Conflict-driven ASP Solving with External Sources and Program Splits

Christoph Redl *
Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
redl@kr.tuwien.ac.at

Abstract

Answer Set Programming (ASP) is a well-known problem solving approach based on nonmonotonic reasoning. HEX-programs extend ASP with *external atoms* for access to arbitrary external sources, which can also introduce constants that do not appear in the program (*value invention*). In order to determine the relevant constants during (pre-)grounding, external atoms must in general be evaluated under up to exponentially many possible inputs. While program splitting techniques allow for eliminating exhaustive pre-grounding, they prohibit effective conflict-driven solving. Thus, current techniques suffer either a grounding or a solving bottleneck. In this work we introduce a new technique for conflict-driven learning over multiple program components. To this end, we identify reasons for inconsistency of program components wrt. input from predecessor components and propagate them back. Experiments show a significant, potentially exponential speedup.

1 Introduction

Answer-Set Programming (ASP) is a declarative programming paradigm based on nonmonotonic programs [Gelfond and Lifschitz, 1991]. HEX-programs are an extension of ASP with external sources (ontologies, Web resources, etc). So-called external atoms pass information from the program to an external source, which returns values to the program. Notably, *value invention* allows for domain expansion. For instance, the external atom $\&synonym[car](X)$ might be used to find the synonyms X of car, e.g. $automobile$. Also recursive data exchange between the program and external sources is supported, which leads to high expressiveness of the formalism. We use HEX as a representative for formalisms with expanding domains. Also well-known ASP extensions such as aggregates [Faber *et al.*, 2010] and constraint ASP [Ostrowski and Schaub, 2012] correspond to special cases of HEX.

Suitable safety conditions guarantee the existence of a finite grounding which suffices for answer set computation; a grounding algorithm exists [Eiter *et al.*, 2016b]. However, computing a grounding is often expensive as, in general, the grounder must evaluate external sources under up to exponentially many inputs to ensure that all relevant constants are respected. The situation is relieved by a model-building framework based on *program splitting*, where program components are arranged in a directed acyclic graph [Eiter *et al.*, 2016a]. Then, at the time a component is grounded, its predecessors have already been evaluated and their answer sets can be exploited to skip evaluations. However, splitting deteriorates the performance of the conflict-driven solver since propagation throughout the whole program is inhibited. Therefore, current approaches suffer either a solving or a grounding bottleneck, depending on whether program splitting is used or not.

In this work we propose a **novel learning technique for programs with multiple components**, which is our main contribution. This allows for retaining the splitting technique for the sake of efficient grounding, but at the same time also propagate learned knowledge throughout the whole program. To this end, we identify reasons for inconsistencies in program components in terms of input from predecessors and propagate them back. The results are relevant beyond HEX as they carry over to special cases thereof; also ordinary ASP solving can benefit from splitting [Eiter *et al.*, 2011] and is another possible application of our results.

In more detail, after the preliminaries (Section 2), the organization of the paper and our contributions are as follows:

- In Section 3 we use the notion of *inconsistency reasons (IRs)* for HEX-programs and present an algorithm for computing them for ground programs.
- In Section 4 we extend the algorithm to the nonground case, respecting possible optimizations by the grounder.
- In Section 5 we show how the techniques can be used to realize learning over multiple program components.
- In Section 6 we present our implementation and experimental results, which show a significant – in some cases even exponential – speedup.
- In Section 7 we discuss related work and conclude the paper.

Proofs are outsourced to the extended version at http://www.kr.tuwien.ac.at/research/projects/inthex/tulearning-ext.pdf.

2 Preliminaries

We follow Drescher *et al.* [2008] for basic concepts. We use as our alphabet mutually disjoint sets $\mathcal{P}$ of predicates, $\mathcal{X}$ of

*Supported by the Austrian Science Fund (FWF) project P27730.

external predicats, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. The set of all terms is given by $\mathcal{T} = \mathcal{C} \cup \mathcal{V}$.

An (ordinary) atom a is of form $p(t_1, \ldots, t_\ell)$ with predicate symbol $p \in \mathcal{P}$ and terms $t_1, \ldots, t_\ell \in \mathcal{T}$, abbreviated as $p(\mathbf{t})$; for a list of terms $\mathbf{t} = t_1, \ldots, t_\ell$ we write $t \in \mathbf{t}$ if $t = t_i$ for some $1 \leq i \leq \ell$. We call an atom (or other objects) *ground* if it does not contain variables. A (signed) literal is of type $\mathbf{T}a$ or $\mathbf{F}a$, where a is an atom. We let $\overline{\sigma}$ denote the negation of a literal, i.e. $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* $\mathbf{A}$ over the (finite) set A of ground atoms is a set of literals over A, where $\mathbf{T}a \in \mathbf{A}$ expresses that a is true, also denoted $\mathbf{A} \models a$, and $\mathbf{F}a \in \mathbf{A}$ that a is false, also denoted $\mathbf{A} \not\models a$. A *nogood* is a set $\{L_1, \ldots, L_n\}$ of ground literals $L_i, 1 \leq i \leq n$. An assignment $\mathbf{A}$ is a *solution* to a nogood δ if $\delta \not\subseteq \mathbf{A}$, and to a set of nogoods Δ if $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$. Note that according to this definition, partial assignments (i.e., assignments such that for some $a \in A$ we have $\mathbf{T}a \notin A$ and $\mathbf{F}a \notin A$) might be solutions to nogoods, even if supersets thereof are not. This definition is by intend and does not harm in the following.

HEX**-Programs**. We briefly recall HEX-programs, which generalize (disjunctive) logic programs under the answer set semantics [Gelfond and Lifschitz, 1991]; for more details and background, see Eiter *et al.* [2016a].

Syntax. HEX-programs extend ordinary ASP-programs by *external atoms*, which enable a bidirectional interaction between a program and external sources of computation. An *external atom* is of the form $\&g[\mathbf{Y}](\mathbf{X})$, where $\&g \in \mathcal{X}$ is an external predicate, $\mathbf{Y} = p_1, \ldots, p_k$ is a list of input parameters (predicate names from $\mathcal{P}$ or constants[1] from $\mathcal{C} \cup \mathcal{V}$), called *input list*, and $\mathbf{X} = t_1, \ldots, t_l$ are output terms.

Definition 1. *A* HEX*-program P consists of rules*

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

where each a_i is a ground atom and each b_j is either an ordinary atom or an external atom.

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

Semantics. We first define the semantics of ground programs. In the following, assignments are over the set $A(P)$ of ordinary atoms that occur in the program P at hand. The semantics of a ground external atom $\&g[\mathbf{y}](\mathbf{x})$ wrt. an assignment $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary *Boolean oracle function* $f_{\&g}$ that is defined for all possible values of $\mathbf{A}$, $\mathbf{y}$ and $\mathbf{x}$. Thus, $\&g[\mathbf{y}](\mathbf{x})$ is true relative to $\mathbf{A}$ iff $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = \mathbf{T}$. In extension of the semantics by Gelfond and Lifschitz [1991], a ground rule r of form (1) is satisfied under $\mathbf{A}$, denoted $\mathbf{A} \models r$, if $\mathbf{A} \models a_i$ for some $1 \leq i \leq k$, or $\mathbf{A} \not\models b_i$ for some $1 \leq i \leq m$, or $\mathbf{A} \models b_i$ for some $m + 1 \leq i \leq n$. A ground program P is satisfied under $\mathbf{A}$, denoted $\mathbf{A} \models P$, if each $r \in P$ is satisfied under $\mathbf{A}$. The answer sets of a ground HEX-program P are defined using the *FLP-reduct* of P wrt. an assignment $\mathbf{A}$, which is the set $fP^{\mathbf{A}} = \{r \in P \mid \mathbf{A} \models b \text{ for all } b \in B(r)\}$ of all rules whose body is satisfied by $\mathbf{A}$:

Definition 2. *An assignment $\mathbf{A}$ is an answer set of a ground* HEX*-program P, if $\mathbf{A}$ is a subset-minimal model of $fP^{\mathbf{A}}$.* [2]

The set of all answer sets of a program P is denoted $\mathcal{AS}(P)$.

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

The answer sets of a (possibly) nonground program P are given by the answer sets $\mathcal{AS}(grnd_{\mathcal{C}}(P))$ of its grounding $grnd_{\mathcal{C}}(P) = \{\rho(r) \mid \rho\colon \mathcal{V} \to \mathcal{C}\}$, which results from P if all variables are replaced by all constants from $\mathcal{C}$ in all possible ways using *variable substitutions* $\rho\colon \mathcal{V} \to \mathcal{C}$; for a composed object x (atom, rule, program, etc), $\rho(x)$ denotes x after replacing each variable $V \in vars(x)$ in x by $\rho(V)$.

While $grnd_{\mathcal{C}}(P)$ may be infinite in general, suitable safety criteria guarantee that a finite subset suffices to compute the answer sets, cf. Eiter *et al.* [2016b]. In this paper, all programs are assumed to satisfy these criteria.

Grounding and solving algorithm. For the general understanding of this paper it is not necessary to introduce the grounding procedure in detail. Some details about the solving algorithm are introduced later on. However, it is important to know the following properties of the algorithms.

The existing *grounding algorithm* [Eiter *et al.*, 2016b] is typically slow whenever an external atom depends on atoms other than facts and this dependency is neither monotonic nor antimonotonic. Then already the construction of a single rule instance is exponential in the number of atoms. Intuitively, this is because the grounder does not yet know the relevant inputs to the external atom and must evaluate it under exponentially many possibilities to ensure that all relevant output elements are respected. After grounding, the *solving algorithm* [Eiter *et al.*, 2014a] is applied, which uses conflict-driven learning similar to SAT and ordinary ASP solving [Gebser *et al.*, 2012]; the search is interleaved with the evaluation of external sources. Efficiency strongly depends on learning additional nogoods during solving, which are only implicitly in the program. In addition, an *evaluation framework* has been developed, which allows for splitting the program into components [Eiter *et al.*, 2016a]; while splitting may speed up grounding since the grounder already knows parts of the answer set computed by previous components, it is barrier for conflict-driven learning.

3 Inconsistency Analysis – Ground Case

In this section we consider programs P which are extended with facts F_I over atoms $I \subseteq D$ from a given *(input) domain* D, where $F_I = \{a \leftarrow \mid a \in I\}$. This resembles a program (component) P evaluated under input I (e.g. from predecessor components, as discussed more extensively in Section 5).

Following Redl [2017b], we express reasons for inconsistency of $P \cup F_I$ in terms of atoms which must resp. must not occur in I in order to make $P \cup F_I$ inconsistent. Formally:

Definition 3 (Inconsistency Reason (IR))**.** *Let P be a* HEX*-program and D be a set of atoms, called* (input) domain*, such that $H(P) \cap D = \emptyset$. An* inconsistency reason (IR) *of P wrt. D is a pair $R = (R^+, R^-)$ of sets of atoms $R^+ \subseteq D$ and*

[1] Also variables become constants after the grounding step.

[2] For ordinary P, these are Gelfond & Lifschitz' answer sets.

$R^- \subseteq D$ *with* $R^+ \cap R^- = \emptyset$ *s.t.* $P \cup F_I$ *is inconsistent for* $I \subseteq D$ *with* $R^+ \subseteq I$ *and* $R^- \cap I = \emptyset$.

Example 2. *An IR of* $P = \{\leftarrow a, \mathrm{not}\, c;\ d \leftarrow b.\}$ *wrt.* $D = \{a, b, c\}$ *is* $R = (\{a\}, \{c\})$ *because* $P \cup F_I$ *is inconsistent for all* $I \subseteq D$ *whenever* $a \in I$ *and* $c \notin I$, *while* b *can be in* I *or not without influencing (in)consistency.*

Note that in contrast to work on ASP debugging, which we discuss in more detail in Section 7 and focuses on (human-readable) explanations for inconsistency of single programs, out notion rather aims at identifying *classes of program* instances depending on the input facts, which are inconsistent. As a consequence, the techniques developed for ASP debugging cannot directly be used, which is intuitively expected because checking a single program for inconsistency is computationally easier than identifying IRs, cf. Redl [2017b].

Further note that although small IRs are preferred, we do not formally require them to be minimal. This is in view of the application in Section 5, which relies on a fast method for computing IRs, for which we give up minimality.

In general, programs may have no, one or multiple IRs. For instance, $\{\leftarrow a; \leftarrow b\}$ has the inconsistency reasons $R_1 = (\{a\}, \emptyset)$ and $R_2 = (\{b\}, \emptyset))$ wrt. $D = \{a, b\}$.

We introduce now a new technique for computing IRs. Since it is based on an extension of the evaluation algorithm for ground HEX-programs, we recapitulate it in the following paragraph; for details we refer to Eiter *et al.* [2014a]. However, we slightly adopt the presentation of the algorithm as a preparation for the extensions in the paragraph afterwards.

Evaluation of ground HEX-programs. The evaluation follows Algorithm 1. The initialization is at (a). The given HEX-program P is extended with facts F_I and transformed to an ordinary ASP-program $\hat{P}$ by replacing each external atom $\&g[\mathbf{y}](\mathbf{x})$ in P by an ordinary *replacement atom* $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and adding a rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ to guess its value. Program $\hat{P}$ is then represented as set of nogoods Δ, which consists of nogoods $\Delta_{\hat{P}}$ stemming from Clark's completion [Clark, 1977] and singleton loop nogoods [Gebser *et al.*, 2012]; this set will be extended as the search space is traversed. The computed answer set $\hat{\mathbf{A}}$ of the guessing program $\hat{P}$ is initially empty; for the sake of the algorithm, assignments are seen as lists (i.e., the order of assignments is relevant). The decision level (dl) is initially 0 and incremented for every guess. We assume that $dl(a) \in \mathbb{N}_0$ stores the dl of each assigned atom. The initialization immediately assigns facts to true and atoms which do not appear in any heads to false (both at dl 0).

After the initialization, the algorithm performs deterministic propagation at (b) such as unit propagation; further propagation techniques such as unfounded set propagation can be added. Each implied literal is assigned at the maximum dl of the literals which implied it. The algorithm detects conflicts, learns additional nogoods and backtracks at (c). If the conflict is on dl 0, the instance is inconsistent and the callback function h is notified (with nogoods Δ and assignment $\hat{\mathbf{A}}$ over P as input) to determine the return value in case of inconsistency; this callback serves as a preparation for our extension of the algorithm below and is instantiated with just $h_\perp(\Delta, \hat{\mathbf{A}}) = \perp$ for answer set computation. If the assignment is complete at

Algorithm 1: HEX-CDNL

Input: A ground program P, input atoms I, an inconsistency handler function h with a nogood set and an assignment as parameters

Output: An answer set of P if existing, and $h(\Delta, \hat{\mathbf{A}})$ otherwise (where Δ resp. $\hat{\mathbf{A}}$ are the nogood set resp. assignment at the time of discovering the inconsistency)

$P \leftarrow P \cup F_I$; $\Delta \leftarrow \Delta_{\hat{P}}$; $\hat{\mathbf{A}} \leftarrow \emptyset$; $current\_dl \leftarrow 0$ **(a)**

for $a \leftarrow\in P$ **do** $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \circ (\mathbf{T}a)$; $dl[a] \leftarrow current\_dl$

for $a \notin H(r)$ *for all* $r \in P$ **do** $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \circ (\mathbf{F}a)$; $dl[a] \leftarrow current\_dl$

while *true* **do**

- $(\hat{\mathbf{A}}, \Delta) \leftarrow$ Propagation$(\hat{P}, \Delta, \hat{\mathbf{A}})$ **(b)**
- **if** *some nogood* $\delta \in \Delta$ *violated by* $\hat{\mathbf{A}}$ **then** **(c)**
  - **if** $current\_dl = 0$ **then return** $h(\Delta, \hat{\mathbf{A}})$
  - Analyze conflict, learn nogood (Δ), backjump ($current\_dl$)
- **else if** $\hat{\mathbf{A}}$ *is complete* **then** **(d)**
  - **if** *guesses are not correct or not minimal wrt. the reduct* **then**
    - $\Delta \leftarrow \Delta \cup \{\hat{\mathbf{A}}\}$
    - Analyze conflict, learn nogood (Δ), backjump ($current\_dl$)
  - **else**
    - **return** $\hat{\mathbf{A}}|_{A(P)}$
- **else if** *Heuristics evaluates* $\&g[\mathbf{y}]$ *and* $\Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}}) \not\subseteq \Delta$ **then** **(e)**
  - $\Delta \leftarrow \Delta \cup \Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$
- **else** **(f)**
  - Let $\sigma \in \{\mathbf{T}, \mathbf{F}\}$ and $a \in A(\hat{P})$ with $\{\mathbf{T}a, \mathbf{F}a\} \cap \hat{\mathbf{A}} = \emptyset$
  - $current\_dl \leftarrow current\_dl + 1$
  - $dl[a] \leftarrow current\_dl$
  - $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \circ (\sigma a)$

(d), the algorithm must check if the guesses of replacement atoms coincide with the real truth values of external atoms, and if it represents a model which is also subset-minimal wrt. the reduct. If this is not the case, the assignment is added as nogood in order to discard it.[3] Finally, the algorithm may perform theory propagation at (e) or guess a truth value at (f) if no other case is applicable.

Soundness and completeness of Algorithm 1 was shown by Eiter *et al.* [2014a] as follows:

Proposition 1. *Let* P *be a program and* $I \subseteq D$ *be input atoms from a domain* D. *If* HEX-CDNL$(P, I, h_\perp)$ *returns (i) an assignment* $\mathbf{A}$, *then* $\mathbf{A}$ *is an answer set of* $P \cup F_I$; *(ii)* $\perp$, *then* $P \cup F_I$ *is inconsistent.*

Computing inconsistency reasons. We now extend the existing evaluation algorithm for ground HEX-programs from the previous paragraph to an algorithm for computing IRs. Our approach is based on *implication graphs*, which represent the current status and assignment history of the solver following Algorithm 1, cf. e.g. Biere *et al.* [2009]. Intuitively, the nodes of the graph represent (already assigned) literals or conflicts, their decision levels, and the nogoods which implied them. Predecessor nodes represent implicants. Nodes without predecessors represent guesses. Formally:

Definition 4. *An* implication graph *is a directed graph* $\langle V, E\rangle$, *where* V *is a set of triplets* $\langle L, dl, \delta\rangle$, *denoted* $L@dl/\delta$, *where* L *is a signed literal or* $\perp$, $dl \in \mathbb{N}_0$ *is a decision level,* $\delta \in \Delta \cup \{\perp\}$ *is a nogood, and* E *is a set of unlabeled edges.*

Example 3. *Let* $\Delta = \{\delta_1\colon \{\mathbf{T}a, \mathbf{T}b\}, \delta_2\colon \{\mathbf{T}a, \mathbf{F}b, \mathbf{F}c\}, \delta_3\colon \{\mathbf{T}c, \mathbf{T}d, \mathbf{F}e\}\}$. *An implication graph is shown in Figure 1. Here,* $\mathbf{T}a@1/\perp$ *is*

[3] This is for simplicity, actual implementations are more involved.

Algorithm 2: InconsistencyAnalysis

Input: Domain D, set of nogoods Δ representing program P, assignment $\hat{\mathbf{A}}$ conflicting with Δ
Output: Inconsistency reason of P wrt. D

Let $\delta \in \Delta$ such that $\delta \subseteq \hat{\mathbf{A}}$ **(a)**
while *there is a* $\sigma a \in \delta$ *with* $a \notin D$ **do** **(b)**
  Let $\epsilon \in \Delta$ s.t. $\overline{\sigma a} \in \epsilon$ and $\epsilon \setminus \overline{\sigma a} \subseteq \hat{\mathbf{A}}'$ for some $\hat{\mathbf{A}} = \hat{\mathbf{A}}' \circ (\sigma a) \circ \hat{\mathbf{A}}$
  $\delta \leftarrow (\delta \setminus \{\sigma a\}) \cup (\epsilon \cup \overline{\sigma a})$ **(c)**
  $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}}'$
return $(\{a \mid \mathbf{T}a \in \delta\}, \{a \mid \mathbf{F}a \in \delta\})$ **(d)**

a guess on dl 1, $\mathbf{F}b@1/\delta_1$ *is implied by* $\mathbf{T}a$ *using* δ_1, $\mathbf{T}c@1/\delta_2$ *is implied by* $\mathbf{T}a$ *and* $\mathbf{F}b$ *using* δ_2, $\mathbf{T}d@2/\bot$ *is a guess on dl* 2, *and* $\mathbf{T}e@2/\delta_3$ *is implied by* $\mathbf{T}c$ *and* $\mathbf{T}d$ *using* δ_3.

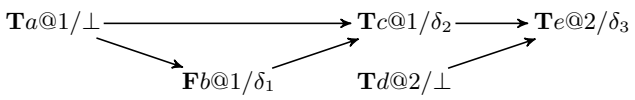

Figure 1: Implication graph of the program in Example 5

In order to compute an IR wrt. a domain D in case of inconsistency, we reuse Algorithm 1 but let it call Algorithm 2 after an inconsistency is detected by using InconsistencyAnalysis$(D, \Delta, \hat{\mathbf{A}})$ instead of $h_\bot$.

Observe that for answer set computation (i.e., using $h = h_\bot$), soundness of Algorithm 1 implies that $h(\Delta, \hat{\mathbf{A}})$ is called at some point because it is the only way to return $\bot$. Note that this can only happen after a violated nogood has been identified on decision level 0. But this implies that at the time the inconsistency of an instance is detected, all assignments are deterministic.

Now the basic idea of Algorithm 2 is as follows. We start at (a) with a nogood $\delta \in \Delta$ which is currently violated; such a nogood always exists because otherwise Algorithm 1 would not have called $h(\Delta, \hat{\mathbf{A}})$. Since each literal σa in this nogood was assigned on decision level 0, each of them was assigned by unit propagation as a consequence of previously assigned literals. Algorithm 2 resolves this nogood at (c) with the cause of one of its literals σa, i.e., with the nogood ϵ which implied literal σa. The loop at (b) repeats this step until the nogood contains only literals over explanation atoms from D. The iterative resolution corresponds to the replacement of the literal by its predecessors in the implication graph (where the implication graph itself is implicitly represented by the nogoods that implied literals). The iteration steps when the nogood δ contains only literals from D; since these literals imply the originally violated nogood and are thus responsible for inconsistency, δ represents an IR which we return at (d).

Using $h^D_{analyse}(\Delta, \hat{\mathbf{A}}) =$ InconsistencyAnalysis$(D, \Delta, \hat{\mathbf{A}})$ allows then for exploiting Algorithm 1 to compute IRs:

Proposition 2. *Let P be a program and $I \subseteq D$ be input atoms from a domain D. If* HEX-CDNL$(P, I, h^D_{analyse})$ *returns (i) an assignment* $\mathbf{A}$, *then* $\mathbf{A}$ *is an answer set of $P \cup F_I$; (ii) a pair $R = (R^+, R^-)$ of sets of atoms, then $P \cup F_I$ is inconsistent and R is an inconsistency reason of P wrt. D.*

4 Inconsistency Analysis – Nonground Case

We now extend the above approach to nonground programs.

Grounding algorithms for ASP and HEX do not use the naive grounding $grnd_\mathcal{C}(P)$ which substitutes variables $\mathcal{V}$ in P by constants in $\mathcal{C}$ in all possible ways. Instead, optimizations are performed to keep the grounding small. However, the exact algorithms for performing such optimizations, and therefore the output, depend on the used grounder. In fact, a correct grounding procedure can output any *optimized ground program* $og_\mathcal{C}(P)$ as long as $\mathcal{AS}(og_\mathcal{C}(P)) = \mathcal{AS}(P)$ holds; it does not even be a subset of $grnd_\mathcal{C}(P)$, which allows for optimizations within rules and introducing auxiliary rules. This is why one cannot give a precise definition of an optimized grounding. However, while multiple valid groundings exist in general, we assume that an arbitrary but fixed algorithm is used, which makes $og_\mathcal{C}(P)$ unique; similarily for $pog_{\mathcal{C},I}(P)$ introduced in the following.

This prohibits the direct reduction of the computation of an IR for a (possibly) nonground program P wrt. a domain D to the ground case. Let $P_g = og_{\mathcal{C},I}(P)$ be an optimized grounding of $P \cup F_I$ for some $I \subseteq D$, where we assume that the facts F_I themselves are not included in the grounding. Then, if HEX-CDNL$(P_g, I, h^D_{analyse})$ returns a pair $R = (R^+, R^-)$, we have by Proposition 2 that R is an IR for P_g, i.e., $P_g \cup F_J$ is inconsistent for all $J \subseteq D$ with $R^+ \subseteq J$ and $R^- \cap J = \emptyset$. However, R is not necessarily an IR for P wrt. D because for a different set of facts $I' \subseteq D$ we may have $og_{\mathcal{C},I'}(P) \neq P_g$.

Example 4. *Consider $P = \{q(X) \leftarrow p(X);\ \leftarrow \text{not } q(1)\}$ and $D = \{p(1)\}$. For input $I = \emptyset$, $P_g \cup F_I = \{\leftarrow \text{not } q(1)\}$ is a valid grounding for evaluation because $\mathcal{AS}(P \cup F_I) = \mathcal{AS}(P_g \cup F_I) = \emptyset$. However, for inconsistency analysis of P, the grounding $P_g \cup F_I$ is not sufficient: $R = (\emptyset, \emptyset)$ is an IR of P_g wrt. D but not of P wrt. D as $P \cup F_{\{p(1)\}}$ is consistent.*

We thus use a *partially optimized grounding* $pog_{\mathcal{C},I}(P)$ for a specific input $I \subseteq D$ with the properties that (i) $pog_{\mathcal{C},I}(P) \subseteq grnd_\mathcal{C}(P)$ and (ii) $\mathcal{AS}(pog_{\mathcal{C},I}(P)) = \mathcal{AS}(P \cup F_I)$. That is, optimization is restricted to the elimination of rules, while changes within or additions of rules are prohibited. A grounding procedure for HEX-progams with these properties was presented by Eiter *et al.* [2016b].

The idea is to use $pog_{\mathcal{C},I}(P)$ for the inconsistency analysis, but respect that unknown rules might provide additional support for atoms. To this end, we introduce for each atom a in $pog_{\mathcal{C},I}(P)$ a rule $a \leftarrow a'$, where a' is a new atom, to represent the situation that a is supported by rules not in $pog_{\mathcal{C},I}(P)$. While structural information of the program might be exploited to further restrict the set of atoms that can have further support by rules that are missing in the current grounding, this is not a trivial task; we leave it for future work in order to focus on the main idea in this paper. Then, if an IR does not contain any a', possibly missing rules are not relevant and it carries over to program P. Formally:

Proposition 3. *Let P be a program and $I \subseteq D$ be input atoms from a domain D. Then an IR $R = (R^+, R^-)$ of $pog_{\mathcal{C},I}(P) \cup \{a \leftarrow a' \mid a \in A(pog_{\mathcal{C},I}(P))\}$ wrt. $D \cup \{a' \mid a \in A(pog_{\mathcal{C},I}(P))\}$ s.t. $(R^+ \cup R^-) \cap \{a' \mid a \in A(pog_{\mathcal{C},I}(P))\} = \emptyset$ is an IR of P wrt. D.*

5 Trans-Unit Propagation

Current evaluation techniques are in particular bad for programs with guesses that are separated from constraints by nonmonotonic external atoms; we focus on programs of this form in the following. While monotonic and antimonotonic external atoms can be efficiently grounded without splitting, grounding nonmonotonic external atoms, which do not depend only on facts, requires exponentially many calls to external sources. Splitting can solve this grounding bottleneck, but it introduces a solving bottleneck because subsequent constraints cannot be propagated back.

Example 5. *Suppose we want to form a committee of employees. Some pairs of persons should not be together in the committee due to conflicts of interests (cf. independent sets of a graph). The competences of the committee depend on the persons involved. For instance, it can decide in technical questions only if a certain number of members has expert knowledge in the field. The competences can depend nonmonotonically on the members. For instance, while overrepresentation of a department might not be forbidden in principle, it can make it lose authorities such as assigning more resources to this department. Constraints define the competences the committee should have. This is encoded as follows:*

$$P = \{r_1\colon in(X) \vee out(X) \leftarrow person(X).$$
$$r_2\colon \leftarrow in(X), in(Y), conflict(X,Y).$$
$$r_3\colon comp(X) \leftarrow \&competences[in](X).$$
$$r_4\colon \leftarrow \text{not } comp(technical), \text{not } comp(financial).\}$$

Here, we do not want committees which can neither decide in technical nor financial affairs.

When grounding P as a whole, since the external atom in r_3 depends nonmonotonically on the guessed committee, the grounder must evaluate it under all possible guesses as each one could introduce other competences. However, many guesses are not relevant because they are already eliminated by r_2. In contrast, an existing evaluation framework for HEX-programs allows for splitting it into components which are separately grounded and solved [Eiter *et al.*, 2016a]. More precisely, based on the dependencies between rules, the program is partitioned into subprograms such that the resulting dependencies between the subprograms form an *acyclic evaluation graph* (cyclically depending rules must be in one component); an example is shown in Figure 2. For the evaluation, the framework first computes the answer sets of components without predecessors. The successor components are then extended with the answer sets of predecessor components (one at a time), grounded and solved. This procedure is repeated recursively; the final answer sets are extracted from the leaf components. For details we refer to Eiter *et al.* [2016a].

Example 6 (cont'd)**.** *Program P from Example 5 can be partitioned into P_1 and P_2, where P_2 depends on P_1, cf. Figure 2.*

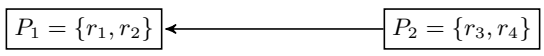

Figure 2: Evaluation graph of the program from Example 5

However, then r_4 is separated from the guess and a conflict with r_4 during the evaluation of P_2 cannot be propagated to exclude guesses. Hence, with current evaluation techniques, there is a bottleneck either in the grounding phase (if the program is not split) or in the solving phase (if it is split).

Propagating inconsistency reasons. The main idea is to associate an IR $R = (R^+, R^-)$ of a later program component with a constraint $c_R = \leftarrow R^+, \{\text{not } a \mid a \in R^-\}$ which we propagate to predecessors. Recall that for a domain D of explanation atoms and an inconsistency reason $R = (R^+, R^-)$ wrt. a program P, all programs $P \cup F_I$ with $I \subseteq D$ such that $R^+ \subseteq I$ and $R^- \cap I = \emptyset$ are inconsistent. Thus:

Proposition 4. *For all* HEX*-programs P and IRs $R = (R^+, R^-)$ of P wrt. a domain D, we have that $\mathcal{AS}(P \cup F_I) = \mathcal{AS}(P \cup \{c_R\} \cup F_I)$ for all $I \subseteq D$.*

Such a constraint can also be added to predecessor components in order to eliminate wrong guesses earlier, which we call *trans-unit (tu-)propagation*.

Proposition 5. *For a program component P and IRs $R = (R^+, R^-)$ of P wrt. a domain D, we have that c_R can be added to a predecessor component P', if all atoms appearing in c_R are defined in P' or one of its own transitive predecessors.*

Example 7 (cont'd)**.** *Assume that joe and sue are the only technicians and $alyson$ is the only economist. Then, P_2 is always inconsistent if none of these three persons is selected, independent of other choices. If the current input to component P_2 is $I = \{in(jack), in(joseph)\}$, then by exploiting nogoods learned from the external source &competences, the IR $R = (\emptyset, \{in(joe), in(sue), in(alyson)\})$ wrt. $P_2 \cup F_I$ can be determined, and the constraint $c_R = \leftarrow \text{not } in(joe), \text{not } in(sue), \text{not } in(alyson)$ can be added to P_1.*

According to Proposition 5, a constraint constructed in this way can be added to a predecessor component P', *if all atoms appearing in c_R are defined in P'*. That is, it can be added to all predecessors which fulfill this criterion. In our implementation and experiments, we add it to the top-most one. This in order to eliminate invalid candidates as early as possible. For analytical reasons, this is always better than adding them to later program components since it eliminates possibly more candidates, while costs for adding the constraint are the same for all components.

6 Implementation and Experiments

For the experiments, we integrated our techniques into the reasoner DLVHEX[4] with GRINGO 4.5.4 and CLASP 3.1.4 as backends. All benchmarks were run on a Linux machine with two 12-core AMD Opteron 6176 SE CPUs and 128 GB RAM; timeout was 300 secs and memout 8 GB per instance. We used the *HTCondor* load distribution system (http://research.cs.wisc.edu/htcondor) to ensure robust runtimes (i.e., deviations of runs on the same instance are negligible).

Setting and hypotheses. Based on the program class we want to assess, we selected our benchmarks such that guessing and checking is separated by nonmonotonic external atoms. For each benchmark we compare three configurations: (i) evaluation as a *monolithic* program, (ii) evaluation using *splitting*,

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

and (iii) evaluation with *trans-unit (tu-)propagation*. Our hypothesis is that (i) suffers a grounding and (ii) suffers (also) a solving bottleneck, whereas (iii) outperforms the other two approaches as it can restrict the grounding and propagate throughout the whole program at the same time. In the tables we show the overall runtime, grounding and solving time, averaged over all instances of the respective size; importantly, other computations besides grounding and solving (e.g. preprocessing and method-specific tasks such as splitting for (ii) and inconsistency analysis for (iii)) can cause the overall runtime to be higher than the sum of grounding and solving time. Numbers in parantheses show the number of timeout instances. The encodings of all presented benchmarks can be found in the official benchmark suite of the system.[5]

Configuration Problem. Consider the following configuration problem. We assemble a server cluster consisting of various components. For a given component selection the cluster has different properties such as its efficiency, power consumption, etc. Properties may depend not only on individual components but also on their interplay, and this dependency can be nonmonotonic (e.g. the selection of an additional component might make it lose the property of low energy consumption). We want the cluster to have certain properties.

In order to capture also similar configuration problems (such as Example 5), we use a more abstract formalization as a quadruple (D, P, m, C), where D is a *domain*, P is a set of *properties*, m is a function which associates each selection $S \subseteq D$ with a set of properties $m(S) \subseteq P$, and C is a set of constraints of form $C_i = (C_i^+, C_i^-)$, where $C_i^+ \subseteq 2^P$ and $C_i^- \subseteq 2^P$ define sets properties which must resp. must not be simultaneously given; a selection $S \subseteq D$ is a *solution* if for all $(C_i^+, C_i^-) \in C$ we have $C_i^+ \not\subseteq m(S)$ or $m(S) \cap C_i^- \neq \emptyset$.

For the experiment we consider for each size n ten randomly generated instances with n domain elements, $\lfloor \frac{n}{5} + 1 \rfloor$ properties, a random function m, and a random number of constraints, such that their count has an expected value of n.

The results are shown in Table 1. One can observe that for all shown instance sizes, the splitting approach is the slowest, monolithic evaluation is the second-slowest, and tu-propagation is the fastest configuration. A look at the grounding and solving times shows that for the monolithic approach, the main source of computation costs is grounding, while for splitting it is the solving phase (although also the grounding costs are high due to repetitive instantiation of program components). In contrast, with tu-propagation both grounding and solving is fast, while most of the computation time is spent on other computations (most importantly, the inconsistency analysis). The observations are in line with our hypotheses.

Diagnosis Problem. Next, we consider the diagnosis problem $\langle \mathcal{O}_d, \mathcal{O}_p, \mathcal{H}, \mathcal{C}, P \rangle$, where sets $\mathcal{O}_d$ and $\mathcal{O}_p$ are definite resp. potential observations, $\mathcal{H}$ is a set of hypotheses, $\mathcal{C}$ is a set of constraints over the hypotheses, and P is a logic program which defines the observations which follow from given hypotheses. Each constraint $C \in \mathcal{C}$ forbids certain combinations of hypotheses. A solution consists of a set $S_\mathcal{H} \subseteq \mathcal{H}$ of hypotheses and a set of potential observations $S_O \subseteq \mathcal{O}_d$ such that (i) all answer sets of $P \cup H$, which contain all of $S_O \cup \mathcal{O}_d$, contain also $S_\mathcal{H}$, and (ii) $C \not\subseteq S_\mathcal{H}$ for all $C \in \mathcal{C}$. Informally, $S_\mathcal{H}$ are necessary hypotheses to explain the observations.

[5] https://github.com/hexhex/core/tree/master/benchmarks

size	monolithic			splitting			tu-propagation		
	total	ground	solve	total	ground	solve	total	ground	solve
6	0.14 (0)	0.01	0.02	0.18 (0)	0.03	0.04	0.15 (0)	<0.005	0.01
8	0.20 (0)	0.05	0.04	0.49 (0)	0.15	0.20	0.19 (0)	<0.005	0.02
10	0.45 (0)	0.22	0.10	2.17 (0)	0.81	1.11	0.33 (0)	<0.005	0.05
12	1.51 (0)	0.90	0.44	9.50 (0)	3.59	5.17	0.86 (0)	0.01	0.10
14	4.84 (0)	3.68	0.82	44.23 (0)	16.58	24.68	1.48 (0)	0.02	0.20
16	19.52 (0)	16.55	2.08	217.46 (0)	86.17	119.93	3.80 (0)	0.07	0.57
18	143.09 (3)	68.89	10.89	300.00 (10)	122.23	165.44	44.76 (1)	0.43	6.29
20	300.00 (10)	300.00	n/a	300.00 (10)	126.40	162.73	37.44 (0)	0.36	3.44
22	300.00 (10)	300.00	n/a	300.00 (10)	122.41	165.68	224.83 (6)	1.03	11.74

Table 1: Configuration Problem

size	monolithic			splitting			tu-propagation		
	total	ground	solve	total	ground	solve	total	ground	solve
5	0.46 (0)	0.15	0.25	0.16 (0)	0.05	0.02	0.36 (0)	0.04	0.04
10	10.16 (0)	5.94	5.08	2.86 (0)	2.29	0.67	6.38 (0)	1.01	0.89
15	276.19 (5)	258.17	40.11	121.73 (1)	96.76	22.94	105.43 (3)	15.06	13.15
20	300.00 (10)	300.00	n/a	294.97 (9)	253.97	53.55	169.56 (5)	26.54	18.64
25	300.00 (10)	300.00	n/a	300.00 (10)	270.64	45.78	187.90 (5)	29.84	18.08
30	300.00 (10)	300.00	n/a	300.00 (10)	273.50	42.07	226.51 (6)	34.92	20.24
35	300.00 (10)	300.00	n/a	300.00 (10)	276.70	37.01	299.71 (9)	44.17	23.71

Table 2: Diagnosis Problem

As a concrete medical example, definite observations are known symptoms and test results, potential observations are possible outcomes of yet unfinished tests and hypotheses are possible causes (e.g. diseases, nutrition behavior, etc). Constraints exclude certain (combinations of) hypotheses because it is known from anamnesis and the patient's declaration that they do not apply. A solution of the diagnosis problem corresponds to a set of possible observations which, if confirmed by tests, imply certain hypotheses (i.e., medical diagnosis), which can be exploited to perform the remaining tests goal-oriented.

We use 10 random instances for each instance size, where the size is given by the number of observations; observations are definite with a probability of 20% and potential otherwise.

The results are shown in Table 2. Unlike for the previous benchmark, monolithic is slower than splitting. This is because the evaluation of the external source (corresponding to evaluating P) is much more expensive now; since monolithic grounding requires exponentially many evaluations, while during solving this is not necessarily the case, the evaluation costs have a larger impact to monolithic than to splitting. However, as before tu-propagation is clearly the fastest configuration. Again, the observations are in line with our hypotheses.

Analysis of Best-Case Potential. Finally, in order to analyze the potential of our approach in the best case, we use a synthetic program. Our program of size n is as follows:

$$P = \{ dom(1..n).\ in(X) \vee out(X) \leftarrow dom(X).$$
$$someIn \leftarrow in(X).$$
$$r(X) \leftarrow \&diff[dom, out](X).\ \leftarrow r(X), someIn \}$$

It uses a domain of size n and guesses a subset thereof. It then uses an external atom to compute the complement set. The final constraint encodes that the guessed set and the complement must not be nonempty at the same time, i.e., there are only two valid guesses: either all elements are in or all are out.

The results are shown in Table 3. While splitting separates the rules in the third line from the others and must handle each guess independently, the monolithic approach must evaluate the external atom under all possible extensions of out. Both

size	monolithic			splitting			tu-propagation		
	total	ground	solve	total	ground	solve	total	ground	solve
5	3.13 (0)	3.02	<0.005	0.19 (0)	0.04	0.04	0.14 (0)	<0.005	0.01
6	13.59 (0)	13.48	<0.005	0.30 (0)	0.09	0.09	0.15 (0)	<0.005	0.02
7	63.11 (0)	63.00	<0.005	0.54 (0)	0.20	0.21	0.16 (0)	<0.005	0.02
8	281.10 (0)	280.99	<0.005	1.07 (0)	0.44	0.47	0.17 (0)	<0.005	0.03
...	...	...	...	...	...	...	...	...	...
12	300.00 (1)	300.00	n/a	19.97 (0)	9.25	9.83	0.24 (0)	<0.005	0.06
13	300.00 (1)	300.00	n/a	41.02 (0)	19.12	20.23	0.28 (0)	<0.005	0.07
14	300.00 (1)	300.00	n/a	85.38 (0)	39.86	42.48	0.30 (0)	<0.005	0.08
15	300.00 (1)	300.00	n/a	183.04 (0)	84.74	91.64	0.32 (0)	<0.005	0.09

Table 3: Synthetic Set Guessing

approaches are exponential. In contrast, tu-propagation learns for each non-empty guess a constraint which excludes all guesses that set $someIn$ to true; after learning a linear number of such constraints, only the two valid guesses remain.

7 Related Work and Conclusion

Related Work. Previous work on inconsistency analysis was mainly in the context of answer set debugging and faulty systems in general, based on symbolic diagnosis [Reiter, 1987]. For instance, Syrjänen [2006] computes inconsistency explanations in terms of either minimal sets of constraints which need to be removed to regain consistency, or of odd loops (in the latter case the program is called *incoherent*). The realization is based on another program. Also the more general approaches by Brain *et al.* [2007] and Gebser *et al.* [2008] rewrite the program to debug into a meta-program using dedicated control atoms. The goal is to support human users to find reasons for undesired program behavior. Possible queries are, for instance, why a certain assignment is not an answer set. These approaches are based on meta-programming while our approach is based on an analysis of the implication graph. The extension to the nonground case, which is crucial for our application, requires further techniques. The challenges arise from the fact that, unlike the approaches mentioned above, we do not only compute reasons for inconsistency of a fixed program, but rather inconsistency wrt. a set of possible extensions by facts. Another difference concerns the goal of the approaches: while the aforementioned ones aim at answer set debugging and therefore at human-readable explanations of the inconsistency, we aim at an explanation in terms of atoms, which can be transformed into a constraint more easily.

Also related are previous evaluation algorithms for HEX-programs. While the previous state-of-the-art algorithm corresponds to Algorithm 1 using $h_\bot(\Delta, \hat{\mathbf{A}}) = \bot$ for parameter h, alternative algorithms have been developed as well. One of them was presented by Eiter *et al.* [2014b], who used *support sets* [Darwiche and Marquis, 2002] to represent sufficient conditions to make an external atom true. Such support sets are used speed up the compatibility check at (d). However, as this technique shows its benefits only during the solving phase, it does not resolve the grounding issue addressed in this paper. Later, another evaluation approach based on support sets was developed; in contrast to the previous one it compiles away external atoms altogether [Redl, 2017a]. However, the size of the resulting rewritten program strongly depends on the type of external sources and is exponential in general. Thus, the technique is only practical for certain external sources which are known to have a small representation by support sets (which is not the case for the benchmarks discussed in this paper).

Conclusion. Our experiments show that inconsistency analysis is a promising technique for realizing conflict-driven learning over multiple program components. For programs where evaluation as a whole causes a grounding bottleneck (e.g. due to presence of value invention in programs with expanding domains), splitting is useful for efficient grounding but introduces a barrier for propagation and a solving bottleneck. Inconsistency analysis is the basis for combining the advantages of both: splitting for the sake of efficient grounding but still propagating throughout the whole program. The results show a significant, potentially exponential speedup for the addressed class of programs.

An interesting starting point for future work is the generalization of nogood propagation to general, not necessarily inconsistent program components. In this work, typical solver optimizations have been disabled for tu-propagation as they can harm soundness of the algorithm in general; although the other approaches used them and tu-propagation was still the fastest, an important extension is the identification of specific solver optimizations which are compatible with inconsistency analysis and might lead to an even greater improvement. Another starting point for future work is to exploit structural information of the program to further restrict the set of atoms that can have further support by rules that are missing in the current grounding. Since this decision must reasonably be made without computing the full grounding

References

[Biere *et al.*, 2009] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Brain *et al.*, 2007] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging ASP Programs by Means of asp. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR'07), Tempe, AZ, USA*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 31–43. Springer, 2007.

[Clark, 1977] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[Darwiche and Marquis, 2002] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.

[Drescher *et al.*, 2008] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-driven disjunctive answer set solving. In *KR'08*, pages 422–432. AAAI Press, 2008.

[Eiter *et al.*, 2011] Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, and Peter Schüller. Pushing efficient evaluation of hex programs by modular decomposition. In James Delgrande and Wolfgang Faber, editors, *11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, BC, Canada, May 16-19, 2011*, volume 6645 of *LNCS*, pages 93–106. Springer, May 2011.

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

[Eiter *et al.*, 2014b] Thomas Eiter, Michael Fink, Christoph Redl, and Daria Stepanova. Exploiting support sets for answer set programs with external evaluations. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1041–1048. AAAI Press, 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. *Theory and Practice of Logic Programming*, 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.

[Faber *et al.*, 2010] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, In Press, Corrected Proof, 2010. Available online 3 April 2010.

[Gebser *et al.*, 2008] Martin Gebser, Joerg Puehrer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In Dieter Fox and Carla P. Gomes, editors, *AAAI-08/IAAI-08 Proceedings*, pages 448–453, 2008.

[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.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3–4):365–386, 1991.

[Ostrowski and Schaub, 2012] Max Ostrowski and Torsten Schaub. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming*, 2012. To appear.

[Redl, 2017a] Christoph Redl. Efficient evaluation of answer set programs with external sources based on external source inlining. In *Proceedings of the Thirty-First AAAI Conference (AAAI 2017), February 4–9, 2016, San Francisco, California, USA*. AAAI Press, February 2017.

[Redl, 2017b] Christoph Redl. Explaining inconsistency in answer set programs and extensions. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning*, July 2017. To appear.

[Reiter, 1987] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987.

[Syrjänen, 2006] Tommi Syrjänen. Debugging inconsistent answer set programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning*, pages 77–84, Lake District, UK, May 2006.