite program grounding that suffices to answer certain reasoning tasks; as a showcase, we discuss *query answering over existential rules*. Another usage is convenient support for parametric notions of data structures (e.g., bounded arithmetic or strings of bounded length), deferring necessary domain restrictions from the HEX-program to the grounder. Furthermore, we consider an application to support *programs with function symbols*.

• We discuss a number of related notions of safety and find that lde-safety is already more general than many approaches using the TBFs presented here, and it allows to accommodoate others.

To summarize, lde-safety is a significant advance for ASP with external source access, which on the one hand improves existing applications, while on the other it empowers new applications that would not be possible without it; to wit, some route planning tasks that we consider are infeasible using ordinary ASP. This demonstrates the potential of our results and of the DLVHEX-system implementing them.

Organization. The remainder of this article is organized as follows. In the next section, we recall HEX-programs and strong domain-expansion safety. In Section 3, we introduce liberal domain safety and consider different ways to instantiate it. In Section 4, we present the new grounding algorithm and discuss its integration into the HEX-model building framework. Section 5 is devoted to implementation and an experimental evaluation. We then consider in Section 6 the generalization of the grounding algorithm, along with an application to query answering from datalog programs with existential quantification in rule heads. After the discussion of related work in Section 7, we conclude in Section 8 with a summary and open issues. In order not to distract from reading, proofs have been moved to A.

2 Preliminaries

We start with basic definitions and recall HEX-programs [18]. The signature consists of mutually disjoint sets $\mathcal{P}$ of predicates, $\mathcal{X}$ of external predicates, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. Note that $\mathcal{C}$ may contain constants that do not occur explicitly in a HEX program and can even be infinite.

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 $p(\mathbf{c})$. An *assignment* $\mathbf{A}$ is a consistent set of literals. We make the convention that if an assignment does not explicitly contain $\mathbf{T}a$ or $\mathbf{F}a$ for some atom a, i.e. the assignment is partial, then a is false wrt. $\mathbf{A}$. An *interpretation* is a complete assignment $\mathbf{A}$, i.e., for every atom a either $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$ holds.

2.1 Syntax

HEX-programs generalize (disjunctive) extended logic programs under the answer set semantics [27] with *external atoms* of form $\&g[\mathbf{X}](\mathbf{Y})$, where $\&g \in \mathcal{X}$, $\mathbf{X} = X_1, \ldots, X_\ell$ and each $X_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ is an *input parameter*, and $\mathbf{Y} = Y_1, \ldots, Y_k$ and each $Y_i \in \mathcal{C} \cup \mathcal{V}$ is an *output term*.

Each $p \in \mathcal{P}$ has arity $ar(p) \geq 0$ and each $\&g \in \mathcal{X}$ has input arity $ar_{\text{I}}(\&g) \geq 0$ and output arity $ar_{\text{O}}(\&g) \geq 0$. Each input argument i of $\&g$ ($1 \leq i \leq ar_{\text{I}}(\&g)$) has *type* $\mathbf{const}$ or $\mathbf{pred}$, denoted $\tau(\&g, i)$, where $\tau(\&g, i) = \mathbf{pred}$ if $X_i \in \mathcal{P}$ and $\tau(\&g, i) = \mathbf{const}$ otherwise.

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

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

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

The *head* of r is $H(r) = \{a_1, \ldots, a_n\}$, the *body* is $B(r) = B^+(r) \cup \text{not } B^-$, 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 O, etc., let $A(O)$ and $EA(O)$ be the set of all ordinary and external atoms occurring in O, respectively.

2.2 Semantics

The semantics of a HEX-program Π is defined via its grounding $grnd(\Pi)$ (over $\mathcal{C}$) as usual, where the value of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an interpretation $\mathbf{A}$ is given by the value $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ of a $k+l+1$-ary Boolean *oracle function* $f_{\&g}$ [18]. The input parameter $p_i \in \mathbf{p}$ is *monotonic* if, $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) \leq f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c})$ whenever $\mathbf{A}'$ increases $\mathbf{A}$ only by literals $\mathbf{T}a$ where a has predicate p_i; otherwise, p_i is *non-monotonic*.

Satisfaction of (sets of) ground literals, rules, programs etc. O wrt. $\mathbf{A}$ (denoted $\mathbf{A} \models O$, i.e., $\mathbf{A}$ is a *model* of O) extends naturally from ordinary logic programs to HEX-programs, by taking external atoms into account. That is, for every ordinary atom a, $\mathbf{A} \models a$ if $\mathbf{T}a \in \mathbf{A}$, and $\mathbf{A} \not\models a$ if $\mathbf{F}a \in \mathbf{A}$, and for every external atom $a = \&g[\mathbf{p}](\mathbf{c})$, $\mathbf{A} \models a$ if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$. For a rule r of form (1), $\mathbf{A} \models r$ if either $\mathbf{A} \models a_i$ for some $1 \leq i \leq k$, $\mathbf{A} \models b_j$ for some $m < j \leq n$, or $\mathbf{A} \not\models b_j$ for some $1 \leq j \leq m$. Finally, $\mathbf{A} \models \Pi$, if $\mathbf{A} \models r$ for every $r \in \Pi$. An *answer set* of a HEX-program Π is any model $\mathbf{A}$ of the *FLP-reduct* $\Pi^{\mathbf{A}}$ of Π wrt. $\mathbf{A}$, given by $\Pi^{\mathbf{A}} = \{r \in grnd(\Pi) \mid \mathbf{A} \models B(r)\}$ [24], whose positive part $\{\mathbf{T}a \in \mathbf{A}\}$ is subset-minimal, i.e., there exists no model $\mathbf{A}'$ of $\Pi^{\mathbf{A}}$ such that $\{\mathbf{T}a \in \mathbf{A}'\} \subset \{\mathbf{T}a \in \mathbf{A}\}$.[1] The set of all answer sets of Π is denoted by $\mathcal{AS}(\Pi)$.

Example 1 (cont'd) Reconsider the program Π from the Introduction, and consider the interpretation $\mathbf{A} = \{\mathbf{T}dom(aa), \mathbf{T}t(a), \mathbf{T}s(aa), \mathbf{T}t(aa), \mathbf{T}s(aaa)\}$ (by the convention from above, all atoms not occurring in $\mathbf{A}$ are false). It can be seen that $\mathbf{A}$ is an answer set of Π. Indeed, $\Pi^{\mathbf{A}}$ (which in abuse of notation denotes the FLP-reduct of the grounding of Π wrt. $\mathbf{A}$) is

$$\Pi^{\mathbf{A}} = \left\{ \begin{array}{ll} t(a). & s(aa) \leftarrow t(a), \&concat[a, a](aa). \\ dom(aa). & s(aaa) \leftarrow t(aa), \&concat[aa, a](aaa). \\ & t(aa) \leftarrow s(aa), dom(aa). \end{array} \right\}.$$

[1] The FLP-reduct is equivalent to the traditional Gelfond-Lifschitz reduct for ordinary logic programs [27], but more attractive for extensions such as aggregates or external atoms.

Clearly, $\mathbf{A} \models \Pi^{\mathbf{A}}$, and no atoms can be switched to false such that the resulting interpretation $\mathbf{A}'$ fulfills $\mathbf{A}' \models \Pi^{\mathbf{A}}$; hence, $\mathbf{A}$ is an answer set of Π. In fact, $\mathbf{A}$ is the only answer set of Π, i.e., $\mathcal{AS}(\Pi) = \{\mathbf{A}\}$. □

Note that in the previous example, a finite portion of the grounding of the program is sufficient to single out its answer sets. This property, which we call *finite restrictability*, is essential for computing the answer in finite time. More formally, let us for programs Π' and Π write $\Pi' \equiv^{pos} \Pi$, if $\mathcal{AS}(\Pi')$ and $\mathcal{AS}(\Pi)$ coincide on positive literals. Then,

Definition 1 *A program* Π *is* finitely restrictable*, if there exists some finite subset* $\Pi' \subseteq grnd(\Pi)$ *such that* $\Pi' \equiv^{pos} \Pi$.

As finite restrictability of a program is clearly undecidable in general, we are interested in decidable cases where Π' can be effectively computed from Π. To this end, suitable notions of safety are useful.

2.3 Safety

The notion of safety has been introduced in the context of logic programming on data bases (viewed as facts of a logic program), to prevent that datalog queries produce infinite results. A program is *safe*, if each variable in a rule r occurs also in a positive body atom in $B^+(r)$. However, due to external atoms, we need additional safety criteria.

Example 2 Let $\&concat[X, a](Y)$ be true iff Y is the string catenation of X and a. Then $\Pi = \{s(a).\ s(Y) \leftarrow s(X), \&concat[X, a](Y)\}$ is safe but not finitely restrictable. Note that Π has the single (infinite) answer set $\{\mathbf{T}s(a^i) \mid i \geq 1\}$. □

Thus the notion of *strong safety* was introduced by [19], which limits the output of cyclic external atoms. In hinges on the concept of *external atom dependencies*.

Definition 2 (External Atom Dependencies) *Let* Π *be a* HEX*-program and leta* $a = \&g[\mathbf{X}](\mathbf{Y})$ *be an external atom in* Π.

- *If* $b = p(\mathbf{Z}) \in \bigcup_{r\in\Pi} H(r)$*, then* a depends external monotonically (resp. nonmonotonically) on b*, denoted* $a \rightarrow^e_m b$ *(resp.* $a \rightarrow^e_n b$*), if* $X_i = p$ *for some monotonic (resp. nonmonotonic) parameter* $X_i \in \mathbf{X}$ $(= X_1, .., X_\ell)$.
- *If* $\{a,\ p(\mathbf{Z})\} \subseteq B^+(r)$*, some* $X_i \in \mathbf{X}$ *occurs in* $\mathbf{Z}$*, and* $\tau(\&g, i) = \mathbf{const}$*, then* $\&g[\mathbf{X}](\mathbf{Y}) \rightarrow^e_m p(\mathbf{Z})$.
- *If* $\{a,\ \&h[\mathbf{V}](\mathbf{U})\} \subseteq B^+(r)$*, some* $X_i \in \mathbf{X}$ *occurs in* $\mathbf{U}$*, and we have* $\tau(\&g, i) = \mathbf{const}$*, then* $\&g[\mathbf{X}](\mathbf{Y}) \rightarrow^e_m \&h[\mathbf{V}](\mathbf{U})$.

Definition 3 (Atom Dependencies) *For a* HEX*-program* Π *and (ordinary or external) atoms* a, b*, we say that*

(i) a depends monotonically on b*, denoted* $a \rightarrow_m b$*, if:*

- *some rule* $r \in \Pi$ *has* $a \in H(r)$ *and* $b \in B^+(r)$*; or*
- *there are rules* $r_1, r_2 \in \Pi$ *such that* $a \in B(r_1)$ *and* $b \in H(r_2)$ *and* a *unifies with* b*; or*
- *some rule* $r \in \Pi$ *has* $a \in H(r)$ *and* $b \in H(r)$.

(ii) a depends nonmonotonically on b, *denoted* $a \rightarrow_n b$, *if there is some rule* $r \in \Pi$ *such that* $a \in H(r)$ *and* $b \in B^-(r)$.

The following definition represents these dependencies.

Definition 4 (Atom Dependency Graph) *Given a* HEX*-program* Π*, its* atom dependency graph $ADG(\Pi)$ *has as nodes* V_A *the (nonground) atoms occurring in non-facts* r *(i.e.,* $k \neq 1$ *or* $n > 0$*) of* Π *and as edges* E_A *the dependency relations* $\rightarrow_m, \rightarrow_n, \rightarrow_m^e, \rightarrow_n^e$ *between these atoms in* Π.

This allows us to introduce strong safety as follows.

Definition 5 (Strong Safety) *An atom* $b = \&g[\mathbf{X}](\mathbf{Y})$ *in a rule* r *of a program* Π *is* strongly safe *wrt.* r *and* Π*, if either there is no cyclic dependency over* b *in* $ADG(\Pi)$*, or every variable in* $\mathbf{Y}$ *occurs also in a positive ordinary atom* $a \in B^+(r)$ *not depending on* b *in* $ADG(\Pi)$*. A program* Π *is strongly safe, if every external atom in a rule* $r \in \Pi$ *is strongly safe wrt.* r *in* Π.

Example 3 (cont'd) Reconsider the program Π of Example 2. The external atom $a = \&concat[X, a](Y)$ is not strongly safe because it is in a cycle and no ordinary body atom contains Y. However, a would be strongly safe, and hence also Π, if the rule body would contain an atom $p(Y)$ and p would occur in Π apart from this only in facts. Likewise, the program Π in the Introduction is not strongly safe as the external atom $\&concat[X, a](Y)$ is not safe. □

The notion of *strong domain-expansion safety* is then as follows.

Definition 6 (Strong Domain-expansion Safety [19]) *A program* Π *is* strongly domain-expansion (sde) safe*, if it is safe and each external atom occurring in any rule* r *of* Π *is strongly safe wrt.* r *and* Π.

For this notion of safety, [19] established the following result.

Proposition 1 *Every strongly domain-expansion safe* HEX*-program* Π *is finitely restrictable.*

More in detail, they gave an algorithm to construct a finite portion Π' of the grounding of such a Π with the property $\Pi' \equiv^{pos} \Pi$; this algorithm has been implemented in the DLVHEX-system.

While many HEX-programs are sde-safe, there are simple programs which lack this property.

Example 4 (cont'd) The program in Example 2 and the program Π in Section 1 are not sde-safe, as external atoms that occur in them are not strongly safe. Furthermore, for both programs adding a domain predicate (as described in Section 1 is infeasible as infinitely many domain facts $d(\sigma)$ would be added, one for each string σ. □

However, the program Π in Section 1 has only finite answer sets, and no infinite domain predicate seems necessary to single out all relevant rules instances; this motivates a more general notion of safety.

3 Liberal Safety

Strong domain-expansion safety is overly restrictive, as it also excludes programs that clearly *are* finitely restrictable.

Example 5 Reconsider the program Π from Section 1. It is not strongly domain-expansion safe because Y in r_3 does not occur in an ordinary body atom that does not depend on $\&concat[X, a](Y)$. However, Π is finitely restrictable as the cycle is "broken" by $dom(X)$ in r_4. □

In this section, we introduce a new notion of *liberal domain-expansion safety* which incorporates both syntactic and semantic properties of the program at hand. In the following, *domain-expansion safety* (*de-safety*) refers to liberal domain-expansion safety, unless we explicitly say strong domain-expansion safety. Compared to the latter, this gives us a larger class of programs which are guaranteed to have a finite grounding that preserves all answer sets. Unlike strong de-safety, liberal de-safety is not a property of entire atoms but of *attributes*, i.e., pairs of predicates and argument positions. Intuitively, an attribute is lde-safe, if the number of different terms in an answer-set preserving grounding (i.e. a grounding which has the same answer sets if restricted to the positive atoms as the original program) is finite. A program is lde-safe, if all its attributes are lde-safe.

Our notion of lde-safety is designed in an extensible fashion, i.e., such that several safety criteria can be easily integrated. For this we parameterize our definition of lde-safety by a *term bounding function (TBF)*, which identifies variables in a rule that are ensured to have only finitely many instantiations in the answer set preserving grounding. Finiteness of the overall grounding follows then from the properties of TBFs. Concrete syntactic and semantic properties are realized in some sample TBFs (cf. Section 3.1).

For an ordinary predicate $p \in \mathcal{P}$, let $p\restriction i$ be the *i-th attribute of p* for all $1 \leq i \leq ar(p)$. For an external predicate $\&g \in \mathcal{X}$ with input list $\mathbf{X}$ in rule r, let $\&g[\mathbf{X}]_r\restriction_T i$ with $T \in \{\text{I}, \text{O}\}$ be the *i-th input resp. output attribute of $\&g[\mathbf{X}]$ in r* for all $1 \leq i \leq ar_T(\&g)$. For a ground program P, the *range* of an attribute is, intuitively, the set of ground terms which occur in the position of the attribute. Formally, for an attribute $p\restriction i$ we have $range(p\restriction i, \Pi) = \{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(\Pi)\}$; for an attribute $\&g[\mathbf{X}]_r\restriction_T i$ we have $range(\&g[\mathbf{X}]_r\restriction_T i, \Pi) = \{x_i^T \mid \&g[\mathbf{x}^{\text{I}}](\mathbf{x}^{\text{O}}) \in EA(\Pi)\}$, where $\mathbf{x}^s = x_1^s, \ldots, x_{ar_s(\&g)}^s$.

Example 6 Reconsider the program Π from Section 1. Examples for attributes are $t\restriction 1$, $\&concat[X, a]_{r_3}\restriction_{\text{I}} 2$ and $\&concat[X, a]_{r_3}\restriction_{\text{O}} 1$. Furthermore, $range(t\restriction 1, \Pi) = \{a\}$.

We use the following monotone operator to compute by fixpoint iteration a finite subset of $grnd(\Pi)$ for a program Π:

$$G_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r\theta \mid \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r\theta)\},$$

where $\mathcal{A}(\Pi') = \{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi')\} \setminus \{\mathbf{F}a \mid a \leftarrow . \in \Pi\}$ and $r\theta$ is the ground instance of r under variable substitution $\theta\colon \mathcal{V} \to \mathcal{C}$. Note that in this definition, $\mathbf{A}$ might be partial, but by convention we assume that all atoms which are not explicitly assigned to true are false. That is, G_Π takes a ground program Π' as input and returns all rules from $grnd(\Pi)$ whose positive body is satisfied under some assignment over the atoms of Π'. Intuitively, the operator iteratively extends the grounding by new rules if they are possibly relevant for the evaluation, where relevance is in terms of satisfaction of the positive rule body under some assignment constructible over the atoms which are possibly derivable so far. Obviously, the least fixpoint $G_\Pi^\infty(\emptyset)$ of this operator is a subset of $grnd(\Pi)$; we will show that it is finite if Π is lde-safe according to our new notion. Moreover, we will show that this grounding preserves all answer sets because all rule instances which are not added have unsatisfied bodies anyway.

Example 7 Consider the following program Π:

$r_1\colon s(a). \quad r_2\colon dom(ax). \quad r_3\colon dom(axx).$
$r_4\colon s(Y) \leftarrow s(X), \&concat[X, x](Y), dom(Y).$

The least fixpoint of G_Π is the following ground program:

$r'_1\colon\ \ s(a).\quad r'_2\colon\ dom(ax).\quad r'_3\colon\ dom(axx).$
$r'_4\colon\ \ s(ax) \leftarrow s(a), \&concat[a,x](ax), dom(ax).$
$r'_5\colon s(axx) \leftarrow s(ax), \&concat[ax,x](axx), dom(axx).$

Rule r'_4 is added in the first iteration and rule r'_5 in the second.

Towards a definition of lde-safety, we say that a term in a rule is *bounded*, if the number of substitutions in $G^\infty_\Pi(\emptyset)$ for this term is finite. This is abstractly formalized using *term bounding functions*.

Definition 7 (Term Bounding Function (TBF)) *A* term bounding function*, denoted $b(\Pi, r, S, B)$, 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 of (bounded) terms $b(\Pi, r, S, B) \supseteq B$, such that every $t \in b(\Pi, r, S, B)$ has finitely many substitutions in $G^\infty_\Pi(\emptyset)$ if (i) the attributes S have a finite range in $G^\infty_\Pi(\emptyset)$ and (ii) each term in $terms(r) \cap B$ has finitely many substitutions in $G^\infty_\Pi(\emptyset)$.*

Intuitively, a TBF receives a set of already bounded terms and a set of attributes that are already known to be lde-safe. Taking the program into account, the TBF then identifies and returns further terms which are also bounded.

Our concept yields lde-safety of attributes and programs from the boundedness of variables according to a TBF. We provide a mutually inductive definition that takes the empty set of lde-safe attributes $S_0(\Pi)$ as its basis. Then, each iteration step $n \geq 1$ defines first the set of bounded terms $B_n(r, \Pi, b)$ for all rules r, and then an enlarged set of lde-safe attributes $S_n(\Pi)$. The set of lde-safe attributes in step $n+1$ thus depends on the TBF, which in turn depends on the domain-expansion safe attributes from step n.

Definition 8 (Liberal Domain-Expansion Safety) *Let b be a term bounding function. The set $B_n(r, \Pi, b)$ of* bounded terms *in a rule $r \in \Pi$ in step $n \geq 1$ 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 all $j \geq 0$, $B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j})$.*

The set of domain-expansion safe attributes $S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$ *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 $p(t_1, \ldots, t_{ar(p)}) \in H(r)$, term $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is* bounded*;*
- *$\&g[\mathbf{X}]_r{\restriction}_\mathrm{I} i \in S_{n+1}(\Pi)$ if each $\mathbf{X}_i$ is a* bounded *variable, or $\mathbf{X}_i$ is a predicate input parameter p and $p{\restriction}1, \ldots, p{\restriction}ar(p) \in S_n(\Pi)$;*
- *$\&g[\mathbf{X}]_r{\restriction}_\mathrm{O} i \in S_{n+1}(\Pi)$ if and only if r contains an external atom $\&g[\mathbf{X}](\mathbf{Y})$ such that $\mathbf{Y}_i$ is bounded, or $\&g[\mathbf{X}]_r{\restriction}_\mathrm{I} 1, \ldots, \&g[\mathbf{X}]_r{\restriction}_\mathrm{I} ar_\mathrm{I}(\&g) \in S_n(\Pi)$.*

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

We sometimes omit "liberally" and refer with domain-expansion safe to lde safe programs. An example is delayed until we have introduced sample TBFs in Section 3.1. However, the intuition is as follows. In each step, the TBF first provides further terms that are bounded (given the information assembled in previous iterations), exploiting e.g. syntactic or semantic criteria. This possibly makes additional attributes lde-safe

(cf. the conditions for $S_n(\Pi)$ in Definition 8 above), which in turn may cause further terms to become bounded in the next iteration step.

We next show that $S_\infty(\Pi)$ is finite, thus the inductive definition can be used for computing $S_\infty(\Pi)$: the iteration can be aborted after finitely many steps. We first note some desired properties.

Proposition 2 *The set $S_\infty(\Pi)$ is finite.*

Moreover, lde-safe attributes have a finite range in $G_\Pi^\infty(\emptyset)$.

Proposition 3 *For every TBF b and $n \geq 0$, if $\alpha \in S_n(\Pi)$, then the range of α in $G_\Pi^\infty(\emptyset)$ is finite.*

Corollary 4 *If $\alpha \in S_\infty(\Pi)$, then $range(\alpha, G_\Pi^\infty(\emptyset))$ is finite.*

This means that such attributes occur with only finitely many arguments in the grounding computed by G_Π. This result implies that also the whole grounding $G_\Pi^\infty(\emptyset)$ is finite.

Corollary 5 *If Π is a lde-safe program, then $G_\Pi^\infty(\emptyset)$ is finite.*

As follows from these propositions, $S_\infty(\Pi)$ is also finitely constructible. Note that the propositions hold independently of a concrete TBF because the properties of TBFs are sufficiently strong. This allows for a modular exchange or combination of the TBFs, as long as the preconditions of TBFs are satisfied, without changing the definition of lde-safety.

We now make use of the results from above to show that lde-safe programs are finitely restrictable. However, we remark that the following proposition does not directly lead to an efficient implementation; the algorithm presented in Section 4 makes use of several optimizations.

Proposition 6 *Every lde-safe program Π is finitely restrictable, and it holds that $G_\Pi^\infty(\emptyset) \equiv^{pos} \Pi$.*

This proposition holds independently of a concrete term bounding function. However, functions that are too liberal are excluded by the preconditions in the definition of TBFs.

3.1 Sample Term Bounding Functions

We now introduce sample term bounding functions that exploit syntactic and semantic properties of external atoms to guarantee boundedness of variables. By our previous result, this ensures also finiteness of the ground program computed by $G_\Pi^\infty(\emptyset)$.

3.1.1 Syntactic Criteria

We first identify syntactic properties that can be exploited for our purposes.

Definition 9 (Syntactic Term Bounding Function) *We define $b_{syn}(\Pi, r, S, B)$ such that $t \in b_{syn}(\Pi, r, S, B)$ iff*

(i) t is a constant in r; or

(ii) there is an ordinary atom $q(s_1, \ldots, s_{ar(q)}) \in B^+(r)$ such that $t = s_j$, for some $1 \leq j \leq ar(q)$ and $q{\restriction}j \in S$; or

(iii) for some external atom $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$*, we have that* $t = Y_i$ *for some* $Y_i \in \mathbf{Y}$*, and for each* $X_i \in \mathbf{X}$*,*

$$\begin{cases} X_i \in B, & \text{if } \tau(\&g, i) = \mathbf{const}, \\ X_i\restriction 1, \ldots, X_i\restriction ar(X_i) \in S, & \text{if } \tau(\&g, i) = \mathbf{pred}. \end{cases}$$

Intuitively, (i) a constant is trivially bounded because it is never substituted by other terms in the grounding. Case (ii) states that terms occurring at lde-safe attribute positions are bounded; more spcifically, the fact that an attribute $q\restriction j$ (where $1 \leq j \leq ar(q)$) is lde-safe, and thus has a finite range in $G^{\infty}_{\Pi}(\emptyset)$, implies that the term at this attribute position is bounded. Case (iii) essentially expresses that if the input to an external atom is finite, then also its output is finite.

Proposition 7 *The function* $b_{syn}(\Pi, r, S, B)$ *is a TBF.*

Example 8 (cont'd) Consider Π from Example 7 to compute the sets of safe attributes. We get $S_1(\Pi) = \{dom\restriction 1, \&concat[X, x]_{r_4}\restriction_{\mathrm{I}} 2\}$, as $B_1(r_2, \Pi, b_{syn}) = \{ax\}$, $B_1(r_3, \Pi, b_{syn}) = \{axx\}$ and $B_1(r_4, \Pi, b_{syn}) = \{x\}$ (by item (i) in Definition 9), i.e., the derived terms in all rules that have $dom\restriction 1$ in their head are known to be bounded. In the next iteration, we get $B_2(r_4, \Pi, b_{syn}) = \{Y\}$ (by item (ii) in Definition 9) as $dom\restriction 1$ is already known to be lde-safe. Since we also have $B_2(r_1, \Pi, b_{syn}) = \{a\}$, the terms derived by r_1 and r_4 are bounded, hence $s\restriction 1 \in S_2(\Pi)$. Moreover, $\&concat[X, x]_{r_4}\restriction_{\mathrm{O}} 1 \in S_2(\Pi)$ because Y is bounded. The third iteration yields $\&concat[X, x]_{r_4}\restriction_{\mathrm{I}} 1 \in S_3(\Pi)$ because $X \in B_3(r_4, \Pi, b_{syn})$ due to item (ii) in Definition 9. Thus, all attributes are lde-safe. □

3.1.2 Semantic Properties

We now define a TBF exploiting meta-information about external sources in four properties.

The first property is based on *malign cycles* in the *positive attribute dependency graphs*, which are the source of any infinite value invention. The *positive attribute dependency graph* $G_A(\Pi)$ has as nodes the attributes of Π and its edges model the information flow between the attributes. For instance, if for rule r we have $p(\mathbf{X}) \in H(r)$ and $q(\mathbf{Y}) \in B^+(r)$ such that $X_i = Y_j$ for some $X_i \in \mathbf{X}$ and $Y_j \in \mathbf{Y}$, then we have a flow from $q\restriction j$ to $p\restriction i$.

Formally, the positive attribute dependency graph is defined as follows.

Definition 10 (Positive Attribute Dependency Graph) *For a given* HEX*-program* Π*, the* positive attribute dependency graph $G_A(\Pi) = \langle Attr, E\rangle$ *has as nodes* $Attr$ *the set of all attributes in* Π *and the least set of edges* E *such that for all* $r \in \Pi$*:*

- *If* $p(\mathbf{X}) \in H(r)$*,* $q(\mathbf{Y}) \in B^+(r)$ *and for some* i, j *we have that* $X_i = Y_j$ *is a variable, then* $(q\restriction j, p\restriction i) \in E$*.*
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$*,* $p(\mathbf{Z}) \in B^+(r)$ *and for some* i, j *we have that* $Z_i = X_j$ *and* $\tau(\&g, i) = \mathbf{const}$*, then* $(p\restriction i, \&g[\mathbf{X}]_r\restriction_{\mathrm{I}} j) \in E$*.*
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$*,* $\&h[\mathbf{V}](\mathbf{U}) \in B^+(r)$ *and for some* i, j *we have that* $V_i = Y_j$ *and* $\tau(\&h, i) = \mathbf{const}$*, then* $(\&g[\mathbf{X}]_r\restriction_{\mathrm{O}} j, \&h\restriction_{\mathrm{I}} i) \in E$*.*
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$ *then* $(\&g[\mathbf{X}]_r\restriction_{\mathrm{I}} i, \&g[\mathbf{X}]_r\restriction_{\mathrm{O}} j) \in E$ *for all* $1 \leq i \leq iar(\&g)$ *and* $1 \leq j \leq oar(\&g)$*..*

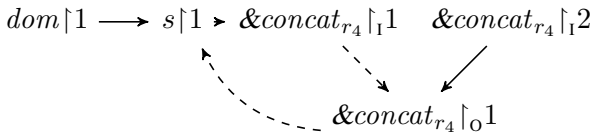


Abbildung 1: Attribute dependency graph for Π in Example 7

- *If* $p(\mathbf{X}) \in H(r)$, $\&g[\mathbf{Y}](\mathbf{Z}) \in B^+(r)$ *and and for some* i, j *we have that* $X_i = Z_j$ *is a variable, then* $(\&g[\mathbf{X}]_r\restriction_{\mathrm{O}} j, p\restriction i) \in E$.
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$ *s.t.* $p = X_i$ *and* $\tau(\&g, i) = \mathbf{pred}$*, then we have* $(p\restriction k, \&g[\mathbf{X}]_r\restriction_{\mathrm{I}} i) \in E$ *for all* $1 \leq k \leq ar(p)$.

The transitive closure of the edge relation in G_A is denoted by $\rightarrow^+_{G_A}$. Intuitively, $G_A(\Pi)$ models the information flow in Π.

Example 9 (cont'd) The (positive) attribute dependency graph of the program from Example 7 has the attributes $Attr = \{s\restriction 1, dom\restriction 1, \&concat_{r_4}\restriction_{\mathrm{I}} 1, \&concat_{r_4}\restriction_{\mathrm{I}} 2, \&concat_{r_4}\restriction_{\mathrm{O}} 1\}$ and the edges $\{(dom\restriction 1,\, s\restriction 1)$, $(s\restriction 1, \&concat_{r_4}\restriction_{\mathrm{I}} 1)$, $(\&concat_{r_4}\restriction_{\mathrm{I}} 1, \&concat_{r_4}\restriction_{\mathrm{O}} 1)$, $(\&concat_{r_4}\restriction_{\mathrm{I}} 2, \&concat_{r_4}\restriction_{\mathrm{O}} 1)$, $(\&concat_{r_4}\restriction_{\mathrm{O}} 1,\, s\restriction 1)\}$ (see Figure 1).

Definition 11 (Benign and Malign Cycles) *A cycle* K *in* $G_A(\Pi)$ *is* benign *wrt. a set of safe attributes* S*, if there exists a well-ordering* $\leq_{\mathcal{C}}$ *of* $\mathcal{C}$*, such that for every attribute* $\&g[\mathbf{X}]_r\restriction_{\mathrm{O}} j \notin S$ *in the cycle,* $f_{\&g}(\mathbf{A}, x_1, \ldots, x_m, t_1, \ldots t_n) = 0$ *whenever*

- *some* x_i *for* $1 \leq i \leq m$ *is a predicate parameter,* $\&g[\mathbf{X}]_r\restriction_{\mathrm{I}} i \notin S$ *is in* K*, and we have* $(s_1, \ldots, s_{ar(x_i)}) \in ext(\mathbf{A}, x_i)$*, and* $t_j \not\leq_{\mathcal{C}} s_k$ *for some* $1 \leq k \leq ar(x_i)$*; or*
- *some* x_i *for* $1 \leq i \leq m$ *is a constant input parameter,* $\&g[\mathbf{X}]_r\restriction_{\mathrm{I}} i \notin S$ *is in* K*, and* $t_j \not\leq_{\mathcal{C}} x_i$.

A cycle in $G_A(\Pi)$ *is called* malign *wrt.* S *if it is not benign.*

Intuitively, a cycle is benign if external atoms never deliver larger values wrt. to their yet unsafe cyclic input. As there is a least element, this ensures a finite grounding.

Example 10 (cont'd) The cycle in $G_A(\Pi)$ (dashed lines in Figure 1) is malign wrt. $S = \emptyset$ because there is no well-ordering as required by Definition 11. Intuitively, this is because the external atom infinitely extends the string.

If we replace $\&concat[X, x](Y)$ in Π by $\&tail[X](Y)$, i.e., we compute the string Y from X with the first character removed, then the cycle in the adapted attribute dependency graph becomes benign using $<$ over the string lengths as well-ordering. □

Two other properties involve meta-information that directly ensures an output attribute of an external source is finite.

Definition 12 (Finite Domain) *An external predicate* $\&g \in \mathcal{X}$ *has the* finite domain property wrt. output argument $i \in \{1, \ldots, ar_{\mathrm{O}}(\&g)\}$*, if* $\{y_i \mid \mathbf{x} \in (\mathcal{P} \cup \mathcal{C})^{ar_{\mathrm{I}}(\&g)}, \mathbf{y} \in \mathcal{C}^{ar_{\mathrm{O}}(\&g)}, f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y}) = 1\}$ *is finite for all assignments* $\mathbf{A}$.

Here, the provider of the external source explicitly states that the output at a certain position in the output tuple is finite. This is perhaps the most direct way to ensure boundedness of the respective term.

Example 11 An external atom $\&md5[S](Y)$ computing the MD5 hash value Y of a string S is finite domain wrt. the (single) output element, as its domain is finite (yet very large). □

While the previous properties derive boundedness of an output term of an external atom from finiteness of its input, we now reverse the direction. An external atom may have the property that only a finite number of different inputs can yield a certain output, which is formalized as follows.

Definition 13 (Finite Fiber) *An external predicate* $\&g \in \mathcal{X}$ *has the* finite fiber property*, if the set* $\{\mathbf{x} \in (\mathcal{P} \cup \mathcal{C})^{ar_{\mathrm{I}}(\&g)} \mid f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y}) = 1\}$ *is finite for every* $\mathbf{A}$ *and* $\mathbf{y} \in \mathcal{C}^{ar_{\mathrm{O}}(\&g)}$.

Example 12 Let $\&sq[X](S)$ be an external atom that computes the square S of the integer X. Then for a given S, there are at most two distinct values for X. □

The four properties above lead to the following TBF.

Definition 14 (Semantic Term Bounding Function) *We define* $b_{sem}(\Pi, r, S, B)$ *such that* $t \in b_{sem}(\Pi, r, S, B)$ *holds iff*

(i) t *is captured by some attribute* α *in* $B^+(r)$ *that is not reachable from malign cycles in* $G_A(\Pi)$ *wrt.* S*, i.e., if* $\alpha = p{\restriction}i$ *then* $t = t_i$ *for some body atom* $p(t_1, \ldots, t_\ell) \in B^+(r)$*, and if* $\alpha = \&g[\mathbf{Y}]_r{\restriction}_T i$ *then* $t = Y_i^T$ *for some external atom* $\&g[\mathbf{Y}^{\mathrm{I}}](\mathbf{Y}^{\mathrm{O}}) \in B^+(r)$ *where* $\mathbf{Y}^T = X_1^T, \ldots, Y_{ar(\&g)}^T$*; or*

(ii) $t = X_i$ *for some* $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$*, where* $\&g$ *has the* finite domain property in i*; or*

(iii) $t = X_i$ *for some* $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$*, where* $\&g$ *has the* relative finite domain property in output argument i and predicate input argument j *and* $Y_j{\restriction}k \in S$ *for all* $1 \leq k \leq ar(Y_j)$*; or*

(iv) $t \in \mathbf{Y}$ *for some* $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$*, where* $X \in B$ *for every* $X \in \mathbf{X}$ *and* $\&g$ *has the* finite fiber property.

This TBF is directly motivated by the properties introduced above.

Proposition 8 *Function* $b_{sem}(\Pi, r, S, B)$ *is a TBF.*

3.2 Modular Combinations of Term-Bounding Functions

For an attractive framework it is important that a certain degree of flexibility is achieved in terms of composability of TBFs. The following proposition allows us to construct TBFs modularly from multiple TBFs and thus ensures future extensibility by, e.g., customized application-specific TBFs.

Theorem 9 *If* $b_i(\Pi, r, S, B)$*,* $1 \leq i \leq \ell$*, are term bounding functions, then the union* $b(\Pi, r, S, B) = \bigcup_{1 \leq i \leq \ell} b_i(\Pi, r, S, B)$ *is a TBF.*

In particular, a TBF which exploits syntactic and semantic properties simultaneously is

$$b_{s^2}(\Pi, r, S, B) = b_{syn}(\Pi, r, S, B) \cup b_{sem}(\Pi, r, S, B),$$

which we will use subsequently.

3.3 Domain predicates

Recall that, as stated in the introduction, using domain predicates d is a common technique to ensure strong safety of a HEX-program (which does, however, not work for the program Π in Section 1). For instance this technique was applied in implementing DL-programs [17] and terminological default theories [1] in DLV-HEX using the DL-plugin, which provides generic external atoms for querying description logic ontologies. However, exploiting lde-safety, sometimes domain predicates may be dropped. We illustrate this with an example.

Example 13 (Bird-Penguin) We consider here a simple DL-program $(\Pi, \mathcal{O})$ which can be viewed as a HEX-program Π (left side) with access to an external ontology $\mathcal{O}$ (right side) containing the conceptual knowledge that penguins are birds and do not fly, and the assertion (data) that lia is a bird; the rules express that birds fly unless the opposite is derivable:

	Rules Π:	Ontology $\mathcal{O}$:	
r_1 :	$birds(X) \leftarrow DL[Bird](X).$	$Flier \sqsubseteq \neg NonFlier$	$Bird(lia)$
r_2 :	$flies(X) \leftarrow birds(X), \text{not } neg\_flies(X).$	$Penguin \sqsubseteq Bird$	
r_3 :	$neg\_flies(X) \leftarrow DL[Flier \uplus flies; \neg Flier](X).$	$Penguin \sqsubseteq NonFlier$	

Here the expressions $DL[\cdots](X)$ are so called DL-atoms, which in the HEX-view are just user-friendly syntax for external atoms $\&dlC[\cdots](X)$ whose input parameters consist of a query (a concept name) and optional additions of facts (assertions) to the ontology prior to query evaluation. To determine the birds that fly, rule r_1 retrieves all birds known by the ontology using a DL-atom $DL[Bird](X)$. Intuitively, the DL-atom $DL[Flier \uplus flies; \neg Flier]$ returns all individuals in $\neg Flier$ assuming that the concept $Flier$ is augmented with the extension of the predicate $flies$. The rules r_2 and r_3 encode then the conclusion that a bird flies by default. In particular, this is concluded for lia, as the program has the single answer set $\{bird(lia), flies(lia)\}$.

While the program Π is safe, it is not strongly safe; this is because the external atom $DL[Flier \uplus flies; \neg Flier](X)$ in rule r_3 is involved in a cycle through negation. This can be remedied using a domain predicate d as described, by adding the literal $d(X)$ in the body of r_3 and the fact $d(lia)$. Alternatively, assuming that all individuals in the ontology are birds, strong safety is gained by using $bird$ as domain predicate and simply adding $bird(X)$ in the body of r_3; we denote the resulting rule by r_3' and the resulting program by Π'. In fact, little reflection reveals that projected on $flies$, the answer sets of Π' remain the same even if this assumption is not made.

On the other hand, Π is lde-safe, as X in $DL[Flier \uplus flies; \neg Flier](X)$ can take only finitely many values. This relieves the user from using a domain predicate, which—even if possible—is often cumbersome in practice. □

4 Grounding Liberally Domain-expansion Safe HEX-Programs

In this section we present a grounding algorithm for liberally domain-expansion safe HEX-programs. It is based on the iteratively grounding the input program and then checking whether 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 that are all demonstrated in Example 14 below. We assume that rules are standardized apart (i.e., share no variables). 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 15 (Liberal Domain-expansion Safety Relevance) *A set R of external atoms is* relevant for lde-safety *of a program Π, if $\Pi|_R$ is lde-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 lde-safety 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 16 (Input Auxiliary Rule) *Let Π be a 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 $b \in B^+(r) \setminus \{\&g[\mathbf{Y}](\mathbf{X})\}$ such that $\&g[\mathbf{Y}](\mathbf{X})$ joins b, and b is lde-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 17 (External Atom Guessing Rule) *Let Π be a program, and let further $\&g[\mathbf{Y}](\mathbf{X})$ 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 lde-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 18 *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 19 *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 program 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 algorithm is formally stated in Algorithm GroundHEX; our naming convention is as follows. Program Π is the non-ground input program, while 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 lde-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 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 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 14 Let Π be the following program:

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

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 lde-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.

$$\begin{aligned} f_1\colon&\ d(a). \quad f_2\colon\ d(b). \quad f_3\colon\ d(c).\\ g_1\colon&\ e_1(Y) \vee ne_1(Y) \leftarrow d(Y).\\ g_2\colon&\ e_2(Y) \vee ne_2(Y) \leftarrow d(Y).\end{aligned} \qquad \begin{aligned} r_1\colon&\ s(Y) \leftarrow e_1(Y), d(Y).\\ r_2\colon&\ n(Y) \leftarrow e_2(Y), d(Y).\\ r_3\colon&\ c(Z) \leftarrow e_3(Z).\end{aligned}$$

Algorithm GroundHEX

Input: An lde-safe HEX-program Π
Output: A ground HEX-program Π_g s.t. $\Pi_g \equiv^{pos} \Pi$

(a) Choose a set R of *lde-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 lde-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}

The ground program Π_{pg} contains no instances of r_3 as 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 lde-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. □

4.1 Soundness and Completeness

We now argue that Algorithm GroundHEX is sound and complete. Towards a proof we first consider a slower but conceptually simpler variant GroundHEXNaive of it, for which we show soundness and completeness; we then prove that the optimizations in GroundHEX do not harm these properties.

Compared to the naive Algorithm GroundHEXNaive, Algorithm GroundHEX contains the following modifications. The first one concerns the ordinary ASP grounder. We allow the grounder to optimize the grounding as formalized by Definition 19.

The second change concerns the external atoms evaluated at (d). Intuitively, an external atom may be skipped if it can only return constants, which are guaranteed to appear also elsewhere in the grounding.

The third optimization concerns the enumeration of assignments. Note that Step (c) in GroundHEXNaive enumerates all models of Π_{pg}. That is, in order to ground the program, an ASP solver must be called, which is computationally expensive and in fact unnecessary. Step (d) in GroundHEX simply enumerates assignments directly extracted from the partial grounding, constructed in a way guaranteeing that all relevant ground instances of the external atoms are represented in the grounding.

We now illustrate the algorithm with an example.

Algorithm GroundHEXNaive:

Input: An lde-safe HEX-program Π
Output: A ground HEX-program Π_g s.t. $\mathcal{AS}(\Pi_g) \equiv^{pos} \mathcal{AS}(\Pi)$

(a) $\Pi_p = \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
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**
  /* partial grounding */
  $\Pi_{pg} \leftarrow grnd_C(\Pi_p)$ with constants C in Π_p
  /* check if the grounding is large enough */
(c) **for** *all models* $\mathbf{A}$ *of* Π_{pg} *over* $A(\Pi_{pg})$ **do**
    /* evaluate all external atoms */
(d) **for** $\&g[\mathbf{Y}](\mathbf{X})$ *in a rule* $r \in \Pi$ **do**
(e) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in \mathbf{A}\}$ **do**
(f) Let $O = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
        /* add the respective ground guessing rules */
        $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid \mathbf{x} \in O\}$
until Π_{pg} *did not change*
(g) $\Pi_g \leftarrow \Pi_{pg}$
Remove input auxiliary rules and external atom guessing rules from Π_g
Replace all $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ in Π_g by $\&g[\mathbf{y}](\mathbf{x})$
return Π_g

Example 15 Let $\Pi = \{d(x); q(Y) \leftarrow d(X), \&concat[X, a](Y)\}$ be the input program, where the external atom $\&concat[c_1, c_2](c_3)$ is true iff c_3 is the concatenation of c_1 and c_2. Then in the first iteration $\Pi_p = \{d(x) \vee d(y); q(Y) \leftarrow d(X), e_{r,\&concat[X,a]}(Y); g_{inp}(X) \leftarrow d(X)\}$, where g_{inp} is the unique auxiliary input predicate for $\&concat[X, a](Y)$. The grounding step yields then $\Pi_{pg} = \{d(x);\ q(Y) \leftarrow d(X), e_{r,\&concat[X,a]}(Y); g_{inp}(X, a) \leftarrow d(X) \mid X, Y \in \{x, y\}\}$. Now the algorithm comes to the checking phase at (c) and (d). Note that $g_{inp}(x, a)$ and $g_{inp}(y, a)$ appear in all models $\mathbf{A}$ of Π_{pg}. Therefore the algorithm evaluates $\&concat$ under inputs x, a and y, a and collects all output tuples $\mathbf{x}$ such that $f_{\&g}(\mathbf{A}, x, a, \mathbf{x}) = 1$ resp. $f_{\&g}(\mathbf{A}, y, a, \mathbf{x}) = 1$ holds. This holds for the output tuples xa and ya. Thus, Step (f) adds the rules $e_{r,\&g[x,a]}(xa) \vee ne_{r,\&g[x,a]}(xa) \leftarrow$ and $e_{r,\&g[y,a]}(xa) \vee ne_{r,\&g[y,a]}(ya) \leftarrow$ to Π_p and grounding starts over again. In the next iteration, the rule instances $q(xa) \leftarrow d(x), e_{r,\&concat[x,a]}(xa)$ and $q(ya) \leftarrow d(y)$, $e_{r,\&concat[y,a]}(ya)$ will appear in Π_{pg}. However, as no new atoms $g_{inp}(\mathbf{y})$ appears in any of the models of the updated Π_{pg}, the loop terminates after the second iteration. □

We now come to the formal proof that this procedure always returns a grounding which has the same answer sets as the original program. As the programs Π_p and Π_{pg} are iteratively updated in the algorithm, we make the following convention. Whenever we write Π_p or Π_{pg} in one of the proofs, we refer to the status after the main loop terminated, i.e., at Step (g) (resp. Step (g) in Algorithm GroundHEX).

A key concept in our proofs will be that of *representation* of external atoms in a ground program.

Definition 20 *For a ground external atom* $\&g[\mathbf{y}](\mathbf{x})$ *in a rule* r*, its* representation degree in a program Π *is* 0*, if* Π *contains a rule* $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$*. It is* $n + 1$*, if* Π *contains a rule with head* $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ *and the maximum representation degree of all* $\&h[\mathbf{w}](\mathbf{v})$ *s.t.* $e_{s,\&h[\mathbf{w}]}(\mathbf{v})$ *occurs in in the body, is* n*. Otherwise (i.e., there is no rule with head* $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$*), the representation degree is* undefined*.*

If the representation degree for some ground external atom is undefined, we also say that the external

atom is *not represented*. Intuitively, if an external atom is represented, this means that the program contains a guessing rule for the respective replacement atom. The representation degree specifies on how many other external atom replacements this guess depends on. Note that in general, an external atom can have multiple representation degrees simultaneously. However, in the following we will only use its *minimum representation degree* and can therefore drop the prefix *minimum*.

Theorem 10 *If* Π *is a lde-safe* HEX*-program and* $\Pi_g = \mathit{GroundHEXNaive}(\Pi)$*, then* $\Pi_g \equiv^{pos} \Pi$.

It appears that also the optimized algorithm is sound and complete.

Theorem 11 *If* Π *is an lde-safe* HEX*-program, then* $\mathit{GroundHEX}(\Pi) \equiv^{pos} \Pi$.

4.2 Integrating the Algorithm into the Model-Building Framework

In this section we discuss the integration of the new grounding algorithm into the existing model-building framework for HEX-programs. However, since the overall evaluation approach is beyond the scope of this paper, we only give an overview and refer to [37] for a detailed description.

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 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 [12].

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 by [40] 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.

Example 16 Consider the following ground HEX-program Π with facts $employee(a)$, $employee(b)$, $employee(c)$ and $qualification(c)$:

$$
\begin{aligned}
r_1\colon&\quad team1(a) \vee team1(b) \leftarrow\\
r_2\colon&\quad team1(b) \vee team1(c) \leftarrow\\
r_3\colon&\quad team2(X) \leftarrow \&\mathit{diff}[employee, team1](X)\\
r_4\colon&\quad team1a(X) \leftarrow \&\mathit{diff}[team1, qualification](X)\\
r_5\colon&\quad team1b(X) \leftarrow team1(X), qualification(X)\\
r_6\colon&\quad bonus(X) \leftarrow team2(X)\\
r_7\colon&\quad bonus(X) \leftarrow team1b(X)
\end{aligned}
$$

Intuitively, the program considers a company with employees defined using predicate $employee$, some of which have a certain $qualification$. The program forms then two teams $team1$ and $team2$ such that certain restrictions concerning the assignment of employees to $team1$, encoded by r_1 and r_2, are satisfied. By r_3, everyone who is not in $team1$ shall be in $team2$. Then $team1$ is further divided into two subteams $team1a$

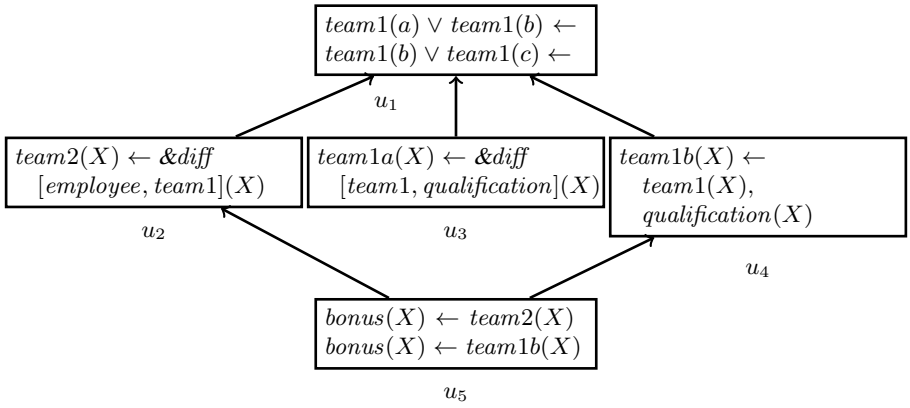


Abbildung 2: Evaluation graph of Example 16 without u_{final}

and $team1b$, where $team1b$ shall consist of all employees who have the *qualification* (r_5); the others are assigned to $team1a$ (r_4). Finally, all employees working in $team2$ or in $team1b$ shall be eligible for a *bonus* (r_6 and r_7). An evaluation graph of Π is visualized in Figure 2.

For the sake of a more convenient presentation, we will denote assignments $\mathbf{A}$ as the set of atoms $\{a \mid \mathbf{T}a \in \mathbf{A}\}$ which are true. For constructing the answer sets of the program, Algorithm BUILDANSWERSETS chooses unit u_1 and computes the set of output models for input model $\emptyset$, which is $o\text{-}ints(u_1) = \{m_2^{\mathrm{O}} = \{team1(a), team1(c)\}, m_3^{\mathrm{O}} = \{team1(b)\}\}$. In the next step, one of the components u_2, u_3 or u_4 can be chosen for evaluation because for each of them the single predecessor unit u_1 has already been processed. For u_2 and input model $m_4^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_6^{\mathrm{O}} = \{team2(b)\}$, and for input model $m_5^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_7^{\mathrm{O}} = \{team2(a), team2(c)\}$. For u_3 and input model $m_8^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_{10}^{\mathrm{O}} = \{team1a(a)\}$, and for input model $m_9^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_{11}^{\mathrm{O}} = \{team1a(b)\}$. For u_4 and input model $m_{12}^{\mathrm{I}} = m_2^{\mathrm{O}} = \{team1(a), team1(c)\}$, the unique output model is $m_{14}^{\mathrm{O}} = \{team1b(c)\}$, and for input model $m_{13}^{\mathrm{I}} = m_3^{\mathrm{O}} = \{team1(b)\}$, the unique output model is $m_{15}^{\mathrm{O}} = \emptyset$.

Then the algorithm chooses u_5 for evaluation. The first step is the computation of the input models of u_5. Because u_5 has two predecessor units u_2 and u_4 and each of them has two output models m_6^{O}, m_7^{O} resp. m_{14}^{O}, m_{15}^{O}, there are four possible combinations. However, only those combinations are considered, which stem from a "common ancestor model" (for details, see [37]), which are $m_{16}^{\mathrm{I}} = \{team2(b), team1b(c)\}$ and $m_{17}^{\mathrm{I}} = \{team2(a), team2(c)\}$. In the second step, the output models of u_5 are determined: for m_{16}^{I} the unique output model is $m_{18}^{\mathrm{O}} = \{bonus(b), bonus(c)\}$ and for m_{17}^{I} it is $m_{19}^{\mathrm{O}} = \{bonus(a), bonus(c)\}$.

Finally, the answer sets of the overall program are determined as the union of those models of all units which step from a common ancestor model (this can be realized as the computation of the input models of an empty unit u_{final}, which depends on all other units). We have five units with two output models each and thus 2^5 possible combinations, but only two of them stem from a common ancestor model, namely $m_{20}^{\mathrm{I}} = \{team_1(a), team_1(c), team_{1a}(a), team_{1b}(c), team_2(b), bonus(b), bonus(c)\}$ and $m_{21}^{\mathrm{I}} = \{team_1(b), team_{1a}(b), team_2(a), team_2(c), bonus(a), bonus(c)\}$. These models are the answer sets of the program. □

The motivation for the evaluation framework in [12] 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 lde-safe, and thus strongly safe, program.

Example 17 Program Π from Example 14 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 lde-safe programs as units. To this end, we define a *generalized evaluation graph* like an evaluation graph in [12], but with lde-safe instead of extended pre-groundable programs as nodes. We can then show that the algorithm BUILDANSWERSETS by [12] remains sound and complete for generalized evaluation graphs, if the grounding algorithm from above is applied:

Theorem 12 *For a generalized evaluation graph $\mathcal{E} = (V, E)$ of an lde-safe* HEX*-program Π,* BUILDANSWERSETS *with* GroundHEX *for grounding returns $\mathcal{AS}(\Pi)$.*

While program decomposition has led to a performance increase for the solving algorithms in [12], it is counterproductive for learning-based algorithms [13] 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 detrimental to conflict learning. Thus, from this angle having few units is an advantage. However, the worst case for the grounding algorithm is a unit containing an external atom that is relevant for lde-safety and receives nonmonotonic input from the same unit; this requires to consider exponentially many assignments.

Example 18 Reconsider program Π from Example 14. Then the algorithm evaluates external atom $\&count[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic and lde-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 [35], 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

Algorithm GreedyGEG

Input: An lde-safe HEX-program Π
Output: A generalized evaluation graph $\mathcal{E} = \langle V, E \rangle$ for Π
(a) Let V be the set of (subset-maximal) strongly connected components of $G = \langle \Pi, \rightarrow_m \cup \rightarrow_n \rangle$
Update E
(b) **while** V *was modified* **do**
(c) **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 $\mathcal{E} = \langle V, E \rangle$

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 19 Reconsider program Π from Example 18. Algorithm GreedyGEG creates a generalized evaluation graph with the 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.

Theorem 13 *For an lde-safe program* Π*, Algorithm GreedyGEG returns a suitable generalized evaluation graph of* Π.

Again, for details and a formal discussion of the integration of the algorithm into the DLVHEX-framework, we refer to [37].

5 Implementation and Evaluation

For implementing our technique, we integrated GRINGO as grounder (implementation of Algorithm GroundASP) and CLASP into our prototype system DLVHEX. The overall evaluation is driven by the evaluation framework described in Section 4.2. The evaluation of single evaluation units use separate instances of GRINGO for grounding and of CLASP for solving. External sources can be easily added by using a convenient API provided by the system. Internally, the system exploits CLASP's SMT interface for realizing external calls. To this end, CLASP makes callbacks to the DLVHEX core whenever its generic propagation methods (exploiting unit clauses and minimality considerations) cannot derive further truth values. The callback is then delegated to external sources to derive further truth values (if possible).

The system is available from http://www.kr.tuwien.ac.at/research/systems/dlvhex as open-source software and provides convenient interfaces for adding external sources and intervening in the algorithms depending on the needs of the application at hand. The system has been successfully applied to a range of applications (e.g. multi-context systems, dl-programs) and runs on multiple platforms, including Linux, Mac OS X and Microsoft Windows.

We now present the problems we are going to use as benchmarks and discuss the results for instances in a separate subsection; the HEX-encodings of all problems can be found in [37].

5.1 Problem Suite

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.

Set Partitioning. In this benchmark we consider a program similar to Example 14, 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, where sel may contain at most two elements. The program is as follows:

$$
\begin{aligned}
&domain(1).\ \ldots\ domain(n).\\
&\quad sel(X) \leftarrow domain(X), \&diff[domain, nsel](X)\\
&\quad nsel(X) \leftarrow domain(X), \&diff[domain, sel](X)\\
&\qquad\quad \leftarrow sel(X), sel(Y), sel(Z), X \neq Y, X \neq Z, Y \neq Z
\end{aligned}
$$

where $\&diff[p, q](X)$ computes the set of all elements X which are in the extension of p but not in the extension of q. Note that under lde-safety, the domain predicate $domain$ is not needed as $\&diff$ does not introduce new constants.

Bird-Penguin Variant. For our experiments, we used a variant of the Bird-Penguin program Π' in Example 13. Structurally, Π' is for grounding similar to the Set Partitioning problem, as the external atom in rule r_3 is monotonic, and grounding is expected to behave similarly. For a worst case which cannot be avoided by the greedy heuristics, we replaced this atom with a slightly more general, nonmonotonic external atom which also outputs a special constant $cons$ if the extended ontology is satisfiable. With a growing number of birds, we get a growing number of cycles with nonmonotonic external atoms which combinatorially intermingle the worst case for the heuristics when $bird(X)$ is dropped from rule r_3'.

Recursive Processing of Data Structures. This problem is representative for a range of applications which process data structures recursively. 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.

Argumentation. In this benchmark, we use a HEX-program which computes specific extensions for Dung-style abstract argumentation frameworks (AFs) [10], given that code for an extension test is available. AFs are directed graphs with the nodes being interpreted as *arguments* and the arcs as *attacks* between arguments. A typical reasoning task is the computation of *extensions*, which are sets of nodes that fulfill certain properties, depending on the semantics being used, and *cautious* and *brave* reasoning, i.e., checking whether an argument is contained in all or at least one extension, respectively. Many reasoning tasks are intractable or even beyond NP, depending on the type of extension considered. Here we consider ideal set extensions, i.e., extensions which have to be ideal sets of the AF at hand; notably, testing whether a set of arguments is an ideal set of an AF is co-NP-complete [11]. Thus a natural guess-and-check computation of an ideal set extension that uses an external atom to decide the extension property reflects this complexity in the ideal set

check, which is done in our ASP program using a standard saturation technique; the encoding is generic and might be adapted for other semantics.

In addition, we perform a processing of the arguments in the computed extension, e.g., by using an external atom for generating LaTeX code for the visualization of the AF (the graph with ideal sets being marked) using a graphics library. The challenge here is that argument processing depends *nonmonotonically* on the ideal sets (the LaTeX code of one ideal set is, in general, incomparable to that of another ideal set). As discussed in Section 4, this is the worst case for the new grounding algorithm if no program decomposition is used, but can be avoided by our new evaluation heuristics in this case.

Route Planning. Inspired by semantically enriched route planning, which has been studied in the MyITS project [20] for smart city applications, we consider here two route planning scenarios using the public transport system of Vienna. The data is available under creative commons license (cc-by) from data.wien.gv.at and contains a map of 158 subway, tram, city bus and rapid transit train lines with a total number of 1701 stations. Since the data does not contain information about the distances between stations, we uniformly assumed costs of 1, 2 and 3 for each stop traveled by subway/rapid transit train, tram or bus, respectively. We further assumed costs of 10 for each necessary change representing walking and waiting time. However, with more detailed data, our encoding would also allow for using different values for each line or station. Access to the data is provided via an external atom $\&path[s, d](a, b, c, l)$, which returns for a start location s and a destination d the shortest direct connection (computed using Dijkstra's algorithm), represented as set edges (a, b) between stations a and b with costs c using line l.

For instance, a journey from $Wien\ Mitte$ to $Taubstummengasse$ is possible using subway line $U4$ from $Wien\ Mitte$ to $Karlsplatz$ (with intermediate stop at *Stadtpark*), changing to line *U1*, and going from $Karlsplatz$ to $Taubstummengasse$ (which is just one stop). This will be represented as follows:

$$\{(Wien\ Mitte, Wien\ Mitte\ (U4), 10, change),$$
$$(Wien\ Mitte\ (U4), Stadtpark\ (U4), 1, U4),$$
$$(Stadtpark\ (U4), Karlsplatz\ (U4), 1, U4),$$
$$(Karlsplatz\ (U4), Karlsplatz\ (U1), 10, change),$$
$$(Karlsplatz\ (U1), Taubstummengasse\ (U1), 1, U1),$$
$$(Taubstummengasse\ (U1), Taubstummengasse, 10, change)\}$$

In order to model changes between lines, our graph has for each station and each line which arrives at this station a separate node, with a label consisting of the actual name of the station and the respective line. To foster a change, the external atom returns a tuple $(a, a', 10, change)$, where a and a' are two nodes representing the same station but for different lines, and $change$ is just a dedicated "line" representing walks between platforms, cf. Figure 3 (dashed lines indicates changes with costs 10, solid lines indicate trips with the costs given in parantheses). In order to relieve the user from writing line-specific names of stations in the input to the program, we further have for each station a generic node which is connected to all line-specific nodes for this station.

Note that there will never be cycles in the direct path between two stations because the costs are minimized, thus the set representation is sufficient and there is no need to formally store the order of the edges. Further note that tuples (1) and (6) do not really represent changes but are merely the connections between the generic stations and the line-specific nodes. This allows the user to use the constants $Wien\ Mitte$ and $Taubstummengasse$ in the input without predetermining which line to take at these stations. However, as these spurious changes at the start and at the destination node are necessary in any route, this does not affect the minimization of the costs.

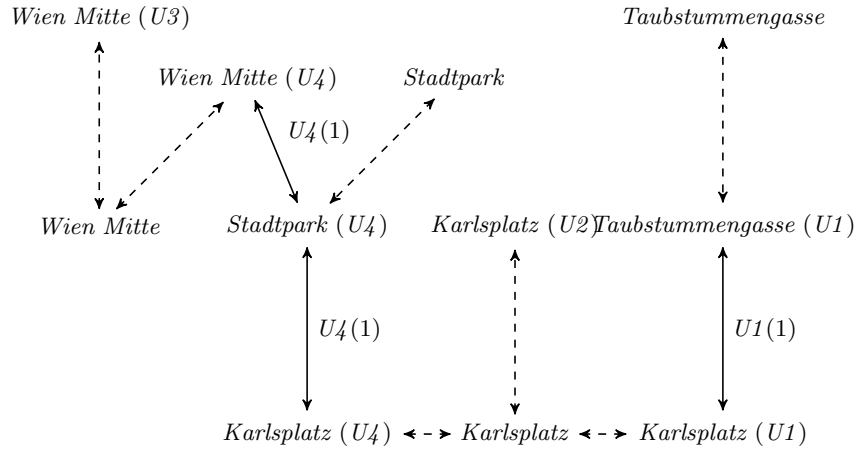


Abbildung 3: Graph representation of stations and lines (dashed: changes with costs 10, solid: given lines and costs)

Route planning can be subject to *side constraints*, where the user not only wants a route connecting two or more locations, but that it satisfies additional semantic conditions, like ending in a restaurant or next to a park; this may be determined using suitable information sources and ontologies [20]. Here we concentrate on a plain setting and consider two concrete applications as show cases.

Single Route Planning. In the first scenario we consider route planning of a single person who wants to visit a number of locations. Additionally, we have the side constraint that the person wants to go for lunch in a restaurant if and only if the tour is longer than the given limit of cost 300. Because the external source allows only for computing direct connections between two locations, it cannot solve the task completely and there needs to be interaction between the HEX-program and the external source.

Pair Route Planning. In our second scenario we consider two persons. Each of them wants to visit a number. Additionally, the two persons want to meet, thus the two tours need to intersect at some point. Possible meeting locations are drawn randomly. We further have the side constraint, that the meeting location shall be a restaurant, if at least one of the tours is longer than the limit of costs 300.

5.2 Benchmark Results

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) when computing the first answer set. 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). The numbers in parentheses indicate the number of instances and timeouts, respectively. For determining lde-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/allbenchmarks.ods.

Reachability. 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. The results are shown in Table 1a. 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

Tabelle 1: Benchmark results in secs; timeout ("—") is 300 secs

(a) Reachability

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
15 (10)	0.59 (0)	0.28 (0)	0.08 (0)	0.49 (0)	0.23 (0)	0.06 (0)
25 (10)	5.78 (0)	4.67 (0)	0.33 (0)	2.94 (0)	1.90 (0)	0.35 (0)
35 (10)	36.99 (0)	33.99 (0)	1.00 (0)	14.02 (0)	11.30 (0)	0.95 (0)
45 (10)	161.91 (0)	155.40 (0)	2.18 (0)	53.09 (0)	47.19 (0)	2.22 (0)
55 (10)	— (10)	— (10)	n/a	171.46 (0)	158.58 (0)	5.74 (0)
65 (10)	— (10)	— (10)	n/a	— (10)	— (10)	n/a

(b) Set Partitioning

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
10 (1)	0.49 (0)	0.01 (0)	0.39 (0)	0.52 (0)	0.02 (0)	0.41 (0)
20 (1)	3.90 (0)	0.05 (0)	3.62 (0)	4.67 (0)	0.10 (0)	4.23 (0)
30 (1)	16.12 (0)	0.18 (0)	15.32 (0)	19.59 (0)	0.36 (0)	18.32 (0)
40 (1)	48.47 (0)	0.48 (0)	46.71 (0)	51.55 (0)	0.90 (0)	48.74 (0)
50 (1)	115.56 (0)	1.00 (0)	112.14 (0)	119.40 (0)	1.79 (0)	114.11 (0)
60 (1)	254.66 (0)	1.84 (0)	248.88 (0)	257.78 (0)	3.35 (0)	248.51 (0)

(c) Bird-penguin

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5 (1)	0.06 (0)	<0.005 (0)	0.01 (0)	0.08 (0)	0.02 (0)	0.01 (0)
10 (1)	0.14 (0)	<0.005 (0)	0.08 (0)	1.32 (0)	1.12 (0)	0.10 (0)
11 (1)	0.27 (0)	<0.005 (0)	0.19 (0)	2.85 (0)	2.43 (0)	0.27 (0)
12 (1)	0.32 (0)	<0.005 (0)	0.23 (0)	6.05 (0)	5.53 (0)	0.26 (0)
13 (1)	0.69 (0)	0.01 (0)	0.60 (0)	12.70 (0)	11.76 (0)	0.61 (0)
14 (1)	0.66 (0)	<0.005 (0)	0.57 (0)	28.17 (0)	26.70 (0)	0.73 (0)
15 (1)	1.66 (0)	0.01 (0)	1.49 (0)	59.73 (0)	57.14 (0)	1.46 (0)
16 (1)	1.69 (0)	0.01 (0)	1.53 (0)	139.47 (0)	131.87 (0)	1.92 (0)
17 (1)	3.83 (0)	0.01 (0)	3.57 (0)	— (1)	— (1)	n/a
18 (1)	4.34 (0)	0.01 (0)	4.08 (0)	— (1)	— (1)	n/a
19 (1)	10.07 (0)	0.01 (0)	9.56 (0)	— (1)	— (1)	n/a
20 (1)	11.36 (0)	0.01 (0)	10.87 (0)	— (1)	— (1)	n/a
24 (1)	95.60 (0)	0.01 (0)	93.35 (0)	— (1)	— (1)	n/a
25 (1)	— (1)	0.01 (0)	— (1)	— (1)	— (1)	n/a

(d) Merge Sort

#	w. domain predicates			w/o domain predicates		
	wall clock	ground	solve	wall clock	ground	solve
5 (10)	0.22 (0)	0.04 (0)	0.10 (0)	0.10 (0)	0.01 (0)	0.04 (0)
6 (10)	1.11 (0)	0.33 (0)	0.54 (0)	0.10 (0)	0.01 (0)	0.04 (0)
7 (10)	9.84 (0)	4.02 (0)	4.42 (0)	0.11 (0)	0.01 (0)	0.05 (0)
8 (10)	115.69 (0)	61.97 (0)	42.30 (0)	0.12 (0)	0.01 (0)	0.05 (0)
9 (10)	— (10)	— (10)	n/a	0.14 (0)	0.01 (0)	0.07 (0)
10 (10)	— (10)	— (10)	n/a	0.15 (0)	0.08 (0)	0.01 (0)
15 (10)	— (10)	— (10)	n/a	0.23 (0)	0.14 (0)	0.01 (0)
20 (10)	— (10)	— (10)	n/a	0.47 (0)	0.35 (0)	0.02 (0)
25 (10)	— (10)	— (10)	n/a	1.90 (0)	1.58 (0)	0.06 (0)
30 (10)	— (10)	— (10)	n/a	4.11 (0)	3.50 (0)	0.12 (0)
35 (10)	— (10)	— (10)	n/a	20.98 (0)	18.45 (0)	0.51 (0)
40 (10)	— (10)	— (10)	n/a	61.94 (0)	54.62 (0)	1.46 (0)
45 (10)	— (10)	— (10)	n/a	144.22 (2)	133.99 (2)	2.26 (0)
50 (10)	— (10)	— (10)	n/a	— (10)	— (0)	n/a

(e) Argumentation

#	monolithic			greedy			#	monolithic			greedy		
	wall clock	ground	solve	wall clock	ground	solve		wall clock	ground	solve	wall clock	ground	solve
4 (30)	0.57 (0)	0.11 (0)	0.38 (0)	0.25 (0)	0.01 (0)	0.18 (0)	10 (30)	— (30)	— (30)	n/a	15.92 (0)	0.02 (0)	15.81 (0)
5 (30)	2.12 (0)	0.67 (0)	1.26 (0)	0.44 (0)	0.01 (0)	0.37 (0)	11 (30)	— (30)	— (30)	n/a	31.19 (0)	0.02 (0)	31.05 (0)
6 (30)	18.93 (0)	7.45 (0)	10.86 (0)	0.88 (0)	0.01 (0)	0.80 (0)	12 (30)	— (30)	— (30)	n/a	63.16 (0)	0.02 (0)	62.95 (0)
7 (30)	237.09 (9)	170.12 (9)	65.12 (0)	1.65 (0)	0.01 (0)	1.57 (0)	13 (30)	— (30)	— (30)	n/a	172.75 (1)	0.03 (0)	172.38 (1)
8 (30)	— (30)	— (30)	n/a	3.13 (0)	0.01 (0)	3.05 (0)	14 (30)	— (30)	— (30)	n/a	256.60 (18)	0.01 (0)	256.44 (18)
9 (30)	— (30)	— (30)	n/a	7.41 (0)	0.02 (0)	7.31 (0)	15 (30)	— (30)	— (30)	n/a	290.01 (29)	<0.005 (0)	290.00 (29)

space during solving.

Set Partitioning. We considered domains with $n*10$ elements, $1 \leq n \leq 6$. The results for the program with and without the domain predicate are 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 lde-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.

Bird-Penguin Variant. We considered ontologies with n distinct birds, for which respective assertions (facts) were added to the ontology, and the non-monotonic variant of the external atom in rule r_3. The results in Table 1c 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. While this example was tailored for the worst case, grounding of the regular Bird-Penguin program is easy, with as well as without domain predicate (similar as Set Partitioning). Furthermore, we could in applications of

DL-programs sometimes even experience a sensible run time improvement by removing domain predicates.

Recursive Processing of Data Structures. 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 unmanageable 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 [15], 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 lde-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*). 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.

Route Planning. For each instance size n we generated 50 instances by randomly drawing n locations to visit *plus* n possible locations for having lunch (the data does not provide information about such locations, but usually there are restaurants or snack bars in the near area of stations). We show for each instance size the averages of the total runtimes, the grounding times, the solving times, the percentage of instances for which a solution was found within the time limit (column *solution (%)*)[2], the average path length (costs) of the instances with solutions (column *length*), the average number of necessary changes, not counting changes between generic and line-specific station nodes (column *changes*), and the percentage of instances with solutions which require a restaurant visit due to length of the tour (column *lunch (%)*). The results are shown in Tables 2a, 2b and 2c using the full map, the map restricted to tram and subway, and the map restricted to subway only, respectively. In addition to the wall clock, grounding and solving time, we further show for the instances which have a solution the average path length (column *length*) average number of necessary changes.

The hardness of the benchmark stems from the side constraint. Without this constraint, the tour could be computed deterministically by successive calls of the external source, once the sequence of locations was guessed. However, due to side constraint, not only the overall tour does depend on the individual locations, but also the individual locations depend on the overall tour (they need to contain a restaurant iff the tour is too long). This leads to a cycle over the external atom $\&path$. With the notion of strong safety, this requires the output variables of this external atom to be bounded by domain predicates, thus the whole map needs to be imported a priori.

Single Route Planning. We considered instances with $1 \leq n \leq 15$ locations to visit. The sequence in which the locations are visited is guessed non-deterministically in the logic program. While the direct connections between two locations are of minimum length by definition of the external atom, the length of the overall tour is only optimal wrt. to the chosen sequence of locations, but other sequences might lead to a shorter overall tour. However, we have the constraint that for visiting n locations there should be at most

[2]The number of instances for which no solution was found include both timeout instances and instances which have no solution.

Tabelle 2: Single Route Planning benchmark, results in secs; timeout ("—") is 300 secs

(a) Full Map

#	w. domain predicates							w/o domain predicates						
	wall clock	ground	solve	solution (%)	length	changes	lunch (%)	wall clock	ground	solve	solution (%)	length	changes	lunch (%)
1 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	2.40 (0)	1.71 (0)	0.54 (0)	100.00	0.00	0.00	0.00
2 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	7.82 (0)	5.00 (0)	2.42 (0)	90.00	82.64	2.24	0.00
3 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	16.44 (0)	9.46 (0)	5.81 (0)	76.00	152.21	3.92	0.00
4 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	36.60 (0)	16.69 (0)	16.90 (0)	52.00	213.00	5.31	3.85
5 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	102.71 (0)	26.63 (0)	69.26 (0)	52.00	281.27	7.58	11.54
6 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	284.69 (38)	236.43 (38)	45.56 (0)	16.00	368.12	9.00	100.00
7 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN

(b) Subway and Tram

#	w. domain predicates							w/o domain predicates						
	wall clock	ground	solve	solution (%)	length	changes	lunch (%)	wall clock	ground	solve	solution (%)	length	changes	lunch (%)
1 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	1.13 (0)	0.83 (0)	0.20 (0)	100.00	0.00	0.00	0.00
2 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	3.50 (0)	2.34 (0)	0.85 (0)	100.00	68.12	1.80	0.00
3 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	8.91 (0)	4.88 (0)	3.01 (0)	96.00	125.19	3.38	0.00
4 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	34.05 (2)	20.11 (2)	11.41 (0)	92.00	192.80	4.65	0.00
5 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	62.98 (0)	13.57 (0)	43.58 (0)	90.00	284.89	7.53	46.67
6 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	192.58 (11)	81.96 (11)	101.66 (0)	74.00	361.86	8.95	100.00
7 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	287.99 (38)	234.55 (38)	49.27 (0)	24.00	410.17	10.50	100.00
8 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	299.75 (48)	289.34 (48)	9.48 (0)	4.00	418.00	12.00	100.00
9 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN

(c) Subway Only

#	w. domain predicates							w/o domain predicates						
	wall clock	ground	solve	solution (%)	length	changes	lunch (%)	wall clock	ground	solve	solution (%)	length	changes	lunch (%)
1 (50)	295.42 (49)	294.73 (49)	0.29 (0)	2.00	0.00	0.00	0.00	0.89 (0)	0.65 (0)	0.15 (0)	100.00	0.00	0.00	0.00
2 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	2.89 (0)	1.92 (0)	0.66 (0)	100.00	64.04	1.22	0.00
3 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	8.08 (0)	4.10 (0)	2.91 (0)	100.00	127.16	2.62	0.00
4 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	23.87 (0)	7.31 (0)	13.72 (0)	100.00	206.78	4.16	12.00
5 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	57.30 (0)	11.69 (0)	39.37 (0)	100.00	317.08	7.20	90.00
6 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	107.05 (0)	17.32 (0)	78.33 (0)	100.00	364.32	8.46	100.00
7 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	225.16 (9)	73.97 (9)	135.58 (0)	82.00	405.27	9.61	100.00
8 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	299.90 (48)	289.15 (48)	9.83 (0)	4.00	353.00	9.00	100.00
9 (50)	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN	— (50)	— (50)	0.00 (0)	0.00	NaN	NaN	NaN

$\lceil n \times 1.5 \rceil$ changes. Due to this constraint not all instances have a solution. It would be easy to extend the scenario to predetermine the sequence of locations by additional constraints, e.g., by global weak constraints in order to minimize the costs.

For each instance size n we generated 50 instances by randomly drawing n locations to visit *plus* n possible locations for having lunch (the data does not provide information about such locations, but usually there are restaurants or snack bars in the near area of stations). We show for each instance size the averages of the total runtimes, the grounding times, the solving times, the percentage of instances for which a solution was found within the time limit (column *solution (%)*)[3], the average path length (costs) of the instances with solutions (column *length*), the average number of necessary changes, not counting changes between generic and line-specific station nodes (column *changes*), and the percentage of instances with solutions which require a restaurant visit due to length of the tour (column *lunch (%)*). The results are shown in Tables 2a, 2b and 2c using the full map, the map restricted to tram and subway, and the map restricted to subway only, respectively. In addition to the wall clock, grounding and solving time, we further show for the instances which have a solution the average path length (column *length*) average number of necessary changes.

Pair Route Planning. We created for each instance size of $1 \leq n \leq 15$ locations (with at most $\lceil n \times 1.5 \rceil$ changes) a number of 50 instances, where the n locations for each person, n possible (non-restaurant)

[3]The number of instances for which no solution was found include both timeout instances and instances which have no solution.

Tabelle 3: Pair Route Planning benchmark, results in secs; timeout ("—") is 300 secs)

(a) Full Map

#	w. domain predicates								w/o domain predicates							
	wall clock		ground		solve		solution (%)		wall clock		ground		solve		solution (%)	
1 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	11.60	(0)	9.32	(0)	1.31	(0)	82.00	150.98
2 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	34.20	(0)	26.00	(0)	6.54	(0)	90.00	300.69
3 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	89.21	(0)	53.31	(0)	29.95	(0)	44.00	449.77
4 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	204.73	(4)	107.60	(4)	83.28	(0)	76.00	592.87
5 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	297.29	(44)	278.85	(44)	15.03	(0)	12.00	710.00
6 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	—	(50)	—	(50)	0.00	(0)	0.00	NaN

#	length	changes	lunch (%)	
1 (50)	4.54	0.00		
2 (50)	8.53	0.00		
3 (50)	11.14	9.09		
4 (50)	15.47	84.21		
5 (50)	17.50	100.00		
6 (50)	NaN	NaN		

(b) Subway and Tram

#	w. domain predicates								w/o domain predicates							
	wall clock		ground		solve		solution (%)		wall clock		ground		solve		solution (%)	
1 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	8.17	(0)	7.32	(0)	0.44	(0)	98.00	133.76
2 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	23.35	(0)	19.09	(0)	2.78	(0)	100.00	269.54
3 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	59.04	(0)	36.47	(0)	17.64	(0)	98.00	390.37
4 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	147.43	(3)	75.46	(3)	59.60	(0)	94.00	582.55
5 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	255.94	(17)	161.93	(17)	76.12	(0)	66.00	636.55
6 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	—	(50)	—	(50)	0.00	(0)	0.00	NaN

#	length	changes	lunch (%)	
1 (50)	14.51	89.36		
2 (50)	16.61	100.00		
3 (50)	NaN	NaN		

(c) Subway Only

#	w. domain predicates								w/o domain predicates							
	wall clock		ground		solve		solution (%)		wall clock		ground		solve		solution (%)	
1 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	7.72	(0)	6.97	(0)	0.30	(0)	98.00	124.51
2 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	21.35	(0)	17.69	(0)	2.19	(0)	100.00	251.66
3 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	60.00	(0)	33.79	(0)	21.05	(0)	100.00	375.08
4 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	167.44	(5)	80.97	(5)	74.14	(0)	90.00	565.33
5 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	267.17	(20)	169.57	(20)	81.00	(0)	60.00	627.70
6 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	299.05	(48)	292.38	(48)	4.88	(0)	4.00	640.00
7 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	—	(50)	—	(50)	0.00	(0)	0.00	NaN

#	length	changes	lunch (%)	
1 (50)	12.30	100.00		
2 (50)	13.00	100.00		
3 (50)	NaN	NaN		

meeting locations and n restaurants are drawn randomly. The results are shown in Tables 3a, 3b and 3c using the full map, the map restricted to tram and subway, and the map restricted to subway only, respectively. Columns *length* and *changes* show the sums of the lengths of the tours and of the necessary changes for both persons.

Observations. In both scenarios we can observe that importing the whole map a priori is merely impossible. Already the grounder fails with a timeout, but due to the large number of (unnecessary) external atoms in the ground program, also solving would not be reasonably possible with the given data. Only liberal safety allows for solving the task in the given time limit by importing only the relevant part of the map during grounding. The external atom implements a cache both for the graph representation of the map and the results of Dijkstra's algorithm. The first external source call needs on our benchmark system approximately 5 seconds to load the map (in case of the full map, but not for single route planning with $n = 1$ which will not call the external source). Moreover, Dijkstra's algorithm computes for a given start node the shortest paths to *all* nodes, thus after the external source has been called for a certain start node, successive calls for the same start node are significantly faster. In particular, the cache is already filled with all relevant data during grounding, thus solving will spend only very little time in external sources and the solving time will mainly be caused by the HEX evaluation algorithms.

For pair route planning, note that even instances with $n = 1$ have a path longer than 0 because the location for the meeting is not included in instance size n.

As expected, a restriction of the map to trams and subway or to subway only usually yields smaller runtimes. Also the number of changes decreases because multiple tram and especially subway lines have usually

more shared stations than bus lines. With increasing number of locations to visit, the number of restaurant visits usually increases as well. However, this is not a strict rule and the tables show some exceptions as the locations were drawn at random and their distances is an important factor.

Summary. Our new grounding algorithm allows for grounding lde-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 Controlled Grounding

Algorithm GroundHEX in Section 4 derives finiteness of the grounding from syntactic and semantic conditions. That is, termination is guaranteed by the program at hand. However, an alternative is to intervene the grounding process itself and guarantee termination by application-specific criteria. To this end, we extend the algorithm with additional *hooks* which can be instantiated for an application at hand.

This leads us to computing a restricted grounding, which enables a form of *bounded model generation* that can be exploited for different purposes. Informally, the bounded grounding algorithm BGroundHEX is explained as follows. The input is a non-ground program Π, and the program Π_p is a non-ground ordinary ASP *prototype program*, which is an iteratively updated variant of Π that is enriched 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} is intended to converge against a fixpoint, i.e., a final *ground* HEX-*program* Π_g. For this purpose, the loop at (b) and the abortion check at (f) introduce two *hooks* (Repeat and Evaluate) which allow for realizing application-specific termination criteria. They need to be substituted by concrete program fragments depending on the reasoning task; for now we assume that the loop at (f) runs exactly once and the check at (f) is always true (which is sound and complete for model computation of lde-safe programs, cf. Theorem 14).

Akin to Algorithm GroundHEX the algorithm first 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). 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 an ordinary ASP grounder GroundASP in the main loop at (b). After the grounding step, it checks whether the grounding contains all relevant constants; to this end, it checks, for all external atoms (d) and relevant input interpretations (e), potential output tuples at (f), whether they contain any new value not yet respected in the grounding; here, $\mathbf{Y}_m, \mathbf{Y}_a, \mathbf{Y}_n$ denote the sets of *monotonic*, *antimonotonic*, and *nonmonotonic* predicate input parameters in $\mathbf{Y}$, respectively. The program adds the relevant constants via guessing rules at (g) to Π_p (this may also be expressed by unstratified negation). Then the main loop starts over again. Eventually, the algorithm is intended to find a program respecting all relevant constants. Then at (h), auxiliary input rules are removed and replacement atoms are translated to external atoms.

The main difference to Algorithm GroundHEX is the addition of the two hooks at (c) (Repeat) and at (f) (Evaluate), which must be defined for a concrete instance of the algorithm (which we do in the following). We assume that the hooks are substituted by code fragments with access to all local variables. Moreover, the set PIT_i contains the input atoms for which the corresponding external atoms have been evaluated in iteration i. Evaluate decides for a given input atom $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ whether the corresponding external atom shall be evaluated under $\mathbf{c}$. This allows for abortion of the grounding even if it is incomplete, which can be

Algorithm BGroundHEX

Input: A HEX-program Π
Output: A ground HEX-program Π_g
(a) $\Pi_p = \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
$i \leftarrow 0$
(b) **while** *Repeat()* **do**
  $i \leftarrow i+1$ `// Remember already processed input tuples at iteration` i
(c) Set $NewInputTuples \leftarrow \emptyset$ and $PIT_i \leftarrow \emptyset$
  repeat
    $\Pi_{pg} \leftarrow$ GroundASP(Π_p) `// partial grounding`
(d) **for** $\&g[\mathbf{Y}](\mathbf{X})$ *in a rule* $r \in \Pi$ **do** `// evaluate all external atoms`
(e) `// do this under all relevant assignments`
      $\mathbf{A}_{ma} = \{\mathbf{T}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
      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**
        $\mathbf{A} = (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow\in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow\in \Pi_{pg}\}$
(f) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})$ *s.t.* *Evaluate*$(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})) = true$ **do**
(g) `// add ground guessing rules and remember` $\mathbf{y}$`-evaluation`
          $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
          $NewInputTuples \leftarrow NewInputTuples \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{y})\}$
    $PIT_i \leftarrow PIT_i \cup NewInputTuples$
  until Π_{pg} *did not change*
(h) Remove input auxiliary rules and external atom guessing rules from Π_{pg}
Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π_{pg} by $\&g[\mathbf{y}](\mathbf{x})$
return Π_{pg}

exploited for reasoning tasks over programs with infinite groundings where a finite subset of the grounding is sufficient. The second hook Repeat allows for repeating the core algorithm multiple times such that Evaluate can distinguish between input tuples processed in different iterations. Naturally, soundness and completeness of the algorithm cannot be shown in general, but depends on concrete instances for (c) and (f) which in turn may vary for different reasoning tasks.

Usage. As already said above, controlled grounding can be used for different purposes, e.g. to bound models to a fragment sufficient for problem solving, or to support parametric data structures. As a show case of the former, we discuss *queries* over (positive) existential rules in Section 6.2, which can be answered by computing a finite part of a canonical model.

As for parametric data structures, we note that using suitable hooks it is not difficult to support e.g. strings of bounded length and trees of bounded depth, where the bound is given by an integer k; this is akin to bounded arithmetic as supported in ASP solvers, where for instance in DLV the range is bounded by a maximum value $\#maxint$. The point is that the actual bound k can be specified separately such that the grounding algorithm can take it into account; this relieves the user from writing clumsy rules within the ASP program to restrict the grounding relative to k, and it increases readability.

Finally, we present with HEX-*programs with function symbols* in Section 6.3 another application of the grounding algorithm.

6.1 HEX$^{\exists}$-Programs

We now model existential quantifiers as an application of Algorithm BGroundHEX, which can also be written in the usual syntax; a rewriting then simulates it by using external atoms which return dedicated *null values* as representatives for the unnamed values introduced by existential quantifiers. We start by introducing a language for HEX-programs with logical existential quantifiers, called HEX$^{\exists}$*-programs*.

A HEX$^{\exists}$*-program* is a finite set of rules of form

$$\forall \mathbf{X} \exists \mathbf{Y} : p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \tag{2}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables, $\mathbf{X}' \subseteq \mathbf{X}$, $p(\mathbf{X}', \mathbf{Y})$ is an atom, and $\mathbf{conj}[\mathbf{X}]$ is a conjunction of atoms possibly under not, containing exactly the variables $\mathbf{X}$; without confusion, we may omit $\forall \mathbf{X}$.

Intuitively speaking, whenever $\mathbf{conj}[\mathbf{X}]$ holds for some vector of constants $\mathbf{X}$, then there should exist a vector $\mathbf{Y}$ of (unnamed) individuals such that $p(\mathbf{X}', \mathbf{Y})$ holds. Existential quantifiers are simulated by using *new* null values which represent the introduced unnamed individuals. Formally, we assume that $\mathcal{N} \subseteq \mathcal{C}$ is a set of dedicated null values, denoted by ω_i with $i \in \mathbb{N}$, which do not appear in the program.

We transform HEX$^{\exists}$-programs to HEX-programs as follows. For a HEX$^{\exists}$-program Π, let $T_{\exists}(\Pi)$ be the HEX-program with each rule r of form (2) replaced by

$$p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \&exists^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y}),$$

where $f_{\&exists^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 1$ iff $\mathbf{y} = \omega_1, \ldots, \omega_m$ is a vector of *fresh and unique null values* for $r, \mathbf{x}$, and $f_{\&exists^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 0$ otherwise. Each occurrence of an existential quantifier is replaced by an external atom of kind $\&exists^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y})$ of appropriate input and output arity which exploits value invention for simulating the logical existential quantifier similar to the *chase* algorithm.

We call a HEX$^{\exists}$-program Π lde-safe if the program $T_{\exists}(\Pi)$ is lde-safe.

Example 20 The following set of rules is a HEX$^{\exists}$-program Π:

$$\begin{aligned} \Pi: \quad & employee(john). \quad employee(joe). \\ r_1: \quad & \exists Y : office(X, Y) \leftarrow employee(X). \\ r_2: \quad & room(Y) \leftarrow office(X, Y). \end{aligned}$$

Then $T_{\exists}(\Pi)$ is the following lde-safe program:

$$\begin{aligned} T_{\exists}(\Pi): \quad & employee(john). \quad employee(joe). \\ r_1': \quad & office(X, Y) \leftarrow employee(X), \&exists^{1,1}[r_1, X](Y). \\ r_2: \quad & room(Y) \leftarrow office(X, Y). \end{aligned}$$

Intuitively, each employee X has an unnamed office Y of X, which is a room. The unique answer set of $T_{\exists}(\Pi)$ is $\{employee(john), employee(joe), office(john, \omega_1), office(joe, \omega_2), room(\omega_1), room(\omega_2)\}$.

For grounding lde-safe programs, we simply let Repeat be the test $i < 1$ and let Evaluate return $true$. Explicit model computation is in general infeasible for non-lde-safe programs. However, the resulting algorithm GroundDESafeHEX always terminates for lde-safe programs; for non-lde-safe programs, we can support bounded model generation by other hook instantiations. This is exploited e.g. for query answering over cyclic programs (described next). We establish the following result.

Theorem 14 *If Π is an lde-safe program then $\mathcal{AS}(\mathsf{GroundDESafeHEX}(\Pi)) \equiv^{pos} \mathcal{AS}(\Pi)$, where $\equiv^{pos}$ denotes equivalence of the answer sets on positive atoms.*

6.2 Application: Query Answering over Positive HEX$^{\exists}$-Programs

The basic idea for query answering over programs with possibly infinite models is to compute a ground program with a single answer set that can be used for answering the query. Positive programs with existential variables are essentially grounded by simulating the *parsimonious chase procedure* by [33], which uses null values for each existential quantification. However, for termination of BGroundHEX we need to provide specific instances of the hooks in the grounding algorithm.

We start by restricting the discussion to a fragment of HEX$^{\exists}$ called $Datalog^{\exists}$ [33]. A $Datalog^{\exists}$-program is a HEX$^{\exists}$-program where every rule body $\mathbf{conj}[\mathbf{X}]$ consists of positive ordinary atoms. Thus compared to HEX$^{\exists}$-programs, default negation and external atoms are excluded.

Example 21 The following set Π of rules is a simple $Datalog^{\exists}$-program about persons and fathers:

$$\begin{aligned} \Pi: \quad & person(john). \quad person(joe). \\ r_1: \quad & \exists Y : father(X,Y) \leftarrow person(X). \\ r_2: \quad & person(Y) \leftarrow father(X,Y). \end{aligned} \tag{3}$$

Intuitively, each person X has some unnamed father Y of X which is also a person. □

We recall *homomorphisms* as used for defining $Datalog^{\exists}$-semantics and query answering over $Datalog^{\exists}$-programs. A *homomorphism* h is a mapping $h\colon \mathcal{N} \cup \mathcal{V} \to \mathcal{C} \cup \mathcal{V}$. For a homomorphism h, let $h|_S$ be its restriction to $S \subseteq \mathcal{N} \cup \mathcal{V}$, i.e., $h|_S(X) = h(X)$ if $X \in S$ and is undefined otherwise. For any atom a, let $h(a)$ be the atom where each variable and null value V in a is replaced by $h(V)$; this is likewise extended to $h(S)$ for sets S of atoms and/or vectors of terms. A homomorphism h is a *substitution*, if $h(N) = N$ for all $N \in \mathcal{N}$. An atom a is *homomorphic* (*substitutive*) to atom b, if some homomorphism (substitution) h exists such that $h(a) = b$. An isomorphism between two atoms a and b is a bijective homomorphism h s.t. $h(a) = b$ and $h^{-1}(b) = a$.

A set M of atoms is a model of a $Datalog^{\exists}$-program Π, denoted $M \models \Pi$, if is holds that $h(B(r)) \subseteq M$ for some substitution h and $r \in \Pi$ of form (2) implies that $h|_{\mathbf{X}}(H(r))$ is substitutive to some atom in M; the set of all models of Π is denoted by $mods(\Pi)$.

Next, we can introduce queries over $Datalog^{\exists}$-programs. A *conjunctive query* (CQ) q is an expression of form $\exists \mathbf{Y} : \leftarrow \mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$, where $\mathbf{Y}$ and $\mathbf{X}$ (the free variables) are disjoint sets of variables and $\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$ is a conjunction of ordinary atoms containing all and only the variables $\mathbf{X} \cup \mathbf{Y}$.

The answer of a CQ q with free variables $\mathbf{X}$ wrt. a model M is defined as follows:

$$ans(q, M) = \{h|_{\mathbf{X}} \mid h \text{ is a substitution and } h(\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]) \subseteq M\}.$$

Intuitively, this is the set of assignments to the free variables such that the query holds wrt. the model. The answer of a CQ q wrt. a program Π is then defined as the set $ans(q, \Pi) = \bigcap_{M \in mods(\Pi)} ans(q, M)$.

Query answering wrt. Π can be carried out over some *universal model* U of Π that is embeddable into each of its models by applying a suitable homomorphism. Formally, a model U of a program Π is called *universal* if, for each $M \in mods(\Pi)$, there is a homomorphism h s.t. $h(U) \subseteq M$. Thus, a universal model may be obtained using null values for unnamed individuals introduced by existential quantifiers. Moreover, it can be used to answer any query thanks to the following result.

Proposition 15 ([25]) *Let Π be a $Datalog^{\exists}$-program and let U be a universal model. Then, for any CQ q, it holds that $h \in ans(q, \Pi)$ iff $h \in ans(q, U)$ and $h : \mathcal{V} \to \mathcal{C} \setminus \mathcal{N}$.*

Intuitively, the set of all answers to q wrt. U which map all variables to non-null constants is exactly the set of answers to q wrt. Π.

Example 22 (cont'd) Reconsider Π consisting of the rules (3). The conjunctive query $\exists Y : \leftarrow person(X)$, $father(X, Y)$ asks for all persons who have a father. Its universal model is $U = \{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2), \ldots\}$. Hence, $ans(q, \Pi)$ contains answers $h_1(X) = john$ and $h_2(X) = joe$. □

Thus, computing a universal model is a key issue for query answering. A common approach for this step is the chase procedure. Intuitively, it starts from an empty interpretation and iteratively adds the head atoms of all rules with satisfied bodies, where existentially quantified variables are substituted by fresh nulls. However, in general this procedure does not terminate. Thus, a restricted *parsimonious chase procedure* was introduced by [33], which derives less atoms, and which is guaranteed to terminate for the class of *Shy-programs*. Moreover, it was shown that the interpretation computed by the parsimonious chase procedure is, although not a model of the program in general, still sound and complete for query answering and a *bounded model* in our view.

For query answering over $Datalog^{\exists}$-programs we reuse the translation in Section 6.1.

Example 23 (cont'd) The $Datalog^{\exists}$-program Π in Example 21, has the following HEX-translation:

$$\begin{aligned} T_{\exists}(\Pi): \quad & person(john). \quad person(joe). \\ & father(X, Y) \leftarrow person(X), \&exists^{1,1}[r_1, X](Y). \\ & person(Y) \leftarrow father(X, Y). \end{aligned}$$

While $T_{\exists}(\Pi)$ is *not* lde-safe in general, the hooks in Algorithm BGroundHEX can be used to still guarantee termination. We define the algorithm GroundDatalog$^{\exists}(\Pi, k)$ as BGroundHEX$(T_{\exists}(\Pi))$ where Repeat tests for $i < k + 1$ where k is the number of existentially quantified variables in the query, and Evaluate$(PIT_i, x) = true$ iff atom x is *not* homomorphic to any $a \in PIT_i$.

The produced program has a single answer set, which essentially coincides with the result of $pChase$ [33] that can be used for query answering. Thus, query answering over Shy-programs is reduced to grounding and solving of a HEX-program.

Theorem 16 *For a Shy-program Π, GroundDatalog$^{\exists}(\Pi, k)$ has a unique answer set which is sound and complete for answering CQs with up to k existential variables.*

The main difference to $pChase$ by [33] is essentially due to the homomorphism check. Actually, $pChase$ instantiates existential variables in rules with satisfied body to new null values only if the resulting head atom is not homomorphic to an already derived atom. In contrast, our algorithm performs the homomorphism check for the input to $\&exists^{n,m}$ atoms. Thus, homomorphisms are detected when constants are cyclically sent to the external atom. Consequently, our approach may need one iteration more than $pChase$, but allows for a more elegant integration into our algorithm.

Example 24 (cont'd) For the program Π in Example 23, the algorithm computes a program with the single answer set $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2)\}$. In contrast, $pChase$ stops earlier with the answer set $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2)\}$ as the atoms $person(\omega_1), person(\omega_2)$ are homomorphic to $person(john), person(joe)$.

Formally, one can show that $\mathsf{GroundDatalog}^{\exists}(\Pi, k)$ yields, for a Shy-program Π, a program with a single answer set that is equivalent to $pChase(\Pi, k+1)$ by [33]. Lemma 4.9 by [33] implies that the resulting answer set can be used for answering queries with k different existentially quantified variables, which proves Theorem 16.

While $pChase$ intermingles grounding and computing a universal model, our algorithm cleanly separates the two stages; modularized program evaluation by the solver will however also effect such intermingling. We nevertheless expect the more clean separation to be advantageous for extending Shy-programs to programs that involve existential quantifiers and other external atoms, or existential quantifiers in presence of disjunction; this remains for future work.

6.3 HEX-Programs with Function Symbols

In this section we show how to process terms with function symbols by a rewriting to lde-safe HEX-programs, which is another application of the grounding algorithm. We will briefly discuss advantages of our approach compared to a direct implementation of function symbols.

We consider HEX-programs, where the arguments X_i for $1 \leq i \leq \ell$ of ordinary atoms $p(X_1, \ldots, X_\ell)$, and the constant input arguments in $\mathbf{X}$ and the output $\mathbf{Y}$ of an external atom $\&g[\mathbf{X}](\mathbf{Y})$ are from a set of *terms* $\mathcal{T}$, that is the least set $\mathcal{T} \supseteq \mathcal{V} \cup \mathcal{C}$ such that $f \in \mathcal{C}$ (constant symbols are also used as function symbols) and $t_1, \ldots, t_n \in \mathcal{T}$ imply $f(t_1, \ldots, t_n) \in \mathcal{T}$.

Following [8], we introduce for all $k \geq 0$ external predicates $\&comp_k$ and $\&decomp_k$ with parameters $ar_{\text{I}}(\&comp_k) = 1+k$, $ar_{\text{O}}(\&comp_k) = 1$, $ar_{\text{I}}(\&decomp_k) = 1$, and $ar_{\text{O}}(\&decomp_k) = 1+k$. We define

$$f_{\&comp_k}(\mathbf{A}, f, X_1, \ldots, X_k, T) = f_{\&decomp_k}(\mathbf{A}, T, f, X_1, \ldots, X_k) = 1,$$

iff $T = f(X_1, \ldots, X_k)$.

Composition and decomposition of function terms can be simulated using these external predicates. Function terms are replaced by new variables and appropriate additional external atoms with predicate $\&comp_k$ or $\&decomp_k$ in rule bodies to compute their values. More formally, the rewriting is as follows.

For any HEX-program Π with function symbols, let $T_f(\Pi)$ be the program where each occurrence of a term $t = f(t_1, \ldots, t_n)$ in a rule r s.t. $B(r) \neq \emptyset$ is recursively replaced by a new variable V, and if V occurs afterwards in $H(r)$ or the input list of an external atom in $B(r)$, we add $\&comp_n[f, t_1, \ldots, t_n](V)$ to $B(r)$; otherwise (i.e., V occurs afterwards in some ordinary body atom or the output list of an external atom), we add $\&decomp_n[V](f, t_1, \ldots, t_n)$ to $B(r)$.

Intuitively, $\&comp_n$ is used to construct a nested term from a function symbol and arguments, which can be nested terms themselves, and $\&decomp_n$ is used to extract the function symbol and the arguments from a nested term. The translation can be optimized wrt. evaluation efficiency, which we disregard here.

Example 25 Consider the following HEX-program Π with function symbols and its translation:

$$\begin{array}{lll}
\Pi\colon & q(z).\ q(y). & \\
& p(f(f(X))) \leftarrow q(X). & \\
& r(X) \leftarrow p(X). & \\
& r(X) \leftarrow r(f(X)). &
\end{array}
\qquad
\begin{array}{lll}
T_f(\Pi)\colon & q(z).\ q(y). & \\
& p(V) & \leftarrow q(X), \&comp_1[f, X](U), \\
& & \quad \&comp_1[f, U](V). \\
& r(X) & \leftarrow p(X). \\
& r(X) & \leftarrow r(V), \&decomp_1[V](f, X).
\end{array}$$

Intuitively, $T_f(\Pi)$ builds $f(f(X))$ for any X on which q holds using two atoms over $\&comp_1$, and it extracts terms X from derived $r(f(X))$ facts using an appropriate $\&decomp_1$-atom. □

Note that $\&decomp_n$ supports a well-ordering on term depth such that its output has always a strictly smaller depth than its inputs. This is an important property for proving finite groundability of a program by exploiting the TBFs introduced by [15].

Example 26 The program $\Pi = \{q(f(f(a)));\ q(X) \leftarrow q(f(X))\}$ is translated to program $T_f(\Pi) = \{q(f(f(a)));\ q(X) \leftarrow q(V), \&decomp_1[V](f, X)\}$. Since the external source $\&decomp_1$ supports a well-ordering, the cycle is *benign* [15], i.e., it cannot introduce infinitely many values because the nesting depth of terms is strictly decreasing with each iteration. □

The realization of function symbols via external atoms has the advantage that their processing can be controlled. For instance, new nested terms may be restricted by additional conditions which can be integrated in the semantics of the external predicates $\&comp_k$ and $\&decomp_k$. A concrete example is *data type checking*, i.e., testing whether the arguments of a function term are from a certain domain. In particular, values might be rejected (e.g., for generation bounded up to a maximal term depth). Another example is computing some of the term arguments from others, e.g., constructing the function term $num(7, vii)$ from 7, where the second argument is the Roman representation of the first argument.

A further advantage of using external atoms for function term processing is that the expressive framework of lde-safety of HEX-programs can be exploited to guarantee finiteness of the grounding; no safety criteria specific for function terms have to be enforced.

7 Related Notions of Safety

Our notion of lde-safety using b_{s^2} compares to the traditionally used strong de-safety and to other formalizations

7.1 Strong Safety

One can now show that liberal de-safety is strictly less restrictive than strong de-safety.

Proposition 17 *Every strongly de-safe program* Π *is lde-safe.*

The converse does not hold, as there are clearlyx lde-safe programs that are not strongly safe, cf. Example 5.

7.2 VI-Restricted Programs

[8] introduced the notion of *VI-restrictedness* for *VI programs*, which amount to the class of HEX-programs in which all input parameters to external atoms are of type $\mathbf{const}$. Their notion of attribute dependency graph is related to ours, but our notion is more fine-grained for attributes of external predicates. While we use a node $\&g[\mathbf{X}]_r\restriction_T i$ for each external predicate $\&g$ with input list $\mathbf{X}$ in a rule r and $T \in \{\text{I}, \text{O}\}$, $1 \leq i \leq ar_T(\&g)$, Calimeri et al. use just one attribute $\&g\restriction i$ for each $i \in \{1, \ldots, ar_{\text{I}}(\&g)+ar_{\text{O}}(\&g)\}$ independent of $\mathbf{X}$. Thus, neither multiple occurrences of $\&g$ with different input lists in a rule, Thus, neither multiple occurrences of $\&g$ with different input lists in a rule, nor of the same attribute in multiple rules are distinguished; this collapses distinct nodes in our attribute dependency graph into one.

Example 27 Consider the following program $\Pi = \{r_1\colon\ t(X) \leftarrow s(X), \&e[X](Y),$
$r_2\colon\ r(X) \leftarrow t(X), \&e[X](Y)\}$. We have the attributes $s{\restriction}1, t{\restriction}1, r{\restriction}1, \&e[X]_{r_1}{\restriction}_{\mathrm{I}}1, \&e[X]_{r_1}{\restriction}_{\mathrm{O}}1, \&e[X]_{r_2}{\restriction}_{\mathrm{I}}1,$ $\&e[X]_{r_2}{\restriction}_{\mathrm{O}}2$ with edges $(s{\restriction}1, \&e[X]_{r_1}{\restriction}_{\mathrm{I}}1)$, $(\&e[X]_{r_1}{\restriction}_{\mathrm{I}}1, \&e[X]_{r_1}{\restriction}_{\mathrm{O}}1)$, and $(\&e[X]_{r_1}{\restriction}_{\mathrm{O}}1, t{\restriction}1)$ originating from the first rule of Π, and the edges $(t{\restriction}1, \&e[X]_{r_2}{\restriction}_{\mathrm{I}}1)$, $(\&e[X]_{r_2}{\restriction}_{\mathrm{I}}1, \&e[X]_{r_2}{\restriction}_{\mathrm{O}}1)$, $(\&e[X]_{r_2}{\restriction}_{\mathrm{O}}1, t{\restriction}r)$ originating from the second rule of Π.

In contrast, [8] have attributes $s{\restriction}1, t{\restriction}1, r{\restriction}1, \&e{\restriction}1, \&e{\restriction}2$ with edges $(s{\restriction}1, \&e{\restriction}1)$, $(\&e{\restriction}1, \&e{\restriction}2)$, $(\&e{\restriction}2, t{\restriction}1), (t{\restriction}1, \&e{\restriction}1), (\&e{\restriction}2, r{\restriction}1)$.

Using b_{s^2}, we establish the following result.

Proposition 18 *Every VI-restricted program* Π *is lde-safe.*

The converse does not hold, as there are clearly lde-safe VI-programs (due to semantic criteria) that are not VI-restricted.

7.3 Logic Programs with Function Symbols

[41] defined ω-restricted logic programs, which allow function symbols in an ordinary logic program under a level mapping to control the introduction of new terms with function symbols to ensure decidability. [8] observe that such programs Π can be rewritten to VI-programs $F(\Pi)$ using special external predicates that compose/decompose terms from/into function symbols and a list of arguments, such that $F(\Pi)$ is VI-restricted. As every VI-restricted program, viewed as a HEX-program, is by Proposition 18 also lde-safe, we obtain:

Proposition 19 *If an ordinary logic program* Π *is* ω*-restricted, then* $F(\Pi)$ *is lde-safe.*

As lde-safety is strictly more liberal than VI-restrictedness, it is also more liberal than ω-restrictedness. More expressive variants of ω-restricted programs are λ-restricted programs [26], argument-restricted programs [34] and Γ-acyclic programs [30]. These notions can be captured within our framework as well, but argument-restricted programs Π exist such that $F(\Pi)$ is *not* lde-safe wrt. b_{s^2}. The reason is that specific properties of the external atoms resp. sources for term (de)composition are needed, while our TBF b_{s^2} builds on general external sources. However, tailored TBFs can be used (which shows the flexibility of our modular approach). A further generalization of Γ-acyclic programs are *bounded programs*, which do not track the propagation of single terms through the program but consider whole rule bodies [29]. This allows for deriving termination even if the term depth of single terms increases, as long as the set of terms in a rule does not increase. Also the notions of *model-faithful (MFA)* and *model-summarising acyclicity (MSA)* [28], which have been developed in the context of positive existential rules, can be expressed in our framework. They are both more refined than other notions of acyclicity to single out cases where the *chase* procedure for query answering terminates, but MSA acyclicity is coarser and less complex to check than MFA acyclicity. The key idea is a dependency analysis by examining the actual structure of the so called universal model of the program formed by the existential rules. This analysis, however, can be done within a term bounding function, and thus these notions can be captured in our framework.

Similarly, i.e., by means of dedicated external atoms for (de)composing terms and a specialized TBF, so-called FD programs [9] map into our framework. Finitary programs [4, 3] and FG programs [9], however, differ more fundamentally from our approach and cannot be captured as lde-safe programs using TBFs, as they are not effectively recognizable (and FD-programs are even not finitely restrictable in general).

7.4 Term Rewriting Systems

A term rewriting system is a set of rules for rewriting terms to other terms. Termination is usually shown by proving that the right-hand side of every rule is strictly smaller than its left-hand side [42, 43]. Our notion of benign cycles is similar, but different from term rewriting systems the values do not need to *strictly* decrease. While terms that stay equal may prevent termination in term rewriting systems, they do not harm in our case because they cannot expand the grounding infinitely.

7.5 Other Notions of Safety

Related to our semantic properties are the approaches by [38], [36], and [31]. They exploited finiteness of attributes (cf. item (ii) in Definition 14) in sets of Horn clauses and derive finiteness of further attributes using *finiteness dependencies*. This is related to item (iii) in Definition 14 and item (iii) in Definition 9. However, they did this for query answering over infinite databases but not for model building.

Less related to our work are [7], [32], and [2], where safety resp. argument restrictedness was extended to arbitrary first-order formulas with(out) function symbols under the stable model semantics rather than a generalization of the concepts given.

While a formal comparison with our approach is only possible for concrete other notions of safety, we can at a meta-level say that in general arbitrary notions of safety with finite groundings that are based on the program rules can be easily expressed in our approach. This is because TBFs as in Definition 7 have full access to the program at hand. Thus, the simulation of some approach by a term bounding function is easily possible by declaring all variables in all rules as bounded, if the approach considers the given program to be safe. While this naive simulation works for all approaches which can decide safety based on the program at hand, other notions (e.g. VI-restricted programs) are based on an iterative expansion of sets of safe variables, external sources or rules. For such approaches, a simulation by exploiting the incremental extension of bounded terms as in Definition 8 may be applied.

8 Conclusion

We have presented a framework for obtaining classes of HEX-programs, which are ASP programs with external sources, that allow for finite groundings sufficient for evaluation over an infinite domain (which arises by "value invention" in external atoms). It is based on term bounding functions (TBFs) and enables modular exchange and combination of such functions under the novel notion of *liberal domain expansion (lde) safety*. Hitherto separate syntactic and semantic safety criteria can be combined, which pushes the class of HEX-program with evaluation via finite grounding considerably. We have provided sample TBFs that capture syntactic criteria similar to but more fine-grained than ones by [8], and semantic criteria related to those by [38] and [36], but target model generation (not query answering). Deploying them, classes that strictly enlarger classes available through well-known safety notions for ASP programs are obtained, and other notions can be modularly integrated. An implementation of lde-safety in the DLVHEX-framework is available for use.

Together with lde-safety, we have also presented a new grounding algorithm for HEX-programs. In contrast to previous grounding techniques for ASP and HEX-programs, it can handle all lde-safe programs directly and does not rely on program decomposition. This is an advantage, as program splitting negatively affects learning techniques described in [13]. However, in the worst case the algorithm needs exponentially many external source calls to determine the relevant constants for grounding. We have 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 DLVHEX evaluation framework for its use. An experimental evaluation of our implementation on synthetic and real applications shows a clear benefit.

Furthermore, we have presented a generalization of the grounding algorithm which allows for controlled grounding by injecting application-specific stopping criteria; this can be fruitfully exploited e.g. for bounded model generation (as shown for query answering from existential rules), for parametric data structures, and for support of function symbols in HEX-programs.

Open Issues and Future Work. While lde-safe HEX-programs are ready for use via the DLVHEX-system, several issues remain naturally for future work. One such isses is to identify further TBFs and suitable well-orderings of domains in practice. Of particular interest are external atoms that provide built-in functions and simulate, in a straightforward manner, particular interpreted function symbols. On the implementation side, further refinement and optimizations are an issue, as well as a library of TBFs and a plugin architecture that supports creating customized TBFs easily, to make our framework more broadly usable. Connected with this are refinements of our algorithm and heuristics. Here, meta-information about external sources to identify programs that allow for a better grounding and to reduce worst case costs of inputs are of interest. A particular challenge is also a sensitive integration of grounding and solving under decomposition, which are currently not much aligned. Finally, exploring on the application side the benefits of our results e.g. for domain-specific value invention (i.e., existential quantifiers under restrictions, which occurs prominently in configuration problems) appears to be an interesting direction.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF) via the project P24090.

Literatur

[1] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. *Journal of Automated Reasoning*, 14(1):149–180, 1995.

[2] Michael Bartholomew and Joohyung Lee. A decidable class of groundable formulas in the general theory of stable models. In *12th International Conference on the Principles of Knowledge Representation and Reasoning (KR'10)*, pages 477–485. AAAI Press, 2010.

[3] Piero A. Bonatti. Reasoning with infinite stable models ii: Disjunctive programs. In *18th International Conference on Logic Programming (ICLP'02)*, volume 2401 of *LNCS*, pages 333–346. Springer, 2002.

[4] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75—111, 2004.

[5] Gerd Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In Robert C. Holte and Adele Howe, editors, *22nd AAAI Conference on Artificial Intelligence (AAAI'07)*, pages 385–390. AAAI Press, 2007.

[6] Gerd Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[7] Pedro Cabalar, David Pearce, and Agustín Valverde. A revised concept of safety for general answer set programs. In *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 58–70. Springer, 2009.

[8] Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3–4):333–361, 2007.

[9] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in asp: Theory and implementation. In *International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008.

[10] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.

[11] Paul E. Dunne. The computational complexity of ideal semantics. *Artificial Intelligence*, 173(18):1559–1591, 2009.

[12] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, and Peter Schüller. Pushing efficient evaluation of hex programs by modular decomposition. In James Delgrande and Wolfgang Faber, editors, *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *LNAI*, pages 93–106. Springer, May 2011.

[13] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, July 2012.

[14] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Grounding HEX-programs with expanding domains. In David Pearce, Shahab Tasharrofi, Evgenia Ternovska, and Concepción Vidal, editors, *Informal Proceedings of the 2nd Workshop on Grounding and Transformations for Theories with Variables (GTTV'13), Corunna, 15th, September 15, 2013, Corunna, Spain*, pages 3–15, 2013. Online available at http://gttv13.irlab.org/sites/10.56.35.200.gttv13/files/gttv13.pdf.

[15] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Liberal Safety Criteria for HEX-Programs. In Marie desJardins and Michael Littman, editors, *Twenty-Seventh AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA*, pages 267–275. AAAI Press, July 2013.

[16] Thomas Eiter, Michael Fink, Christoph Redl, and Thomas Krennwallner. HEX-programs with existential quantification. In *20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), Kiel, Germany, September 11-13, 2013, Preproceedings*, 2013.

[17] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[18] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 90–96. Professional Book Center, 2005.

[19] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In York Sure and John Domingue, editors, *3rd European Conference on Semantic Web (ESWC'06)*, volume 4011 of *LNCS*, pages 273–287. Springer, 2006.

[20] Thomas Eiter, Thomas Krennwallner, Matthias Prandtstetter, Christian Rudloff, Patrik Schneider, and Markus Straub. Semantically enriched multi-modal routing. *International Journal of Intelligent Transportation Systems Research*, August 2014. Published online: 05 August 2014.

[21] Thomas Eiter, Thomas Krennwallner, and Patrik Schneider. Lightweight spatial conjunctive query answering using keywords. In Philipp Cimiano, Óscar Corcho, Valentina Presutti, Laura Hollink, and Sebastian Rudolph, editors, *ESWC*, volume 7882 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2013.

[22] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. In Didier Dubois, Christopher Welty, and Mary-Anne Williams, editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 141–151. AAAI Press, 2004.

[23] Esra Erdem, Yelda Erdem, Halit Erdogan, and Umut Öztok. Finding answers and generating explanations for complex biomedical queries. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[24] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, January 2011.

[25] Ronald Fagin, Phokion Kolaitis, Renée Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[26] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483, pages 266–271. Springer, 2007.

[27] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3–4):365–386, 1991.

[28] Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *CoRR*, abs/1406.4110, 2014.

[29] Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Bounded programs: A new decidable class of logic programs with function symbols. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI 2013)*, IJCAI 2013, pages 926–931. AAAI Press, 2013.

[30] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. On the termination of logic programs with function symbols. In Agostino Dovier and Vítor Santos Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 323–333. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[31] Ravi Krishnamurthy, Raghu Ramakrishnan, and Oded Shmueli. A framework for testing safety and effective computability. *Journal of Computer and System Sciences*, 52(1):100–124, 1996.

[32] Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Safe formulas in the general theory of stable models (preliminary report). In *24th International Conference on Logic Programming (ICLP'08)*, volume 5366 of *LNCS*, pages 672–676. Springer, 2008.

[33] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Efficiently computable datalog$^{\exists}$ programs. In *KR*, 2012.

[34] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *25th International Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 489–493. Springer, 2009.

[35] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[36] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *6th Symposium on Principles of Database Systems (PODS'87)*, pages 328–339. ACM, 1987.

[37] Christoph Redl. *Answer Set Programming with External Sources: Algorithms and Efficient Evaluation*. PhD thesis, Vienna University of Technology, Knowledge-Based Systems Group, A-1040 Vienna, Karlsplatz 13, April 2014.

[38] Y. Sagiv and M. Y. Vardi. Safety of datalog queries over infinite databases. In *8th Symposium on Principles of Database Systems (PODS'89)*, pages 160–171. ACM, 1989.

[39] P. Schüller, V. Patoglu, and E. Erdem. Levels of integration between low-level reasoning and task planning. In *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013) – Intelligent Robotic Systems, Bellevue, WA, July 14th, 2013*. AAAI Press, 2013. to appear.

[40] Peter Schüller. *Inconsistency in Multi-Context Systems: Analysis and Efficient Evaluation*. PhD thesis, Vienna University of Technology, Vienna, Austria, August 2012.

[41] Tommi Syrjänen. Omega-restricted logic programs. In *6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 267–279. Springer, 2001.

[42] Hans Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994.

[43] Hans Zantema. The termination hierarchy for term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):3–19, 2001.

A Proofs

Proof of Proposition 2. There are only finitely many ordinary and external predicates with finite (input and output) arity. □

Proof of Proposition 3. We prove this by induction on n.

For $n = 0$ we have $S_0(\Pi) = \emptyset$ and the proposition holds trivially.

For the induction step $n \mapsto n+1$, assume that the attributes in $S_n(\Pi)$ are domain-expansion safe (outer induction hypothesis). We first show that for each rule r and term $t \in B_{n+1}(r, \Pi, b)$, the set of ground instances of r in $G^\infty_\Pi(\emptyset)$ contains only finitely many different substitutions for t. We consider $B_{n+1,j}(r, \Pi, b)$ and again prove this by induction on j. For $j = 0$ we have $B_{n+1,0}(r, \Pi, b) = \emptyset$ and the proposition holds trivially. For the induction step $j \mapsto j+1$, assume that the terms in $B_{n+1,j}(r, \Pi, b)$ are bounded (inner induction hypothesis). Let $t \in B_{n+1,j+1}(r, \Pi, b)$. If $t \in B_{n+1,j}(r, \Pi)$ then the claim follows from the inner induction hypothesis. Otherwise t is added in step $j+1$. By the outer induction hypothesis all attributes in $S_n(\Pi)$ have a finite range in $G^\infty_\Pi(\emptyset)$. By the inner induction hypothesis there are only finitely many substitutions for all terms $t \in B_{n+1,j}(r, \Pi, b)$ in $G^\infty_\Pi(\emptyset)$. This fulfills the conditions of TBFs (Definition 7). Since b is a TBF, this implies that there are also only finitely many substitutions for all $t \in b(r, S_n(\Pi), B_{n+1,j})$. This proves the inner induction statement and, by definition of $B_n(r, \Pi, b)$, also that for each $t \in B_{n+1}(r, \Pi, b)$ the set of ground instances of r in $G^\infty_\Pi(\emptyset)$ contains only finitely many different substitutions for t.

If $p{\restriction}i \in S_{n+1}(\Pi)$, then for each rule $r \in \Pi$ and atom $p(t_1, \ldots, t_{ar(p)}) \in H(r)$ we have $t_i \in B_{n+1}(r, \Pi, b)$. As we have shown, this means that there are only finitely many different substitutions for t_i in the ground instances of r in the set $G^\infty_\Pi(\emptyset)$. As there are also only finitely many different rules in Π, and the number of substitutions for the term t_i in the head of r is finite, this implies that also the set $\{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(G^\infty_\Pi(\emptyset))\}$ is finite.

If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} i \in S_{n+1}(\Pi)$, then the i-th input parameter is either of type constant and Y_i is a constant or a variable, or it is of type predicate. If it is of type constant and Y_i is a constant, then there exists only one ground instance. If it is of type constant and Y_i is a variable, then $Y_i \in B_{n+1}(r, \Pi, b)$, for which we have shown that there are only finitely many different substitutions for Y. If it is of type predicate input parameter p, then the range of all attributes $p{\restriction}1, \ldots, p{\restriction}ar(p)$ in $G^\infty_\Pi(\emptyset)$ is finite by the (outer) induction hypothesis.

If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{O} i \in S_{n+1}(\Pi)$, then either $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} 1, \ldots, \&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} ar_\mathrm{I}(\&g) \in S_n(\Pi)$, or r contains some $\&g[\mathbf{Y}](\mathbf{X})$ s.t. Y_i is bounded.

If $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} 1, \ldots, \&g[\mathbf{Y}]_r{\restriction}_\mathrm{I} ar_\mathrm{I}(\&g) \in S_n(\Pi)$, then the range of all input parameters in $G^\infty_\Pi(\emptyset)$ is finite by the (outer) induction hypothesis. But then there exist only finitely many oracle calls to $\&g$. As each such call can introduce only finitely many new values, also the range of each output parameter in $G^\infty_\Pi(\emptyset)$ is finite. If r contains an external atom $\&g[\mathbf{Y}](\mathbf{X})$ such that Y_i is bounded, then only finitely many substitutions for $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{O} i$ can satisfy the rule body, thus $G^\infty_\Pi(\Pi)$ will also contain only finitely many values for $\&g[\mathbf{Y}]_r{\restriction}_\mathrm{O} i$. Thus, the (outer) induction hypothesis holds for $n+1$, which proves the statement. □

Proof of Corollary 4. If $a \in S_\infty$ then $a \in S_n$ for some $n \geq 0$ and the claim follows from Proposition 3. □

Proof of Corollary 5. Since Π is domain-expansion safe by assumption, $a \in S_\infty(\Pi)$ for all attributes a of Π. Then by Corollary 4, the range of all attributes of Π in $G^\infty_\Pi(\emptyset)$ is finite. But then there exists also only a finite number of ground atoms in $G^\infty_\Pi(\emptyset)$. Since the original non-ground program Π is finite, this implies that also the grounding is finite. □

Proof of Proposition 6. We construct the grounding $grnd_C(\Pi)$ as the least fixpoint $G^\infty_\Pi(\emptyset)$ of the grounding operator $G_\Pi(X)$, which is known to be finite by Corollary 5. The set C is then implicitly given by the set of constants appearing in $grnd_C(\Pi)$. It remains to show that indeed $grnd_C(\Pi) \equiv^{pos} grnd_\mathcal{C}(\Pi)$. We will show the more general proposition $grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$ for any $C' \supseteq C$.

($\Rightarrow$) Suppose $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Let $\mathbf{A}' = \mathbf{A} \cup \big\{\mathbf{F}a \mid a \in A\big(grnd_{C'}(\Pi)\big),\ \mathbf{T}a \notin \mathbf{A}\big\}$, i.e., the completion of $\mathbf{A}$ to all atoms in $grnd_{C'}(\Pi)$ by setting all additional atoms to false. Then $\{\mathbf{T}a \in \mathbf{A}\} =$

$\{\mathbf{T}a \in \mathbf{A}'\}$. We show now that $\mathbf{A}'$ is an answer set of $grnd_{C'}(\Pi)$. First observe that $\mathbf{A}' \not\models B^+(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$; otherwise $r \in G_\Pi\big(grnd_C(\Pi)\big)$, which contradicts the assumption that $grnd_C(\Pi)$ is the least fixpoint of $G_\Pi(\emptyset)$. Therefore, $\mathbf{A}' \models grnd_{C'}(\Pi)$. Moreover $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}'}$, hence $\mathbf{A}'$ is a subset-minimal model of the FLP-reduct of $grnd_{C'}(\Pi)$ iff $\mathbf{A}$ is a subset-minimal model of the FLP-reduct of $grnd_C(\Pi)$, which is the case because $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Therefore $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Now suppose $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Then we have $\mathbf{A}' = \mathbf{A} \cap \big\{\mathbf{T}a, \mathbf{F}a \mid a \in A\big(grnd_C(\Pi)\big)\big\}$ is a model of $grnd_C(\Pi)$. Let $\mathbf{A}'' = \mathbf{A}' \cup \{\mathbf{F}a \mid a \in A\big(grnd_{C'}(\Pi)\big), \mathbf{T}a \notin \mathbf{A}'\}$, i.e., the completion of $\mathbf{A}'$ to all atoms in $grnd_{C'}(\Pi)$ by setting all additional atoms to false. Then we have $\mathbf{A}'' \not\models B^+(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$; otherwise $r \in G_\Pi\big(grnd_C(\Pi)\big)$, which contradicts the assumption that $grnd_C(\Pi)$ is the least fixpoint of $G_\Pi(\emptyset)$. Therefore, $\mathbf{A}'' \models grnd_{C'}(\Pi)$. But this implies that $\mathbf{A} = \mathbf{A}''$: by construction of $\mathbf{A}''$ we have $\mathbf{A}''^{\mathbf{T}} \subseteq \mathbf{A}^{\mathbf{T}}$, and $\mathbf{A}''^{\mathbf{T}} \subsetneq \mathbf{A}^{\mathbf{T}}$ would imply that $\mathbf{A}$ is not subset-minimal, which contradicts the assumption that $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Moreover, $fgrnd_C(\Pi)^{\mathbf{A}'} = fgrnd_{C'}(\Pi)^{\mathbf{A}''}$. Therefore $\mathbf{A}'$ is a subset-minimal model of the FLP-reduct of $grnd_C(\Pi)$ iff $\mathbf{A}''$ is a subset-minimal model of the FLP-reduct of $grnd_{C'}(\Pi)$, which is the case because $\mathbf{A}'' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. Thus we have $\mathbf{A}' \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. The observation $\{\mathbf{T}a \in \mathbf{A}'\} = \{\mathbf{T}a \in \mathbf{A}''\}$ concludes the proof. □

Proof of Proposition 7. If t is in the output of $b_{syn}(\Pi, r, S, B)$, then one of the conditions holds. If Condition (i) holds, then t is a constant, hence there is only one ground instance. If Condition (ii) holds, then t must also occur as value for $q\restriction j$, which has a finite range by assumption.

If Condition (iii) holds, then t is output of an external atom such that there are only finitely many substitutions of its constant inputs and the attributes of all predicate inputs have a finite range by assumption. Thus there are only finitely many different oracle calls with finite output each. □

Proof of Proposition 8. If t is in the output of $b_{sem}(\Pi, r, S, B)$, then one of the conditions holds. If Condition (i) holds, then there is no information flow from malign cycles wrt. S to t. However, such cycles are the only source of infinite groundings: the attributes in S have a finite domain by assumption. For the remaining attributes in the cycle, the well-ordering guarantees that only finitely many different values can be produced in the cycle.

If Condition (ii) holds, then the claim follows immediately from finiteness of the domain of the respective external atom.

If Condition (iii) holds, then the external atom cannot introduce new constants. Because the set of constants in the extension of the respective input parameter Y_j is finite by assumption that $Y_j\restriction k \in S$ for all $1 \leq k \leq ar(Y_j)$, it follows that also the set of constants in the output of the external atom is finite.

If Condition (iv) holds, then there are only finitely many different substitutions for t because the output of the respective external atom is bound by the precondition of TBFs and the finite fiber ensures that there are only finitely many different inputs for each output. □

Proof of Theorem 9. For $t \in b(\Pi, r, S, B)$, $t \in b_i(\Pi, r, S, B)$ for some $1 \leq i \leq \ell$. Then there are only finitely many substitutations for t in $G^\infty_\Pi(\emptyset)$ because b_i is a TBF. □

Towards a proof of Theorem 10, we first prove the following lemma.

Lemma 20 *Let $\Pi_g = \mathit{GroundHEXNaive}(\Pi)$ and C be the constants which appear in Π_g. Then for any $C' \supseteq C$ and each model $\mathbf{A}$ of $grnd_C(\Pi)$, $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$.*

Proof. Let $\mathbf{A}$ be a model of $grnd_C(\Pi)$. Then it can be extended to a model $\mathbf{A}_{pg}$ of Π_{pg} as follows:

- For all $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.

- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$, for all predicates $g^{\&g}_{inp}$ occurring in the head of some r^a_{inp} (for an external atom $a = \&g[\mathbf{Y}](\mathbf{X})$).

This satisfies each ground instance of each input auxiliary rule r^a_{inp} because the head $g^{\&g}_{inp}(\mathbf{y})$ is true. Moreover, because $\mathbf{A}$ is a model of $grnd_C(\Pi) = \Pi_g$ and Π_{pg} contains $e_{r,\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ to true iff $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$, it satisfies also all remaining rules.

We show now that $\mathbf{A}$ is also a model of $grnd_{C'}(\Pi)$. Let $r \in grnd_{C'}(\Pi)$ and suppose $\mathbf{A} \not\models r$, then $\mathbf{A} \not\models H(r)$ and $\mathbf{A} \models B(r)$. Since $\mathbf{A} \models B(r)$, we have $\mathbf{A} \models a$ for each ordinary literal $a \in B(r)$. If there would be only ordinary literals in $B(r)$, then Π_g would also contain this rule instance because all constants in $B(r)$ must appear in the atoms which are true in $\mathbf{A}$ und thus in Π_g. Hence, $\mathbf{A}$ could not be a model of Π_g. Therefore there must be external atoms in $B(r)$.

We show now that each positive external atom in r is represented in Π_{pg} (with degree 0). Suppose there is an external atom in $B(r)$ which is not represented in Π_{pg}. Then, due to safety of r, which forbids cyclic passing of constant input within a rule body, there is also a 'first' unrepresented external atom $\&g[\mathbf{v}](\mathbf{u})$, i.e., one such that all its input constants in $\mathbf{v}$ either: (1) appear in a positive ordinary atom, (2) appear in the output list of a represented external atom, or (3) were already constants in the input program. In all three cases, the input auxiliary rule for $\&g[\mathbf{v}](\mathbf{u})$ is instantiated for this $\mathbf{v}$ because its body atoms are potentially true (they are ordinary atoms or replacement atoms of represented external atoms), i.e., $g^{\&g}_{inp}(\mathbf{v})$ appears in the program and is therefore true in $\mathbf{A}_{pg}$. Thus, the loop at (e) would evaluate $\&g$ with $\mathbf{A}_{pg}$ and $\mathbf{v}$ and determine all tuples $\mathbf{w}$ s.t. $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{w}) = 1$. However, $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{u}) = 0$, because otherwise rule $e_{r,\&g[\mathbf{v}]}(\mathbf{u}) \vee ne_{r,\&g[\mathbf{v}]}(\mathbf{u}) \leftarrow$ would have been added at (f) to Π_p and thus $\&g[\mathbf{v}](\mathbf{u})$ would be represented in Π_{pg}, which contradicts our assumption. But if $f_{\&g}(\mathbf{A}_{pg}, \mathbf{v}, \mathbf{u}) = 0$ then also $f_{\&g}(\mathbf{A}, \mathbf{v}, \mathbf{u}) = 0$ because $\mathbf{A}_{pg}$ and $\mathbf{A}$ differ only on input auxiliary atoms and external atom replacement atoms, which would imply $\mathbf{A} \not\models B(r)$.

Thus, all positive external atoms are represented in Π_{pg}. But as default-negated ones cannot introduce new values due to ordinary safety, all constants in r also appear in Π_g, thus $r \in \Pi_g$. But then $\mathbf{A}$ could not be a model of Π_g if $\mathbf{A} \models B(r)$, hence $\mathbf{A} \not\models B(r)$. □

Now we can show the result of Theorem 10.

Proof of Theorem 10. Let $\Pi_g = \mathsf{GroundHEXNaive}(\Pi)$. For the proof, observe that $\Pi_g = grnd_C(\Pi)$ where C is the set of all constants which appear in Π_g. We show now

$$grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$$

for any $C' \supseteq C$. Because $\Pi \equiv^{pos} grnd_{\mathcal{C}}(\Pi)$ for Herbrand universe $\mathcal{C} \supseteq C$ by definition of the HEX-semantics, this implies the proposition.

Termination of the algorithm follows from Theorem 11, where we will prove that an optimized version of the algorithm, which may produce a larger grounding (wrt. the number of constants) but need less iterations, terminates. As the grounding produced by this algorithm is even smaller, it terminates as well.

($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. By Lemma 20 it is also a model of $grnd_{C'}(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Since $C \subseteq C'$, $fgrnd_C(\Pi)^{\mathbf{A}} \subseteq fgrnd_{C'}(\Pi)^{\mathbf{A}}$. By Lemma 20, $\mathbf{A} \not\models B(r)$ for any $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, thus $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. But since $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, it is a minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$, thus also of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$, i.e., $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Let $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. We show that $\mathbf{A} = \mathbf{A}' \cap A\big(grnd_C(\Pi)\big)$ is an answer set of $grnd_C(\Pi)$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$, it is trivial that $\mathbf{A}$ is a model of $grnd_C(\Pi)$. It remains to show that

it is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. By Lemma 20, $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. Clearly, $\mathbf{A} \subseteq \mathbf{A}'$. But $\mathbf{A} \subsetneq \mathbf{A}'$ would imply that $\mathbf{A}'$ is not subset-minimal, which contradicts the assumption that it is an answer set of $grnd_{C'}(\Pi)$, thus $\mathbf{A} = \mathbf{A}'$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$ and $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, it holds that $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Because $\mathbf{A}$ is a subset-minimal model of $grnd_{C'}(\Pi)^{\mathbf{A}}$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$. □

In order to prove soundness and completeness of our optimized algorithm, we first show a lemma analogous to Lemma 20.

Lemma 21 *Let $\Pi_g = \mathsf{GroundHEX}(\Pi)$ and C be the constants which appear in Π_g. Then for any $C' \supseteq C$ and each model $\mathbf{A}$ of $grnd_C(\Pi)$, $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$.*

Proof. Let $\mathbf{A}$ be an model of $grnd_C(\Pi)$. Then it can be extended to a model $\mathbf{A}_{pg}$ of Π_{pg} as follows:

- For all $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}_g, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$, for all predicates $g^{\&g}_{inp}$ occurring in the head of some r^a_{inp} (for an external atom $a = \&g[\mathbf{Y}](\mathbf{X})$).

This satisfies all guessing rules as for each $\&g[\mathbf{y}](\mathbf{x})$ one of the atoms $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ or $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is true, and each input auxiliary rule r^a_{inp} because the head $g^{\&g}_{inp}(\mathbf{y})$ is true. Moreover, because $\mathbf{A}$ is a model of $grnd_C(\Pi)$, it is also a model of the (possibly) less restrictive program Π_g. Since Π_{pg} contains $e_{r,\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ in $\mathbf{A}_{pg}$ to true iff $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$, $\mathbf{A}_{pg}$ satisfies also all remaining rules in program Π_{pg}.

We show now that $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. Let $r \in grnd_{C'}(\Pi)$ and suppose $\mathbf{A} \not\models r$, i.e., $\mathbf{A} \not\models H(r)$ but $\mathbf{A} \models B(r)$.

As we have seen in the proof of Theorem 10, all lde-safety relevant positive external atoms in r are represented with degree 0 in the program computed by Algorithm GroundHEXNaive. As such external atoms are handled equivalently by our optimized algorithm, they are also represented in Π_{pg}. We show that this holds also for positive external atoms which are not lde-safety relevant.

Suppose r contains an external atom which is not lde-safety relevant and which is not represented in Π_{pg}. Then there is a 'first' such external atom $\&g[\mathbf{v}](\mathbf{u})$ in $B(r)$, i.e., its input list only contains constants which (1) appear in ordinary atoms, (2) appear in lde-safety relevant external atoms, or (3) were already constants in the input program. In all three cases, the input auxiliary rule for $\&g[\mathbf{v}](\mathbf{u})$ is instantiated for this $\mathbf{v}$ because its body atoms are potentially true (ordinary atoms appear also in $B(r)$ and are potentially true, otherwise r would not have been added to Π_g; external atoms are all not lde-safety relevant and are potentially true since they are represented with degree 0), i.e., $g^{\&g}_{inp}(\mathbf{v})$ appears in the program. Moreover, the respective external atom guessing rule is instantiated for $\mathbf{v}$ and $\mathbf{u}$ because all its body atoms are potentially true (with the same argument as for input auxiliary rules). Thus, $\&g[\mathbf{v}](\mathbf{u})$ would be represented in Π_p with some degree > 0, and thus also in Π_{pg} and Π_g.

Thus, all positive external atoms are represented in Π_{pg}. But as default-negated ones cannot introduce new values due to ordinary safety, all constants in r also appear in Π_g, thus a strengthening of r would be in Π_g. But then $\mathbf{A}$ could not be a model of Π_g if $\mathbf{A} \models B(r)$, hence $\mathbf{A} \not\models B(r)$. □

We now show some additional lemmas to simplify the proof of soundness and completeness of the optimized algorithm GroundHEX.

Lemma 22 *Let* $\Pi_g = \mathit{GroundHEX}(\Pi)$ *and* C *be the constants which appear in* Π_g*. Every answer set* $\mathbf{A}$ *of* Π_g *can be extended to an answer set* $\mathbf{A}_{pg}$ *of* Π_{pg}*.*

Proof. Let $\mathbf{A} \in \mathcal{AS}(\Pi_g)$. Then $\mathbf{A}_{pg}$ is constructed by iteratively adding additional atoms to $\mathbf{A}$ as follows:

- If the body B of an external atom guessing rule $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in Π_{pg} is satisfied by $\mathbf{A}_{pg}$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{r,\&g\mathbf{y}}(\mathbf{x})$ otherwise.

- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A(\Pi_{pg})$ if the body of the respective input auxiliary rule is satisfied by $\mathbf{A}_{pg}$.

Note that this operation is monotonic because input auxiliary rules and external atom guessing rules contain only positive body literals.

Then the fixpoint of this operation $\mathbf{A}_{pg}$ is by construction a model of all input auxiliary rules and external atom guessing rules. Moreover, it is also a model of all remaining rules because $\mathbf{A}$ is a model of the corresponding rules in Π_g with external atoms in place of replacement atoms, and we set the truth values of the external atom replacement atoms exactly to the truth values of the external atoms in $\mathbf{A}$. Note that there might be external atoms in Π_g for which neither $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ nor $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is added to $\mathbf{A}_{pg}$, but then the body of the respective external atom guessing rule is unsatisfied by $\mathbf{A}_{pg}$. But since the body of an external atom guessing rule is a subset of the body of the rule where this external atom occurs, also this rule is satisfied.

It remains to show that $\mathbf{A}_{pg}$ is also a subset-minimal model of ${f\Pi_{pg}}^{\mathbf{A}_{pg}}$. Suppose there is a smaller model $\mathbf{A}'_{pg} \subsetneq \mathbf{A}_{pg}$. Then $\mathbf{A}_{pg} \setminus \mathbf{A}'_{pg}$ must contain at least one atom which is not a replacement atom or an input auxiliary atom, because by construction of $\mathbf{A}_{pg}$ such atoms are only set to true if necessary, i.e., if they are supported by $\mathbf{A}$, and all rules used to derive such atoms are also in ${f\Pi_{pg}}^{\mathbf{A}_{pg}}$. We now show that the restriction of $\mathbf{A}$ to ordinary atoms $\mathbf{A}' \subsetneq \mathbf{A}$ (i.e., without replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ and without external atom input atoms $g^{\&g}_{inp}(\mathbf{y})$) is a model of ${f\Pi_g}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g.

Observe that, except for the external atom guessing and input auxiliary rules, the reduct ${f\Pi_{pg}}^{\mathbf{A}_{pg}}$ contains the same rules as ${f\Pi_g}^{\mathbf{A}}$ with replacement atoms instead of external atoms. Thus, for $r \in {f\Pi_g}^{\mathbf{A}}$, the corresponding $r_{pg} \in {f\Pi_{pg}}^{\mathbf{A}_{pg}}$ contains the same ordinary literals in the head and body.

We show now that $\mathbf{A}'_{pg} \models r_{pg}$ implies $\mathbf{A}' \models r$. If $\mathbf{A}'_{pg}$ is a model of r_{pg}, then we have either (1) $\mathbf{A}'_{pg} \models h$ for some $h \in H(r_{pg})$, or (2) $\mathbf{A}'_{pg} \not\models b$ for some $b \in B(r_{pg})$. In Case (1), we also have $h \in H(r)$. Since $\mathbf{A}'_{pg}$ and $\mathbf{A}'$ coincide on non-replacement and non-input atoms, this implies $\mathbf{A}' \models r$. In Case (2), b is either (2a) a non-replacement literal, or (2b) a (positive or default-negated) external atom replacement. In Case (2a), we also have $b \in B(r)$. Since $\mathbf{A}'_{pg}$ and $\mathbf{A}'$ coincide on such atoms, this implies $\mathbf{A}' \models r$. In Case (2b), we either have (2b′) $\mathbf{A}_{pg} \not\models b$, or (2b″) b is positive (since a default-negated atom cannot become false by removing atoms from the interpretation) and some literal b' in the body of the external atom guessing or in the input rule for b is false in $\mathbf{A}'_{pg}$; in this case b is represented in Π_p with some degree n. In Case (2b′), $\mathbf{A}$ falsifies by construction of $\mathbf{A}_{pg}$ the external atom in $B(r)$ which corresponds to the replacement atom b. In Case (2b″), b' also appears in $B(r_{pg})$. Note that b' can be another external replacement atom. But in this case, the external atom corresponding to b' is represented with some degree $< n$. Thus, we start the case distinction for b' again. However, because the degree is reduced with every iteration, we will eventually end up in one of the other cases.

Thus, $\mathbf{A}'$ would be a model of ${f\Pi_g}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g. This shows that $\mathbf{A}_{pg}$ is an answer set of Π_{pg}. □

Lemma 23 *Let* $\Pi_g = \mathsf{GroundHEX}(\Pi)$ *and* C *be the constants which appear in* Π_g*. Every answer set* $\mathbf{A}$ *of* $grnd_C(\Pi)$ *can be extended to an answer set* $\mathbf{A}_p$ *of* $grnd_C(\Pi_p)$*.*

Proof. Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. Then $\mathbf{A}_p$ is constructed by iteratively adding additional atoms to $\mathbf{A}$ as follows:

- If the body B of an external atom guessing rule $e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in $grnd_C(\Pi_p)$ is satisfied by $\mathbf{A}_p$, add $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ otherwise.
- Add all $g^{\&g}_{inp}(\mathbf{y}) \in A\big(grnd_C(\Pi_p)\big)$ if the body of the respective input auxiliary rule is satisfied by $\mathbf{A}_p$.

Note that this operation is monotonic because input auxiliary rules and external atom guessing rules contain only positive body literals.

Then the fixpoint of this operation $\mathbf{A}_p$ is by construction a model of all ground input auxiliary rules and external atom guessing rules. Moreover, it is also a model of all remaining rules in $grnd_C(\Pi_p)$ because $\mathbf{A}$ is a model of the corresponding rules in $grnd_C(\Pi)$ with external atoms in place of replacement atoms, and we set the truth values of the external atom replacement atoms exactly to the truth values of the external atoms in $\mathbf{A}$. Note that there might be external atoms $\&g[\mathbf{y}](\mathbf{x})$ for which neither $e_{r,\&g[\mathbf{y}]}(\mathbf{x})$ nor $ne_{r,\&g[\mathbf{y}]}(\mathbf{x})$ is added to $\mathbf{A}_p$, but then the body of the respective external atom guessing rule is unsatisfied by $\mathbf{A}_p$. But since the body of an external atom guessing rule is a subset of the body of the rule where this external atom occurs, also this rule is satisfied.

Thus $\mathbf{A}_p$ is a model of $grnd_C(\Pi_p)$. It remains to show that it is also a subset-minimal model of $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$. Suppose there is a smaller model $\mathbf{A}'_p \subsetneq \mathbf{A}_p$. Then $\mathbf{A}_p \setminus \mathbf{A}'_p$ must contain at least one atom which is not a replacement atom or an input auxiliary atom, because by construction of $\mathbf{A}_p$ such atoms are only set to true if necessary, i.e., if they are supported by $\mathbf{A}$, and all rules used to derive such atoms are also in $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$. We now show that the restriction of $\mathbf{A}$ to ordinary atoms $\mathbf{A}' \subsetneq \mathbf{A}$ (i.e., without replacement atoms $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ and without external atom input atoms $g^{\&g}_{inp}(\mathbf{y})$) is a model of $fgrnd_C(\Pi)^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$.

Observe that, except for the external atom guessing and input auxiliary rules, the reduct $fgrnd_C(\Pi_p)^{\mathbf{A}_p}$ contains the same rules as $fgrnd_C(\Pi)^{\mathbf{A}}$ with replacement atoms instead of external atoms. Thus, for $r \in fgrnd_C(\Pi)^{\mathbf{A}}$, the corresponding $r_p \in fgrnd_C(\Pi_p)^{\mathbf{A}_p}$ contains the same ordinary literals in the head and body.

We show now that $\mathbf{A}'_p \models r_p$ implies $\mathbf{A}' \models r$. If $\mathbf{A}'_p$ is a model of r_p, then we have either (1) $\mathbf{A}'_p \models h$ for some $h \in H(r_p)$, or (2) $\mathbf{A}'_p \not\models b$ for some $b \in B(r_p)$. In Case (1), we also have $h \in H(r)$. Since $\mathbf{A}'_p$ and $\mathbf{A}'$ coincide on non-replacement and non-input atoms, this implies $\mathbf{A}' \models r$. In Case (2), b is either (2a) a non-replacement literal, or (2b) a (positive or default-negated) external atom replacement. In Case (2a), we also have $b \in B(r)$. Since $\mathbf{A}'_p$ and $\mathbf{A}'$ coincide on ordinary atoms, this implies $\mathbf{A}' \models r$. In Case (2b), we either have (2b′) $\mathbf{A}_p \not\models b$, or (2b″) b is positive (since a default-negated atom cannot become false by removing atoms from the interpretation) and some literal b' in the body of the external atom guessing or in the input rule for b is false in $\mathbf{A}'_p$; in this case b is represented in $grnd_C(\Pi_p)$ with some degree n. In Case (2b′), $\mathbf{A}$ falsifies by construction of $\mathbf{A}_p$ the external atom in $B(r)$ which corresponds to the replacement atom b. In Case (2b″), b' also appears in $B(r_p)$. Note that b' can be another external replacement atom. But in this case, the external atom corresponding to b' is represented with some degree $< n$. Thus, we start the case distinction for b' again. However, because the degree is reduced with every iteration, we will eventually end up in one of the other cases.

Thus, $\mathbf{A}'$ would be a model of $fgrnd_C(\Pi)^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$. This shows that $\mathbf{A}_p$ is an answer set of $grnd_C(\Pi_p)$. □

Lemma 24 *Let $\Pi_g = \mathsf{GroundHEX}(\Pi)$ and C be the constants which appear in Π_g. It holds that $\mathcal{AS}(\Pi_g) = \mathcal{AS}(grnd_C(\Pi))$.*

Proof. ($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}(\Pi_g)$. Then by Lemma 22 it can be extended to an answer set $\mathbf{A}_{pg}$ of Π_{pg}. By Definition 19, $\mathbf{A}_{pg}$ is also an answer set of $grnd_C(\Pi_p)$.

Let $r \in grnd_C(\Pi)$ and let r' be the respective rule in $grnd_C(\Pi_p)$ with replacement atoms instead of external atoms. (1) If there is no strengthening of r in Π_g, then there is also no strengthening of r' in Π_{pg}. Then by Definition 19, every answer set of $grnd_C(\Pi_p)$ falsifies some ordinary body literal of r'. Thus this holds also for $\mathbf{A}_{pg}$. But all ordinary literals of r' are also in r and $\mathbf{A}$ coincides with $\mathbf{A}_{pg}$ on ordinary literals, thus $\mathbf{A}$ is a model of r. (2) If there is a strengthening $\bar{r}$ of r in Π_g, then there is also a strengthening $\bar{r'}$ of r' in Π_{pg} from which $\bar{r}$ was generated by replacing external replacement atoms by external atoms. Because $\mathbf{A}_{pg}$ is an answer set of $grnd_C(\Pi_p)$, it is also a model of $\bar{r'}$. Moreover, by Definition 19, it satisfies also all ordinary literals $B(r') \setminus B(\bar{r'})$. This is the same set as $B(r) \setminus B(\bar{r})$. Because $\mathbf{A}$ and $\mathbf{A}_{pg}$ coincide on ordinary literals, also $\mathbf{A}$ is a model of r. Thus, $\mathbf{A}$ is a model of $grnd_C(\Pi)$.

We show now that $\mathbf{A}$ is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Because we have seen that $\mathbf{A} \not\models B(r)$ for every $r \in grnd_C(\Pi)$ which has no strengthening in Π_g, it follows that $f{\Pi_g}^{\mathbf{A}}$ contains a strengthening $\bar{r}$ for every rule $r \in fgrnd_C(\Pi)^{\mathbf{A}}$. Conversely, by Definition 19 every rule in $f{\Pi_g}^{\mathbf{A}}$ is a strengthening of some rule $r \in fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, the rules in $fgrnd_C(\Pi)^{\mathbf{A}}$ are even more restrictive, i.e., every model of $fgrnd_C(\Pi)^{\mathbf{A}}$ is also a model of $f{\Pi_g}^{\mathbf{A}}$. Thus, if there would be a smaller model $\mathbf{A}' \subsetneq \mathbf{A}$ of $fgrnd_C(\Pi)^{\mathbf{A}}$, it would also be a model of $f{\Pi_g}^{\mathbf{A}}$, which contradicts the assumption that $\mathbf{A}$ is an answer set of Π_g.

($\Leftarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, then it is also a model of Π_g because this program is (possibly) less restrictive. It remains to show that $\mathbf{A}$ is also a subset-minimal model of $f{\Pi_g}^{\mathbf{A}}$. By Lemma 23, $\mathbf{A}$ can be extended to an answer set $\mathbf{A}_p$ of $grnd_C(\Pi_p)$.

Let for every $r \in grnd_C(\Pi)$ be r' the respective rule in $grnd_C(\Pi_p)$ with replacement atoms instead of external atoms. Note that the rules in $f{\Pi_g}^{\mathbf{A}}$ are strengthenings of the rules in $fgrnd_C(\Pi)^{\mathbf{A}}$. Let $r \in grnd_C(\Pi)$. (1) If there is no strengthening of r in Π_g, then also r' has no strengthening in Π_{pg}. But this means, that every answer set of $grnd_C(\Pi_p)$ falsifies an ordinary body literal in r', thus also $\mathbf{A}_p$. Because $\mathbf{A}$ and $\mathbf{A}_p$ coincide on ordinary literals, also $\mathbf{A}$ falsifies some ordinary literal in $B(r)$, thus r is not in $fgrnd_C(\Pi)^{\mathbf{A}}$. (2) Now suppose there is a strengthening $\bar{r}$ of r in Π_g. Then $\mathbf{A} \models B(r)$ implies $\mathbf{A} \models B(\bar{r})$. Conversely, if $\mathbf{A} \models B(\bar{r})$, then the missing literals in $B(r) \setminus B(\bar{r})$ are satisfied as well because they are satisfied by all answer sets of $grnd_C(\Pi_p)$, including $\mathbf{A}_p$, which coincides with $\mathbf{A}$ on ordinary atoms (otherwise the literal would not have been removed by the optimizer).

Now suppose $f{\Pi_g}^{\mathbf{A}}$ has a smaller model $\mathbf{A}' \subsetneq \mathbf{A}$. Then $\mathbf{A}'$ is a model of $fgrnd_C(\Pi)^{\mathbf{A}}$ because we have seen that the missing literals are satisfied as well. □

Now we can prove Theorem 11.

Proof of Theorem 11. Let $\Pi_g = \mathsf{GroundHEX}(\Pi)$ and let C be the constants which appear in Π_g.

The differences to Algorithm GroundHEXNaive are that we (i) use a faithful (optimized) ASP grounding procedure to compute Π_g instead of the naive $grnd_C(\Pi_p)$; (ii) consider only lde-safety relevant external atoms in Part (b); and (iii) a different set of interpretations in Part (c). We will now show that the algorithm is still correct.

We first ignore modification (iii) and show that the algorithm is still correct if only modifications (i) and (ii) are active.

We need to show that $\mathcal{AS}(\Pi_g) = \mathcal{AS}(\Pi)$. Recall that we have $\mathcal{AS}(\Pi_g) = \mathcal{AS}\big(grnd_C(\Pi)\big)$ by Lemma 24, thus it is sufficient to show that $\mathcal{AS}\big(grnd_C(\Pi)\big) = \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$ for any $C' \supseteq C$.

($\Rightarrow$) Let $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. By Lemma 21, $\mathbf{A}$ is a model of $grnd_{C'}(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$. As $C \subseteq C'$, $fgrnd_C(\Pi)^{\mathbf{A}} \subseteq fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Moreover, by Lemma 21 $\mathbf{A} \not\models B(r)$ for any $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$, thus $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. But since $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$, thus it is also a model of $fgrnd_{C'}(\Pi)^{\mathbf{A}}$, i.e., $\mathbf{A} \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$.

($\Leftarrow$) Let $\mathbf{A}' \in \mathcal{AS}\big(grnd_{C'}(\Pi)\big)$. We show that $\mathbf{A} = \mathbf{A}' \cap A\big(grnd_C(\Pi)\big)$ is an answer set of $grnd_C(\Pi)$.

Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$, it is trivial that $\mathbf{A}$ is a model of $grnd_C(\Pi)$. It remains to show that it is also a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$.

By Lemma 21, $\mathbf{A}$ is also a model of $grnd_{C'}(\Pi)$. But then $\mathbf{A} = \mathbf{A}'$ because $\mathbf{A} \subsetneq \mathbf{A}'$ would imply that $\mathbf{A}'$ is not subset-minimal, which contradicts the assumption that it is an answer set of $grnd_{C'}(\Pi)$, thus $\mathbf{A} = \mathbf{A}'$. Because $grnd_C(\Pi) \subseteq grnd_{C'}(\Pi)$ and $\mathbf{A} \not\models B(r)$ for all $r \in grnd_{C'}(\Pi) \setminus grnd_C(\Pi)$ by Lemma 21, we have $fgrnd_C(\Pi)^{\mathbf{A}} = fgrnd_{C'}(\Pi)^{\mathbf{A}}$. Because $\mathbf{A}$ is a subset-minimal model of $grnd_{C'}(\Pi)^{\mathbf{A}}$, it is a subset-minimal model of $fgrnd_C(\Pi)^{\mathbf{A}}$. Thus, $\mathbf{A}$ is an answer set of $grnd_C(\Pi)$.

Finally, consider modification (iii). While Algorithm GroundHEXNaive loops for all models of Π_{pg}, the optimized algorithm constructs the considered assignments such that the output of the external atoms is maximized: all monotonic input atoms are set to true, all antimonotonic input atoms to false, and for nonmonotonic input atoms all combinations are checked (except facts, which are always true). Every model of Π_{pg} considered by Algorithm GroundHEX, is contained in some assignment enumerated by Algorithm GroundHEXNaive. The output of the external atom wrt. this assignment may be larger, but never smaller. Thus, the optimized algorithm only produces larger but never smaller groundings wrt. the set of constants. As we have shown in Lemma 21, this guarantees that the program has the same answer sets.

We also need to show that the algorithm terminates. But this follows from the observation that each run of the loop at (c) corresponds to a (restricted) application of operator G_Π; while G_Π instantiates rules whenever their positive body is satisfied by some of the enumerated assignments, our algorithm also respects the negative part of the rule body, i.e., it is even more restrictive. But by Corollary 5, $G_\Pi^\infty(\emptyset)$ is finite for lde-safe programs, thus the grounding produced by our algorithm is finite as well. Therefore the algorithm terminates. □

Proof of Theorem 13. The initial set of nodes defined at (a) is the set of all subset-maximal strongly connected components of the rules of Π wrt. $\rightarrow_m \cup \rightarrow_n$. This ensures that the graph is acyclic, that every rule (including constraints) is contained in exactly one unit, and that unit dependencies are updated according to the rule dependencies. Thus the initial decomposition forms a generalized evaluation graph.

Loop (b) then iteratively merges two different units, where Condition (a) ensures that the graph remains acyclic. As the algorithm also updates E according to the rule dependencies, all conditions of a generalized evaluation graph remain satisfied. □

Proof of Theorem 14. By definition of the hooks, GroundDESafeHEX behaves like Algorithm GroundHEX. □

We first introduce some lemmas to simplify the proof of Theorem 16.

Lemma 25 *If Π is a shy $Datalog^\exists$-program then the unique answer set of GroundDESafeHEX$(T_\exists(\Pi), 0)$ is a universal model of Π.*

Proof. Let $\mathbf{A}$ be the unique answer set of Π_g and let $U = chase(\Pi)$ be the universal model computed by the $chase$ procedure. We first show that there exists a homomorphism from $\mathbf{A}$ to U. Then we prove that $\mathbf{A}$ is a model of Π, which concludes the proof that it is a universal model as well.

For a rule $r \in \Pi$, let $r_{T_\exists} \in T_\exists(\Pi)$ be the according rule in $T_\exists(\Pi)$. We stepwise construct an isomorphism h, beginning with the empty one, and show by induction that the following holds for this isomorphism. Whenever $chase$ adds an atom $a = \hat{\sigma}(H(r))$ to U, then our algorithm adds an o-strengthening $r'_{T_\exists}$ of an instance of $r_{T_\exists}$ to the grounding Π_g s.t. $a = h(H(r'_{T_\exists}))$ (and $h^{-1}(a) = H(r'_{T_\exists})$) and such that for the unique answer set $\mathbf{A}$ of Π_g it holds that $\mathbf{A} \models B(r'_{T_\exists})$ (and thus, as it is an answer set, also $\mathbf{A} \models H(r'_{T_\exists})$).

Suppose some atom $p(\mathbf{u}, \mathbf{e})$ is added to U by $chase$ in the n-th iteration of the loop after (c), where $\mathbf{u}$ are the substitutions for universally quantified variables and $\mathbf{e}$ the substitutions for existentially quantified ones.

For $n = 0$, all (possibly existentially quantified) facts in Π are added to U. Facts without existential variables are trivially also part of the grounding Π_g, thus the proposition holds for any isomorphism, thus also for h. Facts with existential variables do not contain universal variables (otherwise they would not be safe), i.e., they are of form $r = \exists \mathbf{X} : p(\mathbf{X})$. Then $chase$ adds $p(\mathbf{e})$ for some vector of fresh nulls $\mathbf{e}$ to U. Thus $r_{T_\exists} = p(\mathbf{X}) \leftarrow \&exists^{0,|\mathbf{X}|}[](\mathbf{X})$. Since the external atom has no input parameters, the input auxiliary rule degenerates to fact $r_{inp}^{\&exists^{0,|\mathbf{X}|}[r,\mathbf{Y}](\mathbf{X})}()$ and the algorithm introduces at (f) a rule $e_{r,\&g[]}(\mathbf{x}) \vee ne_{r,\&g[]}(\mathbf{x}) \leftarrow$ for a fresh vector of nulls $\mathbf{x}$. Since $\mathbf{x}$ is new in Π_p and $\mathbf{e}$ is new in U, we can easily extend h s.t. $h(p(\mathbf{x})) = p(\mathbf{e})$ and $h^{-1}(p(\mathbf{e})) = p(\mathbf{x})$.

For $n \mapsto n+1$, suppose there is a firing substitution σ for some r wrt. U and suppose $U \cup NewAtoms \not\models \sigma(H(r))$, i.e., $chase$ adds $\hat{\sigma}(H(r))$ to $NewAtoms$ and thus to U. Because σ is a firing substitution, $\sigma(B(r)) \subseteq U$, i.e., all body atoms of r under σ have been added to U in some earlier iteration. Thus, by induction hypothesis, our algorithms algorithm adds for all $b \in \sigma(B(r))$ a rule r_b with $h(H(r_b)) = b$ and $H(r_b) = h^{-1}(b)$ to Π_g, s.t. $B(r_b)$ and $H(r_b)$ are satisfied under the unique answer set of Π_g. All ordinary atoms in $B(r_{T_\exists})$ occur also in $B(r)$, thus $\mathbf{A}$ satisfies the ordinary atoms in $B(r_{T_\exists})$ under substitution $h^{-1} \circ \sigma$. For existentially quantified variables, $r_{T_\exists}$ contains an additional atom $\&exists[\mathbf{c}](\mathbf{X})$ in the rule body and we show now that this atom is satisfied by $\mathbf{A}$ as well for an appropriate vector $\mathbf{c}$ in place of $\mathbf{Y}$ some fresh vector of nulls $\mathbf{x}$ in place of $\mathbf{X}$. Because all ordinary atoms in $B(r_{T_\exists})$ under substitution $h^{-1} \circ \sigma$ are satisfied by $\mathbf{A}$, our algorithm will add an o-strengthening of the input auxiliary rule of $\&exists[r, \mathbf{Y}](\mathbf{X})$ and atom $r_{inp}^{\&exists[r,\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ will appear in $A(\Pi_{pg})$, where $\mathbf{c} = h^{-1}(\mathbf{u})$ is the vector of substitutions for the universally quantified variables in r as defined by $h^{-1} \circ \sigma$. But then by definition of $\&exists$, $\&exists[r, \mathbf{c}](\mathbf{x})$ holds for a vector of nulls $\mathbf{x}$, which is unique for $\mathbf{c}$ and r and new in Π_p. But then, the algorithm will add a guessing rule for $\&exists[\mathbf{c}](\mathbf{x})$ to Π_p and subsequently the desired instance $r'_{T_\exists}$ with $H(r'_{T_\exists}) = p(\mathbf{c}, \mathbf{x})$ will appear in Π_g. Because $\mathbf{x}$ is new in Π_p and $\mathbf{e}$ is new in U, h does not define any mapping for them so far, thus we can easily extend h s.t. $h^{-1}(p(\mathbf{u}, \mathbf{e})) = p(\mathbf{c}, \mathbf{x})$ and $p(\mathbf{u}, \mathbf{e}) = h(p(\mathbf{c}, \mathbf{x}))$.

This shows that for any atom $a = \hat{\sigma}(H(r))$ added to U, our algorithm adds an according rule to Π_g. Conversely, the algorithm may add additional rules to Π_g. However, their bodies remain unsatisfied by $\mathbf{A}$ because otherwise $chase(\Pi, 0)$ would have identified firing substitutions and added the respective rule heads. Thus, the additional rules do not hurt.

Moreover, the definition of $\&exists$ correctly models the semantics of the existential quantifier. Thus, $\mathbf{A}$ is also a model of Π. Since we have defined a homomorphism from $\mathbf{A}$ to U, we have shown that $\mathbf{A}$ is also a universal model of Π. □

Lemma 26 *If Π is a shy $Datalog^\exists$-program, then* GroundDatalog$^\exists(\Pi, \infty)$ *yields the same (possibly infinite) program Π_g as* GroundDESafeHEX$(T_\exists(\Pi))$.

Proof. Resetting PIT to $\emptyset$ at the beginning of each iteration of the loop at (b) has the same effect as disabling the homomorphism check. □

Lemma 27 *If Π is a shy $Datalog^{\exists}$-program and $\Pi_g = \mathsf{GroundDatalog}^{\exists}(\Pi, k)$, then the unique answer set of Π_g is complete for conjunctive query answering with queries with up to k existentially quantified variables.*

Proof. Resetting PIT to $\emptyset$ after every iteration of the main loop at (b) behaves like freezing of nulls as by [33] and the loop at (b) runs $k + 1$ times. Thus, the lemma follows from Lemma 4.10 by [33]. □

Lemma 28 *Algorithm $\mathsf{GroundDatalog}^{\exists}(\Pi, k)$ terminates.*

Proof. Since all newly introduced values are null values, the loop at (f) introduces only finitly many new values because all remaining vectors $\mathbf{y}$ will eventually become homomorphic to some previously processed input vector. Thus each iteration of the loop at (c) terminates. As k is finite, also the loop at (b) terminates. □

We are now ready to prove our main result on Shy-programs in Theorem 16.

Proof of Theorem 16. Soundness of the algorithm follows from Lemmas 25 and 26 ($\mathsf{GroundDatalog}^{\exists}(\Pi, k)$ yields a subset of a universal model of Π), in combination with Proposition 2.4 by [33]. Completeness follows from Lemma 27. Termination follows from Lemma 28. □

Proof of Proposition 17. Suppose Π is strongly safe. We show that for any attribute α of Π, we have $a \in S_n(\Pi)$ for some $n \geq 0$, i.e., a is domain-expansion safe.

Let a be an attribute of Π and let j be the number of malign cycles wrt. $\emptyset$ in $G_A(\Pi)$ from which a is reachable. We prove by induction that if a is reachable from $j \geq 0$ malign cycles wrt. $\emptyset$ in $G_A(\Pi)$, then a is domain-expansion safe.

If $j = 0$ we make a case distinction. Case 1: if a is of form $p{\restriction}i$, then there is no information flow from a malign cycle wrt. $\emptyset$ to $p{\restriction}i$. Therefore, for every rule r with $p(t_1, \ldots, t_\ell) \in H(r)$ we have that $t_i \in B_{n+1}(r, \Pi, b_{synsem})$ for all $n \geq 0$ due to Condition (i) in Definition 14. But then $p{\restriction}i$ is domain-expansion safe.

Case 2: if a is of form $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$, then for every $Y_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{const}$ we have $Y_i \in B_{n+1}(r, \Pi, b_{synsem})$ due to Condition (i) in Definition 14, and for every predicate $p_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{pred}$ we have that $p_i{\restriction}j$ is domain-expansion safe for every $1 \leq j \leq ar(p_i)$ by Case 1; note that $p_i{\restriction}j$ is not reachable from any malign cycle wrt. $\emptyset$ because this would by transitivity of reachability mean that also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$ is reachable from such a cycle, which contradicts our assumption. But then also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}i$ is domain-expansion safe by Definition 8.

Case 3: if a is of form $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$, then no $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{I}}j$ for $1 \leq j \leq ar_{\mathrm{I}}(\&g)$ is reachable from a malign cycle wrt. $\emptyset$, because then also $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ would be reachable from such a cycle. But then by Definition 8, $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ is domain-expansion safe. Hence, attributes of any kind, which are not reachable from malign cycles wrt. $\emptyset$, are domain-expansion safe.

Induction step $j \mapsto j + 1$: If a is reachable from $j + 1$ malign cycles wrt. $\emptyset$, then there is an attribute α' in such a cycle C from which a is reachable. The malign cycle C wrt. $\emptyset$ contains an attribute of kind $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$, corresponding to an external atom $\&g[\mathbf{Y}](\mathbf{X})$ in rule r. Since $\&g[\mathbf{Y}]_r{\restriction}_{\mathrm{O}}i$ is cyclic in $G_A(\Pi)$, $\&g[\mathbf{Y}](\mathbf{X})$ is cyclic in $ADG(\Pi)$. Then by strong safety of Π, each variable in Y occurs in a body atom $p(t_1, \ldots, t_\ell) \in B^+(r)$ which is not part of C, i.e., it is captured by $p{\restriction}k$ for some $1 \leq k \leq ar(p)$. But since

$p(t_1, \ldots, t_\ell)$ is not part of the cycle C in $ADG(\Pi)$, also $p\restriction k$ is not part of it. Therefore $p\restriction k$ is reachable from (at least) one malign cycle wrt. $\emptyset$ less than a, i.e., it is reachable from at most j malign cycles. Thus $p\restriction k$ is domain-expansion safe by induction hypothesis. But then by Condition (ii) in Definition 9, also a is domain-expansion safe. □

Proof of Proposition 18. We first reformulate the definitions of *blocking* and *savior attributes* in an inductive way, which is possible because criteria are monotonic.

Blocking:

- $blocked_0(r) = \emptyset$ for all $r \in \Pi$
- $blocked_{n+1}(r) = \big\{p\restriction i \mid p\restriction i$ is dangerous in r and $p\restriction i$ captures X in r and
  for every $\&g[\mathbf{Y}](\mathbf{X})$ with $X \in \mathbf{X}$,
  for every $Y \in \mathbf{Y}$ there is a body atom
  $q(t_1, \ldots, t_\ell)$ such that $X = t_i$
  for some $1 \leq i \leq ar(q)$ and $q\restriction i \in savior_n\big\}$,
  for all $n \geq 0$
- $blocked_\infty(r) = \bigcup_{n\geq 0} blocked_n(r)$

Savior attributes:

- $savior_0 = \emptyset$
- $savior_{n+1} = \big\{p\restriction i \mid$ for all $r \in \Pi$ with $p(t_1, \ldots, t_\ell) \in H(r)$, either
  t_i is a constant; or
  t_i is captured by some $q\restriction j \in savior_n$ in $B^+(r)$; or
  $p\restriction i \in blocked_n(r)\big\}$,
  for all $n \geq 0$
- $savior_\infty = \bigcup_{n\geq 0} savior_n$

We show now by induction on n for all $n \geq 0$:

- If $p\restriction i \in blocked_n(r)$ and $p\restriction i$ captures X in r, then $X \in B_n(r, \Pi, S, b_{synsem})$.
- If $p\restriction i \in savior_n$ for some $n \geq 0$, then $p\restriction i \in S_n(\Pi)$.

For $n = 0$ this is trivial.

For the induction step $n \mapsto n + 1$, suppose $p\restriction i \in blocked_{n+1}(r)$. Then $p\restriction i$ is dangerous and captures some X in r. For every $\&g[\mathbf{Y}](\mathbf{X})$ with $X \in \mathbf{X}$ and for every variable $Y \in \mathbf{Y}$ there is a body atom $q(t_1, \ldots, t_\ell)$ such that $X = t_j$ for some $1 \leq j \leq ar(q)$ and $q\restriction j \in savior_n$ Then, by the induction hypothesis, $q\restriction j$ is domain-expansion safe. But then by Condition (ii) in Definition 9 all input variables $Y \in \mathbf{Y}$ are declared bounded in the first step, i.e., $Y \in B_{n+1,1}(r, \Pi, b_{synsem})$. Then by Condition (iii) in Definition 9 also all output variables $X \in \mathbf{X}$ are declared bounded in the second step, i.e., $X \in B_{n+1,2}(r, \Pi, b_{synsem})$. Thus we have $X \in B_{n+1}(r, \Pi, S_n(\Pi), b_{synsem})$.

Now suppose $p\restriction i$ in $savior_{n+1}$. Then we have for all $r \in \Pi$ with $p(t_1, \ldots, t_\ell) \in H(r)$ that

(i) t_i is a constant; or

(ii) t_i is captured by some $q{\restriction}j \in savior_n$ in $B^+(r)$; or

(iii) $p{\restriction}i \in blocked_n(r)$.

In Case (i), $t_i \in B_{n+1}(r, \Pi, S_n(\Pi), b_{synsem})$ by Condition (i) in Definition 9. In Case (ii), $q{\restriction}j$ is domain-expansion safe by the induction hypothesis and thus t_i is declared bounded by Condition (ii) in Definition 9. In Case (iii), it holds that $t_i \in B_{n+1}(r, \Pi, S, b_{synsem})$ as shown above.

This shows that all dangerous (but blocked) attributes are domain-expansion safe. It remains to show that also all non-dangerous attributes are lde-safe. Let a be such an attribute. If it occurs in a cycle in $G_A(\Pi)$, then it occurs also in a cycle in $G_{\bar{A}}(\Pi)$ because in this graph nodes from $G_A(\Pi)$ may be merged, i.e., the graph is less fine-grained. If it is of type $p{\restriction}i$, then it is dangerous and we already know that it is domain-expansion safe. Otherwise it is an external input attribute of form $\&g[\mathbf{X}]_r{\restriction}_{\mathrm{I}}i$ or an output attribute of form $\&g[\mathbf{X}]_r{\restriction}_{\mathrm{O}}i$. If it is an input attribute, then we know that its cyclic input depends (possibly transitively) on domain-expansion safe ordinary attributes. As the output attributes of external atoms become domain-expansion safe as soon as the input becomes domain-expansion safe by Definition 8, domain-expansion safety will be propagated by Condition (iii) in Definition 9 along the cycle, beginning at the ordinary predicates, i.e., the input parameter will be declared domain-expansion safe after finitely many steps (since the cycle is of finite length). This shows that all attributes in cycles in $G_A(\Pi)$ are domain-expansion safe.

As all attributes in cycles are domain-expansion safe, the remaining attributes (attributes which depend on a cycle but are not in a cycle) will also be declared domain-expansion safe after finitely many steps by Definition 8. □

Proof of Proposition 19. By Theorem 7 by [8] $F(\Pi)$ is VI-restricted, and thus by Proposition 18 it is also lde-safe using $b_{synsem}(\Pi, r, S, B)$. □