Domain Expansion for ASP-Programs with External Sources*

Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl

Institut für Informationssysteme, Technische Universität Wien, Favoritenstrasse 9-11, A-1040 Vienna, Austria

Abstract

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 directly without the need for program decomposition. While we present our approach on top of a proposed formalism in order to make the formalization precise, the general concepts carry over to related formalisms and important special cases as well. An experimental evaluation of lde-safety on various applications confirms the practicability of our approach.

Keywords: Answer Set Programming, Knowledge Representation Formalisms, Nonmonotonic Reasoning, External Source Access, Grounding, Computational Logic.

1. Introduction

Answer Set Programming (ASP) is a declarative programming approach which due to expressive and efficient systems like CLASP (Gebser et al., 2011)[2], DLV (Leone et al., 2006)[3], and SMODELS (Simons et al., 2002)[4], has been gaining popularity in several application areas, and in particular in artificial intelligence (Brewka et al., 2011). 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

Email addresses: eiter@kr.tuwien.ac.at (Thomas Eiter), fink@kr.tuwien.ac.at (Michael Fink), tkren@kr.tuwien.ac.at (Thomas Krennwallner), redl@kr.tuwien.ac.at (Christoph Redl)

* Preliminary results of this work have been presented at AAAI 2013 (Eiter et al., 2013b), the 2nd GTTV workshop (Eiter et al., 2013a), INAP 2013 (Eiter et al., 2014c), and in (Eiter et al., 2014a).

[2] http://potassco.sourceforge.net

[3] http://www.dlvsystem.com

[4] http://www.tcs.hut.fi/Software/smodels

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* (Eiter et al., 2008), which integrate rules with description logic ontologies in such a way 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 (Schüller et al., 2013); 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 (Eiter et al., 2014d) 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 simply added to the ASP program. Related to this is light-weight data access on the Web (e.g., XML, RDF (Lassila and Swick, 1999) 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 (Erdem et al., 2011) 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 (Dung, 1995) (this will be more discussed below) or of multi-context systems (Brewka and Eiter, 2007), fall in this class.

To cater for the need of external source access, HEX-*programs* (Eiter et al., 2005) extend ASP with so-called *external atoms*, through which the user can couple any external data source with a logic program. Roughly, an external atom can be seen as an API that passes information from the program, given by predicate extensions, to an external source and receives output values of an (abstract) function which that source computes, implemented in whatever language. This powerful abstraction is, for specific data sources that amount to logical theories, related in spirit to *SAT modulo theories* (Biere et al., 2009) but more versatile due to the possibility to use variables. It also generalizes a number of ASP extensions, among them *VI-programs* (Calimeri et al., 2007), ASP with monotone constraint atoms (Marek et al., 2004), constraint ASP (Gebser et al., 2009), programs with aggregates (Faber et al., 2011), programs with function symbols (Syrjänen, 2001)), as well as other formalisms which aim at the integration of external sources (e.g., DL-programs). The abstract nature of HEX-*programs* makes them a representative of a class of formalisms that is useful to discuss the new techniques on a concrete yet general level; the results carry over to the less general formalisms (which we shall exemplify) and are thus relevant beyond HEX-programs alone.

HEX-programs and instantiations such as DL-programs have already been considered for a range of applications; besides those mentioned above they include e.g. complaint management in e-government Zirtiloğlu and Yolum (2008), material culture analysis Mosca and Bernini (2008), user interface adap-

tation Zakraoui and Zagler (2012), or ontology integration in the biomedical domain Hoehndorf et al. (2007) (see also Redl (2014)). However, a wider takeup in practice requires efficient evaluation methods. Due to the abstract nature of HEX-programs, providing such methods is non-trivial; scalable algorithms have been introduced only recently (cf. Eiter et al. (2012, 2014b)). In a sense, the situation is akin to ordinary ASP: although the formal semantics was around since the early 1990's, only with the advent of efficient solvers (such as SMODELS (Simons et al., 2002) and DLV (Leone et al., 2006)) ASP could be widely established; broader deployment to real-world applications still took further time. [5]

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. In this paper, we focus on the grounding step. This step is non-trivial and significantly more involved than for ordinary ASP because external atoms may introduce new constants that are not present in the program; this is commonly referred to as *value invention*.

In particular, a naive support of value invention may easily lead to infinite program groundings and answer sets, as the following example demonstrates. Consider the program

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

where the external atom $\&appendA[X](Y)$ returns in Y the string in X with character 'A' appended. Due to the cycle over r_3 and r_4, the start string a will be expanded infinitely many times, i.e., the program has an infinite grounding and answer set (assuming that the external source processes all finite strings over an alphabet).

Moreover, even in cases where a finite subset of the grounding suffices to compute the answer sets of the original program, the set of relevant constants is often not known a priori. For instance, let Π' be program Π where rule r_3 is replaced by $r_3' \colon s(Y) \leftarrow r(X), \&reach[X](Y)$, where the external atom $\&reach[X](Y)$ computes for a (fixed and finite) graph that is stored externally the set of all nodes Y which are directly reachable from the node X. The overall program then computes the set of nodes reachable from the start node a. It appears that then, due to finiteness of the graph, only finitely many rules are relevant for evaluation. However due to an API style interface, external sources are largely black boxes to an ASP solver. Thus the set of constants that is relevant for evaluation is formally not clear.

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*), and is efficiently computable, traditional approaches impose syntactic safety conditions on a program, such as strong domain-expansion safety (Eiter et al., 2006), which we will recapitulate below, or VI-restrictedness (Calimeri et al., 2007). 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 an example.

In order to evaluate programs which are finitely restrictable but violate the formal safety criteria, a common workaround is to use a *domain predicate* d, where each constant c from the domain of the external source is added as a fact $d(c)$ to the program and type literals $d(X)$ are added for "unsafe" variables X in rule bodies. That is, all constants which *might* be relevant for evaluation are imported a priori into the program. This often requires the user to define an additional external atom which allows

[5] To further support this process, current ongoing work also includes means on the software engineering side for making a prototype system easily accessible, e.g. by providing pre-compiled and ready-to-use binary packages, an online demo, and tutorials. Moreover, additional language features make our implementation compliant with the ASP-Core-2 standard (Calimeri et al., 2013).

for importing the external domain. For instance, in case of program Π' one can define an external atom $\&nodes[](Y)$, which returns all nodes Y in the graph, and add a rule $d(Y) \leftarrow \&nodes[](Y)$ to the program to compute the domain. The domain atom $d(Y)$ can then be added to the bodies of all rules which use the external domain (in Π' this is rule r'_3) in order to make the program strongly safe and allow for evaluating it using the traditional methods. Another application which makes use of domain predicates is the realization of DL-programs (Eiter et al., 2008) via HEX-programs, where the external atoms query a description logic ontology. The individuals (i.e., constants) occurring in the ontology are added to the program in this way. However, this workaround is not only inconvenient, but also infeasible for large external domains. In particular, it often imports many constants that are actually irrelevant for the instance at hand. For example, in the program Π' from above a domain atom imports all nodes of the graph, even if only few nodes might be reachable from the given start node. If Π' is extended to some advanced application on the graph such as route planning, naively grounding the program over a larger graph will be simply infeasible (see Section 5 for a concrete example); the same holds for a similar ordinary ASP program with the graph given straight as facts.

Therefore, the current approach has **two limitations**: it imposes **unnecessary syntactic restrictions** on the program, and, as a consequence, while domain predicates can be used to circumvent these restrictions, this is not only inconvenient but also leads to a **combinatorial explosion of the grounding** which is unmanageable in many practical applications.

This motivates our **two main contributions**. First, we introduce a **new notion of safety** that still ensures finite restrictability of the program without requiring the user to use domain predicates. However, rather than merely to generalize an existing notion of safety, we develop a generic notion which can incorporate besides syntactic also semantic information about sources, and which is flexible with regard to further generalizations and extensions. Second, on top of this notion of safety we then develop a **novel grounding algorithm** which does not rely on a naive import of the full domain.

In more detail, 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 that 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 give 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.

- 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 (and external atoms which import the domain), but also leads to significantly better performance in many cases. In fact, the realization of our applications is in some cases impossible with the traditional notion of safety.

- 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 accommodate 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. After the discussion of related work in Section 6, we conclude in Section 7 with a summary and open issues. In order not to distract from reading, proofs have been moved to Appendix B.

2. Preliminaries

We start with basic definitions and recall HEX-programs from (Eiter et al., 2005). 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. Terms are elements from the set $\mathcal{C} \cup \mathcal{V}$. For the sake of simplicity, we disallow function symbols in this paper, but under appropriate safety criteria, an extension of our approach is straightforward. For any list $\mathbf{t} = t_1, \ldots, t_\ell$ of elements, we let t_i refer to its i-th element; $\mathbf{t}$ is implicitly cast to the set $\{t_1, \ldots, t_\ell\}$ in the context of set operations (thus e.g. $x \in \mathbf{t}$ expresses membership of x in $\mathbf{t}$).

2.1. Syntax

HEX-programs generalize (disjunctive) extended logic programs under the answer set semantics (Gelfond and Lifschitz, 1991) with external atoms. *(Ordinary) atoms* are of the form $p(\mathbf{t})$, where $p \in \mathcal{P}$ is a predicate of arity $\ell = ar(p) \geq 0$ and $\mathbf{t} = t_1, \ldots, t_\ell$ is a list of terms (from $\mathcal{C} \cup \mathcal{V}$). *External atoms* are 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 $\&g \in \mathcal{X}$ has input arity $ar_{\mathrm{I}}(\&g) \geq 0$ and output arity $ar_{\mathrm{O}}(\&g) \geq 0$. Each input argument i of $\&g$ ($1 \leq i \leq ar_{\mathrm{I}}(\&g)$) 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

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

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 $B(r) = B^+(r) \cup \mathrm{not}\, B^-$, 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 $\mathrm{not}\, S = \{\mathrm{not}\, b \mid b \in S\}$. 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; O is *ground*, if no variable occurs in it. For a program Π, let $C_\Pi \subseteq \mathcal{C}$ be the set of constants that occur in Π.

For the sake of a simpler notation, we assume in the following w.l.o.g. that external predicates with input lists are standardized apart, i.e., each occurrence of an external atom $\&g[\mathbf{X}]$ in the program is unique (this can be achieved by renaming variables or by adding dummy input constants if needed).

2.2. Semantics

An *assignment* $\mathbf{A}$ is a set of atoms, where an atom a is said to be *true* in $\mathbf{A}$ if $a \in \mathbf{A}$ and *false* otherwise. 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 assignment $\mathbf{A}$ is given by the value $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ of a decidable $1{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$, where k and l are the lengths of $\mathbf{p}$ and $\mathbf{l}$, respectively. 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 $\mathbf{A}' \supseteq \mathbf{A}$ augments $\mathbf{A}$ only by atoms a with predicate p_i; it is *antimonotonic* if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) \leq f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c})$ whenever $\mathbf{A}' \subseteq \mathbf{A}$ reduces $\mathbf{A}$ only by atoms a with predicate p_i; otherwise p_i is *nonmonotonic*. Let $\mathbf{p_m}$, $\mathbf{p_a}$ and $\mathbf{p_n}$ denote the sets of *monotonic*, *antimonotonic*, and *nonmonotonic* predicate input parameters in $\mathbf{p}$, respectively.

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, an ordinary atom a is *true* wrt. $\mathbf{A}$, denoted $\mathbf{A} \models a$, if $a \in \mathbf{A}$, and it is *false*, denoted $\mathbf{A} \not\models a$, if $a \notin \mathbf{A}$. An external atom $a = \&g[\mathbf{p}](\mathbf{c})$ is *true*, denoted $\mathbf{A} \models a$, if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ and *false*, denoted $\mathbf{A} \not\models a$, otherwise. Importantly, we make the restriction that for any assignment $\mathbf{A}$ and list of terms $\mathbf{p}$, the set $\{\mathbf{c} \mid f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1\}$ is finite and computable.

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 $\Pi^{\mathbf{A}} = \{r \in grnd(\Pi) \mid \mathbf{A} \models B(r)\}$ (Faber et al., 2011), which is subset-minimal, i.e., there exists no model $\mathbf{A}' \subsetneq \mathbf{A}$ of $\Pi^{\mathbf{A}}$.[6] The set of all answer sets of Π is denoted by $\mathcal{AS}(\Pi)$.

Example 1 Consider the program

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

where $\&concat[X, a](Y)$ is true iff Y is the concatenation of X and a (which is true for exactly one value Y), and consider the assignment $\mathbf{A} = \{dom(aa), t(a), s(aa), t(aa), s(aaa)\}$. 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

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

Clearly, $\mathbf{A} \models \Pi^{\mathbf{A}}$, and no atoms can be switched to false such that the resulting assignment $\mathbf{A}'$ 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 compute its answer sets. This property, which we call *finite restrictability*, is essential for computing the answer in finite time. More formally, for programs Π' and Π we write $\Pi' \equiv \Pi$ if $\mathcal{AS}(\Pi') = \mathcal{AS}(\Pi)$. Then,

[6] The FLP-reduct is equivalent to the traditional reduct for ordinary logic programs (Gelfond and Lifschitz, 1991), but more attractive for extensions such as aggregates or external atoms.

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

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 guarantee finite results of datalog queries. 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 concatenation of X and a. Then program $\Pi = \{s(a).\ s(Y) \leftarrow s(X), \&concat[X, a](Y)\}$ is safe but not finitely restrictable. Note that Π has the single (infinite) answer set $\{s(a^i) \mid i \geq 1\}$. □

Thus the notion of *strong safety* was introduced by Eiter et al. (2006), which limits the output of *cyclic external atoms* (to be formalized by the following definitions). In hinges on the concept of *external atom dependencies*. In the following, let $a \rightarrow^e_m b$ (resp. $a \rightarrow^e_n b$) denote that a *depends external monotonically (resp. nonmonotonically) on* b.

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

- *For every* $b = p(\mathbf{Z}) \in \bigcup_{r\in\Pi} H(r)$*, we have* $a \rightarrow^e_m b$ *if* $p \in \mathbf{X_m}$ *and* $a \rightarrow^e_n b$ *otherwise.*
- *If* $\{a,\ p(\mathbf{Z})\} \subseteq B^+(r)$ *for* $r \in \Pi$*, some* $X_i \in \mathbf{X}$ *is in* $\mathbf{Z}$ *and* $\tau(\&g, i) = \mathbf{const}$*, then* $a \rightarrow^e_m p(\mathbf{Z})$.
- *If* $\{a,\ \&h[\mathbf{V}](\mathbf{U})\} \subseteq B^+(r)$ *for some* $r \in \Pi$*, some* $X_i \in \mathbf{X}$ *is in* $\mathbf{U}$ *and* $\tau(\&g, i) = \mathbf{const}$*, then* $a \rightarrow^e_m \&h[\mathbf{V}](\mathbf{U})$.

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

(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* $a \in H(r)$ *and* $b \in B^-(r)$ *for some* $r \in \Pi$.

The following definition represents these dependencies.

Definition 4 (Atom Dependency Graph) *For 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* $\rightarrow_m, \rightarrow_n, \rightarrow^e_m, \rightarrow^e_n$ *between these atoms in* Π.

Example 3 For the program Π from Example 1, we have $\&concat[X, a](Y) \rightarrow^e_m t(X)$, $s(Y) \rightarrow_m \&concat[X, a](Y)$, $t(X) \rightarrow_m s(X)$ and $t(X) \rightarrow_m dom(X)$.

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. Π, if either b is not involved in any cycles in $ADG(\Pi)$, or every variable in $\mathbf{Y}$ occurs also in some positive ordinary atom $a \in B^+(r)$ that does not depend on b in $ADG(\Pi)$. A program Π is strongly safe, if every external atom in a rule $r \in \Pi$ is strongly safe in Π.*

The notion of *strong domain-expansion safety* is then as follows (Eiter et al., 2006).

Definition 6 (Strong Domain-expansion Safety) *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.*

Example 4 (cont'd) Consider 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 $\&reach[X](Y)$ is not safe. □

For this notion of safety, the following result was established.

Proposition 1 (Eiter et al., 2006) *Every strongly domain-expansion safe* HEX*-program Π is finitely restrictable.*

More in detail, Eiter et al. (2006) presented an algorithm which constructs a finite portion Π' of the grounding of a given sde-safe Π such that $\Pi' \equiv \Pi$.

While many HEX-programs are sde-safe, there are also simple and intuitive programs which lack this property, as shown by the program Π' in the introduction. However, this does not necessarily mean that such programs are not finitely restrictable, i.e., the other direction of the proposition does not hold.

Example 5 (cont'd) While the program in Example 2 has indeed no equivalent finite grounding, the program Π' in the introduction is not strongly safe either but in fact it *is* finitely restrictable. □

3. Liberal Safety

To overcome the overly restrictive limitations of strong safety, we introduce the new notion of *liberal domain-expansion (lde) safety* which incorporates both syntactic and semantic properties of the program at hand. 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 domain-expansion safety, liberal domain-expansion safety is not a property of entire atoms but of *attribute positions*, i.e., pairs of predicates and argument positions. Intuitively, an attribute position is lde-safe, if the number of different terms in a grounding which preserves all answer sets (i.e. has the same answer sets as the original program) is finite. A program is lde-safe, if all its attribute positions are lde-safe. The ultimate goal is therefore to identify lde-safe attributes. For this, we will exploit both syntactic and semantic criteria which are sound (i.e., there is actually an equivalent grounding with only finitely many differnet terms at lde-safe attributes), but not necessarily complete. The identification of concrete such criteria is often driven by knowledge about the intuitive meaning of language features; once a candidate for a criterion is available, a formal proof its soundness is usually straightforward.

Our notion of lde-safety is designed in an extensible fashion, 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 3.4).

For an ordinary predicate $p \in \mathcal{P}$, we call a pair (p, i) the *i-th attribute position of p* for all $1 \leq i \leq ar(p)$ and denote it more prominently as $att(p, i)$. As a shortcut for all attribute positions of a given predicate p, let $att(p) = \{att(p, 1), \ldots, att(p, ar(p))\}$. Similarily, we call a triplet $(\&g[\mathbf{X}], T, i)$ with $T \in \{\text{I}, \text{O}\}$ the *i-th input attribute position* if $T = \text{I}$ and the *i-th output attribute position of an external predicate with input list* if $T = \text{O}$ for all $1 \leq i \leq ar_T(\&g)$ and denote it more prominently as $att_T(\&g[\mathbf{X}], i)$. Akin to ordinary predicates, let $att_T(\&g[\mathbf{X}]) = \{att_T(\&g[\mathbf{X}], 1), \ldots, att(\&g[\mathbf{X}], ar_T(\&g))\}$ for $T \in \{\text{I}, \text{O}\}$ be the sets of all input and output attribute positions, respectively. For a ground program P, the *range* of an attribute position $att(p, i)$ is intuitively the set of ground terms which occur at position i of p. Formally, $range(att(p, i), \Pi) = \{t_i \mid p(t_1, \ldots, t_{ar(p)}) \in A(\Pi)\}$; for an attribute position $att_T(\&g[\mathbf{X}], i)$ with $T \in \{\text{I}, \text{O}\}$ we have $range(att_T(\&g[\mathbf{x}], i), \Pi) = \{x_i^T \mid \&g[\mathbf{x}^{\text{I}}](\mathbf{x}^{\text{O}}) \in EA(\Pi)\}$, where $\mathbf{x}^s = x_1^s, \ldots, x_{ar_s(\&g)}^s$.

Example 6 Reconsider program Π from Example 1. Examples for attribute positions are $att(t, 1)$ (the first argument of predicate t), $att_{\text{I}}(\&concat[X, x], 2)$ (the second input argument of external predicate with input list $\&concat[X, x]$) and $att_{\text{O}}(\&concat[X, x], 1)$ (the first output argument of external predicate with input list $\&concat[X, x]$). Furthermore, $range(att(t, 1), \Pi) = \{a\}$ (the set of terms occurring as first argument of predicate t). □

Formally, our approach builds on a *monotone grounding operator*, *boundedness of terms* and *liberal domain-expansion (lde) safety of attribute positions*. Intuitively, the operator is used to define an upper bound for the set of programs which can be considered to be lde-safe, namely all programs which can be finitely grounded by this operator. While this operator provides an operational definition of safety and cannot be used to decide it (the class of lde-safe programs is in general undecidable[7]), the concepts of boundedness and lde-safety then define criteria which can be used to identify (a subset of all) lde-safe programs.

3.1. Monotone Grounding Operator

We first introduce a monotone grounding operator which is intended to serve as a witness for the finite groundability. That is, liberal domain expansion safety will be defined such that for an lde-safe program, the least fixpoint of G_Π is a finite grounding that is equivalent to the original program Π (however, *not* all such programs will be necessarily identified as lde-safe). The operator is defined as follows:

$$G_\Pi(\Pi') = \bigcup_{r \in \Pi} \{r\theta \mid \exists \mathbf{A} \subseteq A(\Pi'), \mathbf{A} \models B^+(r\theta)\},$$

where $r\theta$ is the ground instance of r under a variable substitution $\theta \colon \mathcal{V} \to \mathcal{C}$. 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

[7]The Halting problem can be reduced to the lde-safety check as follows. Consider an external atom which implements the transition function of a deterministic Turing machine and outputs for given machine state the next state (including step count, tape content, etc). A recursive rule can then be used to simulate the machine and the program will be lde-safe iff it halts.

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 $grnd(\Pi)$; after the formal introduction of our new notion of safety, we will show that for an lde-safe program Π, the least fixpoint of the operator is indeed finite and equivalent to the original program Π 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:

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

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

Based on this operator, the general idea of our approach uses two concepts which mutually refer to each other. First, call a term t in a rule *bounded*, if the number of substitutions in $G_\Pi^\infty(\emptyset)$ for term t is finite. Second, we call an attribute position α *de-safe*, if in the program $G_\Pi^\infty(\emptyset)$ only finitely many terms which occur at argument position α.

Even in decidable cases, computing the overall sets of bounded terms and safe attribute positions directly is not straightforward because both the global structure of a program and the semantics of external atoms influence these sets. Thus our approach works incrementally: beginning from the empty sets, terms which are newly identified as bounded may trigger the identification of further lde-safe attribute positions and vice versa.

3.2. Boundedness of Terms

We first focus on boundedness of terms. In order to make the approach extensible as mentioned, the criteria for a term to be bounded are no hard-coded. Instead, to stay flexible, we use abstract *term bounding functions* (concrete instances will be shown below).

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) attribute positions, 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* $G_\Pi^\infty(\emptyset)$*.*

Intuitively, a TBF receives a set of already bounded terms and a set of attribute positions 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.

Example 8 Consider program Π from Example 7 and the function $b(\Pi, r, S, B)$ with $b(\Pi, r_2, S, B) = \{ax\}$ and $b(\Pi, r_3, S, B) = \{aax\}$ (for any S and B) and $b(\Pi, r, S, B) = \emptyset$ for $r \notin \{r_2, r_3\}$ (and any S and B). It identifies (only) the terms ax in rule r_2 and aax in rule r_3 as bounded. Then b is a valid TBF because there are indeed only finitely many substitutions for ax in r_2 and aax in r_3 in the least fixpoint of G_Π as shown in the ground program in Example 7 (in fact, there is only one substitution for each of these terms, namely the term itself).

In contrast, for the program $\Pi' = \{r_1\colon str(a); r_2\colon str(X) \leftarrow str(Y), \&appendA[Y](X)\}$ a function $b'(\Pi, r, S, B)$ with $X \in b(\Pi', r_2, \emptyset, \emptyset)$ would not be a valid TBF because there are infinitely many substitutions for X in $G_{\Pi'}$. □

TBFs are intended to formalize concrete safety criteria based on syntactic or semantic properties (for which we will give examples below). By encapsulating these properties in TBFs, the definition of lde-safety below is independent of such properties and thus extensible.

As $b(\Pi, r, S, B)$ in Example 8 shows, a TBF is *not* required to identify a maximal set of bounded terms (also a in r_1 is bounded). Instead, Definition 7 requires TBFs only to be monotonic and to not return false positives. Because the operator G_Π is fixed, there is also a fixed set of bounded terms for each program and it is in principle possible to define a TBF which simply identifies for each rule a fixed set of bounded terms independent of S and B (as in b from Example 8). However, it turns out that the direct identification of bounded terms is often not trivial in realistic examples. Therefore, our approach is intended to work incrementally such that the identification of bounded terms and lde-safe attribute positions (to be formally introduced below) cyclically trigger each other. For this reason, Definition 7 passes the sets S and B of already safe attribute positions and already bounded terms, respectively, to the TBF in order to inform it about the current status; these sets can (but do not have to) be used by the TBF in order to guide the identification of further bounded terms.

3.3. Liberal Domain-expansion Safety of attribute positions

We now turn to the formalization of lde-safety as our second main concept. We provide a mutually inductive definition that takes the empty set of lde-safe attribute positions $S_0(\Pi)$ as its basis. Then, by using a TBF each iteration step $n \geq 1$ computes first the set of known bounded terms $B_n(r, \Pi)$ for all rules r, and then an enlarged set of known lde-safe attribute positions $S_n(\Pi)$. The set of known lde-safe attribute positions in step $n+1$ thus depends on the TBF, which in turn depends on the known lde-safe attribute positions from step n.

In order to simplify the formal definition, we introduce the following notions. First, we associate with a TBF $b(\Pi, r, S, B)$ and fixed Π, r and S an operator $b_{\Pi,r,S}(B) = b(\Pi, r, S, B)$. This allows us to compactly represent the least fixpoint of a set B under $b(\Pi, r, S, B)$ by $b^\infty_{\Pi,r,S}(\emptyset)$; note that the set of terms in program Π is finite as thus the least fixpoint is finite as well. That is, $b^\infty_{\Pi,r,S}(\emptyset)$ applies the TBF b to Π, r and S and an incrementally extended set B (beginning from the empty set), until no more bounded terms can be identified.

Second, we say that an input attribute position $att_1(\&g[\mathbf{X}], i)$ is *finite with respect to bounded terms B and lde-safe attribute positions S*, if either $\tau(\&g, i) = \mathbf{const}$ and $X_i \in B$ or $\tau(\&g, i) = \mathbf{pred}$ and $att(X_i) \subseteq S$. That is, the input attribute position is either a term which is known to be bounded, or it is a predicate such that all its attribute positions are known to be safe.

Definition 8 (Liberal Domain-Expansion Safety) *Let b be a term bounding function.*

(a) The bounded terms *in $r \in \Pi$ in step $n \geq 1$ are $B_n(r, \Pi) = b^\infty_{\Pi,r,S_{n-1}}(\emptyset)$.*

(b) The lde-safe attribute positions *are $S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$ where $S_0(\Pi) = \emptyset$ and for $n \geq 0$, $a \in S_{n+1}(\Pi)$ if:*

(i) $a = att(p, i)$ and for all $r \in \Pi, p(\mathbf{t}) \in H(r)$ we have $t_i \in B_{n+1}(r, \Pi)$;

(ii) $a = att_1(\&g[\mathbf{X}], i)$ which is finite wrt. bounded terms $B_{n+1}(r, \Pi)$ and lde-safe attribute positions $S_n(\Pi)$;

Figure 1: Visualization of Definition 8 (inductive definition from left to right and top to bottom)

Iteration i	(a) Bounded Terms (iteration i)	(b) LDE-Safe Attribute Positions (iteration i)
	—	$S_0(\Pi) = \emptyset$
1	$B_1(r, \Pi)$	$S_1(\Pi)$
2	$B_2(r, \Pi)$	$S_2(\Pi)$
...	...	...
		$S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$

(iii) $a = att_o(\&g[\mathbf{X}], i)$, $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$ *and either* $Y_i \in B_{n+1}(r, \Pi)$ *or* $att_i(\&g[\mathbf{X}]) \subseteq S_n(\Pi)$.

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

Note that lde-safety is always relative to a given TBF. That is, a program might be lde-safe wrt. one TBF but not wrt. another one; in the following, we will explicitly specify the currently used TBF unless this is clear from context.

Intuitively, the iterative definition of the lde-safe attribute positions $S_\infty(\Pi)$ starts with the empty set $S_0(\Pi) = \emptyset$ of attribute positions. Then, each iteration $n + 1$ consists of two steps in order to define the expanded set of known lde-safe attribute positions $S_{n+1}(\Pi)$.

(a) For each rule $r \in \Pi$, the set $B_{n+1}(r, \Pi)$ of terms in this rule which are known to be bounded is identified, i.e., which are known to have only finitely many substitutions in $G_\Pi^\infty(\emptyset)$. For this purpose, the TBF b is employed which identifies such terms by making use of the fact that the attribute positions $S_n(\Pi)$ from the previous iteration are already known to be lde-safe. The identification of bounded terms is again iterative by applying the TBF until a fixpoint is reached.

(b) Based on the sets of known bounded terms $B_{n+1}(r, \Pi)$ for all $r \in \Pi$ in the $(n+1)$-st iteration, the set of known lde-safe attribute positions $S_{n+1}(\Pi)$ is defined. The intuition here is to identify an attribute position as lde-safe whenever the set of known bounded terms and the set of known lde-safe attribute positions from the previous iteration imply that only finitely many values will occur at this attribute position in $G_\Pi^\infty(\emptyset)$. (b.i) For ordinary attribute positions, this is the case if all terms which occur at this argument position in rule heads (i.e., which may generate values at this argument position) are bounded. (b.ii) For an input attribute position of an external atom, this is the case if it is *finite*, i.e., if it is a constant input parameter which is bounded, or if it is a predicate input parameter with a predicate whose attribute positions are all already known to be lde-safe. (b.iii) For an output attribute position of an external atom, this is the case if either the term at the respective output position is known to be bounded (e.g. because it occurs in another body atom at a position which is known to be lde-safe), or because all of its inputs are already known to be lde-safe attribute positions. In the latter case, since there also finitely many different inputs to the external atom, there can also be only finitely many different outputs (cf. the restriction on the oracle functions introduced in Section 2).

Note that, all sets involved in this definition are defined (and can be computed) in a strictly acyclic fashion. This is visualized in Figure 1, where the sets as by Definition 8 are defined from left to right and top to bottom.

We now give a short example based on the simple TBF from Example 8, while an exhaustive example is delayed until we have introduced realistic TBFs in Section 3.4.

Example 9 (cont'd) Reconsider the program Π and the TBF b from Example 8. Because terms ax in r_2 and aax in r_3 are identified as bounded, we have $B_1(r_2, \Pi) = \{ax\}$ and $B_1(r_3, \Pi) = \{aax\}$. For the set of safe attribute positions identified in the first iteration, we therefore have $att(dom, 1) \in S_1(\Pi)$ by Condition (b.i) of Definition 8. Intuitively this states that attribute position $att(dom, 1)$ is safe, because it is only defined by the terms ax in r_2 and aax in r_3, which are have been identified as bounded by the TBF b. Since the TBF b as introduced in Example 8 does not identify further terms as bounded in the next iteration, there are also no more attribute positions which can be identified as safe and we end up with $S_\infty(\Pi) = \{att(dom, 1)\}$. Note that TBF b from Example 8 is a very simply example TBF; we will reconsider this program in Example 10 but use a stronger (and practical) TBF, which allows for identification of more lde-safe attribute positions (and lde-safety of the overall program). □

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 formalize that a program has only finitely many lde-safe attributes.

Proposition 2 *For every TBF b, the set $S_\infty(\Pi)$ is finite.*

Next, recall that the intuition of an lde-safe attribute is that there are only finitely many terms at this position in a grounding which preserves all answer stes. The grounding $G^\infty_\Pi(\emptyset)$ serves as witness. This can be formalized by stating that lde-safe attribute positions have a finite range in $G^\infty_\Pi(\emptyset)$. We first formalize the idea for attributes which are identified as lde-safe after a certain number of iterations.

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

This results carries over to all lde-safe attributes identified after an arbitrary number of steps.

Corollary 4 *For every TBF b, if $\alpha \in S_\infty(\Pi)$, then $range(\alpha, G^\infty_\Pi(\emptyset))$ is finite.*

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

Corollary 5 *If Π is an lde-safe program, then $G^\infty_\Pi(\emptyset)$ 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 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 $G^\infty_\Pi(\emptyset) \equiv \Pi$.*

Also 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.

3.4. 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 $G^\infty_\Pi(\emptyset)$.

3.4.1. Syntactic Criteria

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

Definition 9 (Syntactic Term Bounding Function) *Let $b_{syn}(\Pi, r, S, B)$ be defined such that we have $t \in b_{syn}(\Pi, r, S, B)$ iff*

(i) t is a constant in r; or

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

(iii) for some $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$, we have $t \in \mathbf{Y}$ and every $a \in att_1(\&g[\mathbf{X}])$ is finite wrt. B and S.

Note that the criteria encoded in this TBF are motivated by the intuitive meaning of rules and other language components. 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 position positions are bounded. This is because the attribute position is already known to be filled by only finitely many terms in a grounding which preserves all answer sets. This property carries over to the term at this position, which has then only finitely many instances. More specifically, it encodes that an attribute position $att(q, j)$ (where $1 \leq j \leq ar(q)$) is lde-safe (thus has a finite range in $G^{\infty}_{\Pi}(\emptyset)$), implies that the term at this attribute position position is bounded. Case (iii) expresses that if all input attribute positions to an external atom are finite, then also its output is finite. This is motivated by the restriction that a given input makes an external atom true for only finitely many differnet outputs (cf. Section 2). Therefore, finiteness of the number of different input values carries over to the number of different output values.

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

Example 10 (cont'd) Reconsider the program

$$\Pi = \left\{ \begin{array}{l} r_1 \colon s(a). \quad r_2 \colon dom(ax). \quad r_3 \colon dom(axx). \\ r_4 \colon s(Y) \leftarrow s(X), \&concat[X, x](Y), dom(Y). \end{array} \right\}$$

from Example 7 to compute the sets of safe attribute positions using TBF b_{syn}. In the first iteration we have $B_1(r_2, \Pi) = \{ax\}$, $B_1(r_3, \Pi) = \{axx\}$ and $B_1(r_4, \Pi) = \{x\}$ by item (i) in Definition 9 as constants are trivially bounded. But then, due to the bounded term ax in r_2 and axx in r_3, the derived terms in all rules that have $att(dom, 1)$ in their head are known to be bounded and thus item (b.i) in Definition 8 applies, which yields

$$S_1(\Pi) = \{att(dom, 1), att_1(\&concat[X, x], 2)\},$$

where the membership of $att(dom, 1)$ in this set intuitively means that dom has only finitely many different arguments in some grounding which preserves all answer sets, because each rule defining dom derives only finitely many different values. On the other hand, the membership of $att_1(\&concat[X, x], 2)$ means that x in the input needs to be instantiated only by finitely many different constants (in fact, as a constant, it will be instantiated only by itself).

In the next iteration, we get $B_2(r_4, \Pi) = \{Y\}$ by item (ii) in Definition 9 because Y appears in $dom(Y)$ in the body of r_4 and $att(dom, 1)$ is already known to be lde-safe from the previous iteration. Intuitively, since dom is known to be instantiated only for finitely many different arguments, this will also be the case for Y. We further have $B_2(r_1, \Pi) = \{a\}$ by item (i) in Definition 9 (actually we already had $B_1(r_1, \Pi) = \{a\}$ in the previous iteration). Therefore, all terms in rule heads which define $att(s, 1)$ (namely those of r_1 and r_4) are bounded, hence, item (b.i) in Definition 8 applies again and

$$att(s, 1) \in S_2(\Pi).$$

Hence, intuitively s needs to be instantiated only for finitely many arguments. Moreover, we also have

$$att_{\mathrm{O}}(\&concat[X, x], 1) \in S_2(\Pi)$$

because the output term at position 1 of this external atom is Y, which is bounded as it also appears in $dom(Y)$ and thus item (ii) in Definition 9 applies. Thus, the second iteration identifies two attribute positions as lde-safe.

In the third iteration we have $X \in B_3(r_4, \Pi)$ due to item (i) in Definition 9. This is because we know from the previous iteration that $att(s, 1)$ is lde-safe, i.e., predicate s will be instantiated only for finitely many arguments; as X occurs at this argument position, there will be only finitely many substitutions for X. But then, the input to the external atom in r_4 is bounded and item (b.ii) in Definition 8 applies, thus

$$att_{\mathrm{I}}(\&concat[X, x], 1) \in S_3(\Pi).$$

At this point, all attribute positions are lde-safe and so is the program. □

Example 10 also demonstrates why we have chosen an incremental approach which cyclically identifies bounded terms and safe attribute positions. Observe that the least fixpoint of G_Π for the program Π is finite and thus all terms in Π are bounded. While it is in principle possible to define a TBF which directly identifies all terms as bounded, this is not trivial as the reasons for a term to be bounded can be complex. For instance, variable X in rule r_4 is bounded because of the following intuitive reason. attribute position $att(dom, 1)$ is safe as it is defined only by the facts. Therefore, variable Y in r_4 is bounded, which implies that only finitely many variables for $s(Y)$ are derived, hence also $att(s, 1)$ is safe. This implies that also X in r_4 is bounded. A direct identification of the bounded terms would result in a complex condition whose correctness would be difficult to assert. In contrast, incrementally enlarging the sets of known bounded terms and safe attribute positions, as formalized by Definitions 7 and 8, allows for formally representing this reasoning chain by a set of much simpler conditions, which are directly motivated by the intuitive meaning of bounded terms and safe attribute positions.

3.4.2. Semantic Properties

We now define a TBF exploiting meta-information about external sources. The first two properties involve meta-information that directly ensures an output attribute position of an external source is finite.

Definition 10 (Finite Domain) *An external predicate $\&g \in \mathcal{X}$ has a* finite domain in (output argument) $i, 1 \leq i \leq ar_{\mathrm{O}}(\&g)$*, if* $\{x_i \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ *for some* $\mathbf{y}$ *and* $\mathbf{x}\}$ *is finite for all* $\mathbf{A}$.

Intuitively, the property states that only finitely many different values occur at a certain output position (independent of the input). 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). □

Example 12 The program Π' from the introduction can be identified as lde-safe, once it is known that the output domain of $\&reach[X](Y)$ is finite due to finiteness of the graph. □

A relaxed notion of finiteness allows for open domains, but forbids constants in the output of an external source which do not already appear in the extension of the respective input predicate parameter, i.e., all output values at a certain position already occur in the input with no new values being invented.

Definition 11 (Relative Finite Domain) *An external predicate $\&g \in \mathcal{X}$ has a* finite domain in (output argument) i, $1 \leq i \leq ar_{\mathrm{O}}(\&g)$, relative to (predicate input) argument j, $1 \leq j \leq ar_{\mathrm{I}}(\&g)$, *if* $\{x_i \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ *for some* $\mathbf{x}\} \subseteq \{c \in \mathbf{c} \mid \mathbf{c} \in ext(\mathbf{A}, y_j)\}$ *is finite for all* $\mathbf{A}$ *and* $\mathbf{y}$.

Example 13 An external atom $\&diff[dom, set](Y)$ has a finite domain in output argument 1 relative to predicate input argument 1 because each constant c must already occur in the extension of dom wrt. $\mathbf{A}$ if $f_{\&g}(\mathbf{A}, dom, set, c) = 1$.

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 12 (Finite Fiber) *An external predicate $\&g \in \mathcal{X}$ has a* finite fiber, *if set* $\{\mathbf{y} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$ *is finite for all* $\mathbf{A}$ *and* $\mathbf{x}$.

Example 14 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. □

In the following, we say that a term t is *captured* by an attribute position α, if t occurs at the (input or output) argument position specified by α. The four properties above lead to the following TBF.

Definition 13 (Semantic Term Bounding Function) *Let $b_{sem}(\Pi, r, S, B)$ be defined such that we have $t \in b_{sem}(\Pi, r, S, B)$ iff*

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

(ii) for a $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, $\&g$ has a relative finite domain in i wrt. j, *$att(Y_j) \subseteq S$ and $t = X_i$; or*

(iii) for a $\&g[\mathbf{Y}](\mathbf{X}) \in B^+(r)$, $\&g$ has a finite fiber, *$\mathbf{X} \subseteq B$ and $t \in \mathbf{Y}$.*

This TBF is directly motivated by the properties introduced above.

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

An extended semantic TBF which takes also cyclicity and well-orderings into account is shown in Appendix A due to its lengthy definition.

3.5. Modular Combinations of Term-Bounding Functions

Note that a program might be lde-safe wrt. one TBF but not wrt. another TBF. Fortunately, multiple TBFs can be combined in order to generate another, even more general TBF. Thus, rather than changing the TBF for every program, our approach is intended to be extended over time by integrating more criteria whenever new properties are discovered that allow for deriving finite groundability. The following proposition allows us to construct TBFs modularly from multiple TBFs and thus ensures this kind of future extensibility.

Theorem 9 *Let $b_i(\Pi, r, S, B)$, $1 \leq i \leq \ell$ be TBFs. Then*

$$b(\Pi, r, S, B) \;=\; b_i(\Pi, r, S, B) \cup \cdots \cup b_\ell(\Pi, r, S, B).$$

That is, the function assessing term boundedness by logical disjunction of the given TBFs, is also a TBF.

In particular, a TBF which exploits syntactic and semantic properties simultaneously is given by $b_{synsem}(\Pi, r, S, B)$ which is defined as follows:

$$b_{synsem}(\Pi, r, S, B) = b_{syn}(\Pi, r, S, B) \cup b_{sem}(\Pi, r, S, B).$$

Example 15 Consider the program

$$\Pi = \left\{ \begin{array}{l} node(a, 0). \\ node(Y) \leftarrow node(X), \&reach[X](Y), \&distance[a, Y](D), \&limit[D](). \end{array} \right\},$$

where $\&reach[X](Y)$ returns the nodes Y that are directly reachable from a node X in some externally given graph (e.g., in a search space), $\&distance[a, Y](D)$ returns the shortest distance D between a and Y, and $\&limit[D]()$ is true for all integers D up to a certain number. Informally, this program imports all nodes within a certain distance from the node a; for a possibly infinite graph, this however might not be feasible in finite time. However, if each node is known to have only finitely many neighbors, by combining syntactic and semantic information, we conclude that Π has a finite grounding as all external atoms have finite fibers. In particular, $\&limit[D]()$ and $\&distance[a, Y](D)$ have finite fibers since only finitely many values for D are within the limit and for each of them finitely many nodes Y with that distance from a exist as each node has only finitely many neighbors. Moreover, $\&reach[X](Y)$ has a finite fiber because each node Y is reachable only from finitely many X. Without proper semantic knowledge about the external sources (e.g., if the nodes can have infinitely many neighbors), finiteness of the grounding is not guaranteed. Moreover, also the syntactic structure of the program is relevant to derive finiteness as after removing $\&limit[D]()$ finiteness is no longer guaranteed.

3.6. 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 (Eiter et al., 2008) and terminological default theories (Baader and Hollunder, 1995) in DLVHEX 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 16 (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 $DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}]$ returns all individuals in $\neg \mathit{Flier}$ 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 $\{\mathit{bird}(\mathit{lia}), \mathit{flies}(\mathit{lia})\}$.

While the program Π is safe, it is not strongly safe; this is because the external atom $DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X)$ 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 $d(\mathit{lia})$. Alternatively, assuming that all individuals in the ontology are birds, strong safety is gained by using bird as domain predicate and simply adding $\mathit{bird}(X)$ 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 $DL[\mathit{Flier} \uplus \mathit{flies}; \neg \mathit{Flier}](X)$ 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 lde-safe programs. It is based on iteratively grounding the program and then checking whether the grounding contains all relevant ground rules. The check works by evaluating external sources under relevant assignments and testing whether they introduce any new constants that were not respected in the grounding. If so, then the set of constants is expanded and the program is grounded again; otherwise, the unrespected constants from $\mathcal{C}$ are irrelevant for ensuring that the grounding has the same answer sets as the original program. For lde-safe programs, this procedure will eventually reach a fixpoint, i.e., all relevant constants are respected in the grounding. While the operator G_Π could be directly used for grounding in principle (the output is finite for lde-safe programs), its complexity is not optimal. In particular, the instantiation of a single rule has exponential runtime, although this is in many cases not necessary. In addition to the singleton usage of each external predicate as introduced in Section 2, we further assume in the following that rules are standardized apart (i.e., share no variables).

Towards a practical grounding algorithm we start with some basic concepts that are all demonstrated in Example 18 below. 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 that are 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 $s \in \{+, -\}$. Furthermore, $\Pi|_R = \bigcup_{r \in \Pi} r|_R$, for a program Π. Furthermore, let $\mathit{var}(r) \subseteq \mathcal{V}$ be the set of variables appearing in a rule r.

Definition 14 (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 $\mathit{var}(r) = \mathit{var}(r|_R)$, 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. While the algorithm works for any such set, a particular choice may influence the efficiency in principle, even if in realistic applications few possibilities seem suggestive. We will use in our experiments a greedy approach which aims at minimizing the set as fewer external atoms usually increase the efficiency. In the following, we assume that for a program Π a fixed set R of lde-safety relevant external atoms has been selected and denote by $\overline{R} = EA(\Pi) \setminus R$ the set of external atoms that are not lde-safety relevant.

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 15 (Input Auxiliary Rule) *Let Π be a program and let $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom 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 $b \in B^+(r) \setminus \{\&g[\mathbf{Y}](\mathbf{X})\}$ s.t. $\&g[\mathbf{Y}](\mathbf{X})$ joins b and $b \notin \overline{R}$.*

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

Definition 16 (External Atom Guessing Rule) *Let Π be a program and let furthermore $\&g[\mathbf{Y}](\mathbf{X})$ be some external atom. Then a rule $r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*

- *the head is $H(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{e_{\&g}(\mathbf{X}), ne_{\&g[\mathbf{Y}]}(\mathbf{X})\}$, and*
- *the body $B(r_{guess}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains*
  - *(i) each $b \in B^+(r) \setminus \{\&g[\mathbf{Y}](\mathbf{X})\}$ s.t. $\&g[\mathbf{Y}](\mathbf{X})$ joins b and $b \notin \overline{R}$; and*
  - *(ii) $g_{inp}(\mathbf{Y})$.*

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

Our approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_C(\Pi)$, which substitutes all constants for all variables in all possible ways, we allow the ASP grounder GroundASP to optimize rules.

Definition 17 *We call rule r' an* o-strengthening *of a rule r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains only ordinary literals and no external atoms replacements.*

For a rule r (resp. a program Π), let $os(r)$ (resp. Π) be the set of all o-strengthenigs of r (resp. rules in Π).

Definition 18 *An algorithm GroundASP is a* faithful ASP grounder*, if for a safe ordinary program Π it returns a ground program $\Pi' \subseteq os(grnd_{C_\Pi}(\Pi))$ s.t. $\Pi' \equiv \Pi$ and for all $r \in grnd_{C_\Pi}(\Pi)$*

- *if $os(r) \cap \Pi' = \emptyset$, then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and*
- *if $r' \in \Pi'$ for some $r' \in os(r)$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies $B(r) \setminus B(r')$.*

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.

Example 17 Let $\Pi = \{r_1\colon a(t).\ \ r_2\colon b \vee c.\ \ r_3 : e \leftarrow b, f.\ \ r_4 : d \leftarrow a(X).\}$. A faithful ASP grounder may optimize this program to $\Pi' = \{r_1\colon a(t).\ \ r_2\colon b \vee c.\ \ r_4'\colon d\}$; both programs have the answer sets $\{a(t), b, d\}$ and $\{a(t), c, d\}$. Moreover, r_3 can be dropped as f is false in all answer sets, and the only instance $d \leftarrow a(t)$ of r_4 can be reduced to its o-strengthening r_4' as $a(t)$ is true in all answer sets. □

Algorithm GroundHEX

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

(a) Let R be a set 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 $e_{\&g[\mathbf{Y}]}(\mathbf{X})$
(b) **repeat**
  $\Pi_{pg} := \mathsf{GroundASP}(\Pi_p)$ `// partial grounding`
(c) **for** $\&g[\mathbf{Y}](\mathbf{X}) \in R$ *in a rule* $r \in \Pi$ **do** `// evaluate lde-safety-relevant external atoms`
    $\mathbf{A}_{ma} := \{p(\mathbf{c}) \in A(\Pi_{pg}) \mid p \in \mathbf{Y_m}\}$
(d) **for** $\mathbf{A}_{nm} \subseteq \{p(\mathbf{c}) \in A(\Pi_{pg}) \mid p \in \mathbf{Y_n}\}$ **do** `// do this under all relevant assignments`
      $\mathbf{A} := \mathbf{A}_{ma} \cup \mathbf{A}_{nm}$
(e) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})\}$ **do**
(f) Let $O = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
        $\Pi_p := \Pi_p \cup \{e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid \mathbf{x} \in O\}$ `// add ground guessing rule`
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 algorithm is formally stated in Algorithm GroundHEX; our naming convention is as follows. Program Π is the non-ground 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 using a heuristics (e.g., 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 guessing rules for external atoms 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* $e_{\&g[\mathbf{Y}]}(\mathbf{X})$. 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). Note that for ordinary ASP programs, the algorithm essentially reduces to a single call of GroundASP and thus does not cause overhead.

We illustrate our grounding algorithm with the following example.

Example 18 Let Π be the following program:

$$\Pi = \left\{ \begin{array}{llllllll} f_1\colon & d(a). & f_2\colon & d(b). & f_3\colon & d(c). & r_1\colon & s(Y) \leftarrow \&diff[d,n](Y), d(Y). \\ & & & & & & r_2\colon & n(Y) \leftarrow \&diff[d,s](Y), d(Y). \\ & & & & & & r_3\colon & c(Z) \leftarrow \&count[s](Z). \end{array} \right\}.$$

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, thus R in step (a) of GroundHEX can be set to $\{\&count[s](Z)\}$. Program Π_p at the beginning of the first iteration, i.e., when reaching (b), 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.

$$\Pi_p = \left\{ \begin{array}{lll} f_1\colon\ d(a). & f_2\colon\ d(b). & f_3\colon\ d(c). \\ g_1\colon\ e_1(Y) \vee ne_1(Y) \leftarrow d(Y). & & r_1\colon\ s(Y) \leftarrow e_1(Y), d(Y). \\ g_2\colon\ e_2(Y) \vee ne_2(Y) \leftarrow d(Y). & & r_2\colon\ n(Y) \leftarrow e_2(Y), d(Y). \\ & & r_3\colon\ c(Z) \leftarrow e_3(Z). \end{array} \right\}.$$

The program Π_{pg} then 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 (step (c)). 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 $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ (step (d)) because the external atom is nonmonotonic in s. In the course of this, in step (f) the algorithm adds the 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. □

It appears that this algorithm is sound and complete. This is independent of a concrete TBF, i.e., the following result (and the lemmas used in its proof) hold for all programs which are lde-safe wrt. *any* TBF.

Theorem 10 *If Π is an lde-safe* HEX*-program, then* GroundHEX*$(\Pi) \equiv \Pi$.*

The new grounding algorithm has all properties that are required in order to use it in the existing model-building framework for HEX-program evaluation, presented in (Eiter et al., 2011). As shown in Example 18, Algorithm GroundHEX needs to evaluate external atoms under exponentially many different interpretations in the worst case. This worst case applies whenever a nonmonotonic external atom is necessary for lde-safety, because then the external atom needs to be evaluated under all possible inputs. However, this worst case can be effectively avoided in most practically relevant cases by program decomposition as supported by the existing framework; see (Eiter et al., 2013a) for details.

5. Implementation and Evaluation

For implementing our technique, we integrated GRINGO (Gebser et al., 2007)[8] as grounder (implementation of Algorithm GroundASP) and CLASP into our prototype system DLVHEX. The evaluation is driven by an evaluation framework that relies on decomposition of the program into multiple *units*, which are independently grounded and solved, where GroundHEX is used for grounding; for more information, see (Eiter et al., 2011; Redl, 2014). 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).

[8] http://potassco.sourceforge.net

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.

Platform and Settings. We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM running an *HTCondor* load distribution system[9] which ensures robust runtimes (i.e., multiple runs of the same instance have negligible deviations) and using DLVHEX version 2.3.0. The grounder and solver backends for all benchmarks are GRINGO 3.0.4 and CLASP 2.1.3. For each instance, we limited the CPU usage to two cores and 8GB RAM. The timeout for all instances was 300 seconds. The encodings of all benchmarks discussed in this section have been included in Appendix C; the instances are available from http://www.kr.tuwien.ac.at/staff/redl/aspext, and the required plugins from the repository (https://github.com/hexhex).

While in principle different grounding and solving backends can be used, we chose GRINGO and CLASP for our experiments. The reason is that our algorithms use these components as black boxes (as explicitly shown for the grounding algorithm) and that a different backends do not influence the behavior of our algorithms; that is, the selection of a certain grounding and solving backend, or a certain version of such as backend, is an independent dimension (similar to e.g. the variable selection heuristics, etc.) which does not provide new insights concerning the two techniques that we want to compare. Moreover, as DLV is only available as binary and does not provide an API, its use within DLVHEX would require interprocess communication and parsing of results; this overhead could bias the performance comparison. Hence, we decided to focus on comparing strong and liberal safety.

5.1. Problem Suite

We now present the problems we are going to use as benchmarks and discuss the results for instances in a separate subsection.

While the modeling advantages of a broader class of supported programs and a less restricted syntax are contributions on their own, the main motivation for liberal safety was to improve efficiency. In contrast to strong safety, lde-safety needs no blind generation and import of a maximal set of possibly relevant constants. Instead, the algorithms compute the domain depending on the actual instance. The hypothesis is that we can expect lde-safety to eliminate large parts of the irrelevant domain and of the ground program in advance, which might lead to a decrease in grounding time and solving time.

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 18, which implements for each domain element x a choice from $sel(x)$ and $nsel(x)$ by an external atom, i.e., a partitioning of

[9] http://research.cs.wisc.edu/htcondor

the domain into two subsets, where sel may contain at most two elements. The program is as follows:

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

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 16. 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. Since the merging operation is implemented as an external source, also the operator for comparing two elements is implicitly given by the external source.

However, this is only a representative of a class of applications and performance cannot be compared to native merge sort implementations. Other applications of a similar structure are the application of algorithms to graph structures (e.g. balance maintenance in AVL trees, insertion into B-trees), binary search in dictionaries, and recursive Web queries.

Argumentation. In this benchmark, we use a HEX-program which computes specific extensions for Dung-style abstract argumentation frameworks (AFs) (Dung, 1995), 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 (Dunne, 2009). 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 (Eiter et al., 2014d) 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:

$$\begin{aligned}\{&(\textit{Wien Mitte}, \textit{Wien Mitte}\ (\textit{U4}), 10, \textit{change}),\\ &(\textit{Wien Mitte}\ (\textit{U4}), \textit{Stadtpark}\ (\textit{U4}), 1, \textit{U4}),\\ &(\textit{Stadtpark}\ (\textit{U4}), \textit{Karlsplatz}\ (\textit{U4}), 1, \textit{U4}),\\ &(\textit{Karlsplatz}\ (\textit{U4}), \textit{Karlsplatz}\ (\textit{U1}), 10, \textit{change}),\\ &(\textit{Karlsplatz}\ (\textit{U1}), \textit{Taubstummengasse}\ (\textit{U1}), 1, \textit{U1}),\\ &(\textit{Taubstummengasse}\ (\textit{U1}), \textit{Taubstummengasse}, 10, \textit{change})\}\end{aligned}$$

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 $(a, a', 10, \textit{change})$, 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 2 (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.

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 (Eiter et al., 2014d). 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.

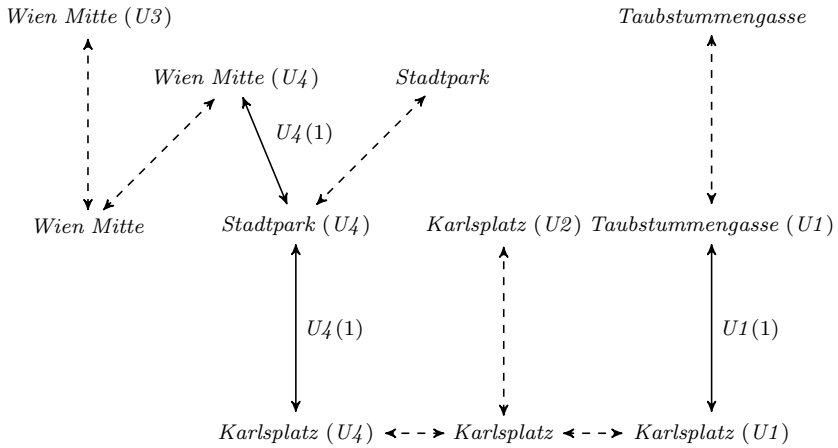

Figure 2: Graph representation of stations and lines (dashed: changes with costs 10, solid: given lines and costs)

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 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

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

(a) Reachability

#	w. domain predicates wall clock	ground	solve	w/o domain predicates 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 wall clock	ground	solve	w/o domain predicates 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 wall clock	ground	solve	w/o domain predicates 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 wall clock	ground	solve	w/o domain predicates wall clock	ground	solve
5 (10)	0.78 (0)	0.10 (0)	0.59 (0)	0.13 (0)	0.04 (0)	0.02 (0)
10 (10)	— (10)	— (0)	n/a (0)	0.25 (0)	0.09 (0)	0.08 (0)
15 (10)	— (10)	— (0)	n/a (0)	0.41 (0)	0.15 (0)	0.18 (0)
20 (10)	— (10)	— (0)	n/a (0)	1.27 (0)	0.43 (0)	0.75 (0)
25 (10)	— (10)	— (0)	n/a (0)	4.08 (0)	1.09 (0)	2.85 (0)
30 (10)	— (10)	— (0)	n/a (0)	34.82 (0)	3.95 (0)	30.14 (0)
35 (10)	— (10)	— (0)	n/a (0)	82.70 (0)	8.37 (0)	72.75 (0)
40 (10)	— (10)	— (0)	n/a (0)	255.91 (7)	213.97 (7)	41.06 (0)
45 (10)	— (10)	— (0)	n/a (0)	— (10)	— (10)	n/a (0)

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 (Eiter et al., 2013b), 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.

Note that the latest `DLV` version from 2012 supports list processing by dedicated built-in predicates. However, while this feature might appear to allow for implementing a similar application at first glance, it in fact comes with several limitations. First, `DLV` supports only low level list processing predicates such as splitting off the first element. Implementing advanced operations such as splitting lists in half and merging two sorted lists is cumbersome; external atoms are more convenient. Second, `DLV`'s built-in comparison predicates $<$ and $>$ impose for non-numeric constants some fixed ordering; the desired ordering must in general be computed within a sub-program. This adds to the complexity and can be problematic in case of unknown values and become even infeasible, as (large) sets of facts encoding the

comparison may have to be provided or computed. Third and most severe, the recursive rules over lists violate DLV's safety criteria, and the safety check must be disabled; but then termination is no longer guaranteed. Therefore, encoding this application in ordinary ASP is not appealing.

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 (%)*)[10], 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, 2c and 2e 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 $\lceil n \times 1.5 \rceil$ 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 (%)*), 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, 2c and 2e 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 results show only the runtimes without domain predicates and liberal safety, as for strong safety we observed only timeout instances.

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

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

(a) Full Map (single)

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

(b) Full Map (pair)

#	w/o domain predicates									
	wall clock		ground		solve		solution (%)	length	changes	lunch (%)
1 (50)	11.60	(0)	9.32	(0)	1.31	(0)	82.00	150.98	4.54	0.00
2 (50)	34.20	(0)	26.00	(0)	6.54	(0)	90.00	300.69	8.53	0.00
3 (50)	89.21	(0)	53.31	(0)	29.95	(0)	44.00	449.77	11.14	9.09
4 (50)	204.73	(4)	107.60	(4)	83.28	(0)	76.00	592.87	15.47	84.21
5 (50)	297.29	(44)	278.85	(44)	15.03	(0)	12.00	710.00	17.50	100.00
6 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	NaN	NaN

(c) Subway and Tram (single)

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

(d) Subway and Tram (pair)

#	w/o domain predicates									
	wall clock		ground		solve		solution (%)	length	changes	lunch (%)
1 (50)	8.17	(0)	7.32	(0)	0.44	(0)	98.00	133.76	3.67	0.00
2 (50)	23.35	(0)	19.09	(0)	2.78	(0)	100.00	269.54	7.62	0.00
3 (50)	59.04	(0)	36.47	(0)	17.64	(0)	98.00	390.37	10.08	0.00
4 (50)	147.43	(3)	75.46	(3)	59.60	(0)	94.00	582.55	14.51	89.36
5 (50)	255.94	(17)	161.93	(17)	76.12	(0)	66.00	636.55	16.61	100.00
6 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	NaN	NaN

(e) Subway Only (single)

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

(f) Subway Only (pair)

#	w/o domain predicates									
	wall clock		ground		solve		solution (%)	length	changes	lunch (%)
1 (50)	7.72	(0)	6.97	(0)	0.30	(0)	98.00	124.51	2.45	0.00
2 (50)	21.35	(0)	17.69	(0)	2.19	(0)	100.00	251.66	4.94	0.00
3 (50)	60.00	(0)	33.79	(0)	21.05	(0)	100.00	375.08	7.46	4.00
4 (50)	167.44	(5)	80.97	(5)	74.14	(0)	90.00	565.33	10.98	82.22
5 (50)	267.17	(20)	169.57	(20)	81.00	(0)	60.00	627.70	12.30	100.00
6 (50)	299.05	(48)	292.38	(48)	4.88	(0)	4.00	640.00	13.00	100.00
7 (50)	—	(50)	—	(50)	0.00	(0)	0.00	NaN	NaN	NaN

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) meeting locations and n restaurants are drawn randomly. The results are shown in Tables 2b, 2d and 2f 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. For single route planning, the results show only the runtimes without domain predicates and liberal safety; for strong safety we again observed only timeout instances.

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, in many cases, the worst case can be avoided by our new decomposition heuristics.

6. Related Notions of Safety

Our notion of lde-safety using $b_{syn}(\Pi, r, S, B) \cup b_{extsem}(\Pi$, r, S, B) compares to the traditionally used sde-safety and to other formalizations

6.1. Strong Safety

One can now show that lde-safety is strictly less restrictive than sde-safety.

Proposition 11 *Every sde-safe program* Π *is lde-safe.*

The converse does not hold, as there are clearlyx lde-safe programs that are not strongly safe, cf. the program Π' from the introduction.

6.2. VI-Restricted Programs

Calimeri et al. (2007) 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 position dependency graph is related to ours, but our notion is more fine-grained for attribute positions of external predicates. While we use a node $att_T(\&g[\mathbf{X}], i)$ 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 position $att(\&g[\mathbf{X}], i)$ 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 position in multiple rules are distinguished; this collapses distinct nodes in our attribute position dependency graph into one.

Example 19 Consider program $\Pi = \{r_1\colon\ t(X) \leftarrow s(X), \&e[X](Y),\ r_2\colon\ r(X) \leftarrow t(Z), \&e[Z](Y)\}$. We have the attribute positions

$att(s, 1)$, $att(t, 1)$, $att(r, 1)$, $att_{\text{I}}(\&e[X], 1)$, $att_{\text{O}}(\&e[X], 1)$, $att_{\text{I}}(\&e[Z], 1)$, $att_{\text{O}}(\&e[Z], 2)$
with edges

$(att(s, 1), att_{\text{I}}(\&e[X], 1))$, $(att_{\text{I}}(\&e[X], 1), att_{\text{O}}(\&e[X], 1))$, and $(att_{\text{O}}(\&e[X], 1), att(t, 1))$
originating from the first rule of Π, and the edges

$(att(t, 1), att_{\text{I}}(\&e[X], 1))$, $(att_{\text{I}}(\&e[Z], 1), att_{\text{O}}(\&e[Z], 1))$, $(att_{\text{O}}(\&e[Z], 1), att(t, r))$

originating from the second rule of Π. In contrast, Calimeri et al. (2007) have attribute positions

$att(s,1)$, $att(t,1)$, $att(r,1)$, $att(\&e,1)$, $att(\&e,2)$

with edges

$(att(s,1), att(\&e,1))$, $(att(\&e,1), att(\&e,2))$, $(att(\&e,2), att(t,1))$, $(att(t,1), att(\&e,1))$, $(att(\&e,2), att(r,1))$. □

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

Proposition 12 *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.

6.3. Logic Programs with Function Symbols

Syrjänen (2001) 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. Calimeri et al. (2007) 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 12 also lde-safe, we obtain:

Proposition 13 *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 (Gebser et al., 2007), argument-restricted programs (Lierler and Lifschitz, 2009) and Γ-acyclic programs Greco et al. (2012). 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 (Greco et al., 2013). 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-summarizing acyclicity (MSA)* (Grau et al., 2013), 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 identify 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 (Calimeri et al., 2008) map into our framework. Finitary programs (Bonatti, 2004, 2002) and FG programs (Calimeri et al., 2008), 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).

6.4. 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 (Zantema, 1994, 2001). 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.

6.5. Programs with Existential Quantification

There are extensions of ASP towards integration of logical existential quantification, see e.g., (Alviano et al., 2012). In order to guarantee finite restrictability, syntactic conditions are employed in order to ensure that a finite subset of the grounding suffices for query answering. The approach by Alviano et al. (2012) makes use of a *chase* procedure adopted from (Calì et al., 2009). Existential quantifiers can also easily be integrated into our algorithm. To this end, a generalized version of the algorithm presented in this paper supports *hooks* which can be used to integrate application-specific termination criteria. As an example, query answering over programs with existential quantifiers can sometimes be done using a finite grounding even if the program has infinite models. To this end, depending on the number of existentially quantified variables in the query, the grounding process is terminated after a finite number of steps; we refer to (Eiter et al., 2014c) for details.

6.6. Other Notions of Safety

Related to our semantic properties are the approaches by Sagiv and Vardi (1989), Ramakrishnan et al. (1987), and Krishnamurthy et al. (1996). They exploited finiteness of attribute positions (cf. item (i) in Definition 13) in sets of Horn clauses and derive finiteness of further attribute positions using *finiteness dependencies*. This is related to item (ii) in Definition 13 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 (Cabalar et al., 2009), (Lee et al., 2008), and (Bartholomew and Lee, 2010), 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.

7. 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 Calimeri et al. (2007), and semantic criteria related to those by Sagiv and Vardi (1989) and Ramakrishnan et al. (1987), 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 by Eiter et al. (2012). The worst case of the new algorithm can be effectively avoided as shown by Eiter et al. (2013a). An experimental evaluation of our implementation on synthetic and real applications shows a clear benefit.

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

We would like to thank the reviewers for their thoughtful constructive comments to improve this article. This work was supported by the Austrian Science Fund (FWF) via the project P24090.

References

Alviano, M., Faber, W., Leone, N., Manna, M., 2012. Disjunctive datalog with existential quantifiers: Semantics, decidability, and complexity issues. Theory and Practice of Logic Programming 12 (4-5), 701–718.

Baader, F., Hollunder, B., 1995. Embedding defaults into terminological knowledge representation formalisms. Journal of Automated Reasoning 14 (1), 149–180.

Bartholomew, M., Lee, J., 2010. A decidable class of groundable formulas in the general theory of stable models. In: 12th International Conf. Principles of Knowledge Representation and Reasoning (KR'10). AAAI Press, pp. 477–485.

Biere, A., Heule, M. J. H., van Maaren, H., Walsh, T. (Eds.), 2009. Handbook of Satisfiability. Vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press.

Bonatti, P. A., 2002. Reasoning with infinite stable models II: Disjunctive programs. In: 18th International Conf. Logic Programming (ICLP'02). LNCS 2401, Springer, pp. 333–346.

Bonatti, P. A., 2004. Reasoning with infinite stable models. Artificial Intelligence 156 (1), 75—111.

Brewka, G., Eiter, T., 2007. Equilibria in heterogeneous nonmonotonic multi-context systems. In: Holte, R. C., Howe, A. (Eds.), 22nd National Conf. Artificial Intelligence (AAAI'07). AAAI Press, pp. 385–390.

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

Cabalar, P., Pearce, D., Valverde, A., 2009. A revised concept of safety for general answer set programs. In: 10th International Conf. Logic Programming and Nonmonotonic Reasoning (LPNMR'09). LNCS 5753, Springer, pp. 58–70.

Calì, A., Gottlob, G., Lukasiewicz, T., 2009. Datalog$^{\pm}$: a unfied approach to ontologies and integrity constraints. In: Proc. 12th International Conf. Database Theory (ICDT '09). ACM, pp. 14–30.

Calimeri, F., Cozza, S., Ianni, G., 2007. External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence 50 (3–4), 333–361.

Calimeri, F., Cozza, S., Ianni, G., Leone, N., 2008. Computable functions in ASP: Theory and implementation. In: International Conf. Logic Programming (ICLP'08). LNCS 5366, Springer, pp. 407–424.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T., 2013. ASP-Core-2: 4th ASP competition official input language format.

Dung, P. M., 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artificial Intelligence 77 (2), 321–357.

Dunne, P. E., 2009. The computational complexity of ideal semantics. Artificial Intelligence 173 (18), 1559–1591.

Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P., May 2011. Pushing efficient evaluation of hex programs by modular decomposition. In: Delgrande, J., Faber, W. (Eds.), 11th Int'l Conf. Logic Programming and Nonmonotonic Reasoning (LPNMR'11). LNCS/LNAI 6645, Springer, pp. 93–106.

Eiter, T., Fink, M., Krennwallner, T., Redl, C., July 2012. Conflict-driven ASP solving with external sources. Theory and Practice of Logic Programming 12 (4-5), special issue on ICLP 2012, 659–679.

Eiter, T., Fink, M., Krennwallner, T., Redl, C., 2013a. Grounding HEX-programs with expanding domains. In: Pearce, D., Tasharrofi, S., Ternovska, E., Vidal, C. (Eds.), Informal Proc. 2nd Workshop on Grounding and Transformations for Theories with Variables (GTTV'13), Sept. 15, 2013, Corunna, Spain. pp. 3–15.

Eiter, T., Fink, M., Krennwallner, T., Redl, C., July 2013b. Liberal Safety Criteria for HEX-Programs. In: desJardins, M., Littman, M. (Eds.), 27th AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA. AAAI Press, pp. 267–275.

Eiter, T., Fink, M., Krennwallner, T., Redl, C., September 2014a. Domain expansion for ASP-programs with external sources. Tech. Rep. INFSYS RR-1843-14-02, Institut f. Informationssysteme, TU Wien.

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

Eiter, T., Fink, M., Redl, C., Krennwallner, T., 2014c. HEX-programs with existential quantification. In: In: Hanus, M., Rocha, R. (Eds.), Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, Sept. 11-13, 2013, Revised Selected Papers. LNCS 8439, Springer, pp. 99–117.

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

Eiter, T., Ianni, G., Schindlauer, R., Tompits, H., 2005. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Kaelbling, L. P., Saffiotti, A. (Eds.), 19th International Joint Conf. Artificial Intelligence (IJCAI'05). Professional Book Center, pp. 90–96.

Eiter, T., Ianni, G., Schindlauer, R., Tompits, H., 2006. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In: Sure, Y., Domingue, J. (Eds.), 3rd European Conf. Semantic Web (ESWC'06). LNCS 4011, Springer, pp. 273–287.

Eiter, T., Krennwallner, T., Prandtstetter, M., Rudloff, C., Schneider, P., Straub, M., 2014d. Semantically enriched multi-modal routing. International Journal of Intelligent Transportation Systems Research. Published online August 5, 2014. URL DOI 10.1007/s13177-014-0098-8

Erdem, E., Erdem, Y., Erdogan, H., Öztok, U., 2011. Finding answers and generating explanations for complex biomedical queries. In: Burgard, W., Roth, D. (Eds.), Proc 25th Conf. Artificial Intelligence (AAAI 2011). AAAI Press.

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

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M., 2011. Potassco: The potsdam answer set solving collection. AI Commun. 24 (2), 107–124.

Gebser, M., Ostrowski, M., Schaub, T., 2009. Constraint answer set solving. In: Hill, P., Warren, D. (Eds.), Proc. 25th Int'l Conf. Logic Programming (ICLP'09). LNCS 5649, Springer, pp. 235–249.

Gebser, M., Schaub, T., Thiele, S., 2007. Gringo: A new grounder for answer set programming. In: Proc. 9th International Conf. Logic Programming and Nonmonotonic Reasoning (LPNMR'07). LNCS 4483. Springer, pp. 266–271.

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

Grau, B. C., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z., 2013. Acyclicity notions for existential rules and their application to query answering in ontologies. Journal of Artificial Intelligence Research 47, 741–808.

Greco, S., Molinaro, C., Trubitsyna, I., 2013. Bounded programs: A new decidable class of logic programs with function symbols. In: Proceedings of the 23rd International Joint Conf. Artificial Intelligence (IJCAI 2013). AAAI Press, pp. 926–931.

Greco, S., Spezzano, F., Trubitsyna, I., 2012. On the termination of logic programs with function symbols. In: Dovier, A., Costa, V. S. (Eds.), ICLP (Technical Communications). Vol. 17 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 323–333.

Hoehndorf, R., Loebe, F., Kelso, J., Herre, H., 2007. Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. BMC Bioinformatics 8 (1), 377.

Heymans, S., Nieuwenborgh, D. V., Vermeir, D., 2007. Open answer set programming for the semantic web. Journal of Applied Logic 5 (1), 144 – 169.

Krishnamurthy, R., Ramakrishnan, R., Shmueli, O., 1996. A framework for testing safety and effective computability. Journal of Computer and System Sciences 52 (1), 100–124.

Lassila, O., Swick, R., 1999. Resource description framework (RDF) model and syntax specification.

Lee, J., Lifschitz, V., Palla, R., 2008. Safe formulas in the general theory of stable models (preliminary report). In: 24th Int'l Conf. Logic Programming (ICLP'08). LNCS 5366, Springer, pp. 672–676.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F., Jul. 2006. The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Logic 7 (3), 499–562.

Lierler, Y., Lifschitz, V., 2009. One more decidable class of finitely ground programs. In: 25th International Conf. Logic Programming (ICLP'09). LNCS 5649, Springer, pp. 489–493.

Marek, V., Niemelä, I., Truszczyński, M., 2004. Logic programs with monotone cardinality atoms. In: Proceedings LPNMR-2004. LNCS 2923, Springer, pp. 154–166.

Mosca, A., Bernini, D., 2008. Ontology-driven geographic information system and dlvhex reasoning for material culture analysis. In: Italian Workshop RiCeRcA 2008.

Ramakrishnan, R., Bancilhon, F., Silberschatz, A., 1987. Safety of recursive Horn clauses with infinite relations. In: 6th Symposium on Principles of Database Systems (PODS'87). ACM, pp. 328–339.

Redl, C., April 2014. Answer set programming with external sources: Algorithms and efficient evaluation. PhD thesis, TU Wien, Austria. URL http:/www.ub.tuwien.ac.at/diss/AC11705117.pdf

Sagiv, Y., Vardi, M. Y., 1989. Safety of datalog queries over infinite databases. In: 8th Symposium on Principles of Database Systems (PODS'89). ACM, pp. 160–171.

Schüller, P., Patoglu, V., Erdem, E., 2013. Levels of integration between low-level reasoning and task planning. In: Workshops at the 27th AAAI Conf. Artificial Intelligence (AAAI 2013) – Intelligent Robotic Systems, Bellevue, WA, July 14th, 2013. AAAI Press.

Simons, P., Niemelä, I., Soininen, T., 2002. Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234.

Syrjänen, T., 2001. Omega-restricted logic programs. In: 6th International Conf. Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Springer, pp. 267–279.

Zakraoui, J., Zagler, W. L., 2012. A method for generating CSS to improve web accessibility for old users. In: Int. Conf. on Computers Helping People with Special Needs (ICCHP). pp. 329–336.

Zantema, H., 1994. Termination of term rewriting: Interpretation and type elimination. Journal of Symbolic Computation 17 (1), 23–50.

Zantema, H., 2001. The termination hierarchy for term rewriting. Applicable Algebra in Engineering, Communication and Computing 12 (1-2), 3–19.

Zirtiloğlu, H., Yolum, P., 2008. Ranking semantic information for e-government: complaints management. In: 1st International Workshop on Ontology-supported Business Intelligence (OBI'08). No. 5 in OBI'08. ACM, p. 7.

Appendix A. Extended Semantic Term Bounding Function

In addition to the properties from Section 3.4.2, cyclicity and wellorderings can be exploited for deriving boundedness of terms. Our concept is based on *malign cycles* in the *positive attribute position dependency graphs*, which are the source of any infinite value invention. The *positive attribute position dependency graph* $G_A(\Pi)$ has as nodes the attribute positions of Π and its edges model the information flow between the attribute positions. 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 $att(q, j)$ to $att(p, i)$.

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

Definition 19 (Positive Attribute Position Dependency Graph) *For a given* HEX*-program* Π*, the* positive attribute position dependency graph $G_A(\Pi) = \langle Attr, E\rangle$ *has as nodes* $Attr$ *the set of all attribute positions 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* $(att(q, j), att(p, i)) \in E$*.*
- *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* $(att(p, i), att_{\text{I}}(\&g[\mathbf{X}], j)) \in E$*.*
- *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* $(att_{\text{O}}(\&g[\mathbf{X}], j), att_{[}(\&h,)X]\text{I}i) \in E$*.*
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$ *then* $(att_{\text{I}}(\&g[\mathbf{X}], i), att_{\text{O}}(\&g[\mathbf{X}], j)) \in E$ *for all* $1 \leq i \leq iar(\&g)$ *and* $1 \leq j \leq oar(\&g)$*..*
- *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* $(att_{\text{O}}(\&g[\mathbf{X}], j), att(p, i)) \in E$*.*
- *If* $\&g[\mathbf{X}](\mathbf{Y}) \in B^+(r)$ *s.t.* $p = X_i$ *and* $\tau(\&g, i) = \mathbf{pred}$*, then* $(att(p, k), att_{\text{I}}(\&g[\mathbf{X}], i)) \in E$ *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 20 (cont'd) The (positive) attribute position dependency graph of the program from Example 7 has the attribute positions

$\{att(s, 1), att(dom, 1), att_{\text{I}}(\&concat[X, x], 1), att_{\text{I}}(\&concat[X, x], 2), \quad att_{\text{O}}(\&concat[X, x], 1)\}$
and the edges

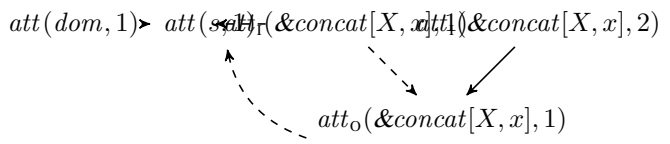

Figure A.3: Attribute position dependency graph for Π in Example 20

$\{(att(dom,1),\ att(s,1)), (att(s,1),\ att_{\text{I}}(\&concat[X,x],1)),$
$(att_{\text{I}}(\&concat[X,x],1),\ att_{\text{O}}(\&concat[X,x],1)),$
$(att_{\text{I}}(\&concat[X,x],2),\ att_{\text{O}}(\&concat[X,x],1)),\ (att_{\text{O}}(\&concat[X,x],1),\ att(s,1))\}$
(see Figure A.3). □

Definition 20 (Benign and Malign Cycles) *A cycle K in $G_A(\Pi)$ is* benign *wrt. a set of safe attribute positions S, if there exists a well-ordering $\leq_{\mathcal{C}}$ of $\mathcal{C}$, such that for every attribute position $att_{\text{O}}(\&g[\mathbf{X}],j) \notin S$ in the cycle, we have that $f_{\&g}(\mathbf{A}, x_1, \ldots, x_m, t_1, \ldots t_n) = 0$ whenever*

- *some x_i for $1 \leq i \leq m$ is a predicate parameter, $att_{\text{I}}(\&g[\mathbf{X}],i) \notin S$ is in K, and $(s_1, \ldots, s_{ar(x_i)}) \in ext(\mathbf{A}, x_i)$, 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, $att_{\text{I}}(\&g[\mathbf{X}],i) \notin S$ 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 21 (cont'd) The cycle in $G_A(\Pi)$ (dashed lines in Figure A.3) is malign wrt. $S = \emptyset$ because there is no well-ordering as required by Definition 20. 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 position dependency graph becomes benign using $<$ over the string lengths as well-ordering. □

We can now define an extended semantic term bounding function which takes into account the properties from Section 3.4.2 and malign cycles.

Definition 21 (Extended Semantic Term Bounding Function) *Let $b_{extsem}(\Pi, r, S, B)$ be defined such that we have $t \in b_{extsem}(\Pi, r, S, B)$ iff*

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

(ii) $t \in b_{sem}(\Pi, r, S, B)$.

Proposition 14 *Function $b_{extsem}(\Pi, r, S, B)$ defined above is a TBF.*

Appendix B. Proofs

Appendix B.1. Liberal Safety (Section 3)

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

For the following proofs, let $B_{n+1,j}$ denote the intermediate result when computing $b^{\infty}_{\Pi,r,S_n(\Pi)}(\emptyset)$ after the j-th iteration. Then $B_n(r,\Pi) = \bigcup_{j\geq 0} B_{n,j}$ and for each $t \in B_n(r,\Pi)$ there is some $j \geq 0$ such that $t \in B_{n,j}(r,\Pi)$.

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 attribute positions in $S_n(\Pi)$ are lde-safe (outer induction hypothesis). We first show that for each rule r and term $t \in B_{n+1}(r,\Pi)$, the set of ground instances of r in $G^{\infty}_{\Pi}(\emptyset)$ contains only finitely many different substitutions for t.

As stated above, since $t \in B_{n+1}(r,\Pi)$ there is some $j \geq 0$ such that $t \in B_{n+1,j}(r,\Pi)$. We consider $B_{n+1,j}(r,\Pi)$ and again prove this by induction on j. For $j = 0$ we have $B_{n+1,0}(r,\Pi) = \emptyset$ and the proposition holds trivially. For the induction step $j \mapsto j+1$, assume that the terms in $B_{n+1,j}(r,\Pi)$ are bounded (inner induction hypothesis). Let $t \in B_{n+1,j+1}(r,\Pi)$. 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 attribute positions 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 $t \in B_{n+1,j}(r,\Pi)$ 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 $B_n(r,\Pi)$, also that for each $t \in B_{n+1}(r,\Pi)$ the set of ground instances of r in $G^{\infty}_{\Pi}(\emptyset)$ contains only finitely many different substitutions for t.

If $att(p,i) \in S_{n+1}(\Pi)$, then for each rule $r \in \Pi$ and atom $p(t_1,\ldots,t_{ar(p)}) \in H(r)$ we have $t_i \in B_{n+1}(r,\Pi)$. 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 $att_{\text{I}}(\&g[\mathbf{X}],i) \in S_{n+1}(\Pi)$, 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 $Y_i \in B_{n+1}(r,\Pi)$, 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 attribute positions $att(p,1),\ldots,att(p,ar(p))$ in $G^{\infty}_{\Pi}(\emptyset)$ is finite by the (outer) induction hypothesis.

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

If $att_{\text{I}}(\&g[\mathbf{X}],1),\ldots,att_{\text{I}}(\&g[\mathbf{X}],ar_{\text{I}}(\&g)) \in S_n(\Pi)$, 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 $att_{\text{O}}(\&g[\mathbf{X}],i)$ can satisfy the rule body, thus $G^{\infty}_{\Pi}(\Pi)$ will also contain only finitely many values for $att_{\text{O}}(\&g[\mathbf{X}],i)$. 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 lde-safe by assumption, $a \in S_\infty(\Pi)$ for all attribute positions a of Π. Then by Corollary 4, the range of all attribute positions of Π in $G_\Pi^\infty(\emptyset)$ is finite. But then there exists also only a finite number of ground atoms in $G_\Pi^\infty(\emptyset)$. 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 $G_\Pi^\infty(\emptyset)$ 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 $grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$. We will show the more general proposition $grnd_C(\Pi) \equiv^{pos} grnd_{C'}(\Pi)$ for any $C' \supseteq C$.

($\Rightarrow$) Suppose $\mathbf{A} \in \mathcal{AS}\big(grnd_C(\Pi)\big)$. 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 $\mathbf{A}' = \mathbf{A} \cap A\big(grnd_C(\Pi)\big)\big\}$ is a model of $grnd_C(\Pi)$. 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 $\mathbf{A}' \subseteq \mathbf{A}$, and $\mathbf{A}' \subsetneq \mathbf{A}$ 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 $\mathbf{A}' = \mathbf{A}$ 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 $att(q, j)$, 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 attribute positions 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 Condition (i) holds, then the claim follows immediately from finiteness of the domain of the respective external atom.

If Condition (ii) 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 $att(Y_j, k) \in S$ 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 (iii) 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 $G_\Pi^\infty(\emptyset)$ because b_i is a TBF. □

Algorithm GroundHEXNaive:

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

(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 $e_{\&g\mathbf{Y}}(\mathbf{X})$
(b) **repeat**
  $\Pi_{pg} \leftarrow grnd_C(\Pi_p)$ with constants C in Π_p `// partial grounding`
(c) **for** *all models* $\mathbf{A}$ *of* Π_{pg} *over* $A(\Pi_{pg})$ **do** `// check if the grounding is large enough`
(d) **for** $\&g[\mathbf{Y}](\mathbf{X})$ *in a rule* $r \in \Pi$ **do** `// evaluate all external atoms`
(e) **for** $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in \mathbf{A}\}$ **do**
(f) Let $O = \{\mathbf{x} \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
  $\Pi_p \leftarrow \Pi_p \cup \{e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid \mathbf{x} \in O\}$ `// add ground guessing rules`
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 $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in Π_g by $\&g[\mathbf{y}](\mathbf{x})$
return Π_g

Appendix B.2. Grounding Liberally Domain-expansion Safe HEX*-Programs (Section 4)*

Towards a proof of the soundness and correctness of Algorithm GroundHEX we first consider a slower but easier to prove 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 18.

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.

Example 22 Let $\Pi = \{d(x); q(Y) \leftarrow d(X), \&concat[X, a](Y)\}$ be the 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 $\Pi_p = \{d(x) \vee d(y); q(Y) \leftarrow d(X), e_{\&concat[X,a]}(Y); g_{inp}(X) \leftarrow d(X)\}$, where g_{inp} is the unique auxiliary input predicate for $\&concat[X, a](Y)$. The grounding step yields then $\Pi_{pg} = \{d(x); q(Y) \leftarrow d(X), e_{\&concat[X,a]}(Y); g_{inp}(X, a) \leftarrow d(X) \mid X, Y \in \{x, y\}\}$. 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 $e_{\&g[x,a]}(xa) \vee ne_{\&g[x,a]}(xa) \leftarrow$ and $e_{\&g[y,a]}(xa) \vee ne_{\&g[y,a]}(ya) \leftarrow$ to Π_p and grounding starts over again. In the next iteration, the rule instances $q(xa) \leftarrow d(x), e_{\&concat[x,a]}(xa)$ and $q(ya) \leftarrow d(y), e_{\&concat[y,a]}(ya)$ 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, whenever we write Π_p or Π_{pg} in one of the proofs, then by convention 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 22 *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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$. It is $n+1$, if* Π *contains a rule with head $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x})$), 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*.

Towards a proof of the soundness and correctness of Algorithm GroundHEXNaive, we first prove the following lemma.

Lemma 15 *Let $\Pi_g =$ GroundHEXNaive(Π) 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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $e_{\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 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 $e_{\&g[\mathbf{v}]}(\mathbf{u}) \vee ne_{\&g[\mathbf{v}]}(\mathbf{u}) \leftarrow$ 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 that Algorithm GroundHEXNaive is sound and complete.

Theorem 16 *If Π is a lde-safe* HEX*-program and $\Pi_g =$ GroundHEXNaive(Π), then $\Pi_g \equiv \Pi$.*

Proof. Let $\Pi_g =$ GroundHEXNaive(Π). 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

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

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

Termination of the algorithm follows from Theorem 10, 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 15 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 15, $\mathbf{A} \not\models B(r)$ 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 15, $\mathbf{A}$ 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 $grnd_C(\Pi)$. □

We now come to the proof of soundness and completeness of our optimized Algorithm GroundHEX. We start with a lemma analogous to Lemma 15.

Lemma 17 *Let $\Pi_g =$ GroundHEX(Π) 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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \in A(\Pi_{pg})$, add $e_{\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}_g, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $e_{\&g[\mathbf{y}]}(\mathbf{x})$ or $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $e_{\&g\mathbf{y}}(\mathbf{x})$ in place of $\&g[\mathbf{y}](\mathbf{x})$ and we set $e_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 16, 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 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 18 *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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in Π_{pg} is satisfied by $\mathbf{A}_{pg}$, add $e_{\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{\&g\mathbf{y}}(\mathbf{x})$ 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 $e_{\&g[\mathbf{y}]}(\mathbf{x})$ nor $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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.

Note 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 assignment) 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 19 *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 $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow B$ in $grnd_C(\Pi_p)$ is satisfied by $\mathbf{A}_p$, add $e_{\&g[\mathbf{y}]}(\mathbf{x})$ if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$ and add atom $ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $e_{\&g[\mathbf{y}]}(\mathbf{x})$ nor

$ne_{\&g[\mathbf{y}]}(\mathbf{x})$ 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 $grnd_C(\Pi)$.

Note 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 assignment) 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 20 *Let Π_g = GroundHEX(Π) and C be the constants which appear in Π_g. It holds that $\mathcal{AS}(\Pi_g) = \mathcal{AS}(grnd_C(\Pi))$.*

Proof. (⇒) Let $\mathbf{A} \in \mathcal{AS}(\Pi_g)$. Then by Lemma 18 it can be extended to an answer set $\mathbf{A}_{pg}$ of Π_{pg}. By Definition 18, $\mathbf{A}_{pg}$ 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 18, 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 18, 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 18 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 19, $\mathbf{A}$ 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 10.

Proof of Theorem 10. 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 assignments 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 20, 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 17, $\mathbf{A}$ 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 17 $\mathbf{A} \not\models B(r)$ 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 17, $\mathbf{A}$ 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 $grnd_{C'}(\Pi)$. 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 17, 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 17, 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. □

Appendix B.3. Related Notions of Safety (Section 6)

Proof of Proposition 11. Suppose Π is strongly safe. We show that for any attribute position α of Π, we have $a \in S_n(\Pi)$ for some $n \geq 0$, i.e., a is lde-safe.

Let a be an attribute position 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 lde-safe.

If $j = 0$ we make a case distinction. Case 1: if a is of form $att(p, i)$, then there is no information flow from a malign cycle wrt. $\emptyset$ to $att(p, i)$. Therefore, for every rule r with $p(t_1, \ldots, t_\ell) \in H(r)$ we have that $t_i \in B_{n+1}(r, \Pi)$ for all $n \geq 0$ due to Condition (i) in Definition 21. But then $att(p, i)$ is lde-safe.

Case 2: if a is of form $att_{\mathrm{I}}(\&g[\mathbf{X}], i)$, then for every $Y_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{const}$ we have $Y_i \in B_{n+1}(r, \Pi)$ due to Condition (i) in Definition 21, and for every predicate $p_i \in \mathbf{Y}$ with $type(\&g, i) = \mathbf{pred}$ we have that $att(p_i, j)$ is lde-safe for every $1 \leq j \leq ar(p_i)$ by Case 1; note that $att(p_i, j)$ is not reachable from any malign cycle wrt. $\emptyset$ because this would by transitivity of reachability mean that also $att_{\mathrm{I}}(\&g[\mathbf{X}], i)$ is reachable from such a cycle, which contradicts our assumption. But then also $att_{\mathrm{I}}(\&g[\mathbf{X}], i)$ is lde-safe by Definition 8.

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

Induction step $j \mapsto j + 1$: If a is reachable from $j + 1$ malign cycles wrt. $\emptyset$, then there is an attribute position α' in such a cycle C from which a is reachable. The malign cycle C wrt. $\emptyset$ contains an attribute position of kind $att_{\mathrm{O}}(\&g[\mathbf{X}], i)$, corresponding to an external atom $\&g[\mathbf{Y}](\mathbf{X})$ in rule r. Since $att_{\mathrm{O}}(\&g[\mathbf{X}], i)$ 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 $att(p, k)$ 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 $att(p, k)$ is not part of it. Therefore $att(p, k)$ 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 $att(p, k)$ is lde-safe by induction hypothesis. But then by Condition (ii) in Definition 9, also a is lde-safe. □

Proof of Proposition 12. We first reformulate the definitions of *blocking* and *savior attribute positions* in an inductive way, which is possible because criteria are monotonic.

Blocking:

- $\mathit{blocked}_0(r) = \emptyset$ for all $r \in \Pi$
- $\mathit{blocked}_{n+1}(r) = \big\{ \mathit{att}(p,i) \mid \mathit{att}(p,i)$ is dangerous in r and $\mathit{att}(p,i)$ 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 $\mathit{att}(q,i) \in \mathit{savior}_n \big\}$,
  for all $n \geq 0$
- $\mathit{blocked}_\infty(r) = \bigcup_{n \geq 0} \mathit{blocked}_n(r)$

Savior attribute positions:

- $\mathit{savior}_0 = \emptyset$
- $\mathit{savior}_{n+1} = \big\{ \mathit{att}(p,i) \mid$ 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 $\mathit{att}(q,j) \in \mathit{savior}_n$ in $B^+(r)$; or
  $\mathit{att}(p,i) \in \mathit{blocked}_n(r) \big\}$,
  for all $n \geq 0$
- $\mathit{savior}_\infty = \bigcup_{n \geq 0} \mathit{savior}_n$

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

- If $\mathit{att}(p,i) \in \mathit{blocked}_n(r)$ and $\mathit{att}(p,i)$ captures X in r, then $X \in B_n(r, \Pi, S)$.
- If $\mathit{att}(p,i) \in \mathit{savior}_n$ for some $n \geq 0$, then $\mathit{att}(p,i) \in S_n(\Pi)$.

For $n = 0$ this is trivial.

For the induction step $n \mapsto n+1$, suppose $\mathit{att}(p,i) \in \mathit{blocked}_{n+1}(r)$. Then $\mathit{att}(p,i)$ 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 $\mathit{att}(q,j) \in \mathit{savior}_n$ Then, by the induction hypothesis, $\mathit{att}(q,j)$ is lde-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., $Y \in B_{n+1,1}(r, \Pi)$. Then by Condition (iii) in Definition 9 also all output variables $X \in \mathbf{X}$ are declared bounded in the second step, i.e., $X \in B_{n+1,2}(r, \Pi)$. Thus we have $X \in B_{n+1}(r, \Pi, S_n(\Pi))$.

Now suppose $\mathit{att}(p,i)$ 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 $\mathit{att}(q,j) \in \mathit{savior}_n$ in $B^+(r)$; or

(iii) $\mathit{att}(p,i) \in \mathit{blocked}_n(r)$.

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

This shows that all dangerous (but blocked) attribute positions are lde-safe. It remains to show that also all non-dangerous attribute positions are lde-safe. Let a be such an attribute position. 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 $att(p, i)$, then it is dangerous and we already know that it is lde-safe. Otherwise it is an external input attribute position of form $att_{\mathrm{I}}(\&g[\mathbf{X}], i)$ or an output attribute position of form $att_{\mathrm{O}}(\&g[\mathbf{X}], i)$. If it is an input attribute position, then we know that its cyclic input depends (possibly transitively) on lde-safe ordinary attribute positions. As the output attribute positions of external atoms become lde-safe as soon as the input becomes lde-safe by Definition 8, lde-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 lde-safe after finitely many steps (since the cycle is of finite length). This shows that all attribute positions in cycles in $G_A(\Pi)$ are lde-safe.

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

Proof of Proposition 13. By Theorem 7 by Calimeri et al. (2007) $F(\Pi)$ is VI-restricted, and thus by Proposition 12 it is also lde-safe using $b_{synsem}(\Pi, r, S, B)$. □

Proof of Proposition 14. If t is in the output of $b_{extsem}(\Pi, r, S, B)$, 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 attribute positions in S have a finite domain by assumption. For the remaining attribute positions 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 from Proposition 8. □

Appendix C. Benchmark Problem Encodings

In this appendix, we give details to some of the encodings used for our benchmarks in Section 5; for the complete set of encodings we refer to Redl (2014). The encodings are very natural representations of the underlying problems in HEX and have not been optimized for our algorithms, thus they have been developed under realistic conditions. For each problem we show the lde-safe version and point out how it can be transformed into a strongly safe program. However, in all programs the underlying idea is the same, namely to import or generate a sufficiently large domain in advance and use this domain to manually bound variables which are otherwise not (strongly) safe.

Appendix C.1. Reachability

We assume that the instance consists of two facts $start(s)$ and $end(e)$. The goal is to check if node e is reachable from node s and to compute a witness in this case (i.e., a path from s to e).

The problem encoding uses the following set of rules to import the relevant part of the graph, consisting of all nodes and edges reachable from s. We use an external atom $\&out[X](Y)$ to get all nodes Y

which are directly reachable from the node X.

$$\begin{aligned} edge(X,Y) &\leftarrow node(X), \&out[X](Y) \\ node(X) &\leftarrow start(X) \\ node(X) &\leftarrow end(X) \\ node(X) &\leftarrow edge(X,Y) \end{aligned}$$

We then guess the path as a set of pairs (n, i), where n is the node visited in step i. Constraints are used to ensure that the path is consecutive and reaches e in the last step.

$$\begin{aligned} path(X,0) &\leftarrow start(X) \\ &\leftarrow path(X,N), path(Y,N), X \neq Y \\ path(Y,N) \vee \overline{path}(Y,N) &\leftarrow node(Y), \#int(N) \\ &\leftarrow path(X,N), path(Y,N2), N2 = N+1, \text{not } edge(X,Y) \\ existspath(N) &\leftarrow path(X,N) \\ &\leftarrow \#int(N), \text{not } existspath(N), existspath(N2), N2 = N+1 \\ lastnode(X) &\leftarrow path(X,N), N2 = N+1, \text{not } existspath(N2) \\ &\leftarrow end(Z), lastnode(Y), Z \neq Y \end{aligned}$$

The strongly safe version of the program needs to import the whole graph a priory because the external atom $\&out[X](Y)$ appears in a cycle and thus needs a domain predicate which bounds Y. This can be realized by adding an external atom $\&allNodes[](Y)$ to the first rule of the encoding, provided that it is true for all nodes Y of the graph.

Appendix C.2. Mergesort

In this benchmark encoding, lists are encoded as string constants. A dedicated separator character allows for decoding the list representations in the external atoms.

We first use a set of rules to split the input list inp, given by a fact $list(inp)$, recursively into half. For each list l, its halves l_1 and l_2 are generated by the use of an external atom $\&splitHalf[l](l_1, l_2)$. We then derive an atom $sublist(l, l_1, l_2)$ that allows us later to reassemble the list from its (sorted) sublists. Let ϵ denote the empty list.

$$\begin{aligned} &sublist(L, H1, H2) \leftarrow list(L), \&splitHalf[L](H1, H2) \\ &sublist(L, H1, H2) \leftarrow sublist(\_, L, \_), \&splitHalf[L](H1, H2), H1 \neq \epsilon, H2 \neq \epsilon \\ &sublist(L, H1, H2) \leftarrow sublist(\_, \_, L), \&splitHalf[L](H1, H2), H1 \neq \epsilon, H2 \neq \epsilon \end{aligned}$$

We declare each sublist of length 1 immediately as sorted. We use an external atom $\&getLength[l](L)$ to get the length L of list l.

$$\begin{aligned} &sorted(L, L) \leftarrow sublist(L, \_, \_), \&getLength[L](1) \\ &sorted(L, L) \leftarrow sublist(\_, L, \_), \&getLength[L](1) \\ &sorted(L, L) \leftarrow sublist(\_, \_, L), \&getLength[L](1) \end{aligned}$$

Finally, we realize the merge step as follows. For each list l and its sublists l_1 and l_2, we determine the sorted versions s_1 and s_2 of l_1 and l_2, respectively, and merge them using external atom $\&merge[s_1, s_2](s)$ to get the sorted version of l.

$$\begin{aligned} sorted(L, S) \leftarrow & sublist(L, H1, H2), sorted(H1, H1s), sorted(H2, H2s), \\ & \&merge[H1s, H2s](S) \\ output(Sorted) \leftarrow & list(Final), sorted(Final, Sorted) \end{aligned}$$

Note that both the splitting and the merging part uses external atoms in cycles, thus the program is not strongly safe. In order to make it strongly safe, we need to generate all lists (i.e., permutations of the input list) a priori and add domain predicates.

For the strongly safe version of the program, note that the domain for sorting a list of size n consists of all $n!$ permutations (we cannot exclude any permutations in advance unless we solve parts of the problem before grounding). That is, for a strongly safe encoding one needs to generate all permutations l and add them as facts of kind $perm(l)$ to the program. This allows for adding atoms of kind $perm(L)$ to the remaining rules in order to make a variable L safe, but is clearly infeasible in practice.

Appendix C.3. Route Planning

We present the encodings for the two route planning scenarios separately. Once the single route planning scenario was introduced, the extension to pair route planning is quite simple. In fact, we will present a more general encoding which allows for planning tours for an arbitrary number of persons.

Appendix C.3.1. Single Route Planning

We start by guessing a sequence of the locations which are defined in the instance by facts of kind $location(loc)$. The following rules ensure that for a set L of n locations, the atoms $seq(i, loc_i)$ for $0 \leq i < n$, order the locations such that $L = \{loc_i \mid 0 \leq i < n\}$.

$$\begin{aligned} seq(I, L) \vee \overline{seq}(I, L) &\leftarrow location(L), \#int(I) && \text{(R1)} \\ &\leftarrow seq(I1, L), seq(I2, L), I1 \neq I2 \\ &\leftarrow seq(I, L1), seq(I, L1), L1 \neq L2 \\ haveSeq(L) &\leftarrow seq(I, L) \\ &\leftarrow location(L), \text{not } haveSeq(L) \\ haveLoc(I) &\leftarrow seq(I, L) \\ &\leftarrow seq(I, L), I1 < I, \#int(I1), \text{not } haveLoc(I1) \end{aligned}$$

The following rules choose exactly one restaurant and add it to the set of locations to visit, if necessary; possible locations for having lunch are assumed to be defined by facts of kind $possibleRestaurant(r)$.

$$\begin{aligned} restaurant(R) \vee \overline{restaurant}(R) &\leftarrow haveLunch, possibleRestaurant(R) \\ restaurantChosen &\leftarrow restaurant(R) \\ &\leftarrow restaurant(R1), restaurant(R2), R1 \neq R2 \\ &\leftarrow haveLunch, \text{not } restaurantChosen \\ location(R) &\leftarrow restaurant(R) \end{aligned}$$

We further need rules to check if our tour has to include a restaurant. The constant $limit$ is to be replaced by an integer which defines the maximal costs of a tour without restaurant. The external atom $\&longerThan[path, limit]()$ is true if the path encoded in the extension of $path$ is longer than $limit$, and false otherwise. Note that despite the rules (R3) and (R4), the choice between $haveLunch$ and $\overline{haveLunch}$ in rule (R2) is not redundant due to the use of the FLP-reduct. [11]

$$haveLunch \vee \overline{haveLunch} \leftarrow \tag{R2}$$

$$haveLunch \leftarrow \&longerThan[path, limit]() \tag{R3}$$

$$\overline{haveLunch} \leftarrow \text{not } \&longerThan[path, limit]() \tag{R4}$$

The following rules plan the tour using the external atom $\&path[L1, L2](X, Y, Cost, Type)$. Atoms of form $path(L1, L2, X, Y, Cost, Type)$ are used to encode the path (consisting of edges (X, Y) with costs $Cost$ and type $Type$) from $L1$ to $L2$. Since the same station may be visited multiple times, the end points $L1$ to $L2$ must be included to make the representation unique.

$$\begin{aligned} path(L1, L2, X, Y, Cost, Type) \leftarrow & seq(Nr, L1), seq(NrNext, L2), \\ & NrNext = Nr + 1, \\ & \&path[L1, L2](X, Y, Cost, Type) \end{aligned} \tag{R5}$$

Finally, we ensure that all pairs of sequent locations are connected. Otherwise, the program would still have answer sets (leaving some locations unconnected) if a location is not reachable.

$$\begin{aligned} pathExists(L1, L2) \leftarrow & seq(Nr, L1), seq(NrNext, L2), NrNext = Nr + 1, \\ & path(L1, L2, X, Y, Cost, Type) \\ \leftarrow & seq(Nr, L1), seq(NrNext, L2), NrNext = Nr + 1, \\ & \text{not } pathExists(L1, L2) \end{aligned} \tag{R6}$$

For the strongly safe version of the program, the full map material needs to be imported in advance. Under the assumption that it is encoded as a set of facts of kind $map(x, y, c, t)$ to represent connections between stations x and y with costs c and type t, adding the atom $map(X, Y, Cost, Type)$ to rule R5 makes the program strongly safe, but obviously blows up the grounding.

Appendix C.3.2. Pair and Group Route Planning

Compared to single route planning, the predicates $location$, seq, $haveSeq$, $haveLoc$ and $path$ are extended by an additional argument P, which is inserted at argument position 1. It allows for discriminating multiple persons. While we considered the special case of two persons in our benchmarks, the encoding is strictly more general and allows arbitrary many persons who need to be defined as facts of

[11] The FLP-reduct $\Pi^{\mathbf{A}}$ wrt. a model $\mathbf{A}$ must not contain constraints, and hence (R6) cannot check the existence of paths in $\Pi^{\mathbf{A}}$. But then, without further techniques, the FLP check could fail as $\Pi^{\mathbf{A}}$ might have a smaller model $\mathbf{A}'$ with disconnected locations. However, as rule (R1) enforces the same sequence of locations in $\Pi^{\mathbf{A}}$ and rule (R5) deterministically computes the shortest paths between two successive locations, this can only happen if $\mathbf{A}'$ and $\mathbf{A}$ encode different sets of locations; this requires that $\mathbf{A}$ and $\mathbf{A}'$ represent tours with and without a restaurant, respectively, or vice versa. But rule (R2) excludes this, as it ensures that the choice between $haveLunch$ and $\overline{haveLunch}$ is the same in both models; without rule (R2), neither $haveLunch$ nor $\overline{haveLunch}$ might be true in the $\mathbf{A}'$ as either rule (R3) or (R4) will be missing in $\Pi^{\mathbf{A}}$.

kind $person(p)$.

$$\begin{aligned}
seq(P,I,L) \vee \overline{seq}(P,I,L) \leftarrow & person(P), location(L), \#int(I) \\
\leftarrow & person(P), seq(P,I1,L), seq(P,I2,L), \\
& I1 \neq I2 \\
\leftarrow & person(P), seq(P,I,L1), seq(P,I,L1), \\
& L1 \neq L2 \\
haveSeq(P,L) \leftarrow & person(P), seq(P,I,L) \\
\leftarrow & person(P), location(P,L), \text{not } haveSeq(P,L) \\
haveLoc(P,I) \leftarrow & person(P), seq(P,I,L) \\
\leftarrow & person(P), seq(P,I,L), I1 < I, \#int(I1), \\
& \text{not } haveLoc(P,I1)
\end{aligned}$$

Computing the path and ensuring its existence is extended to multiple persons as follows.

$$\begin{aligned}
path(P,L1,L2,X,Y,Cost,Type) \leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& \&path[L1,L2](X,Y,Cost,Type) \\
pathExists(P,L1,L2) \leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& path(P,L1,L2,X,Y,Cost,Type) \\
\leftarrow & person(P), \\
& seq(P,Nr,L1), seq(P,NrNext,L2), \\
& NrNext = Nr+1, \\
& \text{not } pathExists(P,L1,L2)
\end{aligned}$$

As above, adding $map(X,Y,Cost,Type)$ to the former rule makes the program strongly safe.

The following rules choose a meeting point and include it in the tour of each person. It is assumed that the possible meeting locations are defined by facts of form $possibleMeeting(m)$.

$$\begin{aligned}
meeting(M) \vee \overline{meeting}(M) \leftarrow & possibleMeeting(M) \\
meetingChosen \leftarrow & meeting(M) \\
\leftarrow & meeting(M1), meeting(M2), M1 \neq M2 \\
\leftarrow & \text{not } meetingChosen \\
location(P,M) \leftarrow & person(P), meeting(M)
\end{aligned}$$

If the tour is longer than the given limit $limit$, then the meeting location should be a restaurant. The external atom $\&longerThanForPerson[path,p,limit]()$ is true if the path for person p encoded in the

extension of $path$ is longer than $limit$, and false otherwise.

$$
\begin{aligned}
haveLunch \vee \overline{haveLunch} \leftarrow & \\
haveLunch &\leftarrow person(P), \&longerThanForPerson[path, P, limit]() \\
\overline{haveLunch} &\leftarrow person(P), \text{not } \&longerThanForPerson[path, P, limit]() \\
&\leftarrow haveLunch, meeting(M), \text{not } possibleRestaurant(M) \\
&\leftarrow \overline{haveLunch}, meeting(M), possibleRestaurant(M)
\end{aligned}
$$

As for the encoding from Section Appendix C.3, the choice between $haveLunch$ and $\overline{haveLunch}$ is not redundant.