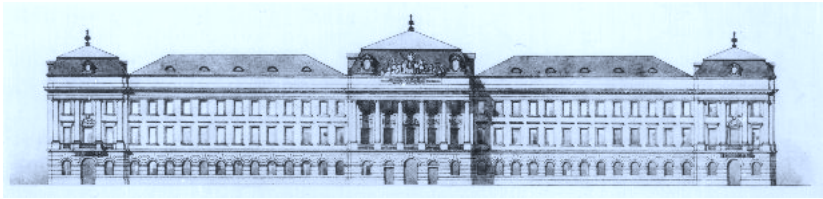
LOGCOMP
RESEARCH
REPORT



INSTITUT FÜR LOGIC AND COMPUTATION

FACHBEREICH WISSENSBASIERTE SYSTEME

# INCONSISTENCY IN ANSWER SET PROGRAMS AND EXTENSIONS

CHRISTOPH REDL

LOGCOMP RESEARCH REPORT 18-04

JUNE 2018

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

LOGCOMP RESEARCH REPORT
LOGCOMP RESEARCH REPORT 18-04, JUNE 2018

# INCONSISTENCY IN ANSWER SET PROGRAMS AND EXTENSIONS

Christoph Redl[1]

**Abstract.** Answer Set Programming (ASP) is a well-known problem solving approach based on nonmonotonic logic programs. HEX-programs extend ASP with *external atoms* for accessing arbitrary external information, which can introduce values that do not appear in the input program. In this work we consider *inconsistent* ASP- and HEX-programs, i.e., programs without answer sets. We study characterizations of inconsistency, introduce a novel notion for explaining inconsistencies in terms of input facts, analyze the complexity of reasoning tasks in context of inconsistency analysis, and present techniques for computing inconsistency reasons. This theoretical work is motivated by two concrete applications, which we also present. The first one is the new modeling technique of *query answering over subprograms* as a convenient alternative to the well-known *saturation technique*. The second application is a new evaluation algorithm for HEX-programs based on *conflict-driven learning for programs with multiple components*: while for certain program classes previous techniques suffer an evaluation bottleneck, the new approach shows significant, potentially exponential speedup in our experiments. Since well-known ASP extensions such as constraint ASP and DL-programs correspond to special cases of HEX, all presented results are interesting beyond the specific formalism.

[1]Institut für Logic and Computation, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: redl@kr.tuwien.ac.at.

**Acknowledgements**: This research has been supported by the Austrian Science Fund (FWF) project P27730.

# Contents

# 1 Introduction

Answer-Set Programming (ASP) is a declarative programming paradigm based on nonmonotonic programs and a multi-model semantics (Gelfond and Lifschitz 1991). The problem at hand is encoded as an ASP-program, which consists of rules, such that its models, called *answer sets*, correspond to the solutions to the original problem. HEX-programs are an extension of ASP with external sources such as description logic ontologies and Web resources (Eiter et al. 2016). So-called external atoms pass information from the logic program, given by predicate extensions and constants, to an external source, which in turn returns values to the program. Notably, *value invention* allows for domain expansion, i.e., external sources might return values which do not appear in the program. 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.

Inconsistent programs are programs which do not possess any answer sets. In this paper we study inconsistent programs: we present a technique for deciding inconsistency of programs by other logic programs, introduce a novel notion for explaining inconsistencies in terms of input facts, analyze the complexity of reasoning tasks in this context, and present techniques for computing inconsistency explanations. The work is motivated by two concrete applications.

The first one is **meta-reasoning** about the answer sets of a *(sub-)program* within another *(meta-)program*, such as aggregation of results. Traditionally, such meta-reasoning requires postprocessing, i.e., the answer sets are inspected after the reasoner terminates. Some simple reasoning tasks, such as brave or cautious query answering, are directly supported by some systems. However, even then the answer to a brave or cautious query is not represented *within* the program but appears only as output on the command-line, which prohibits the direct continuation of reasoning based on the query answer. An existing approach, which allows for meta-reasoning within a program over the answer sets of another program, are *manifold programs*. They compile the calling and the called program into a single one (Faber and Woltran 2011). The answer sets of the called program are then represented within each answer set of the calling program. However, this approach uses weak constraints, which are not supported by all systems. Moreover, the encoding requires a separate copy of the subprogram for each atom occurring in it, which appears to be impractical. Another approach are *nested* HEX*-programs*. Here, dedicated atoms access answer sets of a subprograms and their literals explicitly as accessible objects (Eiter et al. 2013). However, this approach is based on HEX-programs – an extension of ASP – and not applicable if an ordinary ASP solver is used. Our work on inconsistency can be exploited to realize **query answering over subprograms**. We propose this possibility as alternative modeling technique to *saturation* (cf. e.g. Eiter et al. (2009)), which exploits the minimality of answer sets for solving co-NP-hard problems within disjunctive ASP, such as checking if a property holds for *all* objects in a domain. However, the technique is advanced and not easily applicable by average ASP users as using default-negation for checking properties within saturation encodings is restricted.

Later, we extend the study on inconsistency by introducing the **concept of inconsistency reasons (IRs)** for identifying *classes of inconsistent program instances* in terms of *input facts*. The concept is driven by the typical usage of ASP, where the proper rules (IDB; intensional database) are fixed and encode the general problem, while the current instance is specified as facts (EDB; extensional database). It is then interesting to identify sets of instances which lead to inconsistency. The extension is motivated by another application: **optimizing the evaluation algorithm for** HEX**-programs**. Grounding a HEX-program is expensive for certain program classes since in general, already the generation of a single ground instance of a rule requires external sources to be evaluated under up to exponentially many inputs to ensure that all relevant constants are respected (Eiter et al. 2016). 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. 2016). 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 splits act as barriers for propagation. Therefore, for certain program classes, current approaches suffer a bottleneck, which is either due to expensive grounding or due to splitting the guessing part from the checking part. To overcome this bottleneck we propose a novel learning technique for programs with multiple components, which is based on inconsistency reasons.

Previous related work was in context of answer set program debugging and focused on explaining why a *particular interpretation* fails to be an answer set, while we aim at explaining why the overall program is inconsistent. Moreover, debugging approaches also focus on explanations which support the *human user* to find errors in the program. Such reasons can be, for instance, in terms of minimal sets of constraints which need to be removed in order to regain consistency, in terms of odd loops (i.e., cycles of mutually depending atoms which involve negation and are of odd length), or in terms of unsupported atoms, see e.g. Brain et al. (2007) and Gebser et al. (2008) for some approaches. To this end, one usually assumes that a single fixed program is given whose undesired behavior needs to be explained. In contrast, we consider a program whose input facts are subject to change and identify classes of instances which lead to inconsistency.

The organization and contributions of this paper are as follows:

- In Section 2 we present the necessary preliminaries on ASP- and HEX-programs.
- In Section 3 we first discuss restrictions of the saturation technique and point out that using default-negation within saturation encodings would be convenient but is not easily possible. We then show how inconsistency of a normal logic program can be decided within another (disjunctive) program. To this end, we present a saturation encoding which simulates the computation of answer sets of the subprogram and represents the existence of an answer set by a single atom of the meta-program.
- In Section 4 we present our first application, which is meta-reasoning over the answer sets of a subprogram. To this end, we first realize brave and cautious query answering over a subprogram, which can be used as black box such that the user does not need to have deep knowledge about the underlying ideas. Checking a co-NP-complete property can then be expressed naturally by a cautious query.
- In Section 5 we define the novel notion of *inconsistency reasons (IRs)* for HEX-programs wrt. a set of input facts. In contrast to Section 3, which focuses on deciding inconsistency of a single fixed program, we identify classes of program instances which are inconsistent. We then analyze the complexity of reasoning tasks related to the computation of inconsistency reasons.
- In Section 6 we present a meta-programming technique as well as a procedural algorithms for computing IRs for various classes of programs.
- In Section 7 we present our second application, which is a novel evaluation algorithm for HEX-programs based on IRs. To this end, we develop a technique for conflict-driven program solving in presence of multiple program components. We implement this approach in our prototype system and perform an experimental analysis, which shows a significant (potentially exponential) speedup.
- In Section 8 we discuss related work and differences to our approach in more detail.
- In Section 9 we conclude the paper and give an outlook on future work.
- Proofs are outsourced to Appendix A.

A preliminary version of this work has been presented at LPNMR 2017 and IJCAI 2017 (Redl 2017a; Redl 2017d; Redl 2017b). The extensions in this paper comprise of more extensive formalizations and discussions of the theoretical contributions, additional complexity results, and formal proofs of the results.

## 2 Preliminaries

In this section we recapitulate the syntax and semantics of HEX-programs, which generalize (disjunctive) logic programs under the answer set semantics (Gelfond and Lifschitz 1991); for an exhaustive discussion of the background and possible applications we refer to Eiter et al. (2016).

We use as our alphabet the sets $\mathcal{P}$ of predicates, $\mathcal{F}$ of function symbols, $\mathcal{X}$ of external predicates, $\mathcal{C}$ of constants, and $\mathcal{V}$ of variables. We assume that the sets of predicates $\mathcal{P}$ and function symbols $\mathcal{F}$ can share symbols, while the sets are mutually disjoint otherwise. This is by intend and will be needed for our encoding in Section 3. However, it is clear from its position if a symbol is currently used as a predicate or a function symbol. We further let the set of terms $\mathcal{T}$ be the least set such that $\mathcal{C} \subseteq \mathcal{T}$, $\mathcal{V} \subseteq \mathcal{T}$, and whenever $f \in \mathcal{F}$ and $T_1, \ldots, T_\ell \in \mathcal{T}$ then $f(T_1, \ldots, T_\ell) \in \mathcal{T}$.

Each predicate $p \in \mathcal{P}$ has a fixed arity $ar(p) \in \mathbb{N}$. An (ordinary) atom $a$ is of form $p(t_1, \ldots, t_\ell)$, where $p \in \mathcal{P}$ is a predicate symbol with arity $ar(p) = \ell$ and $t_1, \ldots, t_\ell \in \mathcal{T}$ are terms, abbreviated as $p(\mathbf{t})$; for $ar(p) = 0$ we drop parentheses and write $p()$ simply as $p$. For a list of terms $\mathbf{t} = t_1, \ldots, t_\ell$ we write $t \in \mathbf{t}$ if $t = t_i$ for some $1 \leq i \leq \ell$; akin for lists of other objects. We call an atom *ground* if it does not contain variables; similarly for other objects. 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 $\sigma$, i.e. $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$.

An *assignment* $\mathbf{A}$ over a (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 $\mathbf{A} \not\models a$. Assignments $\mathbf{A}$ are called *complete* wrt. $A$, if for every $a \in A$ either $\mathbf{T}a$ or $\mathbf{F}a$ is contained in $\mathbf{A}$, and they are called *partial* otherwise; partial assignments allow for distinguishing false from unassigned atoms, as needed while the reasoner is traversing the search space. Complete assignments $\mathbf{A}$ are also called *interpretations* and, for simplicity, they are also denoted as the set $I = \{a \mid \mathbf{T}a \in \mathbf{A}\}$ of true atoms, while all other atoms are implicitly false (there are no unassigned ones). In Sections 3–5 we will make use of the simplified notation as only complete assignments are relevant, while beginning from Section 6.2 we need partial assignments and use the notation as set of signed literals; we will notify the reader again when we switch the notation.

**HEX-Program 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 parameters from $\mathcal{P}$ or constant parameters[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 $P$ is a set of rules*

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

*where each $a_i$ is an ordinary 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 $B(r) = \{b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n\}$, and its *positive resp. negative body* is $B^+(r) = \{b_1, \ldots, b_m\}$ resp. $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. We let

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

$B_o^+(r)$ resp. $B_o^-(r)$ be the set of ordinary atoms in $B^+(r)$ resp. $B^-(r)$. For a program $P$ we let $X(P) = \bigcup_{r \in P} X(r)$ for $X \in \{H, B, B^+, B^-\}$.

In the following, we call a program *ordinary* if it does not contain external atoms, i.e., if it is a standard ASP-program. Moreover, a rule as by Definition 1 is called *normal* if $k = 1$ and a program is called *normal* if all its rules are normal. A rule $\leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$ (i.e., with $k = 0$) is called a *constraint* and can be seen as normal rule $f \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n, not\ f$ where $f$ is a new atom which does not appear elsewhere in the program.

We will further use *builtin atoms* in rule bodies, which are of kind $o_1 \circ o_2$, where $o_1$ and $o_2$ are constants and $\circ$ is an arithmetic operator (e.g., $<$, $\leq$, $!=$, etc.). More language features (in particular conditional literals) are introduced on-the-fly as needed.

HEX**-Program Semantics** We first discuss the semantics of ground programs $P$. In the following, assignments are always over the set $A(P)$ of ordinary atoms that occur in the program $P$ at hand. The semantics of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an assignment $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary decidable *two-valued (Boolean) oracle function* $f_{\&g}$ that is defined for all possible values of $\mathbf{A}$, $\mathbf{p}$ and $\mathbf{c}$, where $k$ and $l$ are the lengths of $\mathbf{p}$ and $\mathbf{c}$, respectively, and $\mathbf{A}$ must be complete over the ordinary atoms $A(P)$ in the program at hand. We make the restriction that for a fixed complete assignment $\mathbf{A}$ and input $\mathbf{p}$, we have that $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ only for finitely many different vectors $\mathbf{c}$. Thus, $\&g[\mathbf{p}](\mathbf{c})$ is true relative to $\mathbf{A}$, denoted $\mathbf{A} \models \&g[\mathbf{p}](\mathbf{c})$, if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ and false, denoted $\mathbf{A} \not\models \&g[\mathbf{p}](\mathbf{c})$, otherwise.

Satisfaction of ordinary rules and ASP-programs (Gelfond and Lifschitz 1991) is then extended to HEX-rules and -programs as follows. An assignment $\mathbf{A}$ satisfies an atom $a$, denoted $\mathbf{A} \models a$, if $\mathbf{T}a \in \mathbf{A}$, and it does not satisfy it, denoted $\mathbf{A} \not\models a$, otherwise. It satisfies a default-negated atom $not\ a$, denoted $\mathbf{A} \models not\ a$, if $\mathbf{A} \not\models a$, and it does not satisfy it, denoted $\mathbf{A} \not\models not\ a$, otherwise. The truth value of a builtin atom $o_1 \circ o_2$ under $\mathbf{A}$ depends only on $o_1$, $o_2$ and $\circ$ but not on the elements in $\mathbf{A}$ and is defined according to the standard semantics of operators $\circ$; for non-numeric operators this is according to an arbitrary but fixed ordering of constants. A rule $r$ is satisfied under assignment $\mathbf{A}$, denoted $\mathbf{A} \models r$, if $\mathbf{A} \models a$ for some $a \in H(r)$ or $\mathbf{A} \not\models a$ for some $a \in B(r)$. A HEX-program $P$ is satisfied under assignment $\mathbf{A}$, denoted $\mathbf{A} \models P$, if $\mathbf{A} \models r$ for all $r \in P$.

The answer sets of a HEX-program $P$ are defined as follows. Let the *FLP-reduct* of $P$ wrt. an assignment $\mathbf{A}$ be 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 $\mathbf{A}$. Then:

**Definition 2** *A complete assignment $\mathbf{A}$ is an answer set of a* HEX*-program $P$, if $\mathbf{A}$ is a subset-minimal model of $fP^{\mathbf{A}}$ wrt. positive literals in $\mathbf{A}$ (i.e., $\{\mathbf{T}a \in \mathbf{A}\}$).*

**Example 1** *Consider the* HEX*-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 $fP^{\mathbf{A}_1} = \emptyset$.*

For a given ground HEX-program $P$ we let $\mathcal{AS}(P)$ denote the set of all answer sets of $P$.

For ordinary ASP-programs (i.e., HEX-programs without external atoms), the above definition of answer sets based on the FLP-reduct $fP^{\mathbf{A}}$ is equivalent to the original definition of answer sets by Gelfond and Lifschitz (1991) based on the *GL-reduct* $P^{\mathbf{A}} = \{H(r) \leftarrow B^+(r) \mid r \in P, \mathbf{A} \not\models b \text{ for all } b \in B^-(r)\}$. Further note that for a normal ordinary ASP-program $P$ and a complete assignment $\mathbf{A}$, the reduct $P^{\mathbf{A}}$ is always a positive program. This allows for an alternative characterization of answer sets of normal ASP-programs based on fixpoint iteration. For such a program $P$, we let $T_P(S) = \{a \in H(r) \mid r \in P, B^+(r) \subseteq S\}$ be the monotonic *immediate consequence operator*, which derives the consequences of a set $S$ of atoms when applying the positive rules in $P$. Then the least fixpoint of $T_P$ over the empty set, denoted $lfp(T_P)$, is

the *unique* least model of $P$. Hence, an interpretation $I$, denoted as set of true atoms as by our simplified notation, is an answer set of a normal ASP-program $P$ if $I = \mathit{lfp}(T_{P^I})$.

The answer sets of a general, possibly non-ground program $P$ are given by the answer sets $\mathcal{AS}(P) = \mathcal{AS}(grnd_{\mathcal{C}}(P))$ of the *grounding* $grnd_{\mathcal{C}}(P)$ of $P$, which results from $P$ if all variables $\mathcal{V}$ in $P$ are replaced by all constants in $\mathcal{C}$ in all possible ways. In practice, suitable safety criteria guarantee that a finite subset of $\mathcal{C}$ suffices to compute the answer sets (Eiter et al. 2016). For ordinary programs, one can use the (finite) Herbrand universe $HU(P)$ of all constants in $P$ for grounding.

**Saturation Technique** The saturation technique dates back to the $\Sigma_2^{\mathsf{P}}$-hardness proof of disjunctive ASP (Eiter and Gottlob 1995), but was later exploited as a modeling technique, cf. e.g. Eiter et al. (2009). It is applied for solving co-NP-hard problems, which typically involve checking a condition *for all* objects in a certain domain.

To this end, the search space is defined in a program component $P_{guess}$ using disjunctions. Another program component $P_{check}$ checks if the current guess satisfies the property (e.g., being *not* a valid 3-coloring) and derives a dedicated so-called saturation atom $sat$ in this case. A third program component $P_{sat}$ derives all atoms in $P_{guess}$ whenever $sat$ is true, i.e., it *saturates the model*. This has the following effect: if all guesses fulfill the property, all atoms in $P_{guess}$ are derived for all guesses and the so-called *saturation model* $I_{sat} = A(P_{guess} \cup P_{check})$ is an answer set of $P_{guess} \cup P_{check} \cup P_{sat}$; as said above, the complete assignment (interpretation) $I_{sat}$ is denoted as set of true atoms. On the other hand, if there is at least one guess which does not fulfill it, then $sat$ – and possibly further atoms – are not derived. Then, by minimality of answer sets, $I_{sat}$ is not an answer set.

**Example 2** *The program* $P_{non3col} = F \cup P_{guess} \cup P_{check} \cup P_{sat}$ *where*

$$
\begin{aligned}
P_{guess} &= \{r(X) \vee g(X) \vee b(X) \leftarrow node(X)\} \\
P_{check} &= \{sat \leftarrow c(X), c(Y), edge(X,Y) \mid c \in \{r,g,b\}\} \\
P_{sat} &= \{c(X) \leftarrow node(X), sat \mid c \in \{r,g,b\}\}
\end{aligned}
$$

*has the answer set* $I_{sat} = A(P_{non3col})$ *iff the graph specified by facts* $F$ *is not 3-colorable. Otherwise its answer sets are proper subsets of* $I_{sat}$ *which represent valid 3-colorings.*

Importantly, such a check *cannot* be encoded as a *normal* logic program in such a way that the program has an answer set iff the condition holds for all guesses (unless $NP = coNP$). Instead, one can only write a normal program which has *no* answer set if the property holds for all guesses and non-existence of answer sets needs to be determined in the postprocessing. For instance, reconsider non-3-colorability and the following program

$$
\begin{aligned}
P_{3col} = F \cup \{&c_1(X) \leftarrow node(X), not\ c_2(X), not\ c_3(X) \mid \{c_1,c_2,c_3\} = \{r,g,b\} \\
&\leftarrow c(X), c(Y), edge(X,Y) \mid c \in \{r,g,b\}\},
\end{aligned}
$$

where the graph is supposed to be defined by facts $F$ over predicates $node(\cdot)$ and $edge(\cdot,\cdot)$. Its answer sets correspond one-to-one to valid 3-colorings. However, in contrast to the program based on the saturation technique from Example 2, it does *not* have an answer set if there is no valid 3-coloring. For complexity reasons, it is not possible to define a normal program with an answer set that represents that there is *no* such coloring (under the usual assumptions about complexity). This limitation inhibits that reasoning continues *within* the program after checking the property.

# 3 Deciding Inconsistency of Normal Programs in Disjunctive ASP

In this section we first discuss restrictions of the saturation technique concerning the usage of default-negation within the checking part. These restrictions make it sometimes difficult to apply the technique even if complexity considerations imply that it is applicable in principle. In such cases, the technique appears to be inconvenient, especially for standard ASP users who are less experienced with saturation encodings. After discussing the restrictions using some examples, we present an encoding for deciding inconsistency of a normal ASP-program within disjunctive ASP. Different from the one presented by Eiter and Polleres (2006), ours uses conditional literals, which make it conceptually simpler. Moreover, it can also handle non-ground programs.

## 3.1 Restrictions of the Saturation Technique

For complexity reasons, any problem in co-NP can be polynomially reduced to brave reasoning over disjunctive ASP (the latter is $\Sigma_2^{\mathsf{P}}$-complete (Faber et al. 2011)), but the reduction is not always obvious. In particular, the saturation technique is difficult to apply if the property to check cannot be easily expressed without default-negation. Intuitively, this is because saturation works only if $I_{sat}$ is guaranteed to be an answer set of $P_{guess} \cup P_{check} \cup P_{sat}$ whenever no proper subset is one. While this is guaranteed for positive $P_{check}$, interpretation $I_{sat}$ might be unstable otherwise.

**Example 3** *A* vertex cover *of a graph $\langle V, E\rangle$ is a subset $S \subseteq V$ of its nodes s.t. each edge in $E$ is incident with at least one node in $S$. Deciding if a graph has* no *vertex cover $S$ with size $|S| \leq k$ for some integer $k$ is co-*NP*-complete. Consider $P_{vc}$ consisting of facts $F$ over $node$ and $edge$ and the following parts:*

$$
\begin{aligned}
P_{guess} &= \{in(X) \vee out(X) \leftarrow node(X)\} \\
P_{check} &= \{sat \leftarrow edge(X,Y), not\ in(X), not\ in(Y); \\
&\qquad sat \leftarrow in(X_1), \ldots, in(X_{k+1}), X_1 \neq X_2, \ldots, X_k \neq X_{k+1}\} \\
P_{sat} &= \{in(X) \leftarrow node(X), sat;\ out(X) \leftarrow node(X), sat\}
\end{aligned}
$$

*Program $P_{guess}$ guesses a candidate vertex cover $S$, $P_{check}$ derives $sat$ whenever for some edge $(u, v) \in E$ neither $u$ nor $v$ is in $S$ (thus $S$ is invalid), and $P_{sat}$ saturates in this case.*

Observe that for inconsistent instances $F$ (e.g. $\langle\{a, b, c, d\}, \{(a, b), (b, c), (c, d)\}\rangle$ with $k = 1$), this encoding does not work as desired because model $I_{sat} = A(P_{vc})$ is unstable. More specifically, the instances of the first rule of $P_{check}$ are eliminated from the reduct $fP_{vc}^{I_{sat}}$ of $P_{vc}$ wrt. $I_{sat}$ due to default-negation. But then, the least model of the reduct does not contain $sat$ or any atom $in(\cdot)$. Then, $I_< = I_{sat} \setminus (\{sat\} \cup \{in(x) \mid x \in V\})$ is a smaller model of the reduct and $I_{sat}$ is not an answer set of $P_{vc} \cup F$.

In this example, the problem may be fixed by replacing literals $not\ in(X)$ and $not\ in(Y)$ by $out(X)$ and $out(Y)$, respectively. That is, instead of checking if a node is not in the vertex cover, one explicitly checks if it is out. However, the situation is more cumbersome if default-negation is not directly applied to the guessed atoms but to derived ones, as the following example demonstrates.

**Example 4** *A* Hamiltonian cycle *in a directed graph $\langle V, E\rangle$ is a cycle that visits each node in $V$ exactly once. Deciding if a given graph has a Hamiltonian cycle is a well-known* NP*-complete problem; deciding if a graph does not have such a cycle is therefore co-*NP*-complete. A natural attempt to solve the problem*

*using saturation is as follows:*

$$P_{guess} = \{in(X,Y) \vee out(X,Y) \leftarrow arc(X,Y)\} \quad (1)$$

$$P_{check} = \{sat \leftarrow in(Y_1,X), in(Y_2,X), Y1 \neq Y2;\ sat \leftarrow in(X,Y_1), in(X,Y_2), Y1 \neq Y2 \quad (2)$$

$$sat \leftarrow node(X), not\ hasIn(X);\ sat \leftarrow node(X), not\ hasOut(X) \quad (3)$$

$$hasIn(X) \leftarrow node(X), in(Y,X);\ hasOut(X) \leftarrow node(X), in(X,Y)\} \quad (4)$$

$$P_{sat} = \{in(X,Y) \leftarrow arc(X,Y), sat;\ out(X,Y) \leftarrow arc(X,Y), sat\} \quad (5)$$

*Program $P_{guess}$ guesses a candidate Hamiltonian cycle as a set of arcs. Program $P_{check}$ derives $sat$ whenever some node in $V$ does not have exactly one incoming and exactly one outgoing arc, and $P_{sat}$ saturates in this case. The check is split into two checks for at most (rules (2)) and at least (rules (3)) one incoming/outgoing arc. While the check if a node has at most one incoming/outgoing arc is possible using the positive rules (2), the check if a node has at least one incoming/outgoing edge is more involved. In contrast to the check in Example 3, one cannot perform it based on the atoms from $P_{guess}$ alone. Instead, auxiliary predicates $hasIn$ and $hasOut$ are defined by rules (4). Unlike $in(\cdot,\cdot)$, the negation of $hasIn(\cdot)$ and $hasOut(\cdot)$ is not explicitly represented, thus default-negation is used in rules (3) of $P_{check}$. However, this harms stability of $I_{sat}$: the graph $\langle\{a,b,c\},\{(a,b),(b,a),(b,c),(c,b)\}\rangle$, which does not have a Hamiltonian cycle, causes $P_{guess} \cup P_{check} \cup P_{sat}$ to be inconsistent. This is due to default-negation in $P_{check}$, which eliminates rules (3) from the reduct wrt. $I_{sat}$, which has in turn a smaller model.*

Note that in the previous example, for a fixed node $X$, the literal $not\ hasOut(X)$ is used to determine if all atoms $in(X,Y)$ are false (or equivalently: if all atoms $out(X,Y)$ are true). Here, default-negation can be eliminated on the ground level by replacing rule $sat \leftarrow node(X), not\ hasOut(X)$ by $sat \leftarrow node(x), out(x,y_1), \ldots, out(x,y_n)$ for all nodes $x \in V$ and all nodes $y_i$ for $1 \leq i \leq n$ such that $(x,y_i) \in E$.[2] But even this is not always possible, as shown with the next example.

**Example 5** *Deciding if a ground normal ASP-program $P$ is inconsistent is co-*NP*-complete. An attempt to apply the saturation technique is as follows:*

$$P' = \{true(a) \vee false(a) \mid a \in A(P)\} \quad (6)$$

$$\cup \{inReduct(r) \leftarrow \{false(b) \mid b \in B^-(r)\} \mid r \in P\} \quad (7)$$

$$\cup \{leastModel(a) \leftarrow inReduct(r), \{leastModel(b) \mid b \in B^+(r)\} \mid r \in P, a \in H(r)\} \quad (8)$$

$$\cup \{noAS \leftarrow false(a), leastModel(a) \mid a \in A(P)\} \quad (9)$$

$$\cup \{noAS \leftarrow true(a), not\ leastModel(a) \mid a \in A(P)\} \quad (10)$$

$$\cup \{true(a) \leftarrow noAS;\ false(a) \leftarrow noAS \mid a \in A(P)\} \quad (11)$$

$$\cup \{inReduct(r) \leftarrow noAS\} \quad (12)$$

*The idea is to guess all possible interpretations $I$ over the atoms $A(P)$ in $P$ (rules (6)). Next, rules (7) identify the rules $r \in P$ which are in $P^I$ (modulo $B^-(r)$)[3]; these are all rules $r \in P$ whose atoms $B^-(r)$ are all false. Rules (8) compute the least model of the reduct by simulating fixpoint iteration under operator $T_P$ as shown in Section 2. Rules (9) and (10) compare the least model of the reduct to $I$: if this comparison*

[2] On the non-ground level, this might be simulated using *conditional literals* as supported by some reasoners, cf. Gebser et al. (2012) and below.

[3] We can use the GL-reduct here as $P$ is an ordinary program.

*fails, then $I$ is not an answer set and rules (11) and (12) saturate. While the comparison of the least model of the reduct to the original guess in rule (10) is natural, it uses default-negation and destroys stability of $I_{sat}$ in general. In contrast to Example 4, eliminating the negation it is not straightforward, not even on the ground level.*

We conclude that some problems involve checks which can easily be expressed using negation, but such a check within a saturation encoding may harm stability of the saturation model. In the next subsection, we present a valid encoding for checking inconsistency of normal programs (cf. Example 5) within disjunctive ASP. Subsequently, we exploit this encoding for query answering over normal logic programs within disjunctive ASP in the next section.

### 3.2 A Meta-Program for Propositional Programs

We reduce the check for inconsistency of a normal logic program $P$ to brave reasoning over a disjunctive meta-program. The major part $M$ of the meta-program is static and consists of proper rules which are independent of $P$. The concrete program $P$ is then specified by facts $M^P$ which are added to the static part. The overall program $M \cup M^P$ is constructed such that it is consistent for all $P$ and its answer sets either represent the answer sets of $P$, or a distinguished answer set represents that $P$ is inconsistent.

In this subsection we restrict the discussion to ground programs $P$. Moreover, we assume that all predicates in $P$ are of arity 0. This is w.l.o.g. because any atom $p(t_1, \ldots, t_\ell)$ can be replaced by an atom consisting of a new predicate $p'$ without any parameters. In the meta-program defined in the following, we let all atoms be new atoms which do not occur in $P$. We further use each rule $r \in P$ also as a constant in the meta-program.

The static part consists of component $M_{extract}$ for the extraction of various information from the program encoding $M^P$, which we call $M^P_{gr}$ in this subsection to stress that $P$ must be ground, and a saturation encoding $M_{guess} \cup M_{check} \cup M_{sat}$ for the actual inconsistency check. We first show the complete encoding and discuss its components subsequently.

**Definition 3** *We define the* meta-program $M = M_{extract} \cup M_{guess} \cup M_{check} \cup M_{sat}$*, where:*

$$M_{extract} = \{atom(X) \leftarrow head(R,X);\ atom(X) \leftarrow bodyP(R,X);\ atom(X) \leftarrow bodyN(R,X)\} \tag{13}$$

$$\cup \{rule(R) \leftarrow head(R,X);\ rule(R) \leftarrow bodyP(R,X);\ rule(R) \leftarrow bodyN(R,X)\} \tag{14}$$

$$M_{guess} = \{true(X) \vee false(X) \leftarrow atom(X)\} \tag{15}$$

$$M_{check} = \{inReduct(R) \leftarrow rule(R), (false(X) : bodyN(R,X)) \tag{16}$$

$$outReduct(R) \leftarrow rule(R), bodyN(R,X), true(X) \tag{17}$$

$$derivationSeq(X,Y) \vee derivationSeq(Y,X) \leftarrow true(X), true(Y), X \neq Y \tag{18}$$

$$derivationSeq(X,Z) \leftarrow derivationSeq(X,Y), derivationSeq(Y,Z) \tag{19}$$

$$derivationSeq(X_1,X_2) \leftarrow head(R,X_1), (derivationSeq(Y,X_1) : bodyP(R,Y)), \tag{20}$$

$$atom(X_2), (derivationSeq(Y,X_2) : bodyP(R,Y)), X_2 > X_1$$

$$notApp(R) \leftarrow outReduct(R) \tag{21}$$

$$notApp(R) \leftarrow inReduct(R), bodyP(R,X), false(X) \tag{22}$$

$$notApp(R) \leftarrow head(R, X_1), bodyP(R, X_2), derivationSeq(X_1, X_2) \quad (23)$$

$$noAS \leftarrow true(X), (notApp(R) : head(R, X)) \quad (24)$$

$$noAS \leftarrow inReduct(R), head(R, X), false(X), (true(Y) : bodyP(R, Y))\} \quad (25)$$

$$M_{sat} = \{true(X) \leftarrow atom(X), noAS;\ false(X) \leftarrow atom(X), noAS \quad (26)$$

$$derivationSeq(X, Y) \leftarrow atom(X), atom(Y), noAS \quad (27)$$

$$inReduct(R) \leftarrow rule(R), noAS;\ outReduct(R) \leftarrow rule(R), noAS\} \quad (28)$$

This encoding is to be extended by the program-dependent part $M^P_{gr}$. Each rule of $P$ is represented by atoms of form $head(r, a)$, $bodyP(r, a)$, and $bodyN(r, a)$, where $r$ is a rule from $P$ (used as new constant representing the respective rule), and $head(r, a)$, $bodyP(r, a)$ and $bodyN(r, a)$ denote that $a$ is an atom that occurs in the head, positive and negative body of rule $r$, respectively. For the following formalization we assume that in $P$, constraints of form $\leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$ have already written to normal rules of form $f \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n, not\ f$, where $f$ is a new ground atom which does not appear elsewhere in the program, as discussed in Section 2.

**Definition 4** *For a ground normal logic program $P$ we let:*

$$M^P_{gr} = \{head(r, h) \mid r \in P, h \in H(r)\}$$
$$\cup \{bodyP(r, b) \mid r \in P, h \in B^+(r)\} \cup \{bodyN(r, b) \mid r \in P, h \in B^-(r)\}$$

We come now to the explanation of the static part $M$ of the meta-program. The component $M_{extract}$ uses rules (13) and (14) to extract from $M^P_{gr}$ the sets of rules and atoms in $P$.

The structure of the components $M_{guess}$, $M_{check}$ and $M_{sat}$ follows the basic architecture of saturation encodings presented in Section 3. Program $M_{guess}$ uses rule (15) to guess an answer set candidate $I$ of program $P$, $M_{check}$ simulates the computation of the GL-reduct $P^I$ and checks if its least model coincides with $I$, and $M_{sat}$ saturates the model whenever this is *not* the case.[4] If all guesses fail to be answer sets, then every guess leads to saturation and the saturation model is an answer set. On the other hand, if at least one guess represents a valid answer set of $P$, then the saturation model is not an answer set due to subset-minimality. Hence, $M \cup M^P$ has exactly one (saturated) answer set if $P$ is inconsistent, and it has answer sets which are not saturated if $P$ is consistent, but none of them contains $noAS$.

We turn to the checking part $M_{check}$. Rules (16) and (17) compute for the current candidate $I$ the rules in $P^I$: a rule $r$ is in the reduct iff all atoms from $B^-(r)$ are false in $I$. Here, $(false(X) : bodyN(R, X))$ is a *conditional literal* which extends the basic language from Section 2 and evaluates to true iff $false(X)$ holds for all $X$ such that $bodyN(R, X)$ is true, i.e., all atoms in the negative body are false (Gebser et al. 2012). Rules (18)–(20) simulate the computation of the least model $lfp(T_{P^I})$ of $P^I$ using fixpoint iteration. To this end, rule (18) guesses a derivation sequence over the true atoms $a \in I$ during fixpoint iteration under $T_P$. More precisely, an atom $derivationSequence(a, b)$ denotes that atom $a$ is derived before atom $b$; since this is a transitive property, rule (19) compute the closure. Although $T_P$ may derive multiple atoms in the same iteration, we guess a strict sequence here. Since this may lead to repetitive solutions as all permutations of atoms derived in the same iteration are valid, rule (20) enforces atoms to be derived in lexicographical order whenever possible. More precisely, after satisfaction of the positive body $B^+(r)$ of a rule $r$, the head atom $H(r)$ must be derived before any lexicographically larger atom can be derived.

[4] Recall that the GL-reduct is equivalent to the FLP-reduct since $P$ is an ordinary ASP-program.

Next, we check if the current interpretation $I$ is *not* an answer set of $P$ which can be justified by the guessed derivation sequence, and derive $noAS$ in this case. Importantly, $noAS$ is derived both if (i) $I$ is not an answer set at all, and if (ii) $I$ *is* an answer set, but one that cannot be justified using the guessed derivation sequence (i.e., the derivation sequence is invalid). As a preparation for this check, rules (21)–(23) determine the rules $r \in P$ which are *not* applicable in the fixpoint iteration (wrt. the current derivation sequence) to justify their head atom $H(r)$ being true. A rule is not applicable if it is not even in the reduct (rule (21)), if at least one positive body atom is false (rule (22)), or if it has a positive body atom which is derived in a later iteration (rule (23)) because then the rule cannot fire (yet) in the iteration the head atom was guessed to be derived. We can then perform the actual check as follows. Rule (24) checks if all atoms in $I$ are derived by some rule in $P^I$ (i.e., $I \subseteq \mathit{lfp}(T_{P^I})$). Conversely, rule (25) checks if all atoms derived by some rule in $P^I$ are also in $I$ (i.e., $I \supseteq \mathit{lfp}(T_{P^I})$). Overall, the rules (24)–(25) check if $I = \mathit{lfp}(T_{P^I})$, and derive $noAS$ if this is not the case.

Finally, the saturation part in rules (26)–(28) derive all atoms in the program whenever the guess does not represent a valid answer set of $P$.

One can show that atom $noAS$ correctly represents the (in)consistency of $P$, as formalized by the next proposition.

**Proposition 1** *For any ground normal logic program $P$, we have that*
*(1) if $P$ is inconsistent, $M \cup M^P_{gr}$ has exactly one answer set which contains $noAS$; and*
*(2) if $P$ is consistent, $M \cup M^P_{gr}$ has at least one answer set and none of them contains $noAS$.*

### 3.3 A Meta-Program for Non-Ground Programs

We extend the encoding of a ground normal logic program as facts as by Definition 4 to non-ground programs. The program-specific part is called $M^P_{ng}$ to stress that $P$ can now be non-ground (but also ground, which we consider a special case of non-ground programs). In the following, for a rule $r$ let $\mathbf{V}_r$ be the vector of unique variables occurring in $r$ in the order of appearance.

The main idea of the following encoding is to interpret atoms as function terms. That is, for an atom $p(t_1, \ldots, t_\ell)$ we see $p$ as function symbol rather than predicate (recall that Section 2 allows that $\mathcal{P}$ and $\mathcal{F}$ to share elements). Then, atoms, interpreted as function terms, can occur as parameters of other atoms. As before, we assume that constraints in $P$ have been rewritten to normal rules:

**Definition 5** *For a (ground or non-ground) normal logic program $P$ we let:*

$$\begin{aligned} M^P_{ng} = &\{\mathit{head}(r(\mathbf{V}_r), h) \leftarrow \{\mathit{head}(R_d, d) \mid d \in B^+(r)\} \mid r \in P, h \in H(r)\}\\ &\cup \{\mathit{bodyP}(r(\mathbf{V}_r), b) \leftarrow \{\mathit{head}(R_d, d) \mid d \in B^+(r)\} \mid r \in P, b \in B^+(r)\}\\ &\cup \{\mathit{bodyN}(r(\mathbf{V}_r), b) \leftarrow \{\mathit{head}(R_d, d) \mid d \in B^+(r)\} \mid r \in P, b \in B^-(r)\} \end{aligned}$$

For each possibly non-ground rule $r \in P$, we construct a unique identifier $r(\mathbf{V}_r)$ for each ground instance of $r$. It consists of $r$, used as a unique function symbol to identify the rule, and all variables in $r$ as parameters. As for the ground case, the head, the positive and the negative body are extracted from $r$. For ensuring safety, we add a *domain atom* $\mathit{head}(R_d, d)$ for all positive body atoms $d \in B^+(r)$ to the body of the rule in the meta-program in order to instantiate it with all derivable ground instances. More precisely, for each positive body atom $d$ of the current rule, we use $R_d$ as a variable and add atom $\mathit{head}(R_d, d)$, to represents all (other) program rules of $P$ that instantiate the variables in atom $d$. This creates an instance of $r$ for all variable substitutions such that all body atoms of the instance are potentially derivable in the meta-program.

**Example 6** *Let* $P = \{f\colon d(a);\ r_1\colon q(X) \leftarrow d(X), not\, p(X);\ r_2\colon p(X) \leftarrow d(X), not\, q(X)\}$. *We have:*

$$M^P_{ng} = \{head(f, d(a)) \leftarrow \tag{29}$$
$$head(r_1(X), q(X)) \leftarrow head(R_{d(X)}, d(X)) \tag{30}$$
$$bodyP(r_1(X), d(X)) \leftarrow head(R_{d(X)}, d(X)) \tag{31}$$
$$bodyN(r_1(X), p(X)) \leftarrow head(R_{d(X)}, d(X)) \tag{32}$$
$$head(r_2(X), p(X)) \leftarrow head(R_{d(X)}, d(X)) \tag{33}$$
$$bodyP(r_2(X), d(X)) \leftarrow head(R_{d(X)}, d(X)) \tag{34}$$
$$bodyN(r_2(X), q(X)) \leftarrow head(R_{d(X)}, d(X))\} \tag{35}$$

*We explain the encoding with the example of* $r_1$. *Since* $r_1$ *is non-ground, it may represent multiple ground instances, which are determined by the substitutions of* $X$. *We use* $r_1(X)$ *as identifier and define that, for any substitution of* $X$, *atom* $q(X)$ *appears in the head (cf. rule (30)),* $d(X)$ *in the positive body (cf. rule (31)) and* $p(X)$ *in the negative body (cf. rule (32)). The domain of* $X$ *is defined by all atoms* $d(X)$ *which are potentially derivable by any rule (identified by variable* $R_{d(X)}$*), i.e., which occur in the head of a rule. We encode this by atom* $head(R_{d(X)}, d(X))$ *in the bodies of the rules (30)–(32). The encoding works similar for* $f$, *cf. rule (29) and* $r_2$, *cf. rules (33)–(35).*

One can show that the encoding is still sound and complete for non-ground programs:

**Proposition 2** *For any normal logic program* $P$, *we have that*
*(1) if* $P$ *is inconsistent,* $M \cup M^P_{ng}$ *has exactly one answer set which contains* $noAS$; *and*
*(2) if* $P$ *is consistent,* $M \cup M^P_{ng}$ *has at least one answer set and none of them contains* $noAS$.

# 4 Application: Query Answering over Subprograms

We now present a language extension with dedicated *query atoms* which allow for answering queries over a subprogram within another program and accessing their results. This is intended to be a more convenient alternative to the saturation technique. Afterwards we show how the language feature can be reduced to our encoding for deciding inconsistency from the previous section. Finally we demonstrate this language extension with an example.

Note that our approach focuses on simplicity (from user's perspective) and easy usability for typical use cases rather than high expressibility. Other approaches may have higher expressibility, but this comes at the price of a more complex semantics as e.g. in the approach by Bogaerts et al. (2016); we will discuss related approaches in more detail in Section 8.

## 4.1 Programs with Query Atoms

In the following, a ground query $q$ is a set of ground literals (atoms or default-negated atoms) interpreted as conjunction. For an atom or default-negated atom $l$, let $\bar{l}$ be its negation, i.e., $\bar{l} = a$ if $l = not\, a$ and $\bar{l} = not\, a$ if $l = a$. We say that an interpretation $I$ satisfies a query $q$, denoted $I \models q$, if $I \models l$ for all $l \in q$. A HEX-program $P$ *bravely entails* a query $q$, denoted $P \models_b q$, if $I \models q$ for some answer set $I$ of $P$; it *cautiously entails* a query $q$, denoted $P \models_c q$, if $I \models q$ for all answer sets $I$ of $P$.

A ground query $q$ over a subprogram $S$, possibly extended with input from the calling program, is then formalized as follows; in implementations, $S$ may be specified by its filename.

**Definition 6** *A* query atom *is of form $S(\mathbf{p}) \vdash_t q$, where $t \in \{b, c\}$ determines the type of the query, $S$ is a normal ordinary ASP-(sub-)program, $\mathbf{p}$ is a vector of predicates which specify the input, and $q$ is a ground query.*

We allow such query atoms to occur in bodies of general HEX-programs in place of ordinary atoms. Similar to Definition 1 we can then define:

**Definition 7** *A* HEX-program with query atoms *is a set of rules*

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

*where each $a_i$ is an ordinary atom and each $b_j$ is either an ordinary atom, an external atom or a query atom.*

The semantics as introduced in Section 2 is then extended to HEX-programs with query atoms as follows. An interpretation $I$ satisfies a query atom $S(\mathbf{p}) \vdash_t q$, denoted $I \models S(\mathbf{p}) \vdash_t q$, if $S \cup \{p(\mathbf{c}) \leftarrow \mid p \in \mathbf{p}, p(\mathbf{c}) \in I\} \models_t q$. The definitions of modelhood and of answer sets are then as for ordinary ASP-programs.

## 4.2 Reducing Query Answering to Inconsistency Checking

Next, we show how programs with query atoms can be reduced to standard ASP-programs using our encoding for inconsistency checking. The reduction is based on the following observation:

**Proposition 3** *For a normal ASP-program $P$ and a query $q$ we have that (1) $P \models_b q$ iff $P \cup \{\leftarrow \bar{l} \mid l \in q\}$ is consistent; and (2) $P \models_c q$ iff $P \cup \{\leftarrow q\}$ is inconsistent.*

This observation together with Proposition 1 implies the following result.

**Proposition 4** *For a normal ASP-program $P$ be a predicate and query $q$ we have that*
*(1) $M \cup M_{ng}^{P \cup \{\leftarrow \bar{l} | l \in q\}}$ is consistent and each answer set contains $noAS$ iff $P \not\models_b q$; and*
*(2) $M \cup M_{ng}^{P \cup \{\leftarrow q\}}$ is consistent and each answer set contains $noAS$ iff $P \models_c q$.*

The idea of evaluation based reduction is now to apply Proposition 4 to each query atom in the program at hand. That is, for each query atom $S(\mathbf{p}) \vdash_t q$ we create a copy of $M \cup M_{ng}^{S \cup \{\leftarrow \bar{l} | l \in q\}}$ if $t = b$ and of $M \cup M_{ng}^{S \cup \{\leftarrow q\}}$ if $t = c$. Importantly, each copy must use a different name space of atoms, which is realized by adding $S(\mathbf{p}) \vdash_t q$ as an index to every atom. By Proposition 2 we can then use the atom $noAS_{S(\mathbf{p}) \vdash_t q}$ to check consistency of the respective copy of $S \cup \{\leftarrow \bar{l} \mid l \in q\}$ resp. $M \cup M_{ng}^{S \cup \{\leftarrow q\}}$, which corresponds by Proposition 4 to the answer to the query $S \models_t q$.

Formally we define the following translation, where $\mathbf{X}^{ar(p)}$ denotes a vector of variables of length $ar(p)$:

$$[P] = P|_{S(\mathbf{p}) \vdash_c q / noAS_{S(\mathbf{p}) \vdash_c q}, S(\mathbf{p}) \vdash_b q / not\ noAS_{S(\mathbf{p}) \vdash_b q}}|_{not\ not\ a/a} \tag{36}$$

$$\cup \bigcup_{S(\mathbf{p}) \vdash_b q \text{ in } P} \left(M \cup M_{ng}^{S \cup \{\leftarrow \bar{l} | l \in q\}}\right)\big|_{a/a_{S(\mathbf{p}) \vdash_b q}} \cup \bigcup_{S(\mathbf{p}) \vdash_c q \text{ in } P} \left(M \cup M_{ng}^{S \cup \{\leftarrow q\}}\right)\big|_{a/a_{S(\mathbf{p}) \vdash_c q}} \tag{37}$$

$$\cup \bigcup_{S(\mathbf{p}) \vdash_t q \text{ in } P,\ p \in \mathbf{p}} \left(\{head(r_p(\mathbf{X}^{ar(p)}), p(\mathbf{X}^{ar(p)})) \leftarrow p(\mathbf{X}^{ar(p)})\}\right)|_{a/a_{S(\mathbf{p}) \vdash_t q}} \tag{38}$$

Here, the program part in line (36) denotes program $P$ after first replacing every query atom of kind $S(\mathbf{p}) \vdash_c q$ (for some $S$, $\mathbf{p}$ and $q$) by the new ordinary atom $noAS_{S(\mathbf{p})\vdash_c q}$ and every query atom of kind $S(\mathbf{p}) \vdash_b q$ (for some $S$, $\mathbf{p}$ and $q$) by the new ordinary literal $not\ noAS_{S(\mathbf{p})\vdash_b q}$, and then eliminating double negation (which may be introduced by the previous replacement if in the program $P$ a query atom with a brave query appears in the default-negated part of a rule). This part of the rewriting makes sure that the $noAS$ atoms of our encoding from the previous section is accessed in place of the original query atoms following Proposition 4.

Next, the program part in line (37) defines these $noAS$ atoms using the encoding from the previous section. Here, the expression $|_{a/a_{S(\mathbf{p})\vdash_b q}}$ denotes the program after replacing each atom $a$ by $a_{S(\mathbf{p})\vdash_b q}$ (likewise for $S(\mathbf{p}) \vdash_c q$). This realizes different namespaces, i.e., for every query atom $S(\mathbf{p}) \vdash_b q$ resp. $S(\mathbf{p}) \vdash_c q$ in $P$, a separate copy of $M$ and $M_{ng}^{S\cup\{\leftarrow\bar{l}|l\in q\}}$ resp. $M_{ng}^{S\cup\{\leftarrow q\}}$ with disjoint vocabularies is generated. Then each such copy uses also a separate atom $noAS_{S(\mathbf{p})\vdash_b q}$ resp. $noAS_{S(\mathbf{p})\vdash_c q}$ which represents by Proposition 4 the answer to query $q$ as expected by the part in line (36).

Finally, we must add the input facts to $S$. To this end, sets $M_{ng}^{S\cup\{\leftarrow q\}}$ resp. $M_{ng}^{S\cup\{\leftarrow\bar{l}|l\in q\}}$ must be extended. For each input predicate $p \in \mathbf{p}$, each atom of kind $p(\cdots)$ must be added as additional rule $p(\cdots) \leftarrow$, which is encoded akin to Definition 5. This is realized in line (38), which translates the values of $p$-atoms of the calling program to facts in the subprogram.

One can formally show that $[P]$ resembles the semantics of programs with query atoms, as by Definition 7:

**Proposition 5** *For a logic program $P$ with query atoms we have that $\mathcal{AS}(P)$ and $\mathcal{AS}([P])$, projected to the atoms in $P$, coincide.*

Note that while the construction of $[P]$ may not be trivial, this does not harm usability from user's perspective. This is because the rewriting concerns only the implementer of a reasoner for programs with queries over subprograms, while the user can simply use query atoms.

### 4.3 Using Query Atoms

Query answering over subprograms can now be exploited as a modeling technique to check a criterion for all objects in a domain. As observed in Section 3, saturation may fail in cases where the check involves default-negation. Moreover, saturation is an advanced technique which might be not intuitive for less experienced ASP users (it was previously called 'hardly usable by ASP laymen' (Gebser et al. 2011)). Thus, even for problems whose conditions can be expressed by positive rules, an encoding based on query answering might be easier to understand. To this end, one starts with a program $P_{guess}$ which spans a search space of all objects to check. As with saturation, $P_{check}$ checks if the current guess satisfies the criteria and derives a dedicated atom $ok$ in this case. However, instead of saturating the interpretation whenever $ok$ is true, one now checks if $ok$ is cautiously entailed by $P_{guess}\cup P_{check}$ using a rule of kind $allOk \leftarrow P_{guess}\cup P_{check} \vdash_c ok$. Importantly, the user does not need to deal with saturation and can use default-negation in the checking part $P_{check}$.

**Example 7** *Recall Example 4 where we want to check if a graph does not possess a Hamiltonian cycle.*

*Consider the following programs:*

$$P_{guess} = \{in(X,Y) \vee out(X,Y) \leftarrow arc(X,Y)\} \tag{39}$$

$$P_{check} = \{hasIn(X) \leftarrow node(X), in(Y,X);\ hasOut(X) \leftarrow node(X), in(X,Y)\} \tag{40}$$

$$invalid \leftarrow node(X), not\ hasIn(X);\ invalid \leftarrow node(X), not\ hasOut(X) \tag{41}$$

$$invalid \leftarrow in(Y_1,X), in(Y_2,X), Y_1 \neq Y_2;\ invalid \leftarrow in(X,Y_1), in(X,Y_2), Y1 \neq Y2 \tag{42}$$

$$P = \{noHamiltonian \leftarrow P_{guess} \cup P_{check} \cup F \vdash_c invalid\} \tag{43}$$

*Assume that $F$ represents a graph encoded by predicates $node(\cdot)$ and $edge(\cdot,\cdot)$. Then $P_{guess}$ guesses all potential Hamiltonian cycles and $P_{check}$ decides for each of them whether it is valid or invalid; in the latter cae, atom $invalid$ is derived. We use both programs together as a subprogram which we access from $P$. We have that $P$ has a single answer set which contains $noHamiltonian$ if and only if the graph at hand does not contain a Hamiltonian cycle. If there are Hamiltonian cycles, then $P$ has at least one answer set but none of the answer sets contains atom $noHamiltonian$.*

*Recall that the same implementation of the checking part did not work in our saturation encoding from Example 7 due to default-negation. Also note that even if the encoding is adopted such that saturation becomes applicable, the cyclic nature of a saturation encoding makes it more difficult to understand then the strictly acyclic encoding based on query atoms.*

We adopt the example to show how to use input to a subprogram:

**Example 8 (cont'd)** *Consider the following programs:*

$$P_{check} = \{hasIn(X) \leftarrow node(X), in(Y,X);\ hasOut(X) \leftarrow node(X), in(X,Y)\} \tag{44}$$

$$invalid \leftarrow node(X), not\ hasIn(X);\ invalid \leftarrow node(X), not\ hasOut(X) \tag{45}$$

$$invalid \leftarrow in(Y_1,X), in(Y_2,X), Y_1 \neq Y_2;\ invalid \leftarrow in(X,Y_1), in(X,Y_2), Y1 \neq Y2 \tag{46}$$

$$P = \{in(X,Y) \vee out(X,Y) \leftarrow arc(X,Y) \tag{47}$$

$$notHamiltonian \leftarrow P_{check}(node, arc, in) \vdash_c invalid\} \cup F \tag{48}$$

*Here, we make a guess in $P$ in send each candidate separately to a subprogram for verification. To this end, for each guess we add all true atoms over the predicates $node$, $arc$, and $in$ to the subprogram $P_{check}$ and check if the guess is a valid Hamiltonian cycle. If this is not the case, $invalid$ is derived in the subprogram, which satisfies the query in $P$ and derives $notHamiltonian$.*

In summary, we have provided an encoding for a particular $coNP$-hard problem, namely deciding inconsistency of a normal ASP-program. Based on this encoding, have a developed a language extension towards queries over normal subprograms. While our encoding makes use of the saturation technique, the user of the language extension does not get in touch with it. That is, the saturation technique is hidden from the user and needs to be implemented only once, namely as part of a solver for the language extension, but not when solving a concrete problem on top of it.

# 5 Reasons for Inconsistency of HEX-Programs

In this section we consider programs $P$ which are extended by a set of (input) atoms $F \subseteq D$ from a given domain $D$, which are added as facts. More precisely, for a given set $F \subseteq D$, we consider $P \cup facts(F)$, where $facts(F) = \{a \leftarrow \mid a \in F\}$ is the representation of $F$ transformed to facts. We assume that the domain of atoms $D$, which can be added as facts, might only occur in bodies $B(P)$ of rules in $P$, but are not in their heads $H(P)$. This is in spirit of the typical usage of (ASP- and HEX-)programs, where proper rules (IDB; intensional database) encode the problem at hand and are fixed, while facts (EDB; extensional database) specify the concrete instance and are subject to change.

We present a concept which expresses the reasons for the inconsistency of HEX-program $P \cup facts(F)$ in terms of $F$. That is, we want to identify a sufficient criterion wrt. the problem instance $F$ which guarantees that $P \cup facts(F)$ does not possess any answer sets. This leads to a characterization of *classes* of inconsistent instances. While identifying such classes is a natural task by itself, a concrete application can be found in context of HEX-program evaluation and will be presented in the next section.

## 5.1 Formalizing Inconsistency Reasons

Inspired by inconsistency explanations for multi-context systems (Eiter et al. 2014), we propose to make an inconsistency reason dependent on atoms from $D$ which *must occur* resp. *must not occur* in $F$ such that $P \cup facts(F)$ is inconsistent, while the remaining atoms from $D$ might either occur or not occur in $F$ without influencing (in)consistency.

We formalize this idea as follows:

**Definition 8 (Inconsistency Reason (IR))** *Let $P$ be a HEX-program and $D$ be a domain of atoms. An* inconsistency reason (IR) *of $P$ wrt. $D$ is a pair $R = (R^+, R^-)$ of sets of atoms $R^+ \subseteq D$ and $R^- \subseteq D$ with $R^+ \cap R^- = \emptyset$ s.t. $P \cup facts(F)$ is inconsistent for all $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$.*

Here, $R^+$ resp. $R^-$ define the sets of atoms which must be present resp. absent in $F$ such that $P \cup facts(F)$ is inconsistent, while atoms from $D$ which are neither in $R^+$ nor in $R^-$ might be arbitrarily added or not without affecting inconsistency.

**Example 9** *An IR of the program $P = \{\leftarrow a, not\ c;\ d \leftarrow b.\}$ wrt. $D = \{a, b, c\}$ is $R = (\{a\}, \{c\})$ because $P \cup facts(F)$ is inconsistent for all $F \subseteq D$ whenever $a \in F$ and $c \notin F$, while $b$ can be in $F$ or not without affecting (in)consistency.*

Note that in contrast to work on ASP debugging, which we discuss in more detail in Section 8 and aims at finding (human-readable) explanations for inconsistency of single programs, we do *not* focus on debugging a particular program instance. Instead, our 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.

For a given program $P$ and a domain $D$, we say that an IR $R_1 = (R_1^+, R_1^-)$ is *smaller than* an IR $R_2 = (R_2^+, R_2^-)$, if $R_1^+ \subseteq R_2^+$ and $R_1^- \subseteq R_2^-$; it is *strictly smaller* if at least one of the inclusions is proper. An IR is *subset-minimal*, if there is no strictly smaller IR wrt. $P$ and $D$. Although small IRs are typically preferred, we do not formally require them to be minimal in general.

In general there are multiple IRs, some of which might not be minimal. For instance, the program $P = \{\leftarrow a; \leftarrow b\}$ has inconsistency reasons $R_1 = (\{a\}, \emptyset)$, $R_2 = (\{b\}, \emptyset))$ and $R_3 = (\{a, b\}, \emptyset)$ wrt. $D = \{a, b\}$, where $R_1$ and $R_2$ are minimal but $R_3$ is not. On the other hand, a program $P$ might not have any IR

at all if $P \cup facts(F)$ is consistent for all $F \subseteq D$. This is the case, for instance, for the empty HEX-program $P = \emptyset$. However, one can show that there is always at least one IR if $P \cup facts(F)$ is inconsistent for some $F \subseteq D$.

**Proposition 6** *For all* HEX*-programs $P$ and domains $D$ such that $P \cup facts(F)$ is inconsistent for some set $F \subseteq D$ of atoms, then there is an IR of $P$ wrt. $D$.*

## 5.2 Computational Complexity

We now discuss the computational complexity of reasoning problems in context of computing inconsistency reasons. Similarly as for the results by Faber et al. (2011) we assume in the following that oracle functions can be evaluated in polynomial time wrt. the size of their input.

We start with the problem of checking if a candidate IR of a program is a true IR. The complexities correspond to those of inconsistency checking over the respective program class, which is because the problems of candidate IR checking and inconsistency checking can be reduced to each other.

**Proposition 7** *Given a* HEX*-program $P$, a domain $D$, and a pair $R = (R^+, R^-)$ of sets of atoms with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$. Deciding if $R$ is an IR of $P$ wrt. $D$ is*

*(i) $\Pi_2^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $coNP$-complete if $P$ is an ordinary disjunction-free program.*

Based on this result one can derive the complexities for checking the existence of IRs. For all program classes, they are one level higher than checking a single candidate IR. Intuitively this is because the consideration of all potential IRs introduces another level of nondeterminism.

**Proposition 8** *Given a* HEX*-program $P$ and a domain $D$. Deciding if there is an IR of $P$ wrt. $D$ is*

*(i) $\Sigma_3^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $\Sigma_2^P$-complete if $P$ is an ordinary disjunction-free program.*

A related natural question concerns subset-minimal IRs. We recall that for each integer $i \geq 1$, the complexity class $D_i^P$ contains decision problems that are a conjunction of independent $\Sigma_i^P$ and $\Pi_i^P$ problems. More precisely, we have that $D_i^P = \{L_1 \times L_2 \mid L_1 \in \Sigma_i^P, L_2 \in \Pi_i^P\}$.

One can then show the following complexity results. Intuitively, the computational efforts comprise of checking that $R$ is an IR at all (giving lower bounds for the complexities as by Proposition 7), and a further check that there is no smaller IR.

**Proposition 9** *Given a* HEX*-program $P$, a domain $D$, and a pair $R = (R^+, R^-)$ of sets of atoms with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$. Deciding if $R$ is a subset-minimal IR of $P$ wrt. $D$ is*

*(i) $D_2^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $D_1^P$-complete if $P$ is an ordinary disjunction-free program.*

Finally, we further consider the check for the existence of a subset-minimal IR. However, these results are immediate consequences of those by Proposition 8 as there is an IR iff there is a subset-minimal IR:

| Reasoning problem | Program class | | | Prop. |
|---|---|---|---|---|
| | Normal ASP | Disjunctive ASP | General HEX | |
| Checking an IR candidate | $coNP$-c | $\Pi_2^P$-c | $\Pi_2^P$-c | 7 |
| Checking existence of an IR | $\Pi_2^P$-c | $\Pi_3^P$-c | $\Pi_3^P$-c | 8 |
| Checking a subset-minimal IR candidate | $D_1^P$-c | $D_2^P$-c | $D_2^P$-c | 9 |
| Checking existence of a minimal IR | $\Pi_2^P$-c | $\Pi_3^P$-c | $\Pi_3^P$-c | 10 |

Table 1: Summary of Complexity Results

**Proposition 10** *Given a* HEX*-program $P$ and a domain $D$. Deciding if there is a subset-minimal IR of $P$ wrt. $D$ is*

*(i)* $\Sigma_3^P$*-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii)* $\Sigma_2^P$*-complete if $P$ is an ordinary disjunction-free program.*

We summarize the complexity results in Table 1.

# 6 Techniques for Computing Inconsistency Reasons

In this section we discuss various methods for computing IRs. For ordinary normal programs we present a *meta-programming* approach, extending the one from Section 3, which encodes the computation of IRs as a disjunctive program. For general HEX-programs this is not possible due to complexity reasons (except with an exponential blowup) and we present a procedural algorithm instead.

## 6.1 Inconsistency Reasons for Normal Ground ASP-Programs

If the program $P$ at hand is normal and does not contain external atoms, then one can construct a positive disjunctive meta-program $M_{gr}^P$ which checks consistency of $P$, as shown in Section 3. We recapitulate the properties of $M \cup M_{gr}^P$ as by Proposition 1 as follows:

- $M \cup M_{gr}^P$ is always consistent;
- if $P$ is inconsistent, then $M \cup M_{gr}^P$ has a single answer set $I_{sat} = A(M \cup M_{gr}^P)$ containing all atoms in $M \cup M_{gr}^P$ including a dedicated atom $noAS$ which does not appear in $P$; and
- if $P$ is consistent, then the answer sets of $M \cup M_{gr}^P$ correspond one-to-one to those of $P$ and none of them contains $noAS$.

Then, the atom $noAS$ in the answer set(s) of $M \cup M_{gr}^P$ represents inconsistency of the original program $P$. One can then extend $M \cup M_{gr}^P$ in order to compute the inconsistency reasons of $P$ as follows. We construct

$$\tau(D,P) = M \cup M_{gr}^P \tag{49}$$
$$\cup \{a^+ \vee a^- \vee a^x \mid a \in D\} \cup \tag{50}$$
$$\cup \{a \leftarrow a^+;\ \leftarrow a, a^-;\ a \vee \bar{a} \leftarrow a^x \mid a \in D\} \tag{51}$$
$$\cup \{\leftarrow not\ noAS\}, \tag{52}$$

where $a^+$, $a^-$, $a^x$ and $\bar{a}$ are new atoms for all atoms $a \in D$.

Informally, the idea is that rules (50) guess all possible candidate IRs $R = (R^+, R^-)$, where $a^+$ represents $a \in R^+$, $a^-$ represents $a \in R^-$ and $a^x$ represents $a \notin R^+ \cup R^-$. Rules (51) guess all possible sets of input facts wrt. the currently guessed IR, where $\bar{a}$ represents that $a$ is not a fact. We know from Proposition 1 that $M \cup M^P_{gr}$ derives $noAS$ iff $P$ together with the facts from rules (51) is inconsistent. This allows the constraint (52) to eliminate all candidate IRs, for which not all possible sets of input facts lead to inconsistency.

One can show that the encoding is sound and complete wrt. the computation of IRs:

**Proposition 11** *Let $P$ be an ordinary normal program and $D$ be a domain. Then $(R^+, R^-)$ is an IR of $P$ wrt. $D$ iff $\tau(D, P)$ has an answer set $I \supseteq \{a^\sigma \mid \sigma \in \{+, -\}, a \in R^\sigma\}$.*

We provide a tool which allows for computing inconsistency reasons of programs, which is available from https://github.com/hexhex/inconsistencyanalysis. The tool expects as command-line parameters a normal ASP-program $P$ (given as filename) and a comma-separated list of atoms to specify a domain $D$. Its output is another ASP-program $P'$ whose answer sets correspond to the IRs of $P$ wrt. $D$; more precisely, each $I \in \mathcal{AS}(P)$ contains atoms of kind $rp(\cdot)$ and $rm(\cdot)$ to specify the sets $R^+$ and $R^-$ of an IR $(R^+, R^-)$, respectively.

## 6.2 Inconsistency Reasons for General Ground HEX-Programs

Using meta-programming approaches for computing IRs has its limitations depending on the class of the program at hand. Suppose the IRs of a program $P$ wrt. a domain $D$ can be computed by a meta-program; then this meta-program is consistent iff an IR of $P$ wrt. $D$ exists. Therefore, consistency checking over the meta-program must necessarily have a complexity not lower than the one of deciding existence of an IR of $P$. However, we have shown in Proposition 8 that deciding if a general HEX-program has an IR is $\Sigma^P_3$-complete, while consistency checking over a general HEX-program is only $\Sigma^P_2$-complete. But then, unless $\Sigma^P_3 = \Sigma^P_2$, computing the IRs of a general HEX-program cannot be polynomially reduced to a meta-HEX-program (using a non-polynomial reduction is possible but uncommon, which is why we do not follow this possibility). We present two possible remedies.

**An incomplete meta-programming approach** For HEX-programs $P$ without disjunctions we can specify an encoding for computing its IRs, which is sound but not complete. One possibility is to make use of a rewriting of $P$ to an ordinary ASP-program $\hat{P}$, which was previously used for evaluating HEX-programs. In a nutshell, each external atom $\&g[\mathbf{p}](\mathbf{c})$ in $P$ is replaced by an ordinary *replacement atom* $e_{\&g[\mathbf{p}]}(\mathbf{c})$ and rules $e_{\&g[\mathbf{p}]}(\mathbf{c}) \leftarrow not\ ne_{\&g[\mathbf{p}]}(\mathbf{c})$ and $ne_{\&g[\mathbf{p}]}(\mathbf{c}) \leftarrow not\ e_{\&g[\mathbf{p}]}(\mathbf{c})$ are added to guess the truth value of the former external atom. However, the answer sets of the resulting *guessing program* $\hat{P}$ do not necessarily correspond to answer sets of the original program $P$. Instead, for each answer set it must be checked if the guesses are correct. An answer set $I$ of the guessing program $\hat{P}$ is called a *compatible set* of $P$, if $f_{\&g}(\hat{I}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$ iff $e_{\&g[\mathbf{p}]}(\mathbf{c}) \in \hat{I}$ for all external atoms $\&g[\mathbf{p}](\mathbf{c})$ in $P$. Each answer set of $P$ is the projection of some compatible set of $P$.

One can exploit the rewriting $\hat{P}$ to compute (some) IRs of $P$. To this end, one constructs $\tau(D, \hat{P})$ and computes its answer sets. This yields explanations for the inconsistency of the guessing program $\hat{P}$ rather than the actual HEX-program $P$. The HEX-program $P$ is inconsistent whenever the guessing program $\hat{P}$ is, and every inconsistency reason for $\hat{P}$ is also one for $P$. Hence, the approach is sound, but since $\hat{P}$ might be consistent even if $P$ is inconsistent, it follows that the approach is not complete:

**Proposition 12** *Let $P$ be a* HEX*-program and $D$ be a domain. Then each IR of $\hat{P}$ wrt. $D$ is also an IR of $P$ wrt. $D$, i.e., the use of $\hat{P}$ is sound wrt. the computation of IRs.*

The following example shows that using $\hat{P}$ for computing IRs is not complete.

**Example 10** *Consider the* HEX*-program $P = \{p \leftarrow q, \&neg[p]()\}$ and domain $D = \{q\}$. An IR is $(\{q\}, \emptyset)$ because $P \cup facts(F)$ for $F = \{q\}$ is inconsistent. However, the guessing program $\hat{P}$ extended with $F$*

$$\begin{aligned}\hat{P} \cup \{q \leftarrow\} = \{&e_{\&neg[p]} \leftarrow not\ ne_{\&neg[p]}\\ &ne_{\&neg[p]} \leftarrow not\ e_{\&neg[p]}\\ &p \leftarrow q, e_{\&neg[p]}\\ &q \leftarrow\}\end{aligned}$$

*is consistent and has the answer set $\hat{I} = \{e_{\&neg[p]}, p, q\}$; therefore $(\{q\}, \emptyset)$ is not found as an IR when using $\hat{P}$ (actually there is no IR of $\hat{P}$ wrt. $D$).*

In the previous example, the reason why no inconsistency reason is found is that $\hat{I}$ is an answer set of $\hat{P}$ but the value of $e_{\&neg[p]}$ guessed for the external replacement atom differs from the actual value of $\&neg[p]()$ under $\hat{I}$, i.e., $\hat{I}$ is not compatible.

**An incomplete procedural algorithm** Beginning from this section, we need to distinguish unassigned and false atoms and denote assignments as sets of signed literals, as discussed in Section 2.

Our algorithm is an extension of the state-of-the-art evaluation algorithm for ground HEX-programs. The evaluation algorithm is in turn based on conflict-driven nogood learning (CDNL), which has its origins in SAT solving (cf. e.g. Marques-Silva et al. (2009)). Here, 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 $\delta$ if $\delta \not\subseteq \mathbf{A}$, and to a set of nogoods $\Delta$ 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. The basic idea is to represent the program at hand as a set of nogoods and try to construct an assignment, which satisfies all nogoods, where both deterministic propagation and guessing is used. The distinguishing feature of CDNL is that the nogood set is *dynamically extended* with new nogoods, which are learned from conflict situations, such that the same conflict is not constructed again.

The CDNL-based approach for HEX-programs is shown in Algorithm 1. The input is a program $P$, a set of input facts $F$, and an *inconsistency handler function* $h$; the latter is a preparation for our extensions below and returns always $\bot$ for answer set computation. The output is an answer set of $P \cup facts(F)$ if there is one, and $\bot$ otherwise.

The initialization is done at (a). The given HEX-program $P$ is extended with facts $facts(F)$ 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 $\Delta$, which consists of nogoods $\Delta_{\hat{P}}$ stemming from Clark's completion (Clark 1977) and singleton loop nogoods (Gebser et al. 2012); the former basically encode the rules as implications, while the latter control support of atoms. The nogood set will be expanded as the search space is traversed.

**Algorithm 1:** HEX-CDNL-PA

**Input**: ground program $P$, input atoms $F$, an inconsistency handler function $h$ (with a nogood set and an assignment as parameters)
**Output**: an answer set of $P \cup facts(F)$ if existing, and $h(\Delta, \hat{\mathbf{A}})$ otherwise (where $\Delta$ resp. $\hat{\mathbf{A}}$ are the nogood set resp. assignment at the time of inconsistency discovery)

```
// Initialization
(a) P ← P ∪ facts(F); Δ ← Δ_P̂; Â ← ∅; current_dl ← 0
for all facts a ←∈ P do
    Â ← Â ∘ (Ta)
    dl[a] ← current_dl
for a ∉ H(r) for all r ∈ P do
    Â ← Â ∘ (Fa)
    dl[a] ← current_dl

// Main loop
while true do
(b)     (Â, Δ) ← Propagation(P̂, Δ, Â)
(c)     if for some nogood δ ∈ Δ we have by δ ⊆ Â then
            if current_dl = 0 then return h(Δ, Â)  Analyze conflict, add learned nogood to Δ, backjump and update current_dl
(d)     else if Â is complete then
            if guesses are not correct or not minimal wrt. the reduct then
                Δ ← Δ ∪ {Â}
                Analyze conflict, add learned nogood to Δ, backjump and update current_dl
            else
                return Â|_{A(P)}
(e)     else if Heuristics evaluates &g[y] and Λ(&g[y], Â) ⊈ Δ then
            Δ ← Δ ∪ Λ(&g[y], Â)
(f)     else
            Let σ ∈ {T, F} and a ∈ A(P̂) with {Ta, Fa} ∩ Â = ∅
            current_dl ← current_dl + 1
            dl[a] ← current_dl
            Â ← Â ∘ (σa)
```

As further initializations, we set the computed answer set $\hat{\mathbf{A}}$ of the guessing program $\hat{P}$ initially to the empty list; for the sake of the algorithm, assignments are seen as lists such that the order of assignments is retrievable later. The *current decision level (dl)* is initially $0$ and incremented for every guess. The initialization further assigns facts immediately to true and atoms which do not appear in any heads to false (both at dl $0$), where the decision levels of atoms are stored in array $dl$ such that $dl[a] \in \mathbb{N}_0$ for all atoms $a$.

After the initialization, the algorithm performs deterministic propagation such as unit propagation at (b). Further propagation techniques such as unfounded set propagation can be added. Each implied literal is assigned at the maximum current decision of the literals which implied it.

The algorithm detects conflicts, learns additional nogoods and backtracks at (c). If the conflict is on decision level $0$, the instance is inconsistent and the callback function $h$ is notified (with nogoods $\Delta$ and assignment $\hat{\mathbf{A}}$ over $P$ as input) to determine the return value in case of inconsistency; as said above, this callback serves as a preparation for our extension of the algorithm below and is instantiated with just $h_\perp(\Delta, \hat{\mathbf{A}}) = \perp$ (independent of $\Delta$ and $\hat{\mathbf{A}}$) for answer set computation.

If the assignment is complete at (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; actual implementations are more involved to keep memory consumption small (Drescher et al. 2008).

Next, the algorithm may perform theory propagation at (e) driven by a heuristics. That is, an external source $\&g$ with input $\mathbf{y}$ may be evaluated under assignment $\hat{\mathbf{A}}$ to learn parts of its behavior in form of a set of nogoods. We abstractly use a so-called *learning function* $\Lambda$ to refer to the set of nogoods $\Lambda(\&g[\mathbf{y}], \hat{\mathbf{A}})$ learned from such an evaluation step.

Finally, if none of the other cases applies, we have to guess a truth value at (f).

We refer to Eiter et al. (2014) for a more extensive study of this evaluation approach, who also proved

soundness and completeness of Algorithm 1:

**Proposition 13** *Let $P$ be a program and $F \subseteq D$ be input atoms from a domain $D$. If we have that* HEX-*CDNL-PA*$(P, F, h_\bot)$ *returns (i) an assignment* $\mathbf{A}$*, then* $\mathbf{A}$ *is an answer set of $P \cup facts(F)$; (ii) $\bot$, then $P \cup facts(F)$ is inconsistent.*

Our approach for computing IRs for ground programs is based on *implication graphs*, which are used for conceptual representation of the current status and assignment history of the solver, cf. e.g. Biere et al. (2009). However, in Algorithm 1 the implication graph is only implicitly represented by the order of assignments.

Intuitively, the nodes of an implication 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 9** *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 $\bot$, $dl \in \mathbb{N}_0$ is a decision level, $\delta \in \Delta \cup \{\bot\}$ is a nogood, and $E$ is a set of unlabeled edges.*

**Example 11** *Let*
$\Delta = \{\delta_1\colon \{\mathbf{T}a, \mathbf{T}b\}, \delta_2\colon \{\mathbf{T}a, \mathbf{F}b, \mathbf{F}c\}, \delta_3\colon \{\mathbf{T}c, \mathbf{T}d, \mathbf{F}e\}, \delta_4\colon \{\mathbf{T}d, \mathbf{T}e\}\}$. *An implication graph is shown in Figure 1. Here, $\mathbf{T}a@1/\bot$ is a guess at decision level 1, $\mathbf{F}b@1/\delta_1$ is implied by $\mathbf{T}a$ using $\delta_1$, $\mathbf{T}c@1/\delta_2$ is implied by $\mathbf{T}a$ and $\mathbf{F}b$ using $\delta_2$, $\mathbf{T}d@2/\bot$ is a guess at decision level 2, and $\mathbf{T}e@2/\delta_3$ is implied by $\mathbf{T}c$ and $\mathbf{T}d$ using $\delta_3$. Then $\delta_4$ is violated due to $\mathbf{T}d$ and $\mathbf{T}e$.*

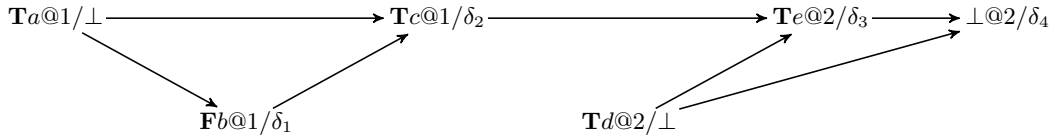



Figure 1: Implication graph of the program in Example 11

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$(D, \Delta, \hat{\mathbf{A}})$ as inconsistency handler function 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 at 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 $\sigma a$ in this nogood was assigned at decision level 0, each of them was assigned by unit propagation as a consequence of previously assigned literals. We then iteratively (cf. loop at (b)) resolve this nogood with the cause of one of its literals $\sigma a$ at (c), i.e., with the nogood $\epsilon$ which implied literal $\sigma a$. The loop at (b) repeats this step until the nogood contains only literals over explanation atoms from $D$; since these literals imply the originally violated nogood and are thus responsible for inconsistency, $\delta$ represents an IR which we return at (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 implicants of literals).

**Algorithm 2:** InconsistencyAnalysis

**Input**: domain $D$, set of nogoods $\Delta$ representing program $P$, assignment $\hat{\mathbf{A}}$ conflicting with $\Delta$
**Output**: inconsistency reason of $P$ wrt. $D$

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

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

**Proposition 14** *Let $P$ be a program and $F \subseteq D$ be input atoms from a domain $D$. If we have that* HEX-CDNL-PA$(P, F, h^D_{analyse})$ *returns (i) an assignment $\mathbf{A}$, then $\mathbf{A}$ is an answer set of $P \cup facts(F)$; (ii) a pair $R = (R^+, R^-)$ of sets of atoms, then $P \cup facts(F)$ is inconsistent and $R$ is an inconsistency reason of $P$ wrt. $D$.*

### 6.3 Computing Inconsistency Reasons for General Non-Ground HEX-Programs

Next, we extend the above approach to non-ground programs. Grounders fo ASP and HEX-programs instantiate non-ground programs in such a way that the resulting ground program is still equivalent to the original one wrt. answer set computation. However, in general the equivalence is not given any more wrt. computation of inconsistency reasons.

Grounding algorithms typically do not use the naive grounding $grnd_{\mathcal{C}}(P)$ which substitutes variables $\mathcal{V}$ in $P$ by constants from $\mathcal{C}$ in all possible ways (and is even infinite in general). Instead, optimizations are performed to keep the grounding small. However, the exact algorithms for performing such optimizations, and therefore the output, depend on the grounder in use. In fact, there is large room for correct grounding procedures as they 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 $grnd_{\mathcal{C}}(P)$. This allows grounders also to optimize *within* rules (and not just drop irrelevant rules as a whole) and introduce auxiliary rules, which is also done in practice. Due to these grounder-specific optimizations we intentionally do not introduce a fixed definition of an optimized grounding here. However, we assume in the following that an arbitrary but fixed optimized grounding of a program $P$ wrt. a set of constants $\mathcal{C}$ accessible via $og_{\mathcal{C}}(P)$.

These optimizations prohibit the direct reduction of the computation of an IR for a (possibly) non-ground program $P$ wrt. a domain $D$ to the ground case because the grounding step may have optimized program parts away, which can be relevant when the input facts change. To demonstrate this, suppose we compute the optimized grounding $P_g = og_{\mathcal{C}}(P \cup facts(F))$ of a program $P$ with input facts $facts(F)$ for some $F \subseteq D$. Then we may try to reuse Algorithm 1 for IR computation by passing $P_g \setminus facts(F)$ as program and $F$ as input, i.e., we call HEX-CDNL-PA$(P_g \setminus facts(F), F, h^D_{analyse})$. If this call returns a pair $R = (R^+, R^-)$, we have then by Proposition 14 that $R$ is an IR for $P_g \setminus facts(F)$, i.e., $(P_g \setminus facts(F)) \cup facts(J)$ 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 $F' \subseteq D$ we may have $og_{\mathcal{C}}(P \cup facts(F')) \setminus facts(F') \neq og_{\mathcal{C}}(P \cup facts(F)) \setminus facts(F)$.

**Example 12** *Consider $P = \{q(X) \leftarrow p(X);\ \leftarrow not\ q(1);\ \leftarrow a\}$ and $D = \{a, p(1)\}$. For input $F = \emptyset$, we have $P_g = og_{\mathcal{C}}(P \cup facts(F)) \setminus facts(F) = \{\leftarrow not\ q(1);\ \leftarrow a\}$. Inconsistency analysis of $P_g$ wrt. $F$ may yield $R = (\emptyset, \emptyset)$, which is an IR of $P_g$ wrt. $D$, but not of $P$ wrt $D$ as $P \cup facts(\{p(1)\})$ is consistent.*

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

The main idea is then as follows. We add for all atoms $a$ a rule $a \leftarrow a'$ to $pog_{\mathcal{C},F}(P)$ in order to encode the potential support by unknown rules, where the new atom $a'$ stands for the situation that such a rule fires. Then an IR $R = (R^+, R^-)$ of the extended program, which does not contain any such atom, does not depend on the absence of $a'$. Hence, adding such a rule cannot invalidate the IR and thus the IR is also valid for $eg_{\mathcal{C},F}(P)$ (and thus for $P$). Formally we can show:

**Proposition 15** *Let $P$ be a program and $F \subseteq D$ be input atoms from a domain $D$. Then an IR $R = (R^+, R^-)$ of $pog_{\mathcal{C},F}(P) \cup \{a \leftarrow a' \mid a \in A(pog_{\mathcal{C},F}(P))\}$ wrt. $D \cup \{a' \mid a \in A(pog_{\mathcal{C},F}(P))\}$ s.t. $(R^+ \cup R^-) \cap \{a' \mid a \in A(pog_{\mathcal{C},F}(P))\} = \emptyset$ is an IR of $P$ wrt. $D$.*

**Example 13 (cont'd)** *For the program $P = \{q(X) \leftarrow p(X);\ \leftarrow not\, q(1);\ \leftarrow a\}$ and domain $D = \{a, p(1)\}$ from Example 12, and $F = \emptyset$ we have $pog_{\mathcal{C},F}(P) = \{\leftarrow not\, q(1);\ \leftarrow a\}$. However, for computing inconsistency reasons we extend the program by the rule $q(1) \leftarrow q(1)'$ to $\{q(1) \leftarrow q(1)'\} \cup pog_{\mathcal{C},F}(P)$ to express that $q(1)$ might be supported by unknown rules, which have been optimized away when grounding wrt. specific facts $F = \emptyset$. Then $\langle \emptyset, \{q(1)'\}\rangle$ is an IR of $\{q(1) \leftarrow q(1)'\} \cup pog_{\mathcal{C},F}(P)$ wrt. $D \cup \{q(1)'\}$, but since it contains $q(1)'$, it is excluded is an IR of $P$ wrt. $D$. In contrast, $\langle \emptyset, \{a\}\rangle$ as an IR of $\{q(1) \leftarrow q(1)'\} \cup pog_{\mathcal{C},F}(P)$ wrt. $D \cup \{q(1)'\}$, and since it does not contain $q(1)'$, it is also an IR of $P$ wrt. $D$.*

# 7 Application: Exploiting Inconsistency Reasons for HEX-Program Evaluation

We now exploit inconsistency reasons for improving the evaluation algorithm for HEX-programs. In this section we focus on a class of programs which cannot be efficiently tackled by existing approaches. Current evaluation techniques inefficient for programs with guesses that are separated from constraints by external atoms which (i) are *nonmonotonic*, and (ii) *introduce new values* to the program (called *value invention*). Here, an external atom $\&g[\mathbf{p}](\cdot)$ is called monotonic if the output never shrinks when more input atoms become true; formally: if for all output vectors $\mathbf{c}$ and assignments $\mathbf{A}$, $\mathbf{A}'$ with $\{\mathbf{T}a \in \mathbf{A}'\} \supseteq \{\mathbf{T} \in \mathbf{A}\}$ it is guaranteed that $\mathbf{A} \models \&g[\mathbf{p}](\mathbf{c})$ implies $\mathbf{A}' \models \&g[\mathbf{p}](\mathbf{c})$. Otherwise it is called nonmonotonic.

We start with a recapitulation of current evaluation techniques and point out their bottlenecks.

## 7.1 Existing Evaluation Techniques for HEX-Programs

Currently, there is a *monolithic* evaluation approach which evaluates the program as a whole, and another one which *splits* it into multiple program components.

**Monolithic Approach** Safety criteria for HEX-programs guarantee the existence of a finite grounding, which allows for evaluating a program by separate grounding an solving phases akin to ordinary ASP. A grounding algorithm was presented by Eiter et al. (2016), the solving algorithm was recapitulated in Algorithm 1 in Section 6.

Since external sources may introduce new constants, determining the set of relevant constants requires the external sources to be evaluated already during grounding. While this set is finite, it is in general expensive to compute because an up to exponential number of evaluation calls is necessary to construct a

single ground instance of a rule. Intuitively, this is the case because nonmonotonic external atoms which provide value invention must be evaluated under all possible inputs to observe all possible output values. We demonstrate this with an example.

**Example 14** *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 involved persons. 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 altogether, 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 by the program*

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

*where $F$ is supposed to be a set of facts over predicate $person$, which specify the available employee.*

*Rule $r_1$ guesses candidate commitees, rule $r_2$ excludes conflicts of interest (defines as facts over predicate $conflict$), $r_3$ determines competences of the candidate, and $r_4$ defines that we do not want committees which can neither decide in technical nor financial affairs.*

Suppose program $P$ from Example 14 is grounded as a whole before solving starts. Then, without further information, the grounder must evaluate $\&competences[in](X)$ under all possible candidate committees, i.e., under all consistent assignments $\mathbf{A} \subseteq \{\mathbf{T}in(a), \mathbf{F}in(a) \mid person(a) \leftarrow\in F\}$ to make sure that all possible competences $X$ are observed. In other words, the grounder must determine the set

$$\{\mathbf{c} \mid \mathbf{A} \models \&competences[in](\mathbf{c}) \text{ for some consistent } \mathbf{A} \subseteq \{\mathbf{T}in(a), \mathbf{F}in(a) \mid person(a) \leftarrow\in F\}\}.$$

While many candidate committees (and their competences) turn out to be irrelevant when solving the ground program because they are already eliminated by $r_2$, this is not discovered in the grounding phase.

Note that this bottleneck occurs only if the external atom is (i) nonmonotonic, and (ii) can introduce new values to the program. If condition (i) would not be given, i.e., if the external atom would be known to the grounder to be monotonic (that is, the set of competences can only increase if more people join the committee), then it would suffice to evaluate it under the maximal assignment $\mathbf{A} = \{\mathbf{T}in(a) \mid person(a) \leftarrow\in F\}$ to observe all relevant constants. If condition (ii) would not be given, i.e., it is known to the grounder that the external atom does not introduce constants which are not in the program, then the grounder would not need to evaluate it at all during grounding since all relevant constants are already in the program.

Hence, for the considered class of programs, the approach usually suffers a bottleneck due to a large number of external calls during grounding.

**Splitting Approach** An alternative evaluation approach is implemented by a *model-building framework* based on program splitting (Eiter et al. 2016). Based on dependencies between rules, the program is split it into components, called *(evaluation) units*, which are arranged in an *acyclic evaluation graph*; acyclicity means informally that a later unit cannot derive an atom (i.e., have it in some rule head) that already occurred in predecessor units. The approach exploits a theorem similar to the splitting theorem by Lifschitz and Turner (1994).

In a nutshell, the evaluation works unit-wise: each unit is separately grounded and solved, beginning from the units without predecessors. The framework first computes the answer sets of units without predecessors. For each answer set, the successor units are extended with the true atoms from the answer set as facts, and it itself grounded and solved. This procedure is repeated recursively, where the final answer sets are extracted from the leaf units.

**Example 15 (cont'd)** *Reconsider program $P$ from Example 14. It can be partitioned into units $u_1 = \{r_1, r_2\}$ and $u_2 = \{r_3, r_4\}$, where $u_2$ depends on $u_1$ because it uses atoms from $u_1$, but does not redefine them, cf. Figure 2. For the evaluation, we first compute the answer sets $\mathcal{AS}(u_1)$ of program component $u_1$, which represent all committee candidates, excluding those with conflicts of interest (due to $r_2 \in u_1$). For each answer set $\mathbf{A} \in \mathcal{AS}(u_1)$, we ground and solve $u_2$ extended with the true atoms from $\mathbf{A}$ as facts, i.e., we compute $\mathcal{AS}(u_2 \cup \{a \leftarrow | \ \mathbf{T}a \in \mathbf{A}\})$. The answer sets of the overall program correspond one-by-one to $\bigcup_{\mathbf{A} \in \mathcal{AS}(u_1)} \mathcal{AS}(u_2 \cup \{a \leftarrow | \ \mathbf{T}a \in \mathbf{A}\})$.*

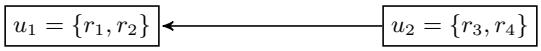



Figure 2: Evaluation graph of the program from Example 14

Observe that when grounding $u_2 \cup \{a \leftarrow | \ \mathbf{T}a \in \mathbf{A}\}$ for a certain answer set $\mathbf{A}$ of $u_1$ in Example 15, the input to the external atom $\&competences[in](X)$ consists of facts only, i.e., it is already fixed. This is exploited to determine all relevant constants with a single external call.

However, the drawback of this approach is that it splits the guessing part from the checks: rule $r_4$ is separated from the guess in rule $r_1$ and a conflict with $r_4$ during the evaluation of $u_2 \cup \{a \leftarrow | \ \mathbf{T}a \in \mathbf{A}\}$ for some $\mathbf{A} \in \mathcal{AS}(u_1)$ cannot be propagated to unit $u_1$ to exclude further guesses, which may lead to a related conflict in $u_2$.

Hence, for the considered class of programs, this technique also suffers a bottleneck, but one of a different nature than for the monolithic approach. While grounding a single program unit for a fixed input from its predecessor unit is now efficient, the problem is now that later units must be instantiated for many different inputs from predecessor units. In the example, unit $u_2$ needs to be grounded for all answer sets of $u_1$, although many of them may make $u_2$ inconsistent because of the same reason.

Although the framework supports arbitrary acyclic evaluation graphs as stated above, we only consider the special case of a list of units in this paper (as in Example 15). This is because it is sufficient for explaining the main idea of our new evaluation technique and saves us from the need to introduce the complete framework formally, which is cumbersome and would distract from the core idea. However, in practice the idea can be easily applied also to general acyclic evaluation graphs.

## 7.2 Trans-Unit Propagation

To overcome the bottleneck we envisage at keeping the program splits for the sake of efficient grounding, but still allow for propagating conflicts back from later units to predecessor units. This can be seen as conflict-driven learning over multiple evaluation units, which is why we call out technique *trans-unit (tu-)propagation*. The main idea is to associate an IR $R = (R^+, R^-)$ of a later unit $u$ with a constraint $c_R = \leftarrow R^+, \{not\, a \mid a \in R^-\}$ which we propagate to predecessors $u'$ of $u$. This is in order to eliminate exactly those interpretations already earlier, which would lead to inconsistency of $u$ anyway.

Recall that for a domain $D$ of explanation atoms and an inconsistency reason $R = (R^+, R^-)$ wrt. a program $P$, all programs $P \cup facts(F)$ with $F \subseteq D$ such that $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ are inconsistent. One can show that adding $c_R$ to $P$ does not eliminate answer sets:

**Proposition 16** *For all* HEX*-programs $P$ and IRs $R = (R^+, R^-)$ of $P$ wrt. a domain $D$, we have that $\mathcal{AS}(P \cup facts(F)) = \mathcal{AS}(P \cup \{c_R\} \cup facts(F))$ for all $F \subseteq D$.*

Moreover and more importantly, such a constraint can also be added to predecessor units: for an inconsistency reason $R$ of a unit $u$ wrt. a domain $D$, one can add the constraint $c_R$ to all predecessor units $u'$ of $u$ such that all atoms in $D$ are defined in $u'$ or its own transitive predecessors. Intuitively, this is the case due to acyclicity of the evaluation graph: since an already defined atom cannot be redefined in a later component, a constraint can be checked in the hierarchical evaluation of the evaluation units as soon as all relevant atoms from $D$ (which are the only ones that can appear in $c_R$) have been defined. This potentially eliminates wrong guesses earlier.

**Proposition 17** *For an evaluation unit $u$ and IR $R = (R^+, R^-)$ of $u$ wrt. a domain $D$, constraint $c_R$ can be added to all predecessor units $u'$ s.t. all atoms in $D$ are defined in $u'$ or one of its own transitive predecessors.*

**Example 16 (cont'd)** *Assume that $joe$ and $sue$ are the only technicians and $alyson$ is the only economist. Then, $u_2$ is always inconsistent if none of these three persons is selected, independent of other choices. If the current input to unit $u_2$ is $F = \{in(jack), in(joseph)\}$, then by exploiting nogoods learned from the external source &competences, the IR $R = (\emptyset, \{in(joe), in(sue), in(alyson)\})$ of $u_2$ wrt. $D = \{in(joe), in(sue), in(alyson)\}$ can be determined, and the constraint $c_R = \leftarrow not\ in(joe), not\ in(sue), not\ in(alyson)$ can be added to $u_1$.*

While in general there can be multiple predecessor units $u'$ satisfying the precondition of Proposition 17, in our implementation and experiments we add $c_R$ to the top-most one. For analytical reasons, this is always better than (and subsumes) adding them to later units since as it eliminates invalid candidates as early as possible. Moreover, the proposition holds for any set $D$ of domain atoms; in practice when computing an inconsistency explanation for a unit $u$, we use the set of all atoms which appear already in some predecessor unit of $u$.

### 7.3 Implementation and Experiments

We now evaluate our approach using a benchmark suite. For the experiments, we integrated our techniques into the reasoner dlvhex[5] with gringo and clasp from the Potassco suite[6] 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 16 GB per instance. We used the *HTCondor* load distribution system[7] 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 with value invention. For each benchmark we compare three configurations: (i) evaluation as a **monolithic** program, (ii) evaluation using

[5] www.kr.tuwien.ac.at/research/systems/dlvhex

[6] https://potassco.org

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

**splitting**, and (iii) evaluation using splitting and **trans-unit (tu-)propagation**. Our hypothesis is that (i) suffers a bottleneck from grounding the program as a whole, (ii) suffer a bottleneck from repetitive instantiation of later program units (where the overall effort is distributed over grounding and solving), 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 time, solving time, and for tu-propagation also the time needed for computing inconsistency reasons as by Algorithms 1 and 2. All time specifications are averaged over all instances of the respective size. Importantly, other computations besides grounding, solving and computing inconsistency reasons (such as preprocessing, running the model-building framework, etc) can cause the overall runtime to be higher than the sum of the other time specifications. Numbers in parentheses show the number of timeout instances. The instances we used for our benchmarks are available from http://www.kr.tuwien.ac.at/research/projects/inthex/inconsistency.

**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 performance, power consumption, disk space, etc. Properties may depend not only on individual components but also on their interplay, and this dependency can be nonmonotonic. For instance, 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 14), 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 $S \subseteq D$ of a domain elements 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 10 randomly generated instances for each size $n$ 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 2. 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. The grounding and solving times show that for the monolithic approach, the main source of computation costs is grounding, which comes from exponentially many external calls. In contrast, for the splitting approach the runtime is more evenly distributed to the grounding and the solving phase, which comes from repetitive instantiation and evaluation of later program units. With **tu-propagation** both grounding and solving is faster. This is because learned nogoods effectively spare grounding and solving of later program units, which would be inconsistent anyway. Hence, the approach clearly outperforms both existing ones. The observations are in line with our hypotheses.

**Diagnosis Problem** We now consider diagnosis problems formalized as follows. We have a quintuple $\langle \mathcal{O}_d, \mathcal{O}_p, \mathcal{H}, \mathcal{C}, P \rangle$, 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. Let $\bar{h}$ be a new atom for each $h \in \mathcal{H}$ which does not appear in $P$. Then a solution consists of a set $S_\mathcal{H} \subseteq \mathcal{H}$ of hypotheses and a set of potential observations $S_{\mathcal{O}_p} \subseteq \mathcal{O}_p$ such that (i) all answer sets of $P \cup \{h \vee \bar{h} \leftarrow \mid h \in \mathcal{H}\}$, which contain all of $\mathcal{O}_d \cup S_{\mathcal{O}_p}$, 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.

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

| size | monolithic | | | splitting | | | tu-propagation | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | total | ground | solve | total | ground | solve | total | ground | solve | analysis |
| 5 | 0.13 (0) | 0.01 | 0.01 | 0.13 (0) | 0.01 | 0.02 | 0.14 (0) | 0.01 | 0.00 | 0.00 |
| 7 | 0.18 (0) | 0.03 | 0.04 | 0.27 (0) | 0.07 | 0.08 | 0.19 (0) | 0.03 | 0.01 | 0.01 |
| 9 | 0.37 (0) | 0.10 | 0.14 | 0.90 (0) | 0.32 | 0.42 | 0.38 (0) | 0.13 | 0.03 | 0.02 |
| 11 | 0.72 (0) | 0.45 | 0.14 | 4.51 (0) | 1.68 | 2.42 | 0.39 (0) | 0.14 | 0.07 | 0.06 |
| 13 | 3.04 (0) | 1.87 | 0.91 | 20.35 (0) | 7.73 | 11.23 | 1.64 (0) | 0.89 | 0.19 | 0.17 |
| 15 | 12.64 (0) | 8.23 | 3.74 | 102.82 (0) | 42.16 | 55.32 | 5.53 (0) | 3.64 | 0.51 | 0.46 |
| 17 | 47.32 (0) | 34.24 | 11.09 | 300.00 (10) | 122.58 | 164.20 | 15.66 (0) | 10.64 | 1.07 | 0.90 |
| 19 | 184.35 (2) | 142.21 | 14.07 | 300.00 (10) | 122.37 | 163.66 | 61.99 (1) | 34.13 | 4.83 | 4.34 |
| 21 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 127.01 | 160.96 | 148.85 (3) | 91.11 | 12.45 | 11.72 |
| 23 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 128.13 | 161.24 | 300.00 (10) | 184.68 | 26.17 | 25.00 |

Table 2: Configuration Problem

| size | monolithic | | | splitting | | | tu-propagation | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | total | ground | solve | total | ground | solve | total | ground | solve | analysis |
| 5 | 1.31 (0) | 0.38 | 0.81 | 0.25 (0) | 0.12 | 0.02 | 0.94 (0) | 0.77 | 0.01 | 0.01 |
| 7 | 3.62 (0) | 1.69 | 1.81 | 0.80 (0) | 0.60 | 0.08 | 2.02 (0) | 1.79 | 0.04 | 0.04 |
| 9 | 12.02 (0) | 7.07 | 4.76 | 2.35 (0) | 1.94 | 0.27 | 4.23 (0) | 3.94 | 0.06 | 0.05 |
| 11 | 75.92 (0) | 42.04 | 32.62 | 13.28 (0) | 11.43 | 1.40 | 49.61 (0) | 48.12 | 0.28 | 0.26 |
| 13 | 300.00 (10) | 300.00 | n/a | 51.29 (1) | 46.51 | 4.26 | 23.19 (0) | 22.54 | 0.12 | 0.11 |
| 15 | 300.00 (10) | 300.00 | n/a | 244.22 (7) | 222.73 | 18.28 | 153.54 (5) | 150.25 | 0.44 | 0.41 |
| 17 | 300.00 (10) | 300.00 | n/a | 260.10 (7) | 237.44 | 19.88 | 158.91 (5) | 156.30 | 0.77 | 0.75 |
| 19 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 274.57 | 22.56 | 179.53 (5) | 175.96 | 0.56 | 0.54 |
| 21 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 272.97 | 24.05 | 86.86 (2) | 84.76 | 0.32 | 0.30 |
| 23 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 271.61 | 25.52 | 164.58 (4) | 159.98 | 0.39 | 0.37 |
| 25 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 270.88 | 26.31 | 241.03 (7) | 235.46 | 0.57 | 0.55 |
| 27 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 243.35 | 26.96 | 275.57 (8) | 270.67 | 0.70 | 0.68 |
| 29 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 269.84 | 27.43 | 203.08 (5) | 200.02 | 0.53 | 0.51 |
| 31 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 269.83 | 27.59 | 268.79 (8) | 264.27 | 0.62 | 0.59 |
| 33 | 300.00 (10) | 300.00 | n/a | 300.00 (10) | 269.94 | 27.61 | 300.00 (10) | 293.03 | 0.70 | 0.68 |

Table 3: Diagnosis Problem

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 3. Unlike for the previous benchmark, **monolithic** is now slower than **splitting**. This is because the evaluation of the external source (corresponding to evaluating $P$) is much more expensive now. Since with **monolithic**, grounding always 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. Due to randomization of the instances, larger instances can in some cases be solved faster than smaller instances. 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,

| size | monolithic | | | splitting | | | tu-propagation | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | total | ground | solve | total | ground | solve | total | ground | solve | analysis |
| 5 | 3.06 (0) | 2.96 | $< 0.005$ | 0.18 (0) | 0.04 | 0.04 | 0.14 (0) | 0.00 | 0.01 | 0.01 |
| 6 | 14.06 (0) | 13.96 | $< 0.005$ | 0.28 (0) | 0.09 | 0.08 | 0.15 (0) | 0.00 | 0.02 | 0.02 |
| 7 | 65.45 (0) | 65.35 | $< 0.005$ | 0.51 (0) | 0.20 | 0.19 | 0.15 (0) | 0.00 | 0.02 | 0.02 |
| 8 | 289.53 (0) | 289.43 | $< 0.005$ | 1.03 (0) | 0.44 | 0.44 | 0.18 (0) | 0.01 | 0.03 | 0.03 |
| 9 | 300.00 (1) | 300.00 | n/a | 2.13 (0) | 0.95 | 0.99 | 0.19 (0) | 0.01 | 0.04 | 0.03 |
| 10 | 300.00 (1) | 300.00 | n/a | 4.62 (0) | 2.16 | 2.10 | 0.20 (0) | 0.01 | 0.04 | 0.04 |
| 11 | 300.00 (1) | 300.00 | n/a | 9.87 (0) | 4.67 | 4.63 | 0.23 (0) | 0.01 | 0.05 | 0.05 |
| 12 | 300.00 (1) | 300.00 | n/a | 19.62 (0) | 9.19 | 9.54 | 0.24 (0) | 0.02 | 0.06 | 0.06 |
| 13 | 300.00 (1) | 300.00 | n/a | 41.36 (0) | 19.21 | 20.51 | 0.27 (0) | 0.02 | 0.07 | 0.07 |
| 14 | 300.00 (1) | 300.00 | n/a | 85.66 (0) | 40.88 | 41.67 | 0.29 (0) | 0.02 | 0.08 | 0.08 |
| 15 | 300.00 (1) | 300.00 | n/a | 178.97 (0) | 85.65 | 86.74 | 0.32 (0) | 0.03 | 0.10 | 0.09 |

Table 4: Synthetic Set Guessing

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 4. While **splitting** separates the rules in the second 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 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.

# 8 Discussion and Related Work

**Related work on the saturation technique** Our encoding for inconsistency checking is related to a technique towards automated integration of guess and check programs (Eiter and Polleres 2006), but using a different encoding. While the main idea of simulating the computation of the least fixpoint of the (positive) reduct by guessing a derivation sequence is similar, our encoding appears to be conceptually simpler thanks to the use of conditional literals. Moreover, they focus on integrating programs, but do not discuss inconsistency checking or query answering over subprograms. We go a step further and introduce a language extension towards query answering over general subprograms, which is more convenient for average users. Also, their approach can only handle ground programs.

**Related work on modular programming** Related to our approach towards query answering over subprograms are *nested* HEX*-programs*, which allow for accessing answer sets of subprograms using dedicated *external atoms* (Eiter et al. 2013). However, HEX is beyond plain ASP and requires a more sophisticated solver. Similar extensions of ordinary ASP exist (Tari et al. 2005b), but unlike our approach, they did not come with a compilation approach into a single program. Instead, *manifold programs* compile both the meta and the called program into a single one, similarly to our approach (Faber and Woltran 2011). But this work

depends on weak constraints, which are not supported by all systems. Moreover, the encoding requires a separate copy of the subprogram for each atom, while ours requires only a copy for each query.

The idea of representing a subprogram by atoms in the meta-program is related to approaches for ASP debugging (cf. Gebser et al. (2008; Oetsch et al. (2010)). But the actual computation is different: while debugging approaches explain why a particular interpretation is not an answer set (and print the explanation to the user), we aim at detecting the inconsistency and continuing reasoning afterwards. which can help users to get a better understanding of the structure of the problem Also the *stable-unstable semantics* supports an explicit interface to (possibly even nested) oracles (Bogaerts et al. 2016). However, there are no query atoms but the relation between the guessing and checking programs is realized via an extension of the semantics.

Modular programming approaches such as by Tari et al. (2005a) or Janhunen et al. (2014) might appear to be related to our evaluation approach at first glance. However, a major difference is that these approaches provide program modules as language features (i.e., offer them to the user) and thus also define a semantics based on modules. In contrast, we use them just within the solver as an evaluation technique, while the semantics of HEX-programs does not use modules.

Finally, we remark that on an abstract level, the component-wise evaluation might be considered to be related to lazy grounding (see e.g. Palù et al. (2009) and Lefèvre et al. (2017)), as pointed out by colleagues in informal discussions. However, different from traditional (more narrow) definitions of lazy grounding, our approach does not interleave the instantiation of single rules with search, but rather ground whole program components using pregrounding algorithms.

**Related work on evaluation algorithms** Our Algorithm 2 is related to *conflict analysis* in existing CDNL-based algorithms, cf. e.g. Gebser et al. (2012). While both are based on iterative resolution of a conflicting nogood with the implicant of one of its literals, the stop criterion is different. Traditional conflict analysis does the resolution based on *learning schemas* such as *first UIP*, which resolves as long as there are multiple literals assigned at the conflicting decision level. In contrast, we have to take the atoms from the domain into account, from which the facts can come.

Also related are previous evaluation algorithms for HEX-programs. While the previous state-of-the-art algorithm corresponds to Algorithm 1 using $h_\perp(\Delta, \hat{\mathbf{A}}) = \perp$ for parameter $h$, alternative algorithms have been developed as well. One of them was presented by Eiter et al. (2014), who used *support sets* (Darwiche and Marquis 2002) to represent sufficient conditions to make an external atom true. Such support sets are used to 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 external atoms away altogether (Redl 2017c). 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).

**Related work on debugging approaches** Debugging approaches explain why a particular interpretation is not an answer set of a certain program instance and print the explanation to the user to help him/her to find the reason why an expected solution is not an answer set. In contrast, our approach of inconsistency reasons aims at detecting classes of instances which make a program inconsistent. This is motivated by applications which evaluate programs with fixed proper rules under different sets of facts. Then, inconsistency reasons can be exploited to skip evaluations which are known to yield no answer sets, as we did for the sake of improving the evaluation algorithm for HEX-programs (Eiter et al. 2016). Here, in order to handle programs with expanding domains (value invention), the overall program is partitioned into multiple *program components* which are arranged in an acyclic *evaluation graph* that is evaluated top-down. In contrast to that, previous

work on inconsistency analysis was mainly in the context of debugging of answer set programs and faulty systems in general, based on symbolic diagnosis (Reiter 1987). For instance, the approach by Syrjänen (2006) computes inconsistency explanations in terms of either minimal sets of constraints which need to be removed in order to regain consistency, or of odd loops (in the latter case the program is called *incoherent*). The realization is based on another (meta-)ASP-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 the human user to find reasons for undesired behavior of the program. Possible queries are, for instance, why a certain interpretation is not an answer set or why a certain atom is not true in an answer set. Explanations are then in terms of unsatisfied rules, only cyclically supported atoms, or unsupported atoms.

**Related work on inconsistency characterizations** Inconsistency management has also been studied for more specialized formalisms such as for multi-context systems (MCSs) (Eiter et al. 2014) and DL-programs (Eiter et al. 2013).

MCSs are sets of knowledge-bases (called *contexts*), which are implemented in possibly different formalisms that are abstractly identified by their belief sets and interconnected by so-called *bridge rules*. Bridge rules look syntactically similar to ASP rules, but can access different contexts and derive atoms in a context based on information from other contexts. Such systems may become inconsistent even if the individual contexts are all consistent. Inconsistency diagnoses for MCSs use pairs of bridge rules which must be removed resp. added unconditionally (i.e., their body is empty) to make the system consistent.

Inconsistent management for DL-programs is about modifying the Abox of the ontology to restore consistency of the program. While the approaches are related to ours on an abstract level, their technical formalizations are not directly comparable to ours as they refer to specific elements of the respective formalism (such as bridge rules of MCSs and the Abox of ontologies), which do not exist in general logic programs.

Most closely related to our work is a decision criterion on the inconsistency of programs (Redl 2017e), which characterizes inconsistency wrt. models and unfounded sets of the program at hand. In contrast, the notion of IRs from this work characterizes it in terms of the input atoms. However, there is a relation which we elaborate in more detail in the following. To this end we use the following definition and result by Faber (2005):

**Definition 10 (Unfounded Set)** *Given a* HEX*-program $P$ and an assignment $I$, let $U$ be any set of atoms appearing in $P$. Then, $U$ is an* unfounded set for $P$ wrt. $I$ *if, for each rule $r \in P$ with $H(r) \cap U \neq \emptyset$, at least one of the following conditions holds:*

*(i) some literal of $B(r)$ is false wrt. $I$; or*

*(ii) some literal of $B(r)$ is false wrt. $I \setminus U$; or*

*(iii) some atom of $H(r) \setminus U$ is true wrt. $I$.*

Unfounded sets have been used to characterize answer sets as follows. A (classical) model $M$ of a HEX-program $P$ is called *unfounded-free* if it does not intersect with an unfounded set of $P$ wrt. $M$. One can then show (Faber 2005):

**Proposition 18** *A model $M$ of a* HEX*-program $P$ is an answer set of $P$ iff it is unfounded-free.*

We make use of the idea to restrict the sets of head and body atoms which may occur in a program component as introduced by Woltran (2008). For sets of atoms $\mathcal{H}$ and $\mathcal{B}$, let $\mathcal{P}^e_{\langle \mathcal{H},\mathcal{B} \rangle}$ denote the set of all

HEX-programs whose head atoms come only from $\mathcal{H}$ and whose body atoms and external atom input atoms come only from $\mathcal{B}$. We then use the following result by Redl (2017e):

**Proposition 19** *Let $P$ be a* HEX*-program. Then $P \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle \mathcal{H},\mathcal{B} \rangle}$ iff for each classical model $I$ of $P$ there is a nonempty unfounded set $U$ of $P$ wrt. $I$ such that $U \cap I \neq \emptyset$ and $U \cap \mathcal{H} = \emptyset$.*

Intuitively, the result states that a program is inconsistent, iff each (classical) model is dismissed as answer set because it is not unfounded-free. Our concept of IRs is related to this result as follows. A pair $(R^+, R^-)$ of disjoint sets of atoms $R^+ \subseteq D$ and $R^- \subseteq D$ is an IR of program $P$ wrt. $D$ iff $P$ is inconsistent for all additions of facts that contain all atoms from $R^+$ but none from $R^-$; this can be expressed by applying the previous result for $\mathcal{B} = \emptyset$ and an appropriate selection of $\mathcal{H}$.

**Lemma 1** *Let $P$ be a* HEX*-program and $D$ be a domain of atoms. A pair $(R^+, R^-)$ of sets of atoms $R^+ \subseteq D$ and $R^- \subseteq D$ with $R^+ \cap R^- = \emptyset$ is an IR of a* HEX*-program $P$ wrt. $D$ iff $P \cup facts(R^+) \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$.*

Now Proposition 19 and Lemma 1 in combination allow for establishing a relation of IRs in terms input atoms on the one hand, and a characterization in terms of of models and unfounded sets on the other hand.

**Proposition 20** *Let $P$ be a ground* HEX*-program and $D$ be a domain. Then a pair of sets of atoms $(R^+, R^-)$ with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$ is an IR of $P$ iff for all classical models $M$ of $P$ either (i) $R^+ \not\subseteq M$ or (ii) there is a nonempty unfounded set $U$ of $P$ wrt. $M$ such that $U \cap M \neq \emptyset$ and $U \cap (D \setminus R^-) = \emptyset$.*

Intuitively, each classical model $M$ must be excluded from being an answer set of $P \cup facts(F)$ for any $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$. This can either be done by ensuring that $M$ does not satisfy $R^+$ or that some atom $a \in M$ is unfounded if $F$ is added because $a \notin D \setminus R^-$.

While the previous result characterizes inconsistency of a program precisely, it is computationally expensive to apply because one does not only need to check a condition for all models of the program at hand, but also for all unfounded sets of all models. We therefore present a second, simpler condition, which refers only to the program's models but not the the unfounded sets wrt. these models.

**Proposition 21** *Let $P$ be a ground* HEX*-program and $D$ be a domain such that $H(P) \cap D = \emptyset$. For a pair of sets of atoms $(R^+, R^-)$ with $R^+ \subseteq D$ and $R^- \subseteq D$, if for all classical models $M$ of $P$ we either have (i) $R^+ \not\subseteq M$ or (ii) $R^- \cap M \neq \emptyset$, then $(R^+, R^-)$ is an IR of $P$.*

However, in contrast to Proposition 20, note that Proposition 21 does *not* hold in inverse direction. That is, the proposition does not characterize inconsistency exactly but provides a practical means for identifying inconsistency in some cases. This is demonstrated by the following example.

**Example 17** *Let $D = \{x\}$ and $P = \{\leftarrow not\, a\}$. Then $R = (\emptyset, \emptyset)$ is an IR of $P$ because $P$ is inconsistent and no addition of any atoms as facts is allowed. However, the classical model $M = \{a\}$ contains $R^+$ but does not intersect with $R^-$, hence the precondition of Proposition 21 is not satisfied.*

# 9 Conclusion and Outlook

**Summary** In this work have have studied inconsistency of HEX-programs in two steps. First, we focused on fixed programs and decide inconsistency of a subprogram within a meta-program. This was in view

of restrictions of the saturation modeling technique, which allows for exploiting disjunctions for solving co-NP-hard problems that involve checking a property all objects in a given domain. The use of default-negation in saturation encodings turns out to be problematic and a rewriting is not always straightforward. On the other hand, complexity results imply that any co-NP-hard problem can be reduced to brave reasoning over disjunctive ASP. Based on our encoding for consistency checking for normal programs, we realized query answering over subprograms, which paves the way for intuitive encodings of checks which involve default-negation.

Second, we identified classes of inconsistent program instances wrt. their input. To this end, we have introduced the novel notion of *inconsistency reasons* for explaining why an ASP- or HEX-program is inconsistent in terms of input facts. In contrast to previous notions from the area of answer set debugging, which usually explain why a certain interpretation is not an answer set of a concrete program, we consider programs with fixed proper rules, which are instantiated with various data parts. Moreover, while debugging approaches aim at explanations which support human users, ours was developed with respect to optimizations in evaluation algorithms for HEX-programs. More precisely, HEX-programs are typically split into multiple components for the sake of efficient grounding. This, however, harms conflict-driven learning techniques as it introduces barriers for propagation. Inconsistency reasons are used to realize conflict-driven learning techniques for such programs and propagate learned nogoods over multiple components. Our experiments show that the technique is promising and leads to an up to exponential speedup.

**Outlook** For the query answering part, future work includes the application of the extension to non-ground queries. Currently, a separate copy of the subprogram is created for every query atom. However, it might be possible in some cases, to answer multiple queries simultaneously. Another possible starting point for future work is the application of our encoding for a more efficient implementation of nested HEX-programs. Currently, nested HEX-programs are evaluated by separate instances of the reasoner for the calling and the called program. While this approach is strictly more expressive (and thus the evaluation also more expensive) due to the possibility to nest programs up to an arbitrary depth, it is possible in some cases to apply the technique from the paper as an evaluation technique (e.g. if the called program is normal and does not contain further nested calls). Another issue is that if the subprogram is satisfiable, then the meta-program has *multiple* answer sets, each of which representing an answer set of the subprogram. If only consistency resp. inconsistency of the subprogram is relevant for the further reasoning in the meta-program, this leads to the repetition of solutions. In an implementation, this problem can be tackled using projected solution enumeration (Gebser et al. 2009).

For the new learning technique, 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. Moreover, exploiting the structure of the program might allow for finding inconsistency reasons in more cases.

## References

[Biere et al. 2009] BIERE, A., HEULE, M. J. H., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

[Bogaerts et al. 2016] BOGAERTS, B., JANHUNEN, T., AND TASHARROFI, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP 16,* 5-6, 570–586.

[Brain et al. 2007] BRAIN, M., GEBSER, M., PÜHRER, J., SCHAUB, T., TOMPITS, H., AND WOLTRAN, S. 2007. Debugging ASP programs by means of ASP. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR'07), Tempe, AZ, USA*, C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer, 31–43.

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

[Darwiche and Marquis 2002] DARWICHE, A. AND MARQUIS, P. 2002. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR) 17*, 229–264.

[Drescher et al. 2008] DRESCHER, C., GEBSER, M., GROTE, T., KAUFMANN, B., KÖNIG, A., OSTROWSKI, M., AND SCHAUB, T. 2008. Conflict-driven disjunctive answer set solving. In *KR'08*. AAAI Press, 422–432.

[Eiter et al. 2016] EITER, T., FINK, M., IANNI, G., KRENNWALLNER, T., REDL, C., AND SCHÜLLER, P. 2016. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming 16,* 4, 418–464.

[Eiter et al. 2016] EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2016. Domain expansion for ASP-programs with external sources. *Artificial Intelligence 233*, 84–121.

[Eiter et al. 2014] EITER, T., FINK, M., KRENNWALLNER, T., REDL, C., AND SCHÜLLER, P. 2014. Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research 49*, 269–321.

[Eiter et al. 2014] EITER, T., FINK, M., REDL, C., AND STEPANOVA, D. 2014. Exploiting support sets for answer set programs with external evaluations. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, C. E. Brodley and P. Stone, Eds. AAAI Press, 1041–1048.

[Eiter et al. 2014] EITER, T., FINK, M., SCHÜLLER, P., AND WEINZIERL, A. 2014. Finding explanations of inconsistency in multi-context systems. *Artif. Intell. 216*, 233–274.

[Eiter et al. 2013] EITER, T., FINK, M., AND STEPANOVA, D. 2013. *Web Reasoning and Rule Systems: 7th International Conference, RR 2013, Mannheim, Germany, July 27-29, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Inconsistency Management for Description Logic Programs and Beyond, 1–3.

[Eiter and Gottlob 1995] EITER, T. AND GOTTLOB, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15,* 3-4, 289–323.

[Eiter et al. 2009] EITER, T., IANNI, G., AND KRENNWALLNER, T. 2009. Answer Set Programming: A Primer. In *5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009*, S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R. A. Schmidt, Eds. LNCS, vol. 5689. Springer, 40–110.

[Eiter et al. 2013] EITER, T., KRENNWALLNER, T., AND REDL, C. 2013. HEX-programs with nested program calls. In *Proceedings of the Nineteenth International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)* (September 28-30, 2011), H. Tompits, Ed. LNAI, vol. 7773. Springer, 1–10.

[Eiter and Polleres 2006] EITER, T. AND POLLERES, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP 6,* 1-2, 23–60.

[Eiter et al. 2016] EITER, T., REDL, C., AND SCHÜLLER, P. 2016. Problem solving using the HEX family. In *Computational Models of Rationality*. College Publications, 150–174.

[Faber 2005] FABER, W. 2005. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005), Diamante, Italy, September 5-8, 2005*. Vol. 3662. Springer, 40–52.

[Faber et al. 2011] FABER, W., LEONE, N., AND PFEIFER, G. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1 (January), 278–298.

[Faber and Woltran 2011] FABER, W. AND WOLTRAN, S. 2011. Manifold answer-set programs and their applications. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*. Lecture Notes in Computer Science, vol. 6565. Springer, 44–63.

[Gebser et al. 2012] GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

[Gebser et al. 2011] GEBSER, M., KAMINSKI, R., AND SCHAUB, T. 2011. Complex optimization in answer set programming. *CoRR abs/1107.5742*.

[Gebser et al. 2009] GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2009. Solution enumeration for projected boolean search problems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*. 71–86.

[Gebser et al. 2012] GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

[Gebser et al. 2008] GEBSER, M., PUEHRER, J., SCHAUB, T., AND TOMPITS, H. 2008. A meta-programming technique for debugging answer-set programs. In *AAAI-08/IAAI-08 Proceedings*, D. Fox and C. P. Gomes, Eds. 448–453.

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

[Janhunen et al. 2014] JANHUNEN, T., OIKARINEN, E., TOMPITS, H., AND WOLTRAN, S. 2014. Modularity aspects of disjunctive stable models. *CoRR abs/1401.3484*.

[Lefèvre et al. 2017] LEFÈVRE, C., BÉATRIX, C., STÉPHAN, I., AND GARCIA, L. 2017. ASPeRiX, a first-order forward chaining approach for answer set computing. *TPLP 17,* 3, 266–310.

[Lifschitz and Turner 1994] LIFSCHITZ, V. AND TURNER, H. 1994. Splitting a Logic Program. In *Proceedings ICLP-94*. MIT-Press, Santa Margherita Ligure, Italy, 23–38.

[Marques-Silva et al. 2009] MARQUES-SILVA, J. P., LYNCE, I., AND MALIK, S. 2009. *Conflict-Driven Clause Learning SAT Solvers*, Chapter 4, 131–153. Volume 185 of Biere et al. Biere et al. (2009).

[Oetsch et al. (2010] OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *TPLP 10,* 4-6, 513–529.

[Palù et al. (2009] PALÙ, A. D., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: answer set programming with lazy grounding. *Fundam. Inform. 96,* 3, 297–322.

[Redl (2017a] REDL, C. 2017a. Answer set programs with queries over subprograms. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning* (July 3–6, 2017). Springer. To appear.

[Redl (2017b] REDL, C. 2017b. Conflict-driven asp solving with external sources and program splits. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (August 19–25, 2017). Springer. To appear.

[Redl (2017c] REDL, C. 2017c. 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* (February 4–9, 2016). AAAI Press.

[Redl (2017d] REDL, C. 2017d. Explaining inconsistency in answer set programs and extensions. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning* (July 3–6, 2017). Springer. To appear.

[Redl (2017e] REDL, C. 2017e. On equivalence and inconsistency of answer set programs with external sources. In *Proceedings of the Thirty-First AAAI Conference (AAAI 2017), February, 2017, San Francisco, California, USA* (February, 2017). AAAI Press. To appear.

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

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

[Tari et al. (2005a] TARI, L., BARAL, C., AND ANWAR, S. 2005a. A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling. In *Proceedings of the 3Proceedings of the 3rd International ASP'05 Workshop, Bath, UK, 27th–29th July 2005*. CEUR Workshop Proceedings, vol. 142. CEUR WS, 277–293.

[Tari et al. (2005b] TARI, L., BARAL, C., AND ANWAR, S. 2005b. A language for modular answer set programming: Application to ACC tournament scheduling. In *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd Intl. ASP'05 Workshop, Bath, UK, September 27-29, 2005*, M. D. Vos and A. Provetti, Eds. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org.

[Woltran (2008] WOLTRAN, S. 2008. A common view on strong, uniform, and other notions of equivalence in answer-set programming. *TPLP 8,* 2, 217–234.

# A Proofs

**Proposition 1**
*For any ground normal logic program $P$, we have that*
*(1) if $P$ is inconsistent, $M \cup M^P_{gr}$ has exactly one answer set which contains $noAS$; and*
*(2) if $P$ is consistent, $M \cup M^P_{gr}$ has at least one answer set and none of them contains $noAS$.*

**Proof**. (1) If $P$ is inconsistent, then each guess by rules (15) represents an interpretation $I$ which is not an answer set of $P$. For each guess, rules (18)–(19) consider all possible derivation sequences under $T_{P^I}$; acyclicity is guaranteed by minimality of answer sets. Since $I$ is not an answer set of $P$, $I$ is not equivalent to the least model of $P^I$. Hence, we either have (i) $I \neq \mathit{lfp}(T_{P^I})$, or (ii) none of the guessed derivation sequences describes the fixpoint iteration under $T_{P^I}$. Rules (21)–(23) identify rules which are not applicable for justifying their head atom being true, either because they are not in the reduct, their positive body is unsatisfied, or the head atom is derived in an earlier iteration than one of the positive body atoms. derive $noAS$ in such cases, as described in the text Based on this information, rule (24) derives $noAS$ if at least one atom in $I$ cannot be derived as no rule which could derive it is applicable (i.e., $I \not\subseteq \mathit{lfp}(T_{P^I})$), and rule (25) derives $noAS$ if a rule has a satisfied body but its head atom is false in $I$ (i.e., $I \not\supseteq \mathit{lfp}(T_{P^I})$). Then, rules (24) and (25) together derive $noAS$ iff $I \neq \mathit{lfp}(T_{P^I})$.

Rules (26) to (28) saturate whenever $noAS$ is true, i.e., whenever $I \neq \mathit{lfp}(T_{P^I})$. Clearly, if $P$ is inconsistent, then all guesses by rules (15) fail to be answer sets, and thus $noAS$ and all atoms in $M^P$ are derived for all gusses. Then, the saturation model $I_{sat} = A(M^P)$ containing all atoms from $M^P$ is the unique answer set of $M^P$.

(2) If $P$ is consistent, then at least one guess by rules rules (15) represents a valid answer set of $P$. Then $noAS$ is not derived as shown in case (1). By minimality of answer sets, $I_{sat}$ is excluded from being an answer set. □

**Proposition 2**
*For any normal logic program $P$, we have that*
*(1) if $P$ is inconsistent, $M \cup M^P_{ng}$ has exactly one answer set which contains $noAS$; and*
*(2) if $P$ is consistent, $M \cup M^P_{ng}$ has at least one answer set and none of them contains $noAS$.*

**Proof**. The proof follows from Proposition 1 and the observation that $M^P_{ng}$ simulates the construction of $M^P_{gr}$ for the grounded version $P_{gr}$ of $P$. □

**Proposition 3**
*For a normal ASP-program $P$ and a query $q$ we have that (1) $P \models_b q$ iff $P \cup \{\leftarrow \bar{l} \mid l \in q\}$ is consistent; and (2) $P \models_c q$ iff $P \cup \{\leftarrow q\}$ is inconsistent.*

**Proof**. (1) By definition, $P \models_b q$ holds iff $q \subseteq I$ for some answer set $I$ of $P$. The constraints $\{\leftarrow \mathit{not}\, a \mid a \in q\}$ eliminate exactly those answer sets $I$ of $P$, for which $I \models q$ does *not* holds, while answer sets $I$ of $P$, for which $I \models q$ holds, are still answer sets of $P \cup \{\leftarrow \mathit{not}\, a \mid a \in q\}$. Moreover, the addition of constraints cannot yield new answer compared to $P$. Hence, $P$ has an answer set $I$ with $I \models q$ iff $P \cup \{\leftarrow \mathit{not}\, a \mid a \in q\}$ is consistent. (2) By definition, $P \models_c q$ holds iff $I \models q$ for all answer sets $I$ of $P$. The constraint $\{\leftarrow q\}$ eliminates exactly those answer sets $I$ of $P$, for which $I \models q$ holds, while answer sets $I$ of $P$, for which $I \models q$ does *not* hold, are still answer sets of $P \cup \{\leftarrow q\}$. Moreover, the addition of constraints cannot yield new answer compared to $P$. Hence, $P$ has *no* answer set iff every answer set $I$ of $P$ satisfies $q$. □

**Proposition 4**
*For a normal ASP-program $P$ be a predicate and query $q$ we have that*
*(1) $M \cup M_{ng}^{P \cup \{\leftarrow \bar{l} | l \in q\}}$ is consistent and each answer set contains $noAS$ iff $P \not\models_b q$; and*
*(2) $M \cup M_{ng}^{P \cup \{\leftarrow q\}}$ is consistent and each answer set contains $noAS$ iff $P \models_c q$.*

**Proof**. (1) By Proposition 3 we have that $P \models_b q$ iff $P \cup \{\leftarrow \bar{l} \mid l \in q\}$ is consistent (because then at least one answer set of $P$ does not satisfy all negations of the query literals, i.e., satisfies the query). If the program is consistent (and thus $P \models_b q$), then by Proposition 1 it has at least one answer set and none of its answer sets contains atom $noAS$. Conversely, if the program is inconsistent (and thus $P \not\models_b q$), then by Proposition 1 it has exactly (and thus at least) one answer set which contains atom $noAS$.

(2) By Proposition 3 we have that $P \models_c q$ iff $P \cup \{\leftarrow q\}$ is inconsistent (because then every answer set of $P$ violates the added constraint, i.e., satisfies the query). If the program is consistent (and thus $P \not\models_c q$), then by Proposition 1 it has at least one answer set which does not contain atom $noAS$. Conversely, if the program is inconsistent (and thus $P \models_c q$), then by Proposition 1 it has exactly (and thus at least) one answer set which contains atom $noAS$. □

**Proposition 5**
*For a logic program $P$ with query atoms we have that $\mathcal{AS}(P)$ and $\mathcal{AS}([P])$, projected to the atoms in $P$, coincide.*

**Proof**. By Proposition 4, we have that for $S(\mathbf{p}) \vdash_b q$, $\left(M \cup M_{ng}^{S \cup \{\leftarrow \bar{l} | l \in q\}}\right)\big|_{a/a_{S(\mathbf{p}) \vdash_b q}}$ has at least one answer set and in each answer set the atom $noAS_{S(\mathbf{p}) \vdash_b q}$ represents satisfaction of the query $S \models_b q$. Similarly, for $S(\mathbf{p}) \vdash_c q$, $\bigcup_{S(\mathbf{p}) \vdash_c q \text{ in } P} \left(M \cup M_{ng}^{S \cup \{\leftarrow q\}}\right)\big|_{a/a_{S(\mathbf{p}) \vdash_c q}}$ has at least one answer set and in each answer set the atom $noAS_{S(\mathbf{p}) \vdash_c q}$ represents satisfaction of the query $S \models_c q$. Since all copies of $M \cup M_{ng}^{S \cup \{\leftarrow \bar{l} | l \in q\}}$ resp. $M \cup M_{ng}^{S \cup \{\leftarrow q\}}$ use disjoint vocabularies, the program line (38) still has at least one answer set and in each answer set atom $noAS_{S(\mathbf{p}) \vdash_t q}$ represents satisfaction of query $q$ over $S$ for all query atoms $S(\mathbf{p}) \vdash_t q$ in $P$.

Next, line (37) translates the values of $p$-atoms of the calling program to facts in the subprogram using the same encoding as by Definition 5 as for the facts which are directly in $S$. Thus, for given values of the $p$-atoms under the current interpretation $I$, the program in lines (37)–(38) behaves as if $\{p(\mathbf{c}) \leftarrow \mid p(\mathbf{c}) \in I\}$ was already in $S$.

Finally, line (36) uses atoms $S(\mathbf{p}) \vdash_t q$ in place of $S(\mathbf{p}) \vdash_t q$, which, however, correspond to each other one-by-one. Thus, program $[P]$ behaves as desired. □

**Proposition 6**
*For all* HEX*-programs $P$ and domains $D$ such that $P \cup facts(F)$ is inconsistent for some set $F \subseteq D$ of atoms, then there is an IR of $P$ wrt. $D$.*

**Proof**. Take some $F$ such that $P \cup facts(F)$ is inconsistent and consider $R = (R^+, R^-)$ with $R^+ = F$ and $R^- = D \setminus F$. Then the only $F'$ with $R^+ \subseteq F'$ and $R^- \cap F' = \emptyset$ is $F$ itself. But $P \cup facts(F)$ is inconsistent by assumption. □

**Proposition 7**
*Given a* HEX*-program $P$, a domain $D$, and a pair $R = (R^+, R^-)$ of sets of atoms with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$. Deciding if $R$ is an IR of $P$ wrt. $D$ is*

*(i) $\Pi_2^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $coNP$-complete if $P$ is an ordinary disjunction-free program.*

**Proof**. **Hardness:** Checking if $P$ does not have any answer set is a well-known problem that is $\Pi_2^P$-complete for (i) and $coNP$-complete for (ii), see Faber et al. (2011).

We reduce the problem to checking if a given $R = (R^+, R^-)$ is an IR of $P$ wrt. a domain $D$, which shows that the latter cannot be easier. Consider $R = (\emptyset, \emptyset)$ and domain $D = \emptyset$. Then the only $F \subseteq D$ with $\emptyset \subseteq F$ and $D \cap F = \emptyset$ is $F = \emptyset$. Then $R$ is an IR iff $P \cup facts(\emptyset) = P$ is inconsistent, which allows for reducing the inconsistency check to the check of $R$ for being an IR of $P$ wrt. $D$.
**Membership:** Consider $P' = P \cup \{x \leftarrow | \ x \in R^+\} \cup \{x \leftarrow not\, nx;\ nx \leftarrow not\, x \mid x \in D \setminus R^-\}$, where $nx$ is a new atom for all $x \in D$. Then $P'$ has an answer set iff for some $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ we have that $P \cup facts(F)$ has an answer set. Conversely, $P'$ does not have an answer set, iff for all $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ we have that $P \cup facts(F)$ has no answer set, which is by Definition 8 exactly the case iff $R$ is an IR of $P$ wrt. $D$. The observation that checking if $P'$ has no answer set – which is equivalent to deciding if $R$ is an IR – is in $\Pi_2^P$ resp. $coNP$ for (i) resp. (ii), because the property of being ordinary and disjunction-free carries over from $P$ to $P'$, concludes the proof. □

Before we analyze the complexity of deciding if a program has some IR, observe:

**Lemma 2** *There is an IR of a* HEX*-program $P$ wrt. $D$ iff $P \cup facts(F)$ is inconsistent for some $F \subseteq D$.*

**Proof**. ($\Rightarrow$) If $R$ is an IR of $P$ wrt. $D$, then by Definition 8 $P \cup facts(F')$ is inconsistent for all $F' \subseteq D$ with $R^+ \subseteq F'$ and $R^- \cap F' = \emptyset$. This holds in particular for $F = R^+$.

($\Leftarrow$) If $P \cup facts(F)$ is inconsistent for some $F \subseteq D$, then choosing $R = (F, D \setminus F)$ is an IR. This is because the only $F' \subseteq D$ with $R^+ \subseteq F'$ and $R^- \cap F' = \emptyset$ is $F$ itself, and we already know that $P \cup facts(F)$ is inconsistent. □

**Proposition 8**
*Given a* HEX*-program $P$ and a domain $D$. Deciding if there is an IR of $P$ wrt. $D$ is*

*(i) $\Sigma_3^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $\Sigma_2^P$-complete if $P$ is an ordinary disjunction-free program.*

**Proof**. **Hardness:** We first show $\Pi_3^P$-hardness (for (i)) resp. $\Pi_2^P$-hardness (for (ii)) of the complementary problem, which is checking if there is *no* IR of $P$ wrt. $D$. To this end, we reduce satisfiability of an appropriate QBF to our reasoning problem.

(i) We reduce checking satisfiability of a QBF of form $\forall \mathbf{X} \exists \mathbf{Y} \forall \mathbf{Z} \phi(\mathbf{X}, \mathbf{Y}, \mathbf{Z})$ to the check for non-existence of an IR of $P$ wrt. $D$. For a fixed vector of truth values $\mathbf{t}$ for the variables $\mathbf{X}$, we construct the following program disjunctive ordinary ASP-program:

$$
\begin{aligned}
P(\mathbf{t}) = \{ & y \vee ny \leftarrow \text{for all } y \in \mathbf{Y} \\
& z \vee nz \leftarrow \text{for all } z \in \mathbf{Z} \\
& sat_i \leftarrow l_{i,j} \text{ for all literals } l_{i,j} \text{ in } \phi_i \text{and all clauses } \phi_i \text{ in } \phi \\
& sat \leftarrow sat_1, \ldots, sat_\ell \\
& z \leftarrow sat \text{ for all } z \in \mathbf{Z} \\
& nz \leftarrow sat \text{ for all } z \in \mathbf{Z} \\
& \leftarrow not\ sat\} \\
\cup \{ & x_i \leftarrow | \ x_i \in \mathbf{X}, t_i = \top\}
\end{aligned}
$$

Then $P(\mathbf{t})$ has an answer set iff $\exists\mathbf{Y}\forall\mathbf{Z}\phi(\mathbf{t},\mathbf{Y},\mathbf{Z})$ is satisfiable. This is because the values of $x_i$ are forced to the truth value $t_i$ in any answer set of $P(\mathbf{t})$: if $t_i = \top$ then the fact $x_i \leftarrow$ forces $x_i$ to be true, if $t_i = \bot$ then there is no support for $x_i$. Moreover, all $y \in \mathbf{Y}$ are guessed nondeterministically, which resembles existential quantification of $\mathbf{Y}$. Finally, the program employs a saturation encoding for checking if for the current guess $\mathbf{y}$ of $\mathbf{Y}$ we have that $\phi(\mathbf{t},\mathbf{y},\mathbf{z})$ holds for all guesses $\mathbf{z}$ of $\mathbf{Z}$. The atom $sat$ is derived whenever $\phi(\mathbf{t},\mathbf{y},\mathbf{z})$ holds for the current guesses $\mathbf{y}$ of $\mathbf{Y}$ and $\mathbf{z}$ of $\mathbf{Z}$. Minimality of answer sets guarantees that the saturated interpretation, containing atom $sat$, is an answer set of $P(\mathbf{t})$ iff some guess for $y \in \mathbf{Y}$ and all subsequent guesses of values for $z \in \mathbf{Z}$ satisfy $\phi(\mathbf{t},\mathbf{Y},\mathbf{Z})$, which resembles the original quantifications of $\mathbf{Y}$ and $\mathbf{Z}$, respectively. Then, $\forall\mathbf{X}\exists\mathbf{Y}\forall\mathbf{Z}\phi(\mathbf{X},\mathbf{Y},\mathbf{Z})$ is satisfiable iff $P(\mathbf{t})$ has an answer set for all values $\mathbf{t}$ for $\mathbf{X}$. By Lemma 2, this is the case iff there is no IR of $P$ wrt. $D$, Because satisfiability checking of a QBF of the given form is $\Pi_3^P$-hard, the same applies to checking the non-existence of an IR of $P$ wrt. $D$.

(ii) We reduce checking satisfiability of a QBF of form $\forall\mathbf{X}\exists\mathbf{Y}\phi(\mathbf{X},\mathbf{Y})$ to the check for non-existence of an IR of an ordinary *disjunction-free* $P$ wrt. $D$. To this end, one can apply the same program as for case (i) by using the empty vector for $\mathbf{Z}$. Observe that then the program is ordinary and disjunction-free. Then, $\forall\mathbf{X}\exists\mathbf{Y}\phi(\mathbf{X},\mathbf{Y})$ is satisfiable iff $P(\mathbf{t})$ has an answer set for all values $\mathbf{t}$ for $\mathbf{X}$. By Lemma 2, this is the case iff there is no IR of $P$ wrt. $D$, Because satisfiability checking of a QBF of the given form is $\Pi_2^P$-hard, the same applies to checking the non-existence of an IR of $P$ wrt. $D$.

Finally, since the complementary problem of checking the non-existence of an IR is $\Pi_3^P$-hard for (i) and $\Pi_2^P$-hard for (ii), checking existence of such an IR is therefore $\Sigma_3^P$-hard for (i) and $\Sigma_2^P$-hard for (ii).

**Membership:** Let $P$ be a program and $D$ be a domain. In order to decide if $P$ has an IR, one can guess an IR $R = (R^+, R^-)$ of $P$ wrt. $D$ and decide in $\Pi_2^P$ for (i) resp. in $coNP$ for (ii) (cf. Proposition 7) that $F$ is an IR. □

**Proposition 9**
*Given a* HEX*-program $P$, a domain $D$, and a pair $R = (R^+, R^-)$ of sets of atoms with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$. Deciding if $R$ is a subset-minimal IR of $P$ wrt. $D$ is*

*(i) $D_2^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii) $D_1^P$-complete if $P$ is an ordinary disjunction-free program.*

**Proof**. **Hardness:** For (i), let $S$ be a QBF-formula of form $\exists\mathbf{X_S}\forall\mathbf{Y_S}\phi_S(\mathbf{X_S},\mathbf{Y_S})$ and let $U$ be a QBF-formula of form $\exists\mathbf{X_U}\forall\mathbf{Y_U}\phi_U(\mathbf{X_U},\mathbf{Y_U})$; we assume that $\phi_S$ and $\phi_U$ are given in clause normal form. We further assume w.l.o.g. that the sets of variables occurring in $S$ and $U$ are disjoint, i.e., they are standardized apart. Deciding if $S$ is satisfiable and $U$ is unsatisfiable are well-known $\Sigma_2^P$- and $\Pi_2^P$-complete problems, respectively. Thus, deciding both together is complete for $D_2^P$.

We reduce the problem of deciding satisfiability of $S$ and unsatisfiability of $U$ to checking a candidate IR $R = (R^+, R^-)$ for being a subset-minimal IR of a program wrt. a domain, which proves hardness of the

latter problem for class $D_2^P$. To this end, we construct the following ordinary disjunctive program:

$$P = \{ \quad x \vee nx \leftarrow \text{for all } x \in \mathbf{X_S} \cup \mathbf{X_U} \tag{53}$$
$$y \vee ny \leftarrow \text{for all } y \in \mathbf{Y_S} \cup \mathbf{Y_U} \tag{54}$$

$$satU_i \leftarrow l_{i,j} \text{ for all literals } l_{i,j} \text{ in } \phi_i \text{ and all clauses } \phi_i \text{ in } \phi_U \tag{55}$$
$$modelU \leftarrow satU_1, \ldots, satU_\ell \text{ for all clauses } \phi_1, \ldots, \phi_\ell \text{ in } \phi_U \tag{56}$$
$$y \leftarrow modelU \text{ for all } y \in \mathbf{Y_U} \tag{57}$$
$$ny \leftarrow modelU \text{ for all } y \in \mathbf{Y_U} \tag{58}$$
$$v_1 \leftarrow not\ modelU, u \tag{59}$$

$$satS_i \leftarrow l_{i,j} \text{ for all literals } l_{i,j} \text{ in } \phi_i \text{ and all clauses } \phi_i \text{ in } \phi_S \tag{60}$$
$$modelS \leftarrow satS_1, \ldots, satS_\ell \text{ for all clauses } \phi_1, \ldots, \phi_\ell \text{ in } \phi_S \tag{61}$$
$$y \leftarrow modelS \text{ for all } y \in \mathbf{Y_S} \tag{62}$$
$$ny \leftarrow modelS \text{ for all } y \in \mathbf{Y_S} \tag{63}$$
$$v_2 \leftarrow not\ modelS \tag{64}$$
$$v_2 \leftarrow s \tag{65}$$

$$\leftarrow v_1, v_2\} \tag{66}$$

The encoding works as follows. Rules (53) guess an assignment of the existential variables, and rules (54) guess an assignment of the universal variables. For the current assignment, rules (55) and (60) check for each clause $\phi_i$ in $U$ and $S$, respectively, if it is currently satisfied. Based on the results for individual clauses, rules (56) and (61) check for the current assignment of the existential variables $\mathbf{x_U}$ and $\mathbf{x_S}$ if they are models of $\forall \mathbf{Y_U} \phi_U(\mathbf{x_U}, \mathbf{Y_U})$ and $\forall \mathbf{Y_S} \phi_S(\mathbf{x_S}, \mathbf{Y_S})$, respectively.

To this end, rules (57)–(58) and (62)–(63) make use of a saturation encoding (see Section 2) to realize the universal quantification over the variables in $\mathbf{Y_S}$ and $\mathbf{Y_U}$, respectively. More precisely, the saturated interpretation including atoms $modelU$ resp. $modelS$ is derived iff all clauses in $U$ resp. $S$ are satisfied under *all* guesses of $\mathbf{Y_S}$ resp. $\mathbf{Y_U}$; if there is at least one guess which does not satisfy at least one clause, then – for this guess – $modelU$ resp. $modelS$ is not derived, which results in a smaller interpretation than the saturated one. Hence, the saturated interpretation is part of an answer set iff the universal quantification is fulfilled.

Rule (59) derives $v_1$ iff, for the current guess $\mathbf{x_U}$ of the existential variables in $U$, $\forall \mathbf{Y_U} \phi_U(\mathbf{x_U}, \mathbf{Y_U})$ is unsatisfiable *and* $u$ is in the facts added to $P$. Rules (64)–(65) derive $v_2$ iff, for the current guess $\mathbf{x_S}$ of the existential variables in $S$, $\forall \mathbf{Y_S} \phi_S(\mathbf{x_S}, \mathbf{Y_S})$ is unsatisfiable *or* $s$ is in the facts added to $P$.

Suppose we evaluate $P \cup facts(F)$ for some $F \subseteq D$. Note that, since the remaining part of the program is positive, the only way to become inconsistent is a derivation of $v_1$ and $v_2$ and a violation of the constraint (66). We show now that $R = (\{u, s\}, \emptyset)$ is a subset-minimal IR of $P$ wrt. $D = \{u, s\}$ iff $S$ is satisfiable and $U$ is unsatisfiable. To this end we consider four cases for the combinations of $S$ and $U$ being satisfiable or unsatisfiable, respectively, and show that in each case, $R$ behaves as stated by the proposition, i.e., it is a subset-minimal IR of $P$ or not as claimed.

- Case 1: Both $S$ and $U$ are satisfiable.

Then there is a guess of the rules (53) such that $modelU$ is derived by the rules (56), and thus $v_1$ is not derived by the rule (59), which is independent of whether $u \in F$ or not. Hence, the constraint (66) is not violated and $P \cup facts(F)$ is consistent. Thus $R$ is not an IR of $P$ wrt. $D$.

- Case 2: Both $S$ and $U$ are unsatisfiable.
  Then for any guess of the rules (53), we have that $modelU$ is not derived by the rules (56) and $modelS$ is not derived by the rules (61), thus rule (59) derives $v_1$ whenever $u$ is a fact and rule (63) always derives $v_2$. In particular, it is *not* necessary that $s \in F$ in order to derive $v_1$ and $v_2$ and violate the constraint (66). Thus, $R' = (\{u\}, \emptyset)$ is an IR of $P$ wrt. $D$, which proves that $R$ is not subset-minimal.

- Case 3: $S$ is unsatisfiable and $U$ is satisfiable.
  Since $U$ is satisfiable, there is some guess of the rules (53) which leads to derivation of $modelU$ by rules (56). In this case, $v_1$ is *not* derived by rule (59) and the constraint (66) is not violated, independent of $F$. But then in any case $R$ is not an IR of $P$ wrt. $D$.

- Case 4: $S$ is satisfiable and $U$ is unsatisfiable.
  We have that $modelU$ is not derived for any guess and thus rule (59) derives $v_1$ iff $u \in F$. Moreover, there is a guess of the rules (53) such that $modelS$ is derived by the rules (61) and thus $v_2$ is not derived the rules (64), hence $s \in F$ is needed to make sure that $v_2$ is dervied by rule (65). Thus, both $u$ and $s$ must be in $F$ in order to make $P \cup facts(F)$ inconsistent, i.e., $P \cup facts(F)$ is inconsistent iff $s, u \in R^+$. But then $R = (\{u, s\}, \emptyset)$ is a subset-minimal IR of $P$ wrt. $D$.

For (ii), let $S$ be a QBF-formula of form $\exists \mathbf{X_S} \phi_S(\mathbf{X_S})$ and let $U$ be a QBF-formula of form $\exists \mathbf{X_U} \phi_U(\mathbf{X_U})$. We assume again w.l.o.g. that the sets of variables occurring in $S$ and $U$ are disjoint, i.e., they are standardized apart. For this type of formula, deciding if $S$ is satisfiable and $U$ is unsatisfiable are well-known NP- and co-NP-complete problems, respectively. Thus, deciding both is complete for $D_1^P$.

The problem is a special case of case (i) with $\mathbf{Y_S} = \mathbf{Y_U} = \emptyset$. In this case, the sets of rules (57)–(58) and (62)–(63) in the above encoding are empty, which makes the program head-cycle free. Then, the guesses by the disjunctive rules (53)–(54) can be rewritten to unstratified negation and the program becomes normal. Hence, the original problem of deciding whether $S$ is satisfiable and $U$ is unsatisfiable can be reduced to checking a candidate IR for being a subset-minimal IR of an ordinary normal program, which proves hardness of the latter problem for class $D_1^P$.

**Membership:** By Proposition 7, checking if a candidate IR $R = (R^+, R^-)$ is an IR is feasible in $\Pi_2^P$ for (i) and in co-NP for (ii). The problem is thus reducible to an UNSAT-instance of a QBF of form $\exists \mathbf{X} \forall \mathbf{Y} \phi(\mathbf{X}, \mathbf{Y})$ for (i) and $\exists \mathbf{X} \phi(\mathbf{X})$ for (ii), respectively.

In order to check if is also subset-minimal, it suffices to check if no $R' = (R'^+, R'^-)$ with $R'^+ \subseteq R^+$ and $R'^- \subseteq R^-$ and $|(R^+ \cup R^-) \setminus (R'^+ \cup R'^-)| = 1$ is an IR; each such check is feasible in $\Sigma_2^P$ for (i) and NP for (ii) and thus reducible to a SAT-instance of a QBF of form $\exists \mathbf{X} \forall \mathbf{Y} \phi(\mathbf{X}, \mathbf{Y})$ for (i) and $\exists \mathbf{X} \phi(\mathbf{X})$ for (ii), respectively. Since only linearly many such checks are necessary, they can be combined into a single SAT-instance which is only polynomially larger than the original instance.

Therefore, the problem of checking a candidate IR for being a subset-minimal IR can be reduced to a pair of independent SAT- and an UNSAT-instances over QBFs of form $\exists \mathbf{X} \forall \mathbf{Y} \phi(\mathbf{X}, \mathbf{Y})$ for (i) and $\exists \mathbf{X} \phi(\mathbf{X})$ for (ii), respectively, which proves membership results as stated. □

**Proposition 10**
*Given a* HEX*-program $P$ and a domain $D$. Deciding if there is a subset-minimal IR of $P$ wrt. $D$ is*

*(i) $\Sigma_3^P$-complete for general* HEX*-programs and for ordinary disjunctive ASP-programs, and*

*(ii)* $\Sigma_2^P$*-complete if* $P$ *is an ordinary disjunction-free program.*

**Proof**. Each subset-minimal IR is in particular an IR, and if there is an IR $R$ then either $R$ or a smaller one is a subset-minimal IR. Thus, there is an IR iff there is a subset-minimal one and the results by Proposition 8 carry over. □

**Proposition 11**
*Let* $P$ *be an ordinary normal program and* $D$ *be a domain. Then* $(R^+, R^-)$ *is an IR of* $P$ *wrt.* $D$ *iff* $\tau(D, P)$ *has an answer set* $I \supseteq \{a^\sigma \mid \sigma \in \{+, -\}, a \in R^\sigma\}$.

**Proof**. The rules (50) guess all possible explanations $(R^+, R^-)$, where $a^+$ represents $a \in R^+$, $a^-$ represents $a \in R^-$ and $a^x$ represents $a \notin R^+ \cup R^-$. The rules (51) then guess all possible sets of input facts $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ wrt. the previous guess for $(R^+, R^-)$: if $a \in R^+$ then $a$ must be true, if $a \in R^-$ then $a$ cannot be true, and if $a \notin R^+ \cup R^-$ then $a$ can either be true or not.

Now by the properties of $M \cup M^P_{gr}$ we know that $A(M \cup M^P_{gr})$ is an answer set of $M \cup M^P_{gr}$ iff $P$ is inconsistent wrt. the current facts computed by rules (51). By minimality of answer sets, $A(\tau(D, P))$ is an answer set of $\tau(D, P)$.

Rules (52) ensure that also the atoms $a^+$, $a^-$ and $a^x$ are true for all $a \in D$. Since $M \cup M^P_{gr}$ is positive, this does not harm the property of being an answer set wrt. $M \cup M^P_{gr}$. □

**Proposition 12**
*Let* $P$ *be a* HEX*-program and* $D$ *be a domain. Then each IR of* $\hat{P}$ *wrt.* $D$ *is also an IR of* $P$ *wrt.* $D$*, i.e., the use of* $\hat{P}$ *is sound wrt. the computation of IRs.*

**Proof**. As each answer set of $P$ is the projection of a compatible set of $\hat{P}$ to the atoms in $P$, inconsistency of $\hat{P} \cup facts(F)$ implies inconsistency of $\hat{P} \cup facts(F)$ for all $F \subseteq D$. □

**Proposition 13**
*Let* $P$ *be a program and* $F \subseteq D$ *be input atoms from a domain* $D$*. If we have that* HEX-*CDNL-PA*$(P, F, h_\perp)$ *returns (i) an assignment* $\mathbf{A}$*, then* $\mathbf{A}$ *is an answer set of* $P \cup facts(F)$*; (ii)* $\perp$*, then* $P \cup facts(F)$ *is inconsistent.*

**Proof**. See Eiter et al. (2014). □

**Proposition 14**
*Let* $P$ *be a program and* $F \subseteq D$ *be input atoms from a domain* $D$*. If we have that* HEX-*CDNL-PA*$(P, F, h^D_{analyse})$ *returns (i) an assignment* $\mathbf{A}$*, then* $\mathbf{A}$ *is an answer set of* $P \cup facts(F)$*; (ii) a pair* $R = (R^+, R^-)$ *of sets of atoms, then* $P \cup facts(F)$ *is inconsistent and* $R$ *is an inconsistency reason of* $P$ *wrt.* $D$*.*

**Proof**. Case (i) follows from case (i) of Proposition 13.

For case (ii), it suffices to focus on Algorithm 2 and show that InconsistencyAnalysis$(D, \Delta, \hat{\mathbf{A}})$ returns an IR of $P$ wrt. domain $D$, where $\Delta$ is the set of nogoods from Algorithm 1 after ending up at with an assignment $\hat{\mathbf{A}}$ which violates $\Delta$ at decision level 0.

The main idea of the proof is then follows. We show that if we currently evaluate the program with input $F \subseteq D$ and Algorithm 2 returns $R = (R^+, R^-)$, then, whenever HEX-CDNL-PA$(P, J, h^D_{analyse})$ is called for some $J \subseteq D$ with $J \subseteq R^+$ and $J \cap R^- = \emptyset$, the nogood $\delta_0$ initially violated during inconsistency analysis (i.e., the value of $\delta$ before entering the loop in Algorithm 2) can also be added during evaluation of $P \cup facts(J)$. This means that the same situation which has led to the detection of inconsistency can be reconstructed, which implies that also $P \cup facts(J)$ is inconsistent as required.

To this end, first observe that the initial nogood set $\Delta_{P\cup\widehat{facts}(F)}$ over the input $F \subseteq D$ used for inconsistency analysis, and the initial nogood set $\Delta_{P\cup\widehat{facts}(J)}$ for a (possibly) different input $J \subseteq D$ can differ only in unary nogoods over atoms in $D$. This is because $\Delta_{P\cup\widehat{facts}(F)} = \Delta_{\hat{P}} \cup \Delta_{facts(F)}$ and $\Delta_{\widehat{P\cup J}} = \Delta_{\hat{P}} \cup \Delta_{facts(J)}$ due to the fact that Clark's completion is created rule-wise (Clark 1977), and that the singleton loop nogoods for an atom $a$ depend only on rules $r$ with $a \in H(r)$ (Gebser et al. 2012); since $D$ and heads of $P$ do not share any atoms by Definition 8, their singleton loop nogoods are independent. In particular, all nogoods in $\Delta_{\hat{P}}$ are independent of $F$ resp. $J$, and $\Delta_{facts(F)}$ resp. $\Delta_{facts(J)}$ contain a nogood $\{\mathbf{F}a\}$ for each atom atom $a \in D$ which also occurs in $F$ resp. $J$, and a nogood $\{\mathbf{T}a\}$ for each atom $a \in D$ which does not occur in $F$ resp. $J$.

Next, note that all nogoods $N$ learned at a later point contain a literal $\mathbf{T}a$ resp. $\mathbf{F}a$ for each $a \in D$ whose presence resp. absence in $F$ was a prerequesite for the nogood to be added (in the sense that for $J \subseteq D$, if $a \in J$ for each $\mathbf{T}a \in N$ and $a \notin J$ for each $\mathbf{F}a \in N$, then $N$ can also be learned during evaluation of $P \cup facts(J)$ without eliminating answer sets). This is shown by induction on the number $n$ of resolution steps performed to derive $N$. For the base case $n = 0$, nogood $N$ must have been added either (1) due to a violation of the compatibility or minimality criterion at (d), or (2) by theory propagation at (e). In case (1) it contains a literal over each $a \in D$ according to its value in $F$, in case (2) it is independent of $F$ and thus can also be learned when the program is evaluated under a (possibly) different input $J$. For $n \to n+1$, nogood $N$ is the resolvent of two other nogoods $N_1$ and $N_2$; these *cannot* be nogoods from $\Delta_{facts(F)}$ because all in $\Delta_{facts(F)}$ are unary over $D$ but literals over $D$ are never resolved due to assignment at decision level 0. But then the claim holds for $N_1$ and $N_2$ either because they come from $\Delta_{\hat{P}}$ (which is independent of $F$), or because they have been added at at later point, in which case the claim holds by induction hypothesis. Moreover, atoms $D$ are never resolved during conflict analysis because they are assigned at decision level 0 at (a), thus all literals over $D$ contained in $N_1$ and $N_2$ are still contained in $N$. But then the claim holds also for nogood $N$ derived by $n+1$ resolution steps.

But then, the initially violated nogood $\delta_0$ selected by Algorithm 2 also contains a literal $\mathbf{T}a$ resp. $\mathbf{F}a$ for each $a \in D$ whose presence resp. absence in $F$ was a prerequesite for this nogood to be added. Akin to resolution during conflict analysis, the loop in Algorithm 2 does never resolve such literals, hence the final nogood $\delta_1$ (i.e., the value of $\delta$ after the loop) still contains all literals over atoms $D$ whose presence resp. absence in $F$ was a prerequisite for the conflict to be derived. Using the sets $R^+ = \{a \mid \mathbf{T}a \in \delta_1\}$ resp. $R^- = \{a \mid \mathbf{F}a \in \delta_1\}$ for the IR exactly guarantees that atoms, whose presence resp. absence in $F$ was a prerequisite to derive the conflict nogood and all its transitive parent nogoods in the resolution, is still given if the program is evaluated under a (possibly) different input $J \subseteq D$, provided that $J \subseteq R^+$ and $J \cap R^- = \emptyset$. But then, for such a $J$, the same conflict can be reproduced during evaluation of $P \cup facts(J)$, hence $P \cup facts(J)$ is inconsistent. This is exactly what $R = (R^+, R^-)$ being an IR means. □

**Preparation for Proposition 15** As a preparation for the upcoming proofs, we observe that one possible solution for computing IRs of a non-ground program is to use a sufficiently large but finite grounding, which is suitable for answer set computation wrt. any set of input atoms from $D$. More precisely, for a program $P$ and a domain $D$, let $eg_{\mathcal{C},D}(P)$ be an *exhaustive grounding* $P$ with the properties that (i) it is finite, (ii) $eg_{\mathcal{C},D}(P) \subseteq grnd_{\mathcal{C}}(P)$, and (iii) $\mathcal{AS}(eg_{\mathcal{C},D}(P) \cup facts(F)) = \mathcal{AS}(P \cup facts(F))$ for all $F \subseteq D$; safety conditions guarantee that such a grounding exists. We assume w.l.o.g. that $eg_{\mathcal{C},D}(P)$ is minimal, i.e., no proper subset has these properties.

**Example 18 (cont'd)** *For the program $P = \{q(X) \leftarrow p(X);\ \leftarrow not\, q(1);\ \leftarrow a\}$ and domain $D = \{a, p(1)\}$ from Example 12, consider the ground program $eg_{\mathcal{C},D}(P) = \{q(1) \leftarrow p(1);\ \leftarrow not\, q(1);\ \leftarrow a\}$;*

*it is correct for answer set computations over $eg_{\mathcal{C},D}(P) \cup facts(F)$ for all $F \subseteq D$.*

One can then show:

**Lemma 3** *For a program $P$ and a domain $D$, an IR for $eg_{\mathcal{C},D}(P)$ wrt. $D$ is also an IR of $P$ wrt. $D$.*

**Proof**. If $R = (R^+, R^-)$ is an IR of $eg_{\mathcal{C},D}(P)$, then $eg_{\mathcal{C},D}(P) \cup facts(F)$ is inconsistent for all $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$. But $\mathcal{AS}(eg_{\mathcal{C},D}(P) \cup facts(F)) = \mathcal{AS}(P \cup facts(F))$ for all $F \subseteq D$, hence also $P \cup facts(F)$ is inconsistent for all $F \subseteq D$. □

However, since this grounding has to be correct for all possible inputs $F \subseteq D$, can be large and – even worse – grounding external atoms for a yet unknown input is expensive because an exponential number of evaluations is necessary to ensure that all relevant constants are respected in the grounding (Eiter et al. 2016).

Our computation of IRs of a (possibly non-ground) program $P$ wrt. a domain $D$ and a given input $F \subseteq D$ is now based on the grounding $pog_{\mathcal{C},F}(P)$. This has the advantage that the grounding must be computed only for the specific input $F$, while the property that $pog_{\mathcal{C},F}(P) \subseteq grnd_{\mathcal{C}}(P)$ can be exploited to ensure that the IR, which is actually computed for the ground program, carries over to the non-ground program.

We first show that it is possible to replace an atom $a$ in the heads of *some* rules of a ground program $P$ by a new atom $a'$, if a rule $a \leftarrow a'$ is added.

**Lemma 4** *Let $P$ be a ground* HEX*-program, $P_s \subseteq P$ be a subprogram, and $a$ be an atom. We define:*

$$P' = P_s \cup \{a \leftarrow a'\} \cup \{H(r)|_{a \to a'} \leftarrow B(r) \mid r \in P \setminus P_s\},$$

*where $a'$ is a new atom and $H(r)|_{a \to a'}$ is $H(r)$ after replacing $a$ by $a'$. Then $\mathcal{AS}(P')$ and $\mathcal{AS}(P)$ coincide modulo $a'$.*

**Proof**. We first consider the program $P'' = P_s \cup \{a \leftarrow a';\ a' \leftarrow a\} \cup \{H(r)|_{a \to a'} \leftarrow B(r) \mid r \in P \setminus P_s\}$ instead of $P'$, i.e., we also add the reverse rule $a' \leftarrow a$. Then $a$ and $a'$ are equivalent and each answer set $\mathbf{A}$ of $P$ corresponds one-by-one to an answer set $\mathbf{A}'' = \mathbf{A} \cup \{\sigma a' \mid \sigma a \in \mathbf{A}\}$ (the equivalence is also given in the search for smaller models in the reduct because $a \leftarrow a'$ is in the reduct wrt. an assignment iff $a' \leftarrow a$ is).

Now obseve that the only difference between $P''$ and $P'$ is that $a' \leftarrow a$ is missing in $P'$. We show now that (i) each answer set of $P''$ is equivalent to an answer set of $P'$ and (ii) each answer set of $P'$ is equivalent to an answer set of $P'$, both modulo $a'$.

(i) We show that for an answer set $\mathbf{A}''$ of $P''$, either $\mathbf{A}''$ or $(\mathbf{A}'' \setminus \{\mathbf{T}a'\}) \cup \{\mathbf{F}a'\}$ is an answer set of $P'$. Obviously, $\mathbf{A}''$ is a model of $P''$ and thus also of $P'(\subseteq P'')$.

- If $\mathbf{A}''$ is a least model of $fP'^{\mathbf{A}''}$, it is an answer set of $P'$.
- Otherwise, there is a smaller model $\mathbf{A}''_<$ of $fP'^{\mathbf{A}''}$. Then, since $\mathbf{A}''$ is a least model of $fP''^{\mathbf{A}''}$, $fP'^{\mathbf{A}''}$ must differ from $fP''^{\mathbf{A}''}$ (only) in the absence of $a' \leftarrow a$. But then $(\mathbf{A}'' \setminus \{\mathbf{T}a'\}) \cup \{\mathbf{F}a'\}$ must be a least model of $fP'^{\mathbf{A}''}$. This is because since removal of $a' \leftarrow a$ caused $\mathbf{A}''$ to be not a minimal model anymore, then obviously $a'$ lost support. Moreover, none of the remaining atoms can be switched to false in addition: since $a'$ does not occur in any rule body (except for the rule $a \leftarrow a'$; but $a$ must be true because it is true in $\mathbf{A}''$ which is a least model $fP''^{\mathbf{A}''}$) switching it to false cannot lead to further atoms losing their support. In this case, $(\mathbf{A}'' \setminus \{\mathbf{T}a'\}) \cup \{\mathbf{F}a'\}$ is also a model of $P'$ and $fP'^{\mathbf{A}''} = fP'^{\mathbf{A}'}$, hence it is an answer set of $P'$.

(ii) We show that for an answer set $\mathbf{A}'$ of $P'$, either $\mathbf{A}'$ or $(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}$ is an answer set of $P''$.

- If $\mathbf{A}'$ is a model of $P''$, since $\mathbf{A}'$ is a subset-minimal model of $fP'^{\mathbf{A}'}$ and $fP''^{\mathbf{A}'} \supseteq fP'^{\mathbf{A}'}$, it is also a subset-minimal model of $fP''^{\mathbf{A}'}$ and thus an answer set of $P''$.

- If $\mathbf{A}'$ is not a model of $P''$, then, since it is a model of $P'$, rule $a' \leftarrow a \in P''$ must be violated under $\mathbf{A}'$; this implies $\mathbf{T}a \in \mathbf{A}'$. But then, $(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}$ satisfies this rule and is a model of $P''$, because no other rule can be violated by switching $a'$ to true (the only rule which contains $a'$ in its body is $a \leftarrow a'$, but $a$ is also true in $\mathbf{A}'$). We have that $\mathbf{A}'$ is a subset-minimal model of $fP'^{\mathbf{A}'}$. We further have that $fP''^{(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}} = fP'^{\mathbf{A}'} \cup \{a' \leftarrow a\}$. Then $(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}$ must be a subset-minimal model of $fP''^{(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}}$: $a'$ cannot be switched to false due to rule $a' \leftarrow a$ and no other atom can be switched to false because $\mathbf{A}'$ (and thus $(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}$, which coincides with $\mathbf{A}'$ on such atoms) is subset-minimal wrt. $fP'^{\mathbf{A}'}$ (to this end, observe again that $a'$ does not appear in any rule bodies in $P'$ other than in rule $a \leftarrow a'$, hence switching it to true cannot allow any other atom to become false). But then $(\mathbf{A}' \setminus \{\mathbf{F}a'\}) \cup \{\mathbf{T}a'\}$ is an answer set of $P''$.

□

We can now show the main proposition:

**Proposition 15**
*Let $P$ be a program and $F \subseteq D$ be input atoms from a domain $D$. Then an IR $R = (R^+, R^-)$ of $pog_{\mathcal{C},F}(P) \cup \{a \leftarrow a' \mid a \in A(pog_{\mathcal{C},F}(P))\}$ wrt. $D \cup \{a' \mid a \in A(pog_{\mathcal{C},F}(P))\}$ s.t. $(R^+ \cup R^-) \cap \{a' \mid a \in A(pog_{\mathcal{C},F}(P))\} = \emptyset$ is an IR of $P$ wrt. $D$.*

**Proof**. In the following, let $Y = \{a_1, \ldots, a_n\}$ be the set $Y = A(pog_{\mathcal{C},F}(P))$ of atoms in $pog_{\mathcal{C},F}(P)$ and $P_g = pog_{\mathcal{C},F}(P)$. Let $P_{eg} = eg_{\mathcal{C},D}(P)$ be the exhaustive grounding of $P$. Then by Lemma 4 we have that $P_{eg} = P_g \cup \{r \mid r \in P_{eg} \setminus P_g\}$ is equivalent to $P_{eg'} = P_g \cup \{a \leftarrow a' \mid a \in Y\} \cup \{H(r)|_{Y \to Y'} \leftarrow B(r) \mid r \in P_{eg} \setminus P_g\}$.[8] Then for all sets of input atoms $J \subseteq D$ we have that $P_{eg} \cup facts(J)$ and $P_{eg'} \cup facts(J)$ have the same answer sets, modulo atoms $a'$ for $a \in Y$.

The idea of the proof is to show for all nogoods, which are added during inconsistency analysis over $P_g \cup \{a \leftarrow a' \mid a \in Y\} \cup facts(F)$ for some $F \subseteq D$, that they can either also be added during evaluation of $P_{eg'} \cup facts(J)$ for any $J \subseteq D$ with $R^+ \subseteq J$ and $R^- \cap J = \emptyset$ for an IR $R = (R^+, R^-)$ of $P_g$, or contain an atom $a'$ or $\beta_{\{a'\}}$ for $a \in Y$. That is, the inconsistency can either be reconstructed during evaluation of $P_{eg'} \cup facts(J)$ and thus $R$ is also an IR of $P$ wrt. $D$, or $R$ contains an atom which violates the precondition of the proposition, i.e., $R$ is found not to carry over to $P$.

To this end, we first show that all initial nogoods in $\Delta_{P_g \cup \{a \leftarrow a' \mid a \in Y\}}$ are either (i) also in $\Delta_{P_{eg'}}$, or (ii) contain $a'$ or an $\beta_{\{a'\}}$ (the auxiliary variable representing the body of $a \leftarrow a'$ (Clark 1977)) for some $a \in Y$. Consider a nogood in $\Delta_{P_g \cup \{a \leftarrow a' \mid a \in Y\}}$. If the nogood comes from Clark's completion of $P_g$, then it is also in $\Delta_{P_{eg'}}$ because $P_{eg'} \supseteq P_g$ and Clark's completion is created rule-wise, cf. Clark (1977). If the nogood comes from Clark's completion of $\{a \leftarrow a' \mid a \in Y\}$, then it contains $a'$ or $\beta_{\{a'\}}$ for some $a \in Y$. If the nogood comes from the singleton loop nogood for an atom other than those in $Y$, it is also in $\Delta_{P_{eg'}}$ since the nogood depends only on rules which contain such an atom in their head (Drescher et al. 2008); however, $P_{eg} \setminus P_g$ does not define such an atom because $Y$ includes all ud-atoms (these are all atoms appearing in a head of $P_{eg} \setminus P_g$) and each $a \in Y$ in a head of $P_{eg} \setminus P_g$ is replaced by $a'$. If the nogood comes from the

[8] Here, $H(r)|_{Y \to Y'}$ abbreviates $H(r)|_{a_1 \to a'_1}|_{\cdots}|_{a_n \to a'_n}$.

singleton loop nogood for an atom $a \in Y$, then it contains $a'$ or $\beta_{\{a'\}}$ due to the rule $a \leftarrow a'$. Therefore, all nogoods in $\Delta_{P_g \cup \{a \leftarrow a' | a \in Y\}}$ are either (i) also in $\Delta_{P_{eg'}}$ or (ii) contain $a'$ or an $\beta_{\{a'\}}$ for some $a \in Y$.

Moreover, all nogoods $N$ learned at a later point either contain a literal $\mathbf{T}a'$ resp. $\mathbf{F}a'$ for some $a \in Y$, or can also be learned when performing inconsistency analysis over $P_{eg'} \cup facts(J)$ for some $J \subseteq D$. This is shown by induction on the number $n$ of resolution steps performed to derive $N$. For the base case $n = 0$, nogood $N$ must have been added either (i) due to a violation of the compatibility or minimality criterion at (d), or (ii) by theory propagation at (e). In case (i) it contains a literal $\mathbf{F}a'$ for all $a \in Y$, in case (ii) the nogood is independent of rules in $P_{eg} \setminus P_g$. For $n \to n+1$, nogood $N$ is the resolvent of two other nogoods $N_1$ and $N_2$. But then the claim holds for $N_1$ and $N_2$ either because they come from $\Delta_{P_g \cup \{a \leftarrow a' | a \in Y\}}$ (in which case they also also in $\Delta_{P_{eg'}}$ as shown above), or because they have been added at at later point, in which case the claim holds by induction hypothesis. Moreover, atoms $a'$ are never resolved during conflict analysis because they are assigned at decision level 0 at (a), thus all literals $a'$ for $a \in Y$ contained in $N_1$ and $N_2$ are still contained in $N$. But then the claim holds also for nogood $N$ derived by $n+1$ resolution steps.

But then each IR $R = (R^+, R^-)$ of $P_g \cup \{a \leftarrow a' \mid a \in Y\}$ with $(R^+ \cup R^-) \cap \{a', \beta_{\{a'\}} \mid a \in Y\} = \emptyset$ is also an IR of $P_{eg'}$ because the conflict can also be derived during conflict analysis over $P_{eg'} \cup facts(J)$ for some $J \subseteq D$. Moreover, since $P_{eg}$ is equivalent to $P_{eg'}$ modulo atoms $a'$ for $a \in Y$, each IR of $P_{eg'}$ is an IR (wrt. $D$) of $P_{eg}$ and thus, by definition of the semantics of non-ground programs, also of $P$. Hence, an IR $R = (R^+, R^-)$ of $P_g \cup \{a \leftarrow a' \mid a \in Y\}$ with $(R^+ \cup R^-) \cap \{a', \beta_{\{a'\}} \mid a \in Y\} = \emptyset$ is also an IR of $P$. Finally, since during conflict analysis all atoms are assigned at decision level 0 and $\beta_{\{a'\}}$ occurs only in binary nogoods for all $a \in Y$, it will always have an implicant and will be resolved away, hence, it suffices to check the candidate IR $R = (R^+, R^-)$ for $(R^+ \cup R^-) \cap \{a' \mid a \in Y\} = \emptyset$. □

**Proposition 16**
*For all* HEX*-programs $P$ and IRs $R = (R^+, R^-)$ of $P$ wrt. a domain $D$, we have that $\mathcal{AS}(P \cup facts(F)) = \mathcal{AS}(P \cup \{c_R\} \cup facts(F))$ for all $F \subseteq D$.*

**Proof**. We have that $\mathcal{AS}(P \cup facts(F)) \supseteq \mathcal{AS}(P \cup \{c_R\} \cup facts(F))$ since the addition of constraints can never generate additional answer sets but only eliminate them. Thus, it suffices to show that all answer sets of $\mathcal{AS}(P \cup facts(F))$ are also answer sets of $\mathcal{AS}(P \cup \{c_R\} \cup facts(F))$.

To this end, note that the constraint $c_R$ is violated by $F \subseteq D$ iff $R^+ \subseteq F$ and $R^- \cap F = \emptyset$. However, since $R$ is an IR, by Definition 8 we have that $P \cup facts(F)$ is inconsistent anyway for such an $F$, hence the addition of $c_R$ cannot eliminate answer sets of $\mathcal{AS}(P \cup facts(F))$. □

**Proposition 17**
*For an evaluation unit $u$ and IR $R = (R^+, R^-)$ of $u$ wrt. a domain $D$, constraint $c_R$ can be added to all predecessor units $u'$ s.t. all atoms in $D$ are defined in $u'$ or one of its own transitive predecessors.*

**Proof**. If $c_R$ is violated under the current input $F$ of unit $u$, $u \cup facts(F)$ is inconsistent by Proposition 16. Due to acyclicity of the dependency graph, atoms defined in $u'$ or one of its own predecessors cannot be redefined in successor units of $u'$. Hence, the truth values of such atoms in $F$ are definitely known once $u'$ was evaluated. But then adding $c_R$ to $u$ to $u'$ does not eliminate answer sets since $u \cup facts(F)$ is inconsistent anyway. □

**Proposition 18**
*A model $M$ of a* HEX*-program $P$ is an answer set of $P$ iff it is unfounded-free.*

**Proof**. See Faber (2005). □

**Proposition 19**
*Let $P$ be a* HEX*-program. Then $P \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle \mathcal{H},\mathcal{B} \rangle}$ iff for each classical model $I$ of $P$ there is a nonempty unfounded set $U$ of $P$ wrt. $I$ such that $U \cap I \neq \emptyset$ and $U \cap \mathcal{H} = \emptyset$.*

**Proof**. See Redl (2017e). □

**Lemma 1**
*Let $P$ be a* HEX*-program and $D$ be a domain of atoms. A pair $(R^+, R^-)$ of sets of atoms $R^+ \subseteq D$ and $R^- \subseteq D$ with $R^+ \cap R^- = \emptyset$ is an IR of a* HEX*-program $P$ wrt. $D$ iff $P \cup facts(R^+) \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$.*

**Proof**. The claim follows basically from the observation that the sets of programs allowed to be added on both sides are the same. In more detail:

($\Rightarrow$) Let $(R^+, R^-)$ be an IR of $P$ with $R^+ \subseteq D$ and $R^- \subseteq D$ with $R^+ \cap R^- = \emptyset$. Let $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$. We have to show that $P \cup facts(R^+) \cup R$ is inconsistent. Take $F = R^+ \cup \{a \mid a \leftarrow\in R\}$; then $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ and thus by our precondition that $(R^+, R^-)$ is an IR we have that $P \cup facts(F)$, which is equivalent to $P \cup facts(R^+) \cup R$, is inconsistent.

($\Leftarrow$) Suppose $P \cup facts(R^+) \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$. Let $F \subseteq D$ such that $R^+ \subseteq F$ and $R^- \cap F = \emptyset$. We have to show that $P \cup facts(F)$ is inconsistent. Take $R = \{a \leftarrow\mid a \in F \setminus R^+\}$ and observe that $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$. By our precondition we have that $P \cup facts(R^+) \cup R$ is inconsistent. The observation that the latter is equivalent to $P \cup facts(F)$ proves the claim. □

**Proposition 20**
*Let $P$ be a ground* HEX*-program and $D$ be a domain. Then a pair of sets of atoms $(R^+, R^-)$ with $R^+ \subseteq D$, $R^- \subseteq D$ and $R^+ \cap R^- = \emptyset$ is an IR of $P$ iff for all classical models $M$ of $P$ either (i) $R^+ \not\subseteq M$ or (ii) there is a nonempty unfounded set $U$ of $P$ wrt. $M$ such that $U \cap M \neq \emptyset$ and $U \cap (D \setminus R^-) = \emptyset$.*

**Proof**. ($\Rightarrow$) Let $(R^+, R^-)$ be an IR of $P$ wrt. $D$. Consider a classical model $M$ of $P$. We show that one of (i) or (ii) is satisfied.

If $M$ is not a classical model of $P \cup facts(R^+)$, then, since $M$ is a model of $P$, we have that $R^+ \not\subseteq M$ and thus Condition (i) is satisfied.

In case that $M$ is a classical model of $P \cup facts(R^+)$, first observe that, since $(R^+, R^-)$ is an IR, by Lemma 1 $P \cup facts(R^+) \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$. By Proposition 19, this is further the case iff for each classical model $M'$ of $P \cup facts(R^+)$ there is a nonempty unfounded set $U$ such that $U \cap M' \neq \emptyset$ and $U \cap (D \setminus R^-) = \emptyset$. Since $M$ is a model of $P \cup facts(R^+)$ it follows that an unfounded set as required by Condition (ii) exists.

($\Leftarrow$) Let $(R^+, R^-)$ be a pair of sets of atoms such that for all classical models $M$ of $P$ either (i) or (ii) holds. We have to show that it is an IR of $P$, i.e., $P \cup facts(F)$ is inconsistent for all $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$.

Assignments which are no classical models of $P \cup facts(F)$ cannot be answer sets, thus it suffices to show for all classical models of $P \cup facts(F)$ that they are no answer sets. Consider an arbitrary but fixed $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$ and let $M$ be an arbitrary classical model of $P \cup facts(F)$. Then $M$ is also a classical model of $P$ and thus, by our precondition, either (i) or (ii) holds.

If (i) holds, then there is an $a \in R^+$ such that $a \notin M$. But since $R^+ \subseteq F$ we have $a \leftarrow\in facts(F)$ and thus $M$ cannot be a classical model and therefore no answer set of $P \cup facts(F)$. If (ii) holds, then there is a nonempty unfounded set $U$ of $P$ wrt. $M$ such that $U \cap M \neq \emptyset$ and $U \cap (D \setminus R^-) = \emptyset$, i.e., $U$ does not contain elements from $D \setminus R^-$. Then $U$ is also an unfounded set of $P \cup facts(R^+)$ wrt. $M$. Then by

Proposition 19 we have that $P \cup facts(R^+) \cup R$ is inconsistent for all $R \in \mathcal{P}^e_{\langle D \setminus R^-, \emptyset \rangle}$. The latter is, by Lemma 1, the case iff $(R^+, R^-)$ be an IR of $P$ wrt. $D$. □

**Proposition 21**
*Let $P$ be a ground* HEX*-program and $D$ be a domain such that $H(P) \cap D = \emptyset$. For a pair of sets of atoms $(R^+, R^-)$ with $R^+ \subseteq D$ and $R^- \subseteq D$, if for all classical models $M$ of $P$ we either have (i) $R^+ \not\subseteq M$ or (ii) $R^- \cap M \neq \emptyset$, then $(R^+, R^-)$ is an IR of $P$.*

**Proof**. Let $(R^+, R^-)$ be a pair of sets of atoms such that for all classical models $M$ of $P$ we either have $R^+ \not\subseteq M$ or $R^- \cap M \neq \emptyset$. We have to show that it is an IR of $P$, i.e., $P \cup facts(F)$ is inconsistent for all $F \subseteq D$ with $R^+ \subseteq F$ and $R^- \cap F = \emptyset$.

Assignments which are no classical models of $P \cup facts(F)$ cannot be answer sets, thus it suffices to show for all classical models that they are no answer sets. Let $M$ be an arbitrary classical model of $P \cup facts(F)$. Then $M$ is also a classical model of $P$ and thus, by our precondition, either (i) or (ii) holds.

If (i) holds, then there is an $a \in R^+$ such that $a \notin M$. But since $R^+ \subseteq F$ we have $a \leftarrow\in facts(F)$ and thus $M$ cannot be a classical model and therefore no answer set of $P \cup facts(F)$. If (ii) holds, then there is an $a \in R^- \cap M$. Since $R^- \cap F = \emptyset$ we have that $a \leftarrow\notin facts(F)$. Moreover, we have that $a \notin H(P)$ by our precondition that $H(P) \cap D = \emptyset$. But then $\{a\}$ is an unfounded set of $P \cup facts(F)$ wrt. $M$ such that $\{a\} \cap M \neq \emptyset$ and thus, by Proposition 18, $M$ is not an answer set of $P$. □