Lazy-Grounding for Answer Set Programs with External Source Access*

Thomas Eiter, Tobias Kaminski, and Antonius Weinzierl
Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, kaminski, weinzierl}@kr.tuwien.ac.at

Abstract

HEX-programs enrich the well-known Answer Set Programming (ASP) paradigm. In HEX, problems are solved using nonmonotonic logic programs with bidirectional access to external sources. ASP evaluation is traditionally based on grounding the input program first, but recent advances in lazy-grounding make the latter also interesting for HEX, as the grounding bottleneck of ASP may be avoided. We explore this issue and present a new evaluation algorithm for HEX-programs based on lazy-grounding solving for ASP. Nonmonotonic dependencies and value invention (i.e., import of new constants) from external sources make an efficient solution nontrivial. However, illustrative benchmarks show a clear advantage of the new algorithm for grounding-intense programs, which is a new perspective to make HEX more suitable for real-world application needs.

1 Introduction

HEX-programs [Eiter et al., 2005; 2016a] enrich answer set programs [Gelfond and Lifschitz, 1991] with the possibility to exchange information with external sources in a bidirectional way. This greatly increases the problem solving capacity of Answer Set Programming (ASP) [Brewka *et al.*, 2016], as in the plugin-architecture of HEX-solvers like DLVHEX, arbitrary sources can be accessed via an API style interface called *external atom*. For example, an external atom $\&prop[config](P)$ might be used in a configuration problem to retrieve from a property checker each property P of a candidate solution, given by selected components in the predicate $config$. External atoms have been fruitfully exploited for a wide range of applications (see [Erdem *et al.*, 2016; Eiter *et al.*, 2016d]).

Efficient evaluation of HEX-programs is challenging, due to the generic (sometimes black-box) nature of external atoms and *value invention*, i.e., the import of new constant symbols from the sources into the program. Advanced algorithms have been developed [Eiter *et al.*, 2016a] which roughly speaking rewrite a given HEX-program into an ordinary ASP program, solve the latter using an ASP solver, and finally check whether the obtained answer sets are compatible with the external sources.

By this approach, HEX-programs inherit the well-known grounding bottleneck of state-of-the-art ASP solving (as e.g. by CLINGO [Gebser *et al.*, 2011a]) which may show up in the grounding phase, i.e. during the computation of a propositional program equivalent to the input program, and can cause an exponential blowup. This makes ASP and likewise HEX incapable of solving a number of real-world problems with larger data volume. To mitigate this problem, several advanced optimization methods and techniques have been developed, cf. [Kaufmann *et al.*, 2016], but the grounded program can still be (too) large. For HEX-programs, grounding is due to external atoms an even bigger challenge, which has been tackled with sophisticated program decomposition and component grounding techniques [Eiter *et al.*, 2016a]. However, decomposition has a trade-off with efficient solving, and without decomposition exponentially many inputs to an external atom may have to be considered during grounding, e.g. for the external atom $\&prop[config](P)$ from above, if properties nonmonotonically depend on the input configuration. Hence, novel evaluation algorithms are an issue.

To overcome the grounding bottleneck of ASP, lazy-grounding algorithms were devised, that ground rules *on-the-fly* [Palù *et al.*, 2009; Lefèvre and Nicolas, 2009a; 2009b; Dao-Tran *et al.*, 2012; Lefèvre *et al.*, 2017]. In an interleaved grounding and solving process, only rules are grounded that are currently useful and thus space explosion is avoided. In this way, problems can be solved that traditional ASP solving cannot handle. Recent advances in lazy grounding, available in prototype solvers, suggest to explore this approach for evaluating HEX-programs. However, an extension to this setting is non-trivial, due to nonmonotonic dependencies of external atoms on absent information, and in particular due to unknown constants from value invention.

Our main contributions are briefly summarized as follows:

- we introduce a novel external source interface to incrementally extend a HEX-program grounding, where new output terms may appear *during solving* (Sec. 3);
- we give a novel evaluation algorithm for HEX-programs that exploits a lazy-grounding ASP solver (Sec. 4); and

*This research has been supported by the Austrian Science Fund (FWF) projects P27730 and W1255-N23.

- we show experimental results which confirm the benefit of the new algorithm on illustrative benchmarks (Sec. 5).

The unprecedented integration of lazy-grounding, external source evaluation and value invention is a new perspective to make HEX and ASP more suitable for real-world applications.

An extended version of this paper containing proofs of the technical results is available [Eiter *et al.*, 2017].

2 Preliminaries

We assume a finite set $\mathcal{P}$ of predicate symbols, a finite set $\mathcal{C}$ of constant symbols and a set $\mathcal{V}$ of variables. Atoms a are of the form $p(\mathbf{t})$ with $p \in \mathcal{P}$ and $\mathbf{t} = t_1, \ldots, t_n$ a list of terms $t_i \in \mathcal{C} \cup \mathcal{V}$; a is ground if each t_i is in $\mathcal{C}$. A (signed) literal is a positive or a negated ground atom $\mathbf{T}a$ (intuitively, a is true) or $\mathbf{F}a$ (a is false). A *nogood* is a set $\{L_1, \ldots, L_n\}$ of literals L_i of type $\mathbf{T}a$ or $\mathbf{F}a$. A *partial assignment* $\mathcal{A}$ over the Herbrand Base $\mathcal{HB}$ of all ground atoms is a set $\mathbf{A}$ of signed literals of kind $\mathbf{T}a$, $\mathbf{F}a$ and $\mathbf{U}a$ (intuitively, a is *unassigned*) such that for every $a \in \mathcal{HB}$, $|\mathbf{A} \cap \{\mathbf{T}a, \mathbf{F}a, \mathbf{U}a\}| = 1$. We call $\mathbf{A}$ *complete*, if no $\mathbf{U}a$ occurs in it.

HEX-programs [Eiter et al., 2005; 2016a] extend ordinary ASP programs by *external atoms* of the form $\&g[\mathbf{p}](\mathbf{t})$, which enable a bidirectional interaction of a program and external computation sources. Here $\&g$ is an external predicate name, $\mathbf{p} = p_1, \ldots, p_k$ is an *input list* of input parameters (predicate names wlog.), and $\mathbf{t} = t_1, \ldots, t_\ell \in \mathcal{C} \cup \mathcal{V}$ are *output terms*; we write $p \in \mathbf{p}$ if $p = p_i$ for some $1 \leq i \leq k$ and analogous for $t \in \mathbf{t}$. An external atom is ground, if $\mathbf{t}$ consists of constants.

We consider normal HEX-programs Π, i.e., sets of rules r

$$a \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$$

where a is an (ordinary, potentially absent) atom and each b_j is either an ordinary or an external atom. The *body* of r is $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$ and the positive body is $B^+(r) = \{b_1, \ldots, b_m\}$. A rule r is *safe*, if each variable occurring in r also occurs in $B^+(r)$. Every rule r in a HEX-program Π must be safe. Moreover, to ensure external atoms introduce only finitely many new constants, we assume Π is *liberal domain-expansion* (*lde*) safe. The notion of lde-safety allows to modularly combine syntactic and/or semantic safety criteria to guarantee that a HEX-program is finitely groundable (cf. [Eiter *et al.*, 2016b] for more details), and it is the most liberal safety notion that has been considered wrt. the HEX formalism. The grounding $grd(r)$ of r is the set of all possible rules $r\sigma$ that result from r by applying a *(ground) substitution* $\sigma : \mathcal{V} \to \mathcal{C}$; the grounding of program Π is $grd(\Pi) = \bigcup_{r \in \Pi} grd(r)$.

Semantics. We follow [Eiter *et al.*, 2016c] for semantics based on partial assignments. The semantics of a *ground external atom* $\&g[\mathbf{p}](\mathbf{c})$ with k input and ℓ output parameters wrt. a partial assignment $\mathbf{A}$ is given by a $1{+}k{+}\ell$-ary *three-valued oracle function* $f_{\&g}$ that is defined for all possible values of $\mathbf{A}$, $\mathbf{p}$ and $\mathbf{c}$. Thus, $\&g[\mathbf{p}](\mathbf{c})$ is true, false or unassigned relative to $\mathbf{A}$, if the value of $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ is $\mathbf{T}$, $\mathbf{F}$ or $\mathbf{U}$, respectively. Moreover, if $\mathbf{A}$ is complete, then $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) \neq \mathbf{U}$ holds and as usual, $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ depends only on the extension (in $\mathbf{A}$) of predicates $p \in \mathbf{p}$.

A partial assignment $\mathbf{A}$ satisfies (or models) a ground atom a, denoted $\mathbf{A} \models a$, if $\mathbf{T}a \in \mathbf{A}$; and it models a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$. Satisfaction of ordinary rules and ASP programs [Gelfond and Lifschitz, 1991] wrt. complete assignments is naturally extended to HEX-rules and programs. The answer sets of a ground HEX-program Π are defined as follows. Let the *FLP-reduct* of Π wrt. a complete assignment $\mathbf{A}$ be the set $f\Pi^{\mathbf{A}} = \{r \in \Pi \mid \mathbf{A} \models b, \text{ for all } b \in B(r)\}$ of all rules whose body is satisfied by $\mathbf{A}$, and let for partial assignments $\mathbf{A}_1$, $\mathbf{A}_2$ denote $\mathbf{A}_1 \leq \mathbf{A}_2$ that $\{\mathbf{T}a \in \mathbf{A}_1\} \subseteq \{\mathbf{T}a \in \mathbf{A}_2\}$. Then,

Definition 1. *A complete assignment* $\mathbf{A}$ *is an answer set of a ground* HEX*-program* Π*, if* $\mathbf{A}$ *is a* $\leq$*-minimal model of* $f\Pi^{\mathbf{A}}$*.*[1]

The answer sets of a non-ground HEX-program Π are those of $grd(\Pi)$.

Example 1. *Consider the program* $\Pi = \{\leftarrow \&size[p](0);\ p(X) \leftarrow d(X);\quad a \leftarrow \text{not } d(c);\quad d(c) \leftarrow \text{not } a\}$ *where the external atom* $\&size[p](Z)$ *computes the cardinality of* p*, i.e.,* $f_{\&size}(\mathbf{A}, p, Z) = \mathbf{U}$ *if* $\mathbf{U}p(X) \in \mathbf{A}$ *for some* X*,* $f_{\&size}(\mathbf{A}, p, Z) = \mathbf{T}$ *if* $|\{p(X) \mid \mathbf{T}p(X) \in \mathbf{A}\}| = Z$*, and* $f_{\&size}(\mathbf{A}, p, Z) = \mathbf{F}$ *otherwise. The single answer set of* Π *is* $\{d(c), p(c)\}$*, because it satisfies the first rule, which expresses that the extension of predicate* p *must not be* 0*.*

3 Evaluation of External Sources Based on Partial Groundings

Lazy-grounding means that the grounding $grd(\Pi)$ of a program Π is computed lazily, i.e., only ground rules deemed necessary are computed. In the following, let $\mathcal{G}_\Pi \subseteq \mathcal{HB}$ denote the set of all atoms occurring in the grounding of Π. Then, partial assignments in the case of lazy-grounding are given with respect to a set of ground atoms $\mathcal{A} \subseteq \mathcal{G}_\Pi \subseteq \mathcal{HB}$.

Definition 2. *A* partial assignment *over a set* $\mathcal{A} \subseteq \mathcal{HB}$ *of atoms is a set* $\mathbf{A}_\mathcal{A}$ *of signed literals* $\mathbf{T}a$*,* $\mathbf{F}a$*, and* $\mathbf{U}a$ *with* $a \in \mathcal{A}$ *s.t. for every* $a \in \mathcal{A}$*,* $|\mathbf{A} \cap \{\mathbf{T}a, \mathbf{F}a, \mathbf{U}a\}| = 1$*; it is* complete *(wrt.* $\mathcal{A}$*), if no* $\mathbf{U}a$ *occurs in it.*

For partial assignments $\mathbf{A}_\mathcal{A}, \mathbf{A}'_{\mathcal{A}'}$ we call $\mathbf{A}'_{\mathcal{A}'}$ an *extension* of $\mathbf{A}_\mathcal{A}$, denoted $\mathbf{A}'_{\mathcal{A}'} \succeq \mathbf{A}_\mathcal{A}$, if $\{\mathbf{T}a \in \mathbf{A}_\mathcal{A}\} \cup \{\mathbf{F}a \in \mathbf{A}_\mathcal{A}\} \subseteq \mathbf{A}'_{\mathcal{A}'}$ and $\mathcal{A} \subseteq \mathcal{A}'$, i.e., some atoms $a \in \mathcal{HB}$ not present in $\mathbf{A}_\mathcal{A}$ may be present in $\mathbf{A}'_{\mathcal{A}'}$ and some unassigned atoms in $\mathbf{A}_\mathcal{A}$ may be flipped to true or false.

Lazy-grounding ASP solving is founded in the notion of a *computation sequence* (cf. [Lefèvre and Nicolas, 2009a]), which is a monotonically growing sequence $(\mathbf{A}_0, \ldots, \mathbf{A}_n)$ of partial assignments such that whenever a (lazily grounded) rule of the input program fires at $\mathbf{A}_i$, it is guaranteed to fire in all later assignments $\mathbf{A}_j$, i.e., $0 \leq i \leq j \leq n$. Furthermore, a grounded rule r only fires in $\mathbf{A}_i$ if it is *applicable*, which means that its positive body $B^+(r)$ is completely true, i.e., $B^+(r) \subseteq \mathbf{A}_i$. Given that ordinary ASP rules are safe, the whole negative body of a ground rule is known once it fires, such that focusing on positive rule bodies is sufficient for completeness of solving under lazy-grounding. For example, if the rule $p(a) \leftarrow q(a), \text{not } r(a)$ fires in a computation

[1] For ordinary Π, these are Gelfond & Lifschitz's answer sets.

sequence at $\mathbf{A}_i$, then $\{\mathbf{T}q(a), \mathbf{F}r(a)\} \subseteq \mathbf{A}_i$ and monotonicity guarantees the same for all later $\mathbf{A}_j$, $i \leq j$.

For (ground) external atoms, it is much harder to ensure that once an output becomes true, it will stay true even for larger input. Eiter et al. [2016c] introduced the notion of assignment-monotonic oracle-function to ensure this in the case where the grounding of a program is generated prior to solving. Formally, a three-valued oracle function $f_{\&g}$ is *assignment-monotonic*, if $f_{\&g}(\mathbf{A}_{\mathcal{G}_\Pi}, \mathbf{p}, \mathbf{c}) = X$, $X \in \{\mathbf{T}, \mathbf{F}\}$, implies $f_{\&g}(\mathbf{A}'_{\mathcal{G}_\Pi}, \mathbf{p}, \mathbf{c}) = X$ for all partial assignments $\mathbf{A}'_{\mathcal{G}_\Pi} \succeq \mathbf{A}_{\mathcal{G}_\Pi}$. Intuitively, this guarantees that the oracle-function cannot treat $\mathbf{T}a \notin \mathbf{A}_{\mathcal{G}_\Pi}$ as being equivalent to $\mathbf{F}a \in \mathbf{A}_{\mathcal{G}_\Pi}$. Observe that all atoms $a \in \mathcal{HB} \setminus \mathcal{G}_\Pi$ must be false in every answer set, simply because there is no rule in the grounding of Π whose head is a. As all assignments for the latter program are over $\mathcal{G}_\Pi$, it is thus guaranteed that each atom a relevant for the external source also occurs in $\mathbf{A}_{\mathcal{G}_\Pi}$, either as $\mathbf{T}a$, $\mathbf{F}a$, or $\mathbf{U}a$. For an $\mathbf{A}_\mathcal{A}$ with $\mathcal{A} \subset \mathcal{G}_\Pi$ and $a \in \mathcal{G}_\Pi \setminus \mathcal{A}$, however, an oracle-function treats a as false, i.e., $\mathbf{A}_\mathcal{A}$ equals $(\mathbf{A} \cup \{\mathbf{F}a\})_{\mathcal{A}\cup\{a\}}$ from the perspective of any oracle-function, even for assignment-monotonic ones.

As oracle-functions are black-boxes, HEX cannot determine the relevant input of an assignment-monotonic oracle-function, i.e.: if $f_{\&g}(\mathbf{A}_\mathcal{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ for an assignment $\mathbf{A}_\mathcal{A}$, then, without knowing the set of ground atoms $\mathcal{G}_\Pi$ that occur in the grounding of a HEX-program Π, we cannot determine whether some atom $a \in \mathcal{G}_\Pi \setminus \mathcal{A}$ exists s.t. $f_{\&g}((\mathbf{A} \cup \{\mathbf{U}a\})_{\mathcal{A}\cup\{a\}}, \mathbf{p}, \mathbf{c}) \neq \mathbf{T}$.

Example 2. *Reconsider Π from Example 1. Assume that $\mathbf{T}a$ is guessed, before grounding the second rule. Then, $\&size[p](0)$ is true under $\mathbf{A}_\mathcal{A} = \{\mathbf{F}d(c), \mathbf{T}a\}$ with $\mathcal{A} = \{d(c), a\}$. However, guessing $\mathbf{T}d(c)$ and grounding the second rule for $X = c$, yields $\mathbf{A}'_{\mathcal{A}'} = \{\mathbf{T}d(c), \mathbf{F}a, \mathbf{T}p(c)\}$ with $\mathcal{A}' = \mathcal{A} \cup \{p(c)\}$. Thus, $f_{\&size}(\mathbf{A}'_{\mathcal{A}'}, p, 0) = \mathbf{F}$, which shows that $\mathbf{A}_\mathcal{A}$ was insufficient for deciding the value of $\&size[p](0)$.*

To address this issue, we must ensure that external atoms are evaluated only with assignments being complete for their input predicates. Intuitively, an assignment is input-complete for a program, if it contains all relevant input to every external atom; a ground atom that occurs in no answer set and is not an input-predicate of any external atom is irrelevant for the truth of external atoms and thus ignored. For a partial assignment $\mathbf{A}_\mathcal{A}$, let its completion wrt. $\mathcal{HB}$ be $\hat{\mathbf{A}}_\mathcal{A} = (\mathbf{A} \cup \{\mathbf{F}a \mid a \in \mathcal{HB} \setminus \mathcal{A}\})_{\mathcal{HB}}$. Then input-completeness is as follows:

Definition 3. *A partial assignment $\mathbf{A}_\mathcal{A}$ is* input-complete *for an external atom $\&g[\mathbf{p}](\mathbf{c})$ occurring in a ground* HEX*-program Π, if $f_{\&g}(\hat{\mathbf{A}}_\mathcal{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ only if every answer set $\mathbf{A}'_{\mathcal{HB}}$ of Π s.t. $\mathbf{A}'_{\mathcal{HB}} \succeq \mathbf{A}_{rel}$ fulfills $f_{\&g}(\mathbf{A}'_{\mathcal{HB}}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$, where $\mathbf{A}_{rel} = \{Xa \in \mathbf{A}_\mathcal{A} \mid a$ has predicate $p \in \mathbf{p}\}$ is the relevant input to $\&g$. A partial assignment $\mathbf{A}_\mathcal{A}$ is* input-complete *wrt. a* HEX*-program Π, if it is* input-complete *for each external atom $\&g[\mathbf{p}](\mathbf{c})$ occurring in $grd(\Pi)$.*

Without restriction to answer sets, in Example 2 no input-complete assignment $\mathbf{A}_\mathcal{A}$ on $\mathcal{A} \subset \mathcal{HB}$ for $\&size[p](0)$ would exist with $f_{\&size}(\mathbf{A}, p, 0) = \mathbf{T}$, as infinitely many constants could be added to $p's$ extension if the grounding is extended.

Example 3 (cont'd)**.** *There is no partial assignment defined over $\{d(c), a\}$ that is input-complete for Π, but the partial assignment $\{\mathbf{T}d(c), \mathbf{F}a, \mathbf{T}p(c)\}$ is input-complete for Π.*

In the following, we characterize syntactically sets of ground atoms that are sufficient for input-completeness, resorting to nonmonotonic inputs to external atoms; monotonic input cannot cause issues with atoms $a \in \mathcal{HB} \setminus \mathcal{A}$ that do not yet occur in an assignment $\mathbf{A}_\mathcal{A}$. Formally, an input predicate $p \in \mathbf{p}$ of an external atom $\&g[\mathbf{p}](\mathbf{c})$ is *monotonic*, if $f_{\&g}(\mathbf{A}_\mathcal{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ implies $f_{\&g}(\mathbf{A}'_{\mathcal{A}'}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ for any $\mathbf{A}'_{\mathcal{A}'} \succeq \mathbf{A}_\mathcal{A}$ that augments a given $\mathbf{A}_\mathcal{A}$ only by atoms with predicate p (cf. [Eiter *et al.*, 2016b]). Then the following proposition can be shown.

Proposition 1. *Let $\&g[\mathbf{p}](\mathbf{c})$ be an external atom occurring in a ground* HEX*-program Π and let each $p \in \mathbf{p}$ be monotonic. Then, any partial assignment $\mathbf{A}_\mathcal{A}$ is input-complete for $\&g[\mathbf{p}](\mathbf{c})$.*

Many external atoms have only monotonic input (e.g., string concatenation, dl-atoms, and the RDF atom in the DLVHEX library). Regarding nonmonotonic input, the set $\mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$ contains all predicates occurring as not monotonic input to some external atom in program Π, i.e., $\mathbf{p}_{\overline{\mathbf{m}}}(\Pi) = \{p \in \mathcal{P} \mid \&g[\mathbf{p}](\mathbf{c})$ occurs in $\Pi, p \in \mathbf{p}, p$ is not monotonic$\}$.

Definition 4. *For a* HEX*-program Π, a (finite) set $\mathcal{A} \subseteq \mathcal{HB}$ of ground atoms is an* input-safe domain *of Π, if it contains each atom $p(\mathbf{X})$ where $p \in \mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$ and $\mathbf{T}p(\mathbf{X}) \in \mathbf{A}$ for some answer set $\mathbf{A}$ of Π.*

Proposition 2. *A partial assignment $\mathbf{A}_\mathcal{A}$ is input-complete wrt. a* HEX*-program Π, if $\mathcal{A}$ is an input-safe domain of Π.*

Notably, Definitions 4 and 3 are in semantic terms, relying on answer sets of a given program. We capture those notions syntactically by the *relevant grounding* $G^\infty_{rel_\Pi(p,\emptyset)}(\emptyset)$ for a not monotonic input predicate p of some external atom. To this end, we compute a partial grounding of a given HEX-program by only considering the subset of (non-ground) rules that is relevant for obtaining all ground instances of p.

Definition 5. *Given a predicate name p, a* HEX*-program Π, and a set S of predicate names, the* relevant rules of Π wrt. *p and S are $rel_\Pi(p, S) = \bigcup_{r \in \Pi_p} \{r\} \cup \{r' \in rel_\Pi(p', S') \mid p' \in \mathcal{P}B^+(r), p' \notin S'\}$ where $S' = S \cup \{p\}$, Π_p contains all rules of Π where p occurs in the head, and $\mathcal{P}B^+(r)$ consists of all predicate names that occur in $B^+(r)$ either as ordinary atom predicate or as an input to an external atom. Furthermore, the* relevant rules of Π wrt. *p are defined by $rel_\Pi(p, \emptyset)$.*

In order to obtain all instances of a predicate possibly true in some answer set of a program, we employ the following monotone grounding operator G_Π from [Eiter *et al.*, 2016b]:

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

where $r\theta$ is the ground instance of r under variable substitution $\theta : \mathcal{V} \to \mathcal{C}$, $\hat{I} = \{\mathbf{T}a \mid a \in I\} \cup \{\mathbf{F}a \mid a \in \mathcal{HB} \setminus I\}$, and $A(\Pi')$ is the set of all ordinary ground atoms occurring in Π'. The least fixpoint of $G^\infty_\Pi(\emptyset)$ contains all atoms that are true in some answer set of Π.

Example 4 (cont'd). *We have $G^{\infty}_{rel_\Pi(p,\emptyset)}(\emptyset)$=$\{d(c) \leftarrow \text{not } a;$ $p(c) \leftarrow d(c)\}$, and a partial assignment $\mathbf{A}_\mathcal{A}$ is input-complete wrt. Π, if $\mathcal{A} \supseteq \{p(c)\}$. Note that in general, $G^{\infty}_{rel_\Pi(p,\emptyset)}(\emptyset)$ would not contain further rules on which p does not depend.*

The rules in $G^{\infty}_{rel_\Pi(p,\emptyset)}(\emptyset)$ contain all ground instances over p that occur in the grounding $grd(\Pi)$ of Π. Since all atoms that occur in some answer set of Π also occur in $grd(\Pi)$, the relevant grounding thus indicates all such atoms.

Proposition 3. *Let Π be a* HEX*-program. If $\mathbf{T}p(\mathbf{X}) \in \mathbf{A}$, for some answer set $\mathbf{A}$ of Π, then $p(\mathbf{X})$ occurs in $G^{\infty}_{rel_\Pi(p,\emptyset)}(\emptyset)$.*

Considering the relevant grounding of those predicates p that occur in Π as a not monotonic input to some external atom, i.e., considering the relevant grounding of all $p \in \mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$, we can obtain an input-safe domain of Π and thus obtain input-complete partial assignments. More formally,

Theorem 1. *A partial assignment $\mathbf{A}_\mathcal{A}$ is input-complete wrt. a* HEX*-program Π if for all $p \in \mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$ it holds that $p(\mathbf{X}) \in \mathcal{A}$ whenever $p(\mathbf{X})$ occurs in $G^{\infty}_{rel_\Pi(p,\emptyset)}(\emptyset)$.*

4 Lazy-Grounding HEX-Evaluation Algorithm

In this section, we present the new evaluation algorithm that interleaves the steps taken by a lazy-grounding solver with the evaluation of external sources, which incrementally introduce new output constants into the program.

Given an input-safe domain $\mathcal{A}$, the algorithm operates on top of a transformation from a HEX-program Π to an ordinary logic program $\alpha(\Pi, \mathcal{A})$, such that an ordinary lazy-grounding solver for ASP can be employed as a host to incrementally ground the rules in $\alpha(\Pi, \mathcal{A})$. Moreover, via a novel interface to external sources, lazy-grounding may import input-output relations over external atoms in form of additional rules.

In the program transformation $\alpha(\Pi, \mathcal{A})$, external atoms are replaced by ordinary atoms, and the program is extended by rules that allow to explicitly derive the negative extension of a given input-safe domain of Π.

Definition 6. *Given a* HEX*-program Π and an input-safe domain $\mathcal{A}$ of Π, the* ordinary program $\alpha(\Pi, \mathcal{A})$ *results from Π by replacing each (non-ground) external atom $\&g[\mathbf{p}](\mathbf{t})$ with an ordinary (non-ground)* replacement atom $e_{\&g[\mathbf{p}]}(\mathbf{t})$*, and by adding for each $p \in \mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$ the rule*

$$\overline{p}(\mathbf{X}) \leftarrow p_d(\mathbf{X}), \text{not } p(\mathbf{X}),$$

and for each $p(\mathbf{X}) \in \mathcal{A}$ a domain fact $p_d(\mathbf{X}) \leftarrow$.

Without loss of generality, we assume that atoms of form $e_{\&g[\mathbf{p}]}(\mathbf{t})$, $\overline{p}(\mathbf{X})$ and $p_d(\mathbf{X})$ do not occur in Π, i.e. they are fresh atoms.

The purpose of the program extension is twofold. On the one hand, it ensures that each atom $p(\mathbf{X})$ in $\mathcal{A}$ is either derived to be true or *explicitly* false (via $\overline{p}(\mathbf{X})$), so that in the end, nonmonotonic external atoms are always evaluated under complete assignments. On the other hand, enabling guessing the values of atoms in $\mathcal{A}$ early during the solving process potentially allows that outputs of nonmonotonic external atoms are derived earlier during search.

Example 5 (cont'd). *Reconsider Π from Example 1. Given $\mathcal{A} = \{p(c)\}$, the first rule is replaced by $\leftarrow e_{\&size[p]}(0)$ in $\alpha(\Pi, \mathcal{A})$, and we add $\{\overline{p}(\mathbf{X}) \leftarrow p_d(\mathbf{X}), \text{not } p(\mathbf{X}); p_d(c) \leftarrow\}$.*

For interleaving the solving algorithm with the evaluation of external sources, we first define a means for constructing a partial assignment which is input-complete wrt. the given program, from an assignment $\mathbf{A}$ derived by the ASP solver.

Values of atoms that are true wrt. $\mathbf{A}$ are taken directly and false atoms are based on atoms of form $\overline{p}(\mathbf{X})$, which represent falsity of $p(\mathbf{X})$ according to Definition 6. All other atoms in the domain are considered unassigned. Formally:

Definition 7. *Given a partial assignment $\mathbf{A}$ and a domain $\mathcal{A}$, the corresponding* external input assignment *is the set*

$$\begin{aligned} i(\mathbf{A}, \mathcal{A}) = &\{\mathbf{T}p(\mathbf{X}) \in \mathbf{A}\} \cup \{\mathbf{F}p(\mathbf{X}) \mid \mathbf{T}\overline{p}(\mathbf{X}) \in \mathbf{A}\} \cup \\ &\{\mathbf{U}p(\mathbf{X}) \mid p(\mathbf{X}) \in \mathcal{A}, \mathbf{T}\overline{p}(\mathbf{X}) \notin \mathbf{A}, \mathbf{T}p(\mathbf{X}) \notin \mathbf{A}\}. \end{aligned}$$

Intuitively, the construction of an *external input assignment* from a given solver assignment ensures that atoms which have not been assigned a truth value during solving but which are in the given input-safe domain, are explicitly declared to be unassigned whenever an external source is queried. This is necessary because the external source requires information about all atoms which can potentially become true in the search later on, in order to only yield outputs that remain correct when the grounding is extended.

Note that if there is no atom $p(\mathbf{X})$ s.t. $\mathbf{T}\overline{p}(\mathbf{X}) \in \mathbf{A}$ and $\mathbf{T}p(\mathbf{X}) \in \mathbf{A}$, then $i(\mathbf{A}, \mathcal{A})$ is a partial assignment. We assume that the previous holds for all external input assignments used in the following.

Example 6. *Consider the partial assignment $\mathbf{A} = \{\mathbf{T}a, \mathbf{U}b, \mathbf{F}c, \mathbf{F}d, \mathbf{T}\overline{e}\}$ and domain $\mathcal{A} = \{d, e, f\}$. The corresponding external input assignment is $i(\mathbf{A}, \mathcal{A}) = \{\mathbf{T}a, \mathbf{U}d, \mathbf{F}e, \mathbf{U}f\}$. Observe that $i(\mathbf{A}, \mathcal{A})$ only depends on the $\mathbf{T}$-part of $\mathbf{A}$ and that $i(\mathbf{A}, \mathcal{A})$ is an assignment.*

The external source interface amounts to a function that yields rules representing the corresponding input-output relations of external atoms. These rules are added to the input program processed by the solver. Accordingly, whenever an output value is obtained based on a solver assignment, a rule is generated that implies the ground replacement atom representing the respective output value relative to the current assignment of the relevant input atoms.

Definition 8. *Given $\&g[\mathbf{p}]$, a partial assignment $\mathbf{A}$ and a domain $\mathcal{A}$, the* external evaluation function η *yields*

$$\begin{aligned} &\eta(\&g[\mathbf{p}], i(\mathbf{A}, \mathcal{A})) = \\ &\quad \{e_{\&g[\mathbf{p}]}(\mathbf{c}) \leftarrow \mathbf{B}_{\mathbf{A},\mathbf{p}} \mid f_{\&g}(i(\mathbf{A}, \mathcal{A}), \mathbf{p}, \mathbf{c}) = \mathbf{T}\}, \end{aligned}$$

where $\mathbf{B}_{\mathbf{A},\mathbf{p}} = \{p'(\mathbf{X}) \mid \mathbf{T}p'(\mathbf{X}) \in \mathbf{A}, p' \in \{\overline{p}, p\}, p \in \mathbf{p}\}$ is a rule body corresponding to the external atom's input.

We denote all possible such evaluations by $\eta(\Pi) = \{r \mid \exists \mathbf{A} \text{ s.t. } r \in \eta(\&g[\mathbf{p}], i(\mathbf{A}, \mathcal{HB})), \&g[\mathbf{p}] \text{ occurs in } \Pi\}$.

Example 7. *Consider Π from Example 1 again. For input-safe domain $\mathcal{A} = \{p(c)\}$ and $\mathbf{A} = \{\mathbf{T}d(c), \mathbf{F}a\}$, the external input assignment $i(\mathbf{A}, \mathcal{A}) = \{\mathbf{T}d(c), \mathbf{U}p(c)\}$ is input-complete for Π, and $\eta(\&size[p], i(\mathbf{A}, \mathcal{A}))$=$\emptyset$. For*

the partial assignment $\mathbf{A}'=\{\mathbf{T}d(c), \mathbf{F}a, \mathbf{T}\overline{p}(c)\}$*, we obtain* $i(\mathbf{A}', \mathcal{A})=\{\mathbf{T}d(c), \mathbf{F}p(c)\}$ *and* $\eta(\&size[p], i(\mathbf{A}', \mathcal{A})) = \{e_{\&size[p]}(0) \leftarrow \overline{p}(c)\}$.

Algorithm 1 allows us to evaluate a HEX-program using lazy grounding. It is based on the lazy-grounding ASP solver ALPHA, which incorporates ideas from OMIGA [Dao-Tran *et al.*, 2012; Weinzierl, 2017]. The algorithm combines *conflict-driven nogood-learning* (*CDNL*) search [Gebser *et al.*, 2012] with lazy-grounding. CDNL applies techniques from *SAT* solving to ASP, by translating a ground program into a set of nogoods, corresponding to clauses in SAT solving, and running a *DPLL-style* search algorithm. In every iteration of the CDNL search procedure, deterministic consequences are propagated first, and in case some nogood is violated, a conflict nogood is added to the nogood store to avoid running into the same conflict again and backjumping is performed. Whenever no deterministic assignments are possible during CDNL search, but the solver assignment is still incomplete, an unassigned atom is guessed to be true or false.

Algorithm 1 receives as input an ordinary program constructed according to Definition 6 from a HEX-program Π and an input-safe domain of Π. In practice, we obtain an input-safe domain based on Definition 4 and Proposition 3, by computing $G^{\infty}_{rel_{\Pi}(p,\emptyset)}(\emptyset)$ for all $p \in \mathbf{p}_{\overline{\mathbf{m}}}(\Pi)$. Note that requesting an input-safe domain of the input program is not a severe restriction on the class of programs that our approach can handle, as an input-safe domain can be obtained for any HEX-program. After initializing, Algorithm 1 explores the search space in one loop, where the first step at each iteration is propagation from the currently known nogoods ∇ and the current assignment $\mathbf{A}$ in (a). If some nogood δ is violated, in (b), a new nogood is learned from the conflict and backjumping is done. If propagation at (a) derived new assignments, lazy-grounding of the input program is done in (c).

In (d), all external sources are queried, employing external evaluation functions and external input assignments. Note that, at this point, it cannot be the case that $\mathbf{T}\overline{p}(\mathbf{X}) \in \mathbf{A}$ and $\mathbf{T}p(\mathbf{X}) \in \mathbf{A}$ both hold for any atom $p(\mathbf{X})$, because atoms of the form $\overline{p}(\mathbf{X})$ are only defined by the rules added in the program transformation of Definition 6 and those rules only fire if $\mathbf{T}p(\mathbf{X}) \notin \mathbf{A}$. In (e), guessing is done, which is different from ordinary CDNL-based guessing: due to the lazy-grounding, not all atoms may be guessed upon but only those corresponding to ground instances of *applicable* rules (cf. Sec. 3). Heuristics may be employed for selecting good guesses. Upon reaching (f), the iterations of lazy-grounding, guessing, and propagating do not yield any more information, i.e., a fixpoint has been reached. In order to complete the assignment (wrt. known atoms), all atoms being unassigned in $\mathbf{A}$ are assigned to *false*. In (g), the assignment is tested for only containing true or false assignments. This is necessary, because the ALPHA solver internally works with *must-be-true* as additional truth value for increased efficiency. For evaluation of external atoms, *must-be-true* is treated as *true*. If the check succeeds, then the current assignment is an answer set of the HEX-program and recorded as such.[2] If the check fails, some *must-be-true* remained and the current assignment is not an answer set, hence backtracking occurs in (h).

[2] In the implementation, false atoms of an answer set $\hat{\mathbf{A}}$ are not stored explicitly.

Algorithm 1: Lazy-Grounding HEX-Evaluation

Input: The ordinary program $\alpha(\Pi, \mathcal{A})$ corresponding to a HEX-program Π, given input-safe domain $\mathcal{A}$ of Π
Output: All answer sets $AS(\alpha(\Pi, \mathcal{A}) \cup \eta(\Pi))$ of $\alpha(\Pi, \mathcal{A}) \cup \eta(\Pi)$

```
AS ← ∅ // found answer sets
A ← {Ua | a ∈ A} // all known atoms unassigned
∇ ← ∅ // dynamic nogood storage
Run lazy grounder (obtain initial nogoods ∇ from facts)
while search space not exhausted do
    (A, ∇) ← Propagation(A, ∇)                                   (a)
    if some nogood δ ∈ ∇ violated by A then                      (b)
        analyze conflict, add learned nogood to ∇, backjump
    else if A changed then                                       (c)
        run lazy grounder wrt. A and extend ∇
    else if external sources not queried for current A then      (d)
        extend ∇ wrt. η(&g[p], i(A, A)) for each &g[p] in Π
    else if there are guesses left then                          (e)
        select a guess
    else if exists Ua ∈ A then                                   (f)
        replace each Ua by Fa in A
    else if all atoms assigned T or F in A then                  (g)
        AS ← AS ∪ {Â}
        add enumeration nogood and backtrack
    else                                                         (h)
        backtrack
return AS
```

Algorithm 1 returns the answer sets of the program transformation together with all rules encoding possibly relevant input-output relations of external atoms:

Proposition 4. *For* HEX*-program* Π *and input-safe domain* $\mathcal{A}$ *of* Π*, Algorithm 1 yields the answer sets of* $\alpha(\Pi, \mathcal{A}) \cup \eta(\Pi)$.

Given a HEX-program Π and an input-safe domain $\mathcal{A}$ of Π, if Algorithm 1 returns an answer set of $\alpha(\Pi, \mathcal{A}) \cup \eta(\Pi)$, we obtain an answer set of Π by using for ordinary atoms occurring in Π the respective truth value and by setting all other atoms in $\mathcal{HB}$ to false. Observe that the resulting assignment maps all atoms of the form $e_{\&g[\mathbf{p}]}(\mathbf{t})$, $\overline{p}(\mathbf{X})$ or $p_d(\mathbf{X})$ to false as they do not occur in Π. Moreover, each answer set of Π is obtained this way.

Theorem 2. *For a* HEX*-program* Π *and an input-safe domain* $\mathcal{A}$ *of* Π*, the answer sets returned by Algorithm 1 correspond exactly to the answer sets of* Π.

To show this result, we rely on the correctness and completeness of ordinary lazy-grounding ASP solving (cf. Theorem 1 in [Weinzierl, 2017]), which needs to be extended to also take external evaluations into account. As external atoms are evaluated under input-complete assignments only, it is ensured that input-output relations returned by the external evaluation function at any point during search are not contradicted by later external evaluations. Since no cyclic dependencies involving external atoms are allowed, their evaluation only depends on a subprogram that does not contain the respective external atom itself. Because of this, the *Splitting*

size	**split.**	**monol.**	**alpha**	**s.** (n=1)	**m.** (n=1)	**a.** (n=1)
4	0.16 (0)	0.16 (0)	1.22 (0)	0.13 (0)	0.13 (0)	1.13 (0)
6	0.80 (0)	0.43 (0)	1.68 (0)	0.61 (0)	0.31 (0)	1.43 (0)
8	7.62 (0)	4.09 (0)	2.48 (0)	5.95 (0)	3.76 (0)	1.98 (0)
10	67.52 (0)	88.54 (0)	4.82 (0)	55.31 (0)	85.11 (0)	3.43 (0)
12	300.00 (10)	189.79 (6)	9.15 (0)	295.73 (9)	158.15 (5)	5.47 (0)
14	300.00 (10)	300.00 (10)	17.37 (0)	300.00 (10)	300.00 (10)	9.52 (0)
16	300.00 (10)	300.00 (10)	27.79 (0)	300.00 (10)	300.00 (10)	14.93 (0)
18	300.00 (10)	300.00 (10)	54.00 (0)	300.00 (10)	300.00 (10)	25.44 (0)
20	300.00 (10)	288.67 (9)	132.08 (0)	300.00 (10)	288.27 (9)	50.67 (0)
22	300.00 (10)	300.00 (10)	225.47 (0)	300.00 (10)	300.00 (10)	66.91 (0)
24	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	300.00 (10)	119.20 (0)

Table 1: Consistent Preferences Results

size	**split.**	**monol.**	**alpha**	**s.** (n=1)	**m.** (n=1)	**a.** (n=1)
10	2.06 (0)	0.40 (0)	1.62 (0)	1.05 (0)	0.31 (0)	1.38 (0)
12	8.71 (0)	1.05 (0)	1.87 (0)	6.67 (0)	0.88 (0)	1.44 (0)
14	39.29 (0)	4.04 (0)	4.56 (0)	22.20 (0)	3.17 (0)	1.83 (0)
16	200.54 (0)	14.83 (0)	3.91 (0)	126.96 (0)	13.89 (0)	2.87 (0)
18	300.00 (10)	57.20 (0)	7.29 (0)	249.94 (8)	55.49 (0)	3.83 (0)
20	300.00 (10)	300.00 (10)	128.01 (4)	233.52 (7)	300.00 (10)	7.42 (0)
22	300.00 (10)	300.00 (10)	133.87 (4)	190.75 (6)	300.00 (10)	5.60 (0)
24	300.00 (10)	300.00 (10)	214.84 (7)	257.36 (8)	300.00 (10)	37.51 (1)
26	300.00 (10)	300.00 (10)	300.00 (10)	212.42 (7)	300.00 (10)	37.96 (1)
28	300.00 (10)	300.00 (10)	243.17 (8)	109.73 (3)	300.00 (10)	6.67 (0)
30	300.00 (10)	300.00 (10)	272.53 (9)	240.28 (8)	300.00 (10)	38.66 (1)

Table 2: Generic Configuration Results

$$
\begin{aligned}
sel(X) &\leftarrow \text{not } n\_sel(X), person(X).\\
n\_sel(X) &\leftarrow \text{not } sel(X), person(X).\\
preferred(X,Y) &\leftarrow \&prefs[sel](X,Y).\\
preferred(X,Y) &\leftarrow preferred(X,Z), preferred(Z,Y).\\
&\leftarrow preferred(X,X).
\end{aligned}
$$

Figure 1: Consistent Preferences Rules

Theorem from [Eiter *et al.*, 2016a] can be applied for proving correctness of Algorithm 1. Completeness intuitively follows from completeness wrt. ordinary programs and the fact that the truth values computed for replacement atoms by Algorithm 1 coincide with the outputs of the respective oracle functions, given identical assignments to ordinary atoms.

5 Implementation and Evaluation

To evaluate the performance of the new HEX-algorithm, we have integrated the ALPHA lazy-grounding solver, which is freely available, with the DLVHEX reasoner [Redl, 2016]. The two components communicate via the interface in Section 4, where DLVHEX bridges to external sources and handles program decomposition, while ALPHA acts as ordinary ASP solver. The program transformation in Definition 6 allows us to omit the usual guessing program [Eiter *et al.*, 2016a] in the implementation. For comparison, we used DLVHEX with GRINGO and CLASP [Gebser *et al.*, 2011b] as backends.

The tests were performed on a Linux machine with two 12-core AMD Opteron 6176 SE CPUs and 128 GB RAM. The timeout for each run was 300 secs and the memory limit 12 GB. We used the *HTCondor* load distribution system[3] to ensure a stable environment that minimizes runtime variations between runs on the same problem instance.

Average runtimes of 10 instances per size (resp. 30 for benchmark #3) are reported in secs for computing all answer sets and one answer set (n=1); timeouts are in parentheses.

Benchmark Configurations. We compared three configurations: • **splitting**: the program is decomposed into independently groundable components, which are processed by an ordinary solver; • **monolithic**: a grounding of the complete program is generated, and an ordinary solver is run; • **alpha**: no program splitting happens and the novel algorithm using ALPHA is applied.

We expected **alpha** to perform better (i) than **splitting**, in case many guesses violate constraints and decomposition splits the latter from the guessing part; and (ii) better than **monolithic**, if generating the respective grounding before solving needs a lot of resources due to nonmonotonic external atoms. Furthermore, we hypothesized that lazy-grounding is beneficial if many constants are imported by value invention but only few of them occur simultaneously in an answer set.

Consistent Preferences. This benchmark considers a problem where many new constants are imported by an external atom based on a guess, which obstructs intelligent grounding techniques. A HEX-program selects a subset P' of a pool P of persons p and checks if the union of individual preferences $pref(p, I) \subseteq I \times I$ over items I is consistent (i.e. acyclic). The answer sets of the rules from Figure 1 plus the facts $\{person(p) \mid p \in P\}$ correspond to all $P' \subseteq P$ where this holds. The item set I and the preferences $pref(p, I)$ are not part of the HEX-program, but imported via an external atom $\&prefs[sel](X, Y)$ for the selected persons. Its oracle function evaluates to true wrt. a partial assignment $\mathbf{A}$ and output (i, i'), if some p fulfills $\mathbf{T}sel(p) \in \mathbf{A}$ and $(i, i') \in pref(p, I)$; to false, if $\mathbf{F}sel(p) \in \mathbf{A}$ holds for all p s.t. $(i, i') \in pref(p, I)$; and to unassigned otherwise. Thus, the input parameter sel is monotonic, but evaluating the external atom under its maximal extension may cause a large amount of constants to be imported into the program.

We ran tests for randomly generated instances with $n = 4, \ldots, 24$ persons and $2n$ items, where each individual preference (i, i') uniformly occurs with 5% probability (Table 1).

Generic Configuration. Using ASP for configuration has a long tradition (e.g. [Soininen *et al.*, 2001] considered product configuration and more recently [Gebser *et al.*, 2015] the railway domain). Here, we address a generic setting[4] that is likely to occur in real-world scenarios as those mentioned.

A *configuration* is a subset C' of a set C of components, which has an associated set $m(C') \subseteq P$ of properties from a set P. An *admissible* C' must fulfill a set R of requirements (e.g. costumer demands) of the form $(R^+, R^-) \in 2^P \times 2^P$, which means that $R^+ \subseteq m(C')$ and $R^- \cap m(C') = \emptyset$ holds.

In the HEX-program, we guess a configuration $C' \subseteq C$ in a predicate $config$ and compute its properties with an external atom $\&prop[config](P)$. As $config$ is a nonmonotonic input parameter, traditional grounding must evaluate it for all possible inputs. For a partial configuration C' and property p, the oracle function is true, if $p \in m(C'')$ for every $C'' \supseteq C'$; false, if $p \notin m(C'')$ for every $C'' \supseteq C'$; and unassigned else.

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

[4] We exploit the benchmark implementation from https://github.com/hexhex/core/tree/master/benchmarks/genericmapping.

size	**split**.	**monol**.	**alpha**	**s**. (n=1)	**m**. (n=1)	**a**. (n=1)
5	0.18 (0)	0.38 (0)	1.39 (0)	0.14 (0)	0.37 (0)	1.13 (0)
10	2.64 (0)	6.33 (0)	9.30 (0)	1.25 (0)	5.89 (0)	1.61 (0)
15	54.54 (1)	224.03 (2)	56.14 (3)	36.38 (0)	218.92 (1)	2.10 (0)
20	273.95 (23)	300.00 (30)	96.49 (9)	262.49 (21)	300.00 (30)	3.40 (0)
25	300.00 (30)	300.00 (30)	111.42 (11)	300.00 (30)	300.00 (30)	8.65 (0)
30	300.00 (30)	300.00 (30)	102.14 (10)	300.00 (30)	300.00 (30)	15.02 (0)
35	300.00 (30)	300.00 (30)	83.31 (8)	300.00 (30)	300.00 (30)	23.92 (0)
40	300.00 (30)	300.00 (30)	55.88 (5)	300.00 (30)	300.00 (30)	27.74 (0)
45	300.00 (30)	300.00 (30)	88.35 (8)	300.00 (30)	300.00 (30)	63.07 (2)
50	300.00 (30)	300.00 (30)	81.79 (7)	300.00 (30)	300.00 (30)	80.34 (6)

Table 3: Failure Diagnosis Results

We tested random instances with $n = 10, \ldots, 30$ components, $n/5+1$ properties and up to $2n$ requirements, where a property occurs in a requirement with probability 30% and depends on a component with probability 10%; each property and requirement was added with probability 50 % (Table 2).

Failure Diagnosis. Another classical use-case of logic programs is abduction-based diagnosis [Kakas *et al.*, 1992]. Suppose possible causes of a machine failure should be given from certain (Boolean) measurement values that are only partially available. The task is to compute, respecting the open measurement values, all necessary causes that entail the measurement values. In that, information about combinations of failure causes that can be excluded is available.

This problem can be modeled[5] using sets M and M' of known resp. unknown measurements, a set H of possible causes, a logic program P relating measurements and possible causes, and a set $\mathcal{C}$ of constraints that exclude specific combinations of causes. We want to compute the intersection $\mathcal{D}$ of all possible diagnoses $D \subseteq H$ wrt. measurement values $\mathcal{M} = M \cup M''$ with $M'' \subseteq M'$, s.t. $\mathcal{D} \not\subseteq C$ for all $C \in \mathcal{C}$. For this, we guess $M'' \subseteq M'$ in the HEX-program and employ a nonmonotonic external atom $\&diags[P, \mathcal{M}](\mathcal{D})$ to obtain the necessary failure causes.

In the tests, we used random instances with $n = 5, \ldots, 50$ measurement values, each available at 20 % (e.g. due to unfinished measurements), and up to $2n$ constraints to exclude combinations of causes, where each occurs in a constraint with probability 30 %. The results are shown in Table 3.

Findings. In all three benchmarks, lazy-grounding (setting **alpha**) exhibits a significant advantage in runtime over **splitting** and **monolithic**. This matches our hypotheses (i) and (ii) as under **monolithic**, the external atom must be grounded for exponentially many input combinations in the last two benchmarks, and under **splitting**, the search space cannot be pruned effectively due to the separation of guesses and constraints. We observe that **splitting** outperforms **monolithic** for Failure Diagnosis because computing the diagnoses is resource-intense and must be executed for every input during the grounding step. Here, this outweighs the costs related to less search space pruning in **splitting**. Note that in general, **alpha** finds the first answer set much faster than the other configurations, and notably, was very fast when no answer set exists. However, in computing all answer sets it often timed out when instances have a large number of solutions. Hence, with increasing instance size, the number of instances with many solutions has a stronger impact on the average runtimes for **alpha**. Methodologically, this suggests to restrict the solution space of a problem by adding further constraints when using lazy-grounding.

Somewhat surprising, **alpha** outperformed **monolithic** for Consistent Preferences, despite feasible grounding for the instance sizes. Our analysis explains this by the large number of guesses usually added for evaluation of external atoms in HEX during grounding. Hence, considerably more time is required for solving. In contrast, no additional guesses must be introduced in our program transformation (Def. 6) as here the external atom only has monotonic input parameters and new constants can be imported on-the-fly.

The benchmark instances and all results are available at http://www.kr.tuwien.ac.at/research/projects/inthex/lazyhex.

[5]We adopt the setting implemented at https://github.com/hexhex/core/tree/master/benchmarks/diagnosis.

6 Discussion and Conclusion

Related Work. Our work builds on partial evaluation of external atoms [Eiter *et al.*, 2016c], and on the recently developed ALPHA solver [Weinzierl, 2017]. It is the first time that lazy-grounding has been considered for the HEX framework. We are not aware of similar approaches for related systems, such as CLINGO, which however, supports no value invention based on the respective answer set as HEX. Lazy-grounding ASP solvers like ASPERIX [Lefèvre and Nicolas, 2009b], GASP [Palù *et al.*, 2009], and OMIGA [Dao-Tran *et al.*, 2012] could in theory be employed, but likely result in worse performance, as they are not based on CDNL-search.

Summary. We have introduced a new algorithm for HEX-programs that interleaves external evaluation plus value invention with lazy-grounding ASP solving. It employs a tailored interface between the two components with a program transformation based on input-safe domains and a novel evaluation function that adds rules for input-output relations over external atoms to the program. Monotonic external atoms are directly evaluated on partial groundings, with the benefit that no additional guesses are needed. Due to the black-box nature of external atoms, computing a restricted grounding wrt. not monotonic inputs is necessary; however, this usually involves only a small subset of the complete grounding.

The benchmark results of our prototype implementation are promising, and show the potential of the new algorithm based on the ALPHA solver. In the special setting of HEX-programs, lazy-grounding exhibits a significant benefit already for relatively small instances since program splits and guessing for monotonic external atoms can be avoided.

Outlook. We plan to integrate an advanced minimality-check into the algorithm, so that cycles over nonmonotonic external atoms can be treated. Moreover, an evaluation mixing full grounding and lazy-grounding, each for a different component of a given program, may increase overall performance.

References

[Brewka *et al.*, 2016] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming: An introduction to the special issue. *AI Magazine*, 37(3):5–6, 2016.

[Dao-Tran *et al.*, 2012] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga : An open minded grounding on-the-fly answer set solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence, JELIA 2012*, volume 7519 of *LNCS*, pages 480–483. Springer, 2012.

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

[Eiter *et al.*, 2014] Thomas Eiter, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller. Efficient HEX-program evaluation based on unfounded sets. *J. of Artificial Intelligence Research*, 49:269–321, 2014.

[Eiter *et al.*, 2016a] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *TPLP*, 16(4):418–464, 2016.

[Eiter *et al.*, 2016b] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. Domain expansion for asp-programs with external sources. *Artificial Intelligence*, 233:84–121, 2016.

[Eiter *et al.*, 2016c] Thomas Eiter, Tobias Kaminski, Christoph Redl, and Antonius Weinzierl. Exploiting partial assignments for efficient evaluation of answer set programs with external source access. In Subbarao Kambhampati, editor, *International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 1058–1065. IJCAI/AAAI Press, 2016.

[Eiter *et al.*, 2016d] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In Christoph Beierle, Gerhard Brewka, and Matthias Thimm, editors, *Computational Models of Rationality, Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday*, pages 150–174. College Publications, 2016.

[Eiter *et al.*, 2017] Thomas Eiter, Tobias Kaminski, and Antonius Weinzierl. Lazy-grounding for answer set programs with external source access. Technical Report INFSYS RR-1843-17-01, Institut für Informationssysteme, Technische Universität Wien, Vienna, Austria, June 2017.

[Erdem *et al.*, 2016] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016.

[Gebser *et al.*, 2011a] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[Gebser *et al.*, 2011b] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[Gebser *et al.*, 2012] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, August 2012.

[Gebser *et al.*, 2015] Martin Gebser, Anna Ryabokon, and Gottfried Schenner. Combining heuristics for configuration problems using answer set programming. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning, LPNMR 2015. Proceedings*, volume 9345 of *LNCS*, pages 384–397. Springer, 2015.

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

[Kakas *et al.*, 1992] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[Kaufmann *et al.*, 2016] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI Magazine*, 37(3):25–32, 2016.

[Lefèvre and Nicolas, 2009a] Claire Lefèvre and Pascal Nicolas. A First Order Forward Chaining Approach for Answer Set Computing. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, LPNMR 2009*, volume 5753 of *LNCS*, pages 196–208. Springer, 2009.

[Lefèvre and Nicolas, 2009b] Claire Lefèvre and Pascal Nicolas. The First Version of a New ASP Solver: ASPeRiX. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming an Nonmonotonic Reasoning, LPNMR 2009*, volume 5753 of *LNCS*, pages 522–527. Springer, 2009.

[Lefèvre *et al.*, 2017] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *TPLP*, pages 1–45, January 2017.

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

[Redl, 2016] Christoph Redl. The DLVHEX system for knowledge representation: Recent advances (system description). *CoRR*, abs/1607.08864, 2016.

[Soininen *et al.*, 2001] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming, ASP'01 Workshop 2001*, 2001.

[Weinzierl, 2017] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In *Logic Programming and Nonmonotonic Reasoning, LPNMR 2017*, LNCS. Springer, 2017. To appear.