Pruning External Minimality Checking for ASP Using Semantic Dependencies*

Thomas Eiter and Tobias Kaminski

Institute of Logic and Computation, TU Wien, Vienna, Austria
{eiter,kaminski}@kr.tuwien.ac.at

Abstract. HEX-programs integrate external computations in ASP. For HEX-evaluation, an external (e)-minimality check is required to prevent cyclic justifications via external sources. As the check is a bottleneck in practice, syntactic information about atom dependencies has been used previously to detect when the check can be avoided. However, the approach largely overapproximates the real dependencies due to the black-box nature of external sources. We show how the dependencies can be approximated more closely by exploiting semantic information, which significantly increases pruning of e-minimality checking. Moreover, we analyze checking and optimization of semantic dependency information. An empirical evaluation exhibits a clear benefit of this approach.

1 Introduction

Answer Set Programming (*ASP*) [10] is a popular approach for declarative problem solving. The HEX-*formalism* [5] extends ASP to address the increasing need for integrating external computation sources. It enables a bidirectional exchange with arbitrary sources via so-called *external atoms*, and has been employed in many areas ranging from *Semantic Web* applications to robot planning [5]. For instance, an external atom $\&concat[X, Y](Z)$ can be used to concatenate strings in a rule $fullname(X) \leftarrow \&concat[X, Y](Z), firstname(X), lastname(Y)$. External atoms may also have predicate input, e.g. in the rule $closeCity(X) \leftarrow \&closeTo[city](X), location(X)$, where the external atom outputs all cities located close to cities in the extension of the predicate *city*.

For HEX-evaluation, advanced reasoning algorithms are required since external atoms must be considered in all solving phases. A notable difference to ordinary ASP is that an *external* (*e-*)*minimality check* is needed to avoid unfounded support by external atoms. For example, if the locations for the above rule are *osaka*, *kobe*, *bratislava* and *vienna*, and the rule $city(X) \leftarrow closeCity(X)$ as well as the fact $city(osaka)$ are added, only the atom $city(kobe)$ should be contained in an answer set in addition. Even though Bratislava and Vienna are located close to each other, the atoms $city(bratislava)$ and $city(vienna)$ can only cyclically

* This research has been supported by the FWF-projects P27730 and W1255-N23. The final authenticated publication is available online at https://doi.org/10.1007/978-3-030-20528-7_24.

support each other via the two rules and the external atom. The e-minimality check of HEX eliminates spurious answer sets containing the latter two atoms.

On the one hand, performing the e-minimality check efficiently is highly non-trivial as it is co-NP-complete already for ground *Horn* programs with polynomial external atoms [4]. On the other hand, if the rule $city(X) \leftarrow closeCity(X)$ is not added above, cyclic support via the external atom can be ruled out independent from the external semantics. Based on this observation, a syntactic criterion was presented in [4] for deciding whether the e-minimality check can be skipped for a program, which often results in significant speedups.

Alternatively, if the external atom $\&closeWest[city](X)$ is used in the example (only retrieving cities close *to the west* of input cities), cyclic support can also be excluded. This cannot be detected by a syntactic criterion, such that the e-minimality check needs to be performed in any case by the previous approach. Moreover, applying a semantic criterion is challenging, as before, external atoms have largely been considered as black-boxes that conceal semantic dependencies.

Skipping e-minimality checks in more cases is of special interest as it can often result in drastic speedups. For this reason, we develop a new approach for pruning e-minimality checking that also exploits semantic dependencies. It relies on additional information about *input-output* (*io-*)*dependencies* of external atoms, which may be provided by a user, or even generated automatically. Hidden io-dependencies are common in applications involving recursive processing, e.g. over external graphs or *Semantic Web* data. At this, supplied dependency information can be incomplete and added flexibly. The overall goal is to increase the efficiency of ASP programs with external atoms to promote their practical applicability.

After preliminaries in Section 2, we present our contributions as follows:

- In Section 3.1, we provide a novel formalization of io-dependencies that encode semantic dependency information, and we show under which condition they can safely be used for pruning the e-minimality check.
- In Section 3.2, we state theoretical properties crucial for checking and optimizing io-dependencies, and show when the associated costs can be reduced.
- In Section 4, we present an experimental evaluation using illustrative benchmark problems that confirms the advantage of exploiting io-dependencies.

Our new approach not only applies to HEX, but may also be employed analogously for other approaches that integrate external sources into ASP, such as CLINGO [8], if external cyclic support is not desired. Proofs and benchmark data can be found at `www.kr.tuwien.ac.at/research/projects/inthex/dep-pruning`.

2 Preliminaries

We assume disjoint sets $\mathcal{P}$, $\mathcal{C}$, $\mathcal{X}$ and $\mathcal{V}$ of predicates, constants, external predicates (prefixed with '&') and variables, respectively. Each $p \in \mathcal{P}$ has fixed arity $ar(p)$, and each $\&g \in \mathcal{X}$ has fixed input and output arity $ar_I(\&g)$ and $ar_O(\&g)$, respectively. An atom is of the form $p(\vec{t})$, where $p \in \mathcal{P}$, $\vec{t} = t_1, \ldots, t_\ell \in \mathcal{C} \cup \mathcal{V}$. A (signed) literal is a positive or a negative ground atom $\mathbf{T}p(\vec{c})$ or $\mathbf{F}p(\vec{c})$. An

assignment $\mathbf{A}$ over a set $\mathcal{A}$ of ground atoms is a set of literals s.t. for each $a \in \mathcal{A}$, either $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$, where $\mathbf{A}(a) = \mathbf{T}$ if $\mathbf{T}a \in \mathbf{A}$, and $\mathbf{A}(a) = \mathbf{F}$ otherwise.

HEX-Programs. HEX-*programs* extend answer set programs with *external atoms* in rule bodies (cf. [5] for more details).

Syntax. An *external atom* is of form $\&g[\vec{X}](\vec{Y})$, where $\&g \in \mathcal{X}$, $\vec{X} = X_1, \ldots, X_k$, with $k = ar_I(\&g)$, are input parameters (variables or predicates w.l.o.g.) and $\vec{Y} = Y_1, \ldots, Y_l$, with $l = ar_O(\&g)$, are output terms. An external atom is *ground* if $\vec{X} = X_1, \ldots, X_k$ are predicates and $\vec{Y} = Y_1, \ldots, Y_l$ are constants. Given a ground external atom $\&g[\vec{X}](\vec{Y})$, we call $\&g[\vec{X}]$ a *ground external (ge-)predicate*.

Definition 1 (HEX-Program). *A* HEX-*program Π is a set of rules of the form $a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$, where each a_i, $1 \leq i \leq k$, is an atom and each b_j, $1 \leq j \leq n$, is either an ordinary atom or an external atom.*

Given a rule r, $H(r) = \{a_1, \ldots, a_k\}$ is its *head*, $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$ its *body*, and $B^+(r) = \{b_1, \ldots, b_m\}$ resp. $B^-(r) = \{b_{m+1}, \ldots, b_n\}$.

Semantics. As safety conditions allow to compute equivalent finite groundings of HEX-programs, in the following we assume assignments are over the set $A(\Pi)$ of atoms that occur in a ground program Π at hand. Moreover, definitions are implicitly parameterized with the according finite vocabulary. Following [6], the semantics of a ground external atom $\&g[\vec{p}](\vec{c})$, wrt. an assignment $\mathbf{A}$, is given by a $1+ar_I(\&g)+ar_O(\&g)$-ary *two-valued (Boolean) oracle function* $f_{\&g}$ defined for all possible values of $\mathbf{A}$, $\vec{p}$ and $\vec{c}$ s.t. $\&g[\vec{p}](\vec{c})$ is true (informally, $\vec{c}$ is an output of $\&g$ for input $\vec{p}$) relative to $\mathbf{A}$ iff $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = \mathbf{T}$. As usual, we assume that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ depends only on the restriction of $\mathbf{A}$ to $\vec{p}$. Satisfaction of ASP rules and programs [10] is extended to HEX-rules and programs in the obvious way.

The answer sets of a HEX-program Π are defined as follows. Let the *FLP-reduct* [7] of Π wrt. an 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 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 2 (Answer Set). *An assignment $\mathbf{A}$ is an answer set of a* HEX-*program Π, if $\mathbf{A}$ is a $\leq$-minimal model of $f\Pi^{\mathbf{A}}$.*

Example 1. Consider $\Pi = \{p \leftarrow \&id[p]()\}$, where $\&id[p]()$ is true iff p is true. Then, Π has the answer set $\mathbf{A}_1 = \emptyset$; indeed it is a $\leq$-minimal model of $f\Pi^{\mathbf{A}_1} = \emptyset$.

Evaluation. A HEX-program Π can be transformed to an ordinary program by replacing each external atom $\&g[\vec{p}](\vec{c})$ in Π by an ordinary *replacement atom* $e_{\&g[\vec{p}]}(\vec{c})$, and by adding a rule $e_{\&g[\vec{p}]}(\vec{c}) \vee ne_{\&g[\vec{p}]}(\vec{c}) \leftarrow$ that guesses its evaluation. An ordinary ASP solver can then be employed to compute the answer sets of the resulting *guessing program* $\hat{\Pi}$, where each answer set $\hat{\mathbf{A}}$ is a *candidate model*. If all truth values for atoms $e_{\&g[\vec{p}]}(\vec{c})$ correspond to $f_{\&g}(\hat{\mathbf{A}}, \vec{p}, \vec{c})$, $\hat{\mathbf{A}}$ is a *compatible set*. Still, the projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ to $A(\Pi)$ is not always an answer set due to the possibility of cyclic support via external atoms.

Example 2 (cont'd). The guessing program $\hat{\Pi}=\{p \leftarrow e_{\&id[p]}(); e_{\&id[p]} \vee ne_{\&id[p]} \leftarrow\}$ has the answer sets $\hat{\mathbf{A}}_1 = \emptyset$ and $\hat{\mathbf{A}}_2 = \{\mathbf{T}p, \mathbf{T}e_{\&id[p]}\}$. Here, $\mathbf{A}_1$ is a $\leq$-minimal model of $f\Pi^{\mathbf{A}_1} = \emptyset$, but $\mathbf{A}_2$ not of $f\Pi^{\mathbf{A}_2} = \Pi$ since $\emptyset \leq \mathbf{A}_2$ is a smaller model.

Consequently, an e-minimality check wrt. $f\Pi^{\mathbf{A}}$ is needed for finding answer sets of HEX-programs. A direct way to ensure minimality of the projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ for a HEX-program Π wrt. $f\Pi^{\mathbf{A}}$ consists in explicitly constructing $f\Pi^{\mathbf{A}}$ and checking that it has no model $\mathbf{A}'$ s.t. $\mathbf{A}' \leq \mathbf{A}$.

3 Pruning the External Minimality Check

Since an answer set $\hat{\mathbf{A}}$ of a guessing program $\hat{\Pi}$ must be a minimal model of the FLP-reduct $f\hat{\Pi}^{\hat{\mathbf{A}}}$, an e-minimality check is under certain conditions redundant. The criterion in [4] for deciding its necessity relies on an atom dependency graph induced by the HEX-program. Informally, an e-minimality check is only needed for programs that allow cyclic support via external atoms, which can be checked efficiently. For instance, the program $\Pi_1 = \{\, p \leftarrow \&id[p]() \,\}$ allows cyclic support for the atom p via $\&id[p]()$, while this is not the case for $\Pi_2 = \{\, p \leftarrow \&id[q](); q \leftarrow r; r \leftarrow q \,\}$, where the truth value of $\&id[q]()$ is independent of the value of p. If cyclic support via external atoms can be ruled out as for Π_2, the e-minimality check can be skipped for a program, potentially avoiding to invest many resources into a redundant check. Note, however, that a minimality check is still needed for computing the answer sets of $\hat{\Pi}$.

In this section, we introduce a new technique for skipping the e-minimality check wrt. a wider class of programs than previous approaches. More precisely, given Π, we present a new sufficient[1] criterion for deciding if every projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ for $\hat{\Pi}$ is an answer set of Π. The criterion exploits that output values of external atoms often do not depend on the complete extensions of their input predicates, which can be determined given additional information concerning dependencies between the inputs and outputs of external atoms.

3.1 Dependency Graph Pruning

We start by defining so-called *io-dependencies*, which specify that certain outputs of external atoms only depend on specific argument values of their inputs. For instance, whether a city c is in the output of $\&closeWest[city](X)$ from Section 1 only depends on cities c' that are located close to the east of c. Hence, the truth value of $\&closeWest[city](kobe)$ clearly only depends on the atom $city(osaka)$, and we want to encode that *kobe* as first output of $\&closeWest[city](X)$ only depends on the element *osaka* as first argument of the first input predicate *city*.

Definition 3 (Io-Dependency). *An* io-dependency *for a ge-predicate* $\&g[\vec{p}]$ *is a tuple* $\delta = \langle i, j : J, k : e \rangle$ *where* $1 \leq i \leq ar_I(\&g)$, $1 \leq j \leq ar(p_i)$, $1 \leq k \leq ar_O(\&g)$, $J \subseteq \mathcal{C}$ *and* $e \in \mathcal{C}$. *The set of all* δ *for* $\&g[\vec{p}]$ *is denoted by* $dep(\&g[\vec{p}])$.

[1] Deciding the sufficient *and necessary* criterion is Π_2^p-complete for polynomial-time decidable external atoms and thus ill-suited for our aim to improve performance.

In the sequel, io-dependencies will be used to constrain the possible dependencies between inputs and outputs of external atoms $\&g[\vec{p}](\vec{c})$. Intuitively, an io-dependency $\langle i, j : J, k : e\rangle$ states that if constant e occurs as the k^{th} output of $\&g[\vec{p}](\vec{c})$, then only those input predicates at position i are relevant for its evaluation where the j^{th} argument matches some $e' \in J$. Thus, the io-dependency $\delta = \langle 1, 1 : \{osaka\}, 1 : kobe\rangle$ could be specified for the example above. Io-dependencies induce atom sets relevant for evaluating respective external atoms:

Definition 4 (Compliant Atoms). *A ground ordinary atom $p_i(\vec{d})$, with $\vec{d} = d_1, ..., d_l$, is* compliant *with a set $D \subseteq dep(\&g[\vec{p}])$ of io-dependencies for a ground external atom $\&g[\vec{p}](\vec{c})$ if $d_j \in J$ for all $\langle i, j : J, k : e\rangle \in D$ with $e = c_k$. The set of all atoms compliant with D for $\&g[\vec{p}](\vec{c})$ is denoted by $comp(D, \&g[\vec{p}](\vec{c}))$.*

For our example, we obtain $comp(\{\delta\}, \&closeWest[city](kobe)) = \{city(osaka)\}$. The semantics of external atoms is related to io-dependencies as follows.

Definition 5 (Faithfulness). *A set $D \subseteq dep(\&g[\vec{p}])$ is* faithful *if for any assignments $\mathbf{A}, \mathbf{A}'$ and ground external atom $\&g[\vec{p}](\vec{c})$, either $\mathbf{A}(p_i(\vec{d})) \neq \mathbf{A}'(p_i(\vec{d}))$ for some $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ or $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.*

Thus, io-dependencies $D \subseteq dep(\&g[\vec{p}])$ constrain the set of atoms that potentially impact the evaluation of $\&g[\vec{p}](\vec{c})$, i.e. if D is faithful, changing only truth values of atoms $p_i(\vec{d}) \notin comp(D, \&g[\vec{p}](\vec{c}))$ has no effect on the value of $\&g[\vec{p}](\vec{c})$.

In the following, we denote by $D(\&g[\vec{p}]) \subseteq dep(\&g[\vec{p}])$ a set of io-dependencies specified for $\&g[\vec{p}]$. By default, we assume that $D(\&g[\vec{p}])$ is empty, but it can be utilized to supply additional dependency information. To ensure correctness of an algorithm that skips e-minimality checks based on $D(\&g[\vec{p}])$, it is important that $D(\&g[\vec{p}])$ is faithful; and we assume in the following that this is the case. Simultaneously, the goal is to approximate the real dependencies between atoms as close as possible for maximal performance gains. Note that while an extensional specification of $D(\&g[\vec{p}])$ might be very verbose, they can often also be specified more concisely in an intensional manner, as in the following example.

Example 3. Consider $\&setDiff[dom, set](c)$, which is true for $c \in \mathcal{C}$ and assignment $\mathbf{A}$ iff $\{\mathbf{T}dom(c), \mathbf{F}set(c)\} \subseteq \mathbf{A}$. Thus, the presence of an output value c only depends on atoms with predicate *dom* or *set* that have c as first argument. Hence, $D(\&setDiff[dom, set]) = \{\langle 1, 1{:}\{c\}, 1{:}c\rangle, \langle 2, 1{:}\{c\}, 1{:}c\rangle \mid c \in \mathcal{C}\}$ is faithful.

We now introduce a notion of atom dependency in HEX-programs that accounts for io-dependencies and generalizes the corresponding notion from [4].

Definition 6 (Atom Dependency). *Given a ground* HEX*-program Π, a set $D(\&g[\vec{p}])$ for each $\&g[\vec{p}]$ in Π, and ordinary ground atoms $p(\vec{d})$ and $q(\vec{e})$, we say*

- *$q(\vec{e})$* depends on *$p(\vec{d})$, denoted $q(\vec{e}) \rightarrow_d p(\vec{d})$ if for some rule $r \in \Pi$ it holds that $q(\vec{e}) \in H(r)$ and $p(\vec{d}) \in B^+(r)$; and*
- *$q(\vec{e})$* depends externally on *$p(\vec{d})$, denoted $q(\vec{e}) \rightarrow_e p(\vec{d})$ if some rule $r \in \Pi$ and some external atom $\&g[\vec{p}](\vec{c}) \in B^+(r) \cup B^-(r)$ with $p \in \vec{p}$ exist such that $q(\vec{e}) \in H(r)$ and $p(\vec{d}) \in comp(D(\&g[\vec{p}]), \&g[\vec{p}](\vec{c}))$.*

Note that Definition 6 generalizes the corresponding one from [4] in that an external dependency is only added if the specified io-dependencies are satisfied. The definitions coincide if $D(\&g[\vec{p}]) = \emptyset$ for all ge-predicates $\&g[\vec{p}]$ in Π.

Example 4. Consider $\&suc[node](n)$, which evaluates to true wrt. an assignment $\mathbf{A}$ and an external directed graph $\mathcal{G} = (V, E)$ iff $n' \to n \in E$ for some node n' s.t. $\mathbf{T}node(n) \in \mathbf{A}$. It is utilized in the following HEX-program Π:

$$node(a).\ \ node(X) \leftarrow\ \ \&suc[node](X).$$

Intuitively, the program computes all nodes reachable from node a via the edges in $\mathcal{G}$. If the external graph has nodes $V = \{a, b, c, d\}$ and directed edges $E = \{a \to b, a \to c, c \to d, e \to d\}$, the grounding of Π produced by the grounding algorithm of the HEX-program solver DLVHEX contains the following rules (omitting facts):

$node(b) \leftarrow \&suc[node](b).\ \ node(c) \leftarrow \&suc[node](c).\ \ node(d) \leftarrow \&suc[node](d).$

Without specifying io-dependencies for $\&suc[node]$, it holds, e.g., that $node(a) \to_e node(b)$ and $node(b) \to_e node(a)$. However, we can specify $D(\&suc[node]) = \big\{\langle 1, 1 : \{c_1 \mid c_1 \to c_2 \in E\}, 1 : c_2\rangle \mid c_2 \in \mathcal{C}\big\}$, exploiting that the presence of output nodes only depends on input nodes to which they are successors. In this case, $node(a) \to_e node(b)$ does not hold according to Definition 6 as $b \to a \notin E$.

We are now ready to introduce the atom dependency graph for a given program Π. From this graph, a property of Π can be derived which is subsequently employed to decide the necessity of the e-minimality check wrt. Π.

Definition 7 (Dependency Graph). *Given a ground* HEX*-program Π, the* dependency graph $\mathcal{G}_\Pi^{dep} = (V, E)$ *has the vertices $V = A(\Pi)$ and directed edges $E = \to_d \cup \to_e$; Π has an* e-cycle*, if $\mathcal{G}_\Pi^{dep}$ has a cycle with an edge $\to_e$.*

While the inverse of $\to_d$ was additionally included in $\mathcal{G}_\Pi^{dep}$ by Eiter et al. [4], we improve their results by showing that our more general definition suffices. Moreover, the following result differs from the previous result for e-minimality check skipping [4] in that it is based on our generalized definition of external dependencies. Consequently, it can be applied to a larger class of HEX-programs.

Theorem 1. *If a ground* HEX*-program Π contains no e-cycle, then every projection $\mathbf{A}$ of a compatible set $\hat{\mathbf{A}}$ for $\hat{\Pi}$ is an answer set of Π.*

Example 5 (cont'd). Figure 1 shows the dependency graphs for Π from Example 4, with and without specified io-dependencies. The full dependency graph has an e-cycle, but the pruned graph does not. Hence, Π does not require e-minimality checks (cf. Theorem 1), but this can only be detected using the pruned graph.

As a result, we obtain a flexible means for increasing the efficiency of evaluating a class of HEX-programs where the e-minimality check is performed due to an overapproximation of the real dependencies between atoms.

Fig. 1. Full and pruned dependency graph for Π from Example 4 (all arrows are "$\rightarrow_e$").

3.2 Properties of Faithful IO-Dependencies

We now consider checking, generating and optimizing io-dependencies.

Informally, given $D_1, D_2 \subseteq dep(\&g[\vec{p}])$, D_1 is better than D_2 if it induces less compliant atoms. We thus say that D_1 *tightens* D_2, denoted $D_1 \leq D_2$, if $comp(D_1, \&g[\vec{p}](\vec{c})) \subseteq comp(D_2, \&g[\vec{p}](\vec{c}))$ holds for all tuples $\vec{c}$. We call D_1 *tight* if no D_2 strictly tightens D_1, i.e., $D_2 \leq D_1$ but $D_1 \not\leq D_2$; furthermore D_1 and D_2 are *equally tight*, denoted $D_1 \equiv D_2$, if $D_1 \leq D_2$ and $D_2 \leq D_1$. We then have:

Proposition 1. *Suppose $D_1, D_2 \subseteq dep(\&g[\vec{p}])$ are such that $D_1 \leq D_2$. If D_1 is faithful, then D_2 is also faithful.*

As a consequence, faithfulness is anti-monotonic wrt. set-inclusion, and it is monotonic wrt. adding subsumed io-dependencies, where $\delta = \langle i, j : J, k : e\rangle$ *subsumes* $\delta' = \langle i, j : J', k : e\rangle$, if $J \subseteq J'$ holds.

Corollary 1. *If $D \subseteq dep(\&g[\vec{p}])$ is faithful, then (i) each $D' \subseteq D$ is faithful and (ii) each $D' = D \cup D''$ where each $\delta'' \in D''$ is subsumed by some $\delta \in D$ is faithful.*

Consequently, we can tighten a faithful set D by sequentially dropping constants c from io-dependencies $\delta = \langle i, j : J, k : e\rangle$ in D, i.e., check whether $D \cup \delta'$ for $\delta' = \langle i, j : J \setminus \{c\}, k : e\rangle$ is faithful and if so, replace D with $(D \setminus \{\delta\}) \cup \{\delta'\}$.

We can simplify D by exploiting the following equivalences; let $\delta^*(i, j, k{:}e) = \langle i, j : \mathcal{C}, k : e\rangle$ for any possible i, j, and $k : e$.

Proposition 2. *For $D \subseteq dep(\&g[\vec{p}])$ and $\langle i, j : J, k : e\rangle \in dep(\&g[\vec{p}])$, we have (i) $D \equiv D \cup \{\delta^*(i, j, k{:}e)\} \equiv D \setminus \{\delta^*(i, j, k{:}e)\}$, and (ii) for any $\delta = \langle i, j : J, k : e\rangle$, $\delta' = \langle i, j : J', k : e\rangle \in D$ that $D \equiv D \cup \{\langle i, j : J \cap J', k : e\rangle\}$.*

That is, $\delta^*(i, j, k{:}e)$ is like a tautology, and we can replace all dependencies for i, j and $k : e$ in D by one which contains the intersection of all their J-sets. We thus can *normalize* D into $nf(D)$ such that for each i, j, and $k : e$ exactly one io-dependency occurs, and then start tightening. We then obtain:

Proposition 3. *Given a faithful $D \subseteq dep(\&g[\vec{p}])$, exhaustive tightening of $nf(D)$ results in a tight faithful D'.*

The set $D = \emptyset$ is trivially faithful, and $nf(\emptyset)$ consists of all $\delta^*(i, j, k{:}c)$; thus even without user input, a tight faithful set D' for $\&g[\vec{p}]$ is constructible. Moreover, semantically faithful sets of compliant atoms have the intersection property.

Proposition 4. *If $D_1, D_2 \subseteq dep(\&g[\vec{p}])$ are faithful, then $D_1 \cup D_2$ is faithful, and for every $\vec{c}$, $comp(D_1 \cup D_2, \&g[\vec{p}](\vec{c})) = comp(D_1, \&g[\vec{p}](\vec{c})) \cap comp(D_2, \&g[\vec{p}](\vec{c}))$.*

Consequently, every ge-predicate has a semantically unique tight set of faithful io-dependencies. However, syntactically, different tight faithful sets may exist.

Example 6. Consider a ge-predicate $\&g[p]$ which is true for output (a, b) wrt. an assignment $\mathbf{A}$ iff $\mathbf{T}p(c) \in \mathbf{A}$, and false for all other output tuples. Then $\{\langle 1, 1 : \{c\}, 1 : a\rangle\}$ and $\{\langle 1, 1 : \{c\}, 2 : b\rangle\}$ are faithful, and both are tight.

To check faithfulness of a set $D \subseteq dep(\&g[\vec{p}])$, formally the oracle function $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c})$ must be evaluated for all evaluations of predicates $p \in \vec{p}$ and output tuples $\vec{c}$, which naively is often not feasible in practice.

Example 7. Reconsider $\&suc[node](X)$ from Example 3. To check faithfulness of the specified io-dependencies wrt. output a, the oracle function needs to be evaluated under all possible assignments to atoms with predicate *node*.

In the worst case, this cannot be avoided by the following result, where we assume that $\&g[\vec{p}](\vec{c})$ is decidable in polynomial time.

Proposition 5. *Checking faithfulness of a given set $D \subseteq dep(\&g[\vec{p}])$ is co-NEXP-complete in general, and co-NP-complete for fixed predicate arities.*

When certain properties of external sources are known, less external calls are needed for faithfulness checking, e.g. for monotonic functions. An input $p_i \in \vec{p}$ of a ge-predicate $\&g[\vec{p}]$ is *monotonic*, if for any assignment $\mathbf{A}$ and output $\vec{c}$, $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = \mathbf{T}$ implies $f_{\&g}(\mathbf{A}', \vec{p}, \vec{c}) = \mathbf{T}$ for every $\mathbf{A}' \geq \mathbf{A}$ s.t. $\mathbf{A}(p_j(\vec{d})) = \mathbf{A}'(p_j(\vec{d}))$ for all predicates $p_j \in \vec{p}$ with $p_j \neq p_i$ (cf. [6]). Based on monotonicity, the number of assignments to consider in a faithfulness check can be decreased.

Proposition 6. *If $p_i \in \vec{p}$ for $\&g[\vec{p}]$ is monotonic, a set $D \subseteq dep(\&g[\vec{p}])$ is* faithful for $\&g[\vec{p}]$ *iff for any assignments $\mathbf{A}$, $\mathbf{A}'$ s.t. $\mathbf{T}p_i(\vec{d}) \in \mathbf{A}$ and $\mathbf{F}p_i(\vec{d}) \in \mathbf{A}'$ for every $p_i(\vec{d}) \notin comp(D, \&g[\vec{p}](\vec{c}))$ and $\mathbf{A}(p_i(\vec{d})) = \mathbf{A}'(p_i(\vec{d}))$ for every $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$, it holds that $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.*

Example 8 (cont'd). As *node* is a monotonic input parameter of $\&suc[node]$, for checking faithfulness wrt. a it suffices to evaluate $f_{\&suc}(\mathbf{A}, node, a)$ under two assignments $\mathbf{A}_t$ and $\mathbf{A}_f$, s.t. $\mathbf{A}_t \subseteq \{\mathbf{T}node(a), \mathbf{T}node(b), \mathbf{T}node(c), \mathbf{T}node(d)\}$ and $\mathbf{A}_f \subseteq \{\mathbf{F}node(a), \mathbf{F}node(b), \mathbf{F}node(c), \mathbf{F}node(d)\}$.

Under additional conditions, we obtain tractability:

Corollary 2. *If all $p_i \in \vec{p}$ for $\&g[\vec{p}]$ are monotonic and $|comp(D, \&g[\vec{p}](\vec{c}))|$ is bounded, then checking faithfulness is polynomial for fixed predicate arities.*

The same holds for computing a tight faithful set D for $\&g[\vec{p}]$. In practice, this applies to Example 3, if the external graph has bounded degree.

Relativized io-dependencies. So far, the context of a given HEX-program has not been exploited for specifying respective io-dependencies. However, without

considering how dependencies in an external source may be affected by input parameters, all io-dependencies that may hold under any possible extension of input predicates must be respected. This is illustrated by the following example.

Example 9. Consider $\&suc[edge, node](X)$, where edges from *edge* are inserted into $\mathcal{G}$ before successor nodes are output. If it is unknown which edges can be added, io-dependencies must account for the complete graph (all edges), which is a maximal overapproximation. Now, consider the following HEX-program.

$$edge(b, c) \vee n\_edge(b, c). \quad node(a). \quad node(b) \leftarrow \&suc[edge, node](b).$$

As $edge(b, c)$ is the only atom with predicate *edge* that can potentially be true in the input of $\&suc[edge, node](b)$ in any answer set, it suffices to specify io-dependencies wrt. the graph $\mathcal{G}' = (V, E \cup \{b \to c\})$ to ensure e-minimality.

To account for the inputs to external sources that are possible in answer sets, we define faithfulness wrt. a HEX-program Π. Let $env(\Pi)$ denote the set of all atoms for Π that are true in some compatible set of Π.

Definition 8 (Relativized Faithfulness). *A set $D \subseteq dep(\&g[\vec{p}])$ is* faithful wrt. a HEX-program Π*, if for any assignments $\mathbf{A}$, $\mathbf{A}'$ s.t. $\{a \mid \mathbf{T}a \in \mathbf{A} \cup \mathbf{A}'\} \subseteq env(\Pi)$, and for any output tuple $\vec{c}$ for $\&g[\vec{p}]$, either $\mathbf{A}(p_i(\vec{d})) \neq \mathbf{A}'(p_i(\vec{d}))$ for some atom $p_i(\vec{d}) \in comp(D, \&g[\vec{p}](\vec{c}))$ or $f_{\&g}(\mathbf{A}, \vec{p}, \vec{c}) = f_{\&g}(\mathbf{A}', \vec{p}, \vec{c})$.*

We show that skipping e-minimality checks based on the relativized definition of faithful io-dependencies is still safe.

Proposition 7. *Theorem 1 still holds if the specified io-dependencies are faithful wrt. to the* HEX*-program Π at hand according to Definition 8.*

The properties of above can be adjusted to this setting.

4 Empirical Evaluation

To empirically evaluate our new technique, we integrated it into the HEX-solver DLVHEX 2.5.0, which uses GRINGO 4.4.0 and CLASP 3.1.1 as backends [9], and tested it on randomly generated instances. Io-depencendies for external atoms are specified by plugin-methods that compute whether a dependency between given input and output values exists. We used a Linux machine with two 12-core AMD Opteron 6238 SE CPUs and 512 GB RAM; the timeout was 300 secs and the memout 8 GB per instance. The average runtime of 10 instances per problem size is reported (in secs) for computing all answer sets; timeouts are in parentheses.

Configurations. To gain insights into how dependency graph pruning and other techniques interact, we consider the frequency of external calls as further factor. While basic evaluation in Section 2 evaluates external atoms wrt. candidate models, we can evaluate them also wrt. partial assignments [6]. At this, investigating how e-minimality check skipping interacts with partial evaluation is of interest as early external evaluation can speed up model search as well as the e-minimality check and thus, potentially influence the impact of our new technique.

Table 1. User Access Selection Results (few cycles)

#	**c-mod**	**c-mod + io-dep**	**part**	**part + io-dep**	**min-part**	**min-part + io-dep**	#cyclic
10	0.46 (0)	**0.43** (0)	0.60 (0)	0.58 (0)	1.53 (0)	1.36 (0)	7/10
15	2.64 (0)	**2.18** (0)	4.58 (0)	3.91 (0)	7.41 (0)	4.43 (0)	3/10
20	16.43 (0)	**14.71** (0)	44.90 (0)	41.93 (0)	43.87 (0)	31.03 (0)	5/10
25	43.85 (0)	**38.25** (0)	102.39 (1)	93.65 (1)	81.51 (0)	67.59 (0)	5/10
30	110.24 (2)	**91.01** (2)	192.48 (4)	180.58 (4)	168.80 (2)	99.53 (2)	4/10
35	111.62 (1)	**79.69** (1)	217.58 (4)	178.62 (2)	161.86 (2)	83.18 (1)	3/10
40	189.64 (2)	**141.12** (2)	262.35 (6)	231.22 (5)	202.95 (3)	143.12 (2)	5/10
45	264.04 (5)	216.89 (4)	269.49 (6)	227.88 (5)	263.40 (5)	**202.55** (4)	5/10
50	300.00 (10)	227.15 (4)	300.00 (10)	249.55 (6)	300.00 (10)	**220.61** (3)	2/10

$$y\_nd(X) \vee n\_nd(X) \leftarrow domain(X). \qquad \leftarrow nd(X), nd\_f(X).$$
$$nd(X) \leftarrow y\_nd(X). \qquad \leftarrow \text{not } nd(X), nd\_a(X).$$
$$nd(X) \leftarrow \&hasAccess[nd](X). \qquad \leftarrow \#count\{X{:}y\_nd(X)\} > 3.$$

Fig. 2. User Access Selection Rules

We compared three different configurations, each with and without dependency graph pruning based on specified io-dependencies (configuration **io-dep**):
• **c-mod**: external atoms are only evaluated wrt. *candidate models* (representing the standard configuration of DLVHEX);
• **part**: external atoms are evaluated wrt. *partial assignments* after every solver guess during the model search; and
• **min-part**: external atoms are evaluated wrt. *partial assignments* after every solver guess during the *e-minimality check*.

In the result tables, we show combinations of configurations where interactions are expected.We predicted **io-dep** to decrease the runtime if e-cycles can be removed from the dependency graph; and that the speedup is larger when **io-dep** is combined with **part** and smaller when combined with **min-part**, whenever partial evaluation is beneficial. If pruning does not skip e-minimality checks, we expected no significant overhead in terms of runtime with **io-dep**.

User Access Selection (UAS). Consider a set of computer nodes C and a set of directed connections A between nodes, where $n_1 \to n_2 \in A$, for $n_1, n_2 \in C$, iff node n_1 has access to node n_2. Hence, a node can be accessed directly, or indirectly via other nodes. Now, suppose a network admin has to assign access rights by selecting nodes $C' \subseteq C$ to which some user will be granted access, s.t. every node in a set $C_a \subseteq C$ (required access) is accessible from some $n \in C'$ and no node in a set $C_f \subseteq C$ (forbidden nodes) is accessible from any $n \in C'$.

We assume the network is not known initially, but each node can be queried for its connections. For this, we use an external atom $\&hasAccess[nodes](n)$, which interfaces external network information, and outputs all nodes that can be accessed by some node in the extension of *nodes*. Accordingly, it evaluates to true for an output node n_2 wrt. an assignment $\mathbf{A}$ iff $\mathbf{T}nodes(n_1) \in \mathbf{A}$ for some $(n_1, n_2) \in A$. Moreover, we specify $D(\&hasAccess[nodes]) = \big\{\langle 2, 1 : \{n_1 \mid n_1 \to n_2 \in A\}, 1 : n_2\rangle \mid n_2 \in C\big\}$, i.e. there is a dependency of an output on an input

Table 2. User Access Selection Results (many cycles)

#	c-mod	c-mod + io-dep	part	part + io-dep	min-part	min-part + io-dep	#cyclic
10	0.41 (0)	0.41 (0)	**0.35** (0)	0.36 (0)	0.46 (0)	0.46 (0)	10/10
15	7.55 (0)	7.61 (0)	**6.17** (0)	6.38 (0)	7.95 (0)	8.15 (0)	10/10
20	44.03 (1)	43.92 (1)	**6.52** (0)	6.57 (0)	44.54 (1)	44.66 (1)	10/10
25	107.50 (2)	107.95 (2)	**51.60** (1)	51.62 (1)	87.53 (1)	87.51 (1)	10/10
30	84.97 (0)	84.64 (0)	**44.23** (0)	44.73 (0)	85.64 (0)	85.42 (0)	10/10
35	223.56 (5)	222.95 (5)	**111.29** (1)	110.98 (1)	223.26 (5)	224.26 (5)	10/10
40	268.27 (7)	268.73 (7)	**152.53** (1)	153.28 (1)	268.86 (7)	269.44 (7)	10/10
45	284.12 (8)	284.33 (8)	**251.08** (4)	252.54 (4)	286.90 (8)	286.56 (8)	10/10
50	300.00 (10)	300.00 (10)	300.00 (10)	**298.61** (9)	300.00 (10)	300.00 (10)	10/10

node whenever the latter has access to the former. The HEX-program in Figure 2 with facts $domain(n)$ for $n \in C$, facts $node\_a(n)$ for $n \in C_a$, and facts $node\_f(n)$ for $n \in C_f$ encodes UAS, where at most three nodes can be accessed directly.

First, we generated networks with $N \in [10, 50]$ nodes, where each node has access to another node with probability $\frac{1}{2 \times N}$ (cf. Table 1). This yields networks about half of which have no cycles and thus, dependency pruning can have an effect on the number of required e-minimality checks. Next, we increased the access probability to $\frac{2}{N}$ (cf. Table 2). This effects that nearly all networks contain cycles, which allowed us to investigate the effect of pruning when this does not impact the need for an e-minimality check. The rightmost column shows the fraction of instances where the computer network has a cycle.

Sequential Allocation of Indivisible Goods (SAIG). Next, we considered a problem from *Social Choice*, namely dividing a set G of m items among two agents a_1 and a_2 by allowing them to pick items in specific sequences $\sigma = o_1 o_2 ... o_m \in \{a_1, a_2\}^m$ [11]. Each agent a_i has a linear preference order $>_i$ over G; and the utility of $g \in G$ for a_i is $u_i(g) = |\{g' \mid g >_i g' \in G\}|$. We assume that an agent always picks the remaining item with maximal utility. The goal is to find a sequence σ resulting in an *envy-free* division of items, i.e. where no agent prefers the items of the other agent over its own items.

We use an external atom to obtain the choices of the agents, while their complete preferences are hidden, and a further one that checks whether an allocation is envy-free. The atom $\&pick[alreadyPicked](a_i, p, g)$ evaluates to true wrt. assignment $\mathbf{A}$ iff $p \in [1, m]$ and $g >_i g'$ for all g' s.t. $\mathbf{T}alreadyPicked(p{-}1, g) \notin \mathbf{A}$, where p represents the positions in a respective sequence. Furthermore, let $G(\mathbf{A}, i, j) = \sum_{g \in \{g \mid \mathbf{T}picked(a_i, p, g) \in \mathbf{A}\}} u_j(g)$. Then, the atom $\&envyFree[picked]()$ is true iff $G(\mathbf{A}, 1, 1) < G(\mathbf{A}, 2, 1)$ and $G(\mathbf{A}, 2, 2) < G(\mathbf{A}, 1, 2)$. The encoding is shown in Figure 3. Together with facts $position(p)$ and $item(g)$ for all $p, g \in [1, m]$, its answer sets encode all sequences that induce an envy-free allocation.

We set $D(\&pick[alreadyPicked]) = \big\{\langle 1, 1{:}\{p_1\}, 2{:}p_2\rangle \mid p_1, p_2 \in [1, m],\ p_2 = p_1{+}1\big\}$, i.e. items already picked at a sequence position only depend on previous positions. The io-dependencies eliminate all cyclic dependencies via external atoms in the instances; thus e-minimality checks can always be skipped. We tested instances with random preference orders and $N \in [3, 10]$ items (cf. Table 3).

Table 3. Sequential Allocation Results

#	**c-mod**	**c-mod + io-dep**	**part**	**part + io-dep**	**min-part**	**min-part + io-dep**
3	**0.19** (0)	**0.19** (0)	0.25 (0)	0.24 (0)	0.38 (0)	**0.19** (0)
4	2.74 (0)	1.73 (0)	0.74 (0)	**0.64** (0)	2.54 (0)	1.72 (0)
5	300.00 (10)	78.28 (0)	152.33 (5)	**2.42** (0)	141.76 (1)	78.02 (0)
6	300.00 (10)	300.00 (10)	300.00 (10)	**8.22** (0)	300.00 (10)	300.00 (10)
7	300.00 (10)	300.00 (10)	300.00 (10)	**26.63** (0)	300.00 (10)	300.00 (10)
8	300.00 (10)	300.00 (10)	300.00 (10)	**89.97** (0)	300.00 (10)	300.00 (10)
9	300.00 (10)	300.00 (10)	300.00 (10)	**284.17** (4)	300.00 (10)	300.00 (10)
10	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)	**300.00** (10)

$$turn(a\_1, P) \vee turn(a\_2, P) \leftarrow position(P).$$
$$picked(A, P, G) \leftarrow \&pick[alreadyPicked](A, P, G), turn(A, P), item(G).$$
$$alreadyPicked(P, G) \leftarrow position(P), position(P1), P1 < P, picked(\_, P1, G).$$
$$\leftarrow \text{not } \&envyFree[picked]().$$

Fig. 3. Sequential Allocation Rules

Findings. When dependency graph pruning skips e-minimality checks, **io-dep** significantly improves the runtimes for all instance sizes and independent from the configuration it is combined with (cf. Tables 1 and 3). In many cases, we are able to solve significantly more instances than before. In Table 2, **io-dep** has only a negligible impact on the runtimes. As **io-dep** has no advantage for cyclic instances, this shows that dependency pruning yields not much overhead. Partial evaluation was only beneficial both in the model search and the e-minimality check for SAIG. As predicted, the speedup for **part+io-dep** is larger than for **min-part+io-dep** since **min-part** already reduces the runtimes required for e-minimality checks, while **part** needs to invest more time in the e-minimality check. The runtimes for **c-mod+io-dep** and **min-part+io-dep** are similar; this is expected as **min-part** only applies to the e-minimality check, which is skipped in both cases. In summary, there is no clear winner among the conditions, but adding **io-dep** is suggestive as a default when io-dependencies can be specified.

5 Discussion and Conclusion

We introduced io-dependencies to formalize semantic dependencies over external atoms that approximate the real dependencies more closely than previously possible. Based on this, more e-minimality checks can be skipped, which proved to be beneficial in practice. We also stated properties for checking and optimizing io-dependencies important for automatically constructing tight faithful dependency sets. While faithfulness checking is intractable in general, we identified cases where the costs can be reduced for certain oracles, or where checking is polynomial.

Our approach is related to *domain independence* techniques in [3], where external atoms are evaluated wrt. subsets of the domain while correct outputs are retained. This is similar to our notions of compliant atoms and faithfulness. Yet, io-dependencies are more general because in [3], only disjoint domain partitions

for external inputs are considered, and dependencies are not used for argument positions. Another important difference is that their approach employs dependencies for *program splitting* as in [13], while we aim at detecting redundant e-minimality checks. They do not analyze the costs for generating dependencies.

Apart from HEX, there are several other approaches that integrate external theories into declarative problem solving, such as CLINGO [8], *SMT* [1] and *Constraint-ASP* [12]. However, to the best of our knowledge, external minimality has not been considered there. Nevertheless, our technique could also be employed directly by related rule-based formalisms if minimality involving external theories is required. Moreover, cyclic support may arise from *external propagators*, e.g. in the *WASP*-solver [2], where our approach could be applied as well.

While we only exploited semantic dependencies for e-minimality checking, additional dependency information is also useful for other parts of HEX-solving such as grounding and *External Behavior Learning* [6]. By limiting oracle calls to compliant input atoms, the number of external calls during HEX-evaluation could potentially be reduced significantly.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artif. Intell. and Applications, vol. 185, pp. 825–885. IOS Press (2009)
2. Dodaro, C., Ricca, F., Schüller, P.: External propagators in WASP: preliminary report. In: Bistarelli, S., Formisano, A., Maratea, M. (eds.) RCRA@AI*IA 2016. CEUR-WS, vol. 1745, pp. 1–9. CEUR-WS.org (2016)
3. Eiter, T., Fink, M., Krennwallner, T.: Decomposition of declarative knowledge bases with external functions. In: Boutilier, C. (ed.) IJCAI 2009. pp. 752–758 (2009)
4. Eiter, T., Fink, M., Krennwallner, T., Redl, C., Schüller, P.: Efficient HEX-program evaluation based on unfounded sets. J. Artif. Intell. Res. **49**, 269–321 (2014)
5. Eiter, T., Kaminski, T., Redl, C., Schüller, P., Weinzierl, A.: Answer set programming with external source access. In: Ianni, G., Lembo, D., Bertossi, L.E., Faber, W., Glimm, B., Gottlob, G., Staab, S. (eds.) Reasoning Web 2017, Tutorial Lectures. LNCS, vol. 10370, pp. 204–275. Springer (2017)
6. Eiter, T., Kaminski, T., Redl, C., Weinzierl, A.: Exploiting partial assignments for efficient evaluation of answer set programs with external source access. J. Artif. Intell. Res. **62**, 665–727 (2018)
7. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. **175**(1), 278–298 (2011)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Carro, M., King, A., Saeedloei, N., Vos, M.D. (eds.) ICLP-TC 2016. OASICS, vol. 52, pp. 2:1–2:15. Schloss Dagstuhl (2016)
9. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: The potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011)
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4), 365–386 (1991)
11. Kalinowski, T., Narodytska, N., Walsh, T., Xia, L.: Strategic behavior when allocating indivisible goods sequentially. In: desJardins, M., Littman, M.L. (eds.) AAAI 2013. AAAI Press (2013)

12. Lierler, Y.: Relating constraint answer set programming languages and algorithms. Artif. Intell. **207**, 1–22 (2014)
13. Lifschitz, V., Turner, H.: Splitting a logic program. In: Hentenryck, P.V. (ed.) ICLP 1994. pp. 23–37. MIT Press (1994)