Answer Set Programs with Queries over Subprograms*

Christoph Redl

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

Abstract. Answer-Set Programming (ASP) is a declarative programming paradigm. In this paper we discuss two related restrictions and present a novel modeling technique to overcome them: (1) Meta-reasoning about the collection of answer sets of a program is in general only possible by external postprocessing, but not within the program. This prohibits the direct continuation of reasoning based on the answer to the query over a (sub)program's answer sets. (2) The saturation programming technique exploits the minimality criterion for answer sets of a disjunctive ASP program to solve co-NP-hard problems, which typically involve checking if a property holds *for all* objects in a certain domain. However, the technique is advanced and not easily applicable by average ASP users; moreover, the use of default-negation within saturation encodings is limited.
In this paper, we present an approach which allows for brave and cautious query answering over normal subprograms within a disjunctive program in order to address restriction (1). The query answer is represented by a dedicated atom within each answer set of the overall program, which paves the way also for a more intuitive alternative to saturation encodings and allows also using default-negation within such encodings, which addresses restriction (2).

Keywords: Answer Set Programming, Nonmonotonic Reasoning, FLP Semantics, Meta Programming, Saturation

1 Introduction

Answer-Set Programming (ASP) is a declarative programming paradigm based on nonmonotonic programs and a multi-model semantics [13]. The problem at hand is encoded as an ASP program whose models, called *answer sets*, correspond one-to-one to the solutions of the problem. In this paper we discuss two reasoning resp. modeling restrictions, which turn out to be related.

The first restriction concerns *meta-reasoning* about the answer sets of a *(sub)program* within another *(meta-)program*, such as aggregation of results. This is usually done during 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,

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

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 [8]. 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 [4]. However, this approach is based on HEX-programs [6] – an extension of ASP – and not applicable if an ordinary ASP solver is used. Moreover, the meta- and the subprogram are evaluated by two isolated reasoner instances, which may harm efficient evaluation.

The second restriction concerns the *saturation technique* (cf. e.g. [3]), which is a modeling technique that allows for solving co-NP-hard problems within disjunctive ASP. To this end, minimality of answer sets is exploited to check if a property holds *for all* objects in a certain domain. However, the technique is advanced and not easily applicable by average ASP users. Moreover, the use of default-negation for checking properties within saturation encodings is restricted as it may harm the support of atoms. Then, default-negation needs to be rewritten, but it is not always obvious how this can be done. This calls for an alternative to saturation, which hides this rewriting from the user.

In this paper, we first present an **encoding which allows for deciding inconsistency of a normal logic program within a disjunctive program**. Inconsistency resp. consistency of the subprogram is represented by a dedicated atom within each answer set of the overall program. This encoding is then exploited to realize **query answering over normal subprograms within disjunctive ASP**; in contrast to related approaches (e.g. [1], see Section 6), ours makes such queries more explicit, which is easier to understand for users. While the encoding itself is based on the saturation technique, once it is defined, it can be flexibly used for query answering without deep knowledge about the saturation technique. This results in a new modeling technique as alternative to saturation, which supports unrestricted use of default-negation.

We proceed as follows:

- In Section 2 we recapitulate answer set programming and the saturation technique.
- In Section 3 we discuss restrictions of saturation and point out that using default-negation within saturation encodings would be convenient but is not easily possible.
- In Section 4 we 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 5 we discuss query answering based on this encoding. To this end, we first realize brave and cautious query answering over a subprogram in Section 5.1. This feature is then further exploited in Section 5.2 for realizing a new modeling technique as an alternative to saturation, but which supports default-negation. The encoding can be used as a black box at this point such that the user does not need to have deep knowledge about the underlying ideas. Instead, checking if a property holds *for all* objects in a domain can be naturally expressed as a cautious query.
- In Section 6 we discuss related work.

– In Section 7 we conclude and give an outlook on future work.

Proofs are outsourced to http://www.kr.tuwien.ac.at/research/projects/inthex/qa-ext.pdf.

2 Preliminaries

We first recapitulate answer set programming and the saturation technique.

Answer Set Programming. Our alphabet consists of possibly infinite sets of constant symbols $\mathscr{C}$ (including all integers), variables $\mathscr{V}$, function symbols $\mathscr{F}$, and predicate symbols $\mathscr{P}$. We assume that $\mathscr{V}$ is disjoint from all other sets, while symbols may be shared between the other sets. We let the set of terms $\mathscr{T}$ be the least set such that $\mathscr{C} \subseteq \mathscr{T}$, $\mathscr{V} \subseteq \mathscr{T}$, and $f \in \mathscr{F}$, $T_1, \ldots, T_\ell \in \mathscr{T}$ implies $f(T_1, \ldots, T_\ell) \in \mathscr{T}$. An (ordinary) atom is of form $p(t_1, \ldots, t_\ell)$ with predicate symbol $p \in \mathscr{P}$ and terms $t_1, \ldots, t_\ell \in \mathscr{T}$, abbreviated as $p(\mathbf{t})$; we write $t \in \mathbf{t}$ if $t = t_i$ for some $1 \leq i \leq \ell$. A term resp. atom is called *ground* if it does not contain variables.

An *interpretation* over the (finite) set $\mathscr{A}$ of ground atoms is a set $I \subseteq \mathscr{A}$, where $a \in I$ expresses that a is true and $a \notin I$ that a is false. A builtin atom is of form $t_1 \circ t_2$ with terms $t_1, t_2 \in \mathscr{T}$ and comparison operator $\circ \in \{=, \neq, <, \leq, \geq, >\}$. For a ground builtin atom $t_1 \circ t_2$ and an interpretation I we have that $I \models t_1 = t_2$ if t_1 is equal to t_2 and $I \not\models t_1 = t_2$ otherwise; conversely for $I \models t_1 \neq t_2$. Operators $<$, $\leq$, $\geq$ and $>$ have the standard semantics and are defined only if t_1 and t_2 are integers. Similarily, arithmetic atoms are of form $t_1 \circ t_2 \circ' t_3$ with integer terms $t_1, t_2, r_3 \in \mathscr{T}$, comparison operator $\circ \in \{=, \neq, <, \leq, \geq, >\}$ and arithmetic operator $\circ' \in \{+, -, *, /\}$, which have the standard semantics.

We now recall disjunctive logic programs under the answer set semantics [13].

Definition 1. *An* answer set program *P consists of rules*

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

where each a_i is an atom, and each b_j is an atom or a builtin atom. A program is called normal *if $k \leq 1$ for all rules, and* disjunctive *otherwise.*

A rule resp. program is ground if it contains only ground atoms. Interpretations I are over the atoms $A(P)$ occurring in the ground program P at hand. A ground rule r of form (1) is satisfied under I, denoted $I \models r$, if $a_i \in I$ for some $1 \leq i \leq k$, or $b_i \notin I$ for some $1 \leq i \leq m$, or $b_i \in I$ for some $m+1 \leq i \leq n$. A ground program P is satisfied under I, denoted $I \models P$, if each $r \in P$ is satisfied under I. For such a rule r we let $H(r) = \{a_1, \ldots, a_k\}$ be its *head*, $B^+(r) = \{b_1, \ldots, b_m\}$ be its *positive body* and $B^-(r) = \{b_{m+1}, \ldots, b_n\}$ be its *negative body*.

The answer sets of a ground program P are defined as follows. The *(GL-)reduct* [13] of P wrt. interpretation I is the set $P^I = \{H(r) \leftarrow B^+(r) \mid r \in \Pi, I \not\models b \text{ for all } b \in B^-(r)\}$.

Definition 2. *An interpretation I is an answer set of a ground program P, if I is a $\subseteq$-minimal model of P^I.*

Note that for a normal program P and an interpretation I, the reduct P^I is a positive program. This allows for an alternative characterization of answer sets of normal logic programs based on fixpoint iteration. For a positive normal 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 is an answer set of a normal logic program P if $I = lfp(T_{P^I})$.

The answer sets of a non-ground program P are given by those of its *grounding* $grnd(P)$, which results from P if all variables are replaced by terms in all possible ways.

Saturation Technique. The saturation technique dates back to the Σ_2^{P}-hardness proof of disjunctive ASP [2], but was later exploited as a modeling technique, cf. e.g. [3]. It is applied for solving co-NP-hard problems, which typically involve checking a condition *for all* objects in a domain. Importantly, such a check *cannot* be encoded in a *normal* logic program such 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. This limitation inhibits that reasoning continues *within* the program after checking the property. Instead, non-existence of answer sets needs to be determined in the postprocessing.

A concrete example is checking if a given graph is *not* 3-colorable. Consider

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

where the graph is supposed to be defined by facts F over predicates *node* and *edge*. Its answer sets correspond one-to-one to valid 3-colorings. Thus, the program does *not* have an answer set if and only if there is no valid 3-coloring. However, it is not possible to define a normal program with an answer set that represents that there is *no* such coloring.

This is only possible with disjunctive programs and the saturation technique. 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 saturation atom *sat* in this case. A third program component P_{sat} derives all atoms from 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 $P_{guess} \cup P_{check} \cup P_{sat}$. 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 1. 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. □

3 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 Σ_2^{P}-complete [7]), 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. This is because saturation works only if I_{sat} is an answer set of $P_{guess} \cup P_{check} \cup P_{sat}$ whenever no proper subset is one. While this is guaranteed if no default-negation occurs in I_{sat}, it might be unstable otherwise.

Example 2. 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:

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

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 $P_{vc}^{I_{sat}}$ 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 $\mathrm{not}\, in(X)$ and $\mathrm{not}\, in(Y)$ 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 does not directly concern the guessed atoms but derived ones.

Example 3. A *Hamiltonian cycle* in a 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:

$$P_{guess} = \{in(X,Y) \vee out(X,Y) \leftarrow arc(X,Y)\} \tag{2}$$
$$P_{check} = \{sat \leftarrow in(Y_1,X), in(Y_2,X), Y1 \neq Y2;\ sat \leftarrow in(X,Y_1), in(X,Y_2), Y1 \neq Y2 \tag{3}$$
$$sat \leftarrow node(X), \mathrm{not}\, hasIn(X);\ sat \leftarrow node(X), \mathrm{not}\, hasOut(X) \tag{4}$$
$$hasIn(X) \leftarrow node(X), in(Y,X);\ hasOut(X) \leftarrow node(X), in(X,Y)\} \tag{5}$$
$$P_{sat} = \{in(X,Y) \leftarrow arc(X,Y), sat;\ out(X,Y) \leftarrow arc(X,Y), sat\} \tag{6}$$

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 (3)) and at least (rules (4)) one incoming/outgoing arc. While the check if a node has at most one incoming/outgoing arcs is possible using the positive rules (3), the check if a node has at least one incoming/outgoing edge is more involved. In contrast

to the check in Example 2, one cannot reasonably perform it based on the atoms from P_{guess} alone. Instead, auxiliary predicates *hasIn* and *hasOut* are defined by rules (5). Unlike $in(\cdot,\cdot)$, the negation of $hasIn(\cdot)$ and $hasOut(\cdot)$ is not explicitly represented, thus default-negation is used in rules (4) 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 (4) from the reduct wrt. I_{sat}, which in turn has a smaller model. □

Note that in the previous example, for a fixed node X the literal $\mathrm{not}\, hasOut(X)$ 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 $sat \leftarrow node(X), \mathrm{not}\, hasOut(X)$ 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 $(x,y_i) \in E$.[1] But this is not always possible:

Example 4. Deciding if a ground normal ASP program P is inconsistent is co-NP-complete. An attempt to apply the saturation technique is as follows:

$$P' = \{true(a) \vee false(a) | a \in A(P)\} \tag{7}$$
$$\cup \{inReduct(r) \leftarrow \{false(b) \mid b \in B^-(r)\} \mid r \in P\} \tag{8}$$
$$\cup \{leastModel(a) \leftarrow inReduct(r), \{leastModel(b) | b \in B^+(r)\} | r \in P, a \in H(r)\} \tag{9}$$
$$\cup \{noAS \leftarrow false(a), leastModel(a) \mid a \in A(P)\} \tag{10}$$
$$\cup \{noAS \leftarrow true(a), \mathrm{not}\, leastModel(a) \mid a \in A(P)\} \tag{11}$$
$$\cup \{true(a) \leftarrow noAS;\ false(a) \leftarrow noAS \mid a \in A(P)\} \tag{12}$$
$$\cup \{inReduct(r) \leftarrow noAS\} \tag{13}$$

The idea is to guess all possible interpretations I over the atoms $A(P)$ in P (rules (7)). Next, rules (8) identify the rules $r \in P$ which are in P^I (modulo $B^-(r)$); these are all rules $r \in P$ whose atoms $B^-(r)$ are all false. Rules (9) compute the least model of the reduct by simulating fixpoint iteration under operator T_P. Rules (10) and (11) compare the least model of the reduct to I: if this comparison fails, then I is not an answer set and rules (12) and (13) saturate. However, the comparison of the least model of the reduct to the original guess in rule (11) uses default-negation. In contrast to Example 3, it is not straightforward how to eliminate the negation, 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 section, we present a valid encoding for checking inconsistency of normal programs, as discussed in Example 4, within disjunctive ASP.

4 Deciding Inconsistency of Normal Programs in Disjunctive ASP

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 dedicated answer sets represents that P is inconsistent.

[1] On the non-ground level, this might be simulated using *conditional literals* as supported by some reasoners, cf. [9] and below.

4.1 A Meta-Program for Propositional Programs

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 only 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 new atom in the meta-program. For simplicity, we further disallow constraints $\leftarrow b_1,\ldots,b_m,\text{not } b_{m+1},\ldots,\text{not } b_n$ (rules with empty head) in P. This is also w.l.o.g. because such a constraint can be seen as an abbreviation for $x \leftarrow b_1,\ldots,b_m,\text{not } b_{m+1},\ldots,\text{not } b_n,\text{not}\, x$, where x is a new ground atom which does not appear elsewhere in the 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 section 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 then discuss its components.

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

$$M_{extract} = \{atom(X) \leftarrow head(R,X);\ atom(X) \leftarrow bodyP(R,X);\ atom(X) \leftarrow bodyN(R,X)\} \tag{14}$$
$$\cup\ \{rule(R) \leftarrow head(R,X);\ rule(R) \leftarrow bodyP(R,X);\ rule(R) \leftarrow bodyN(R,X)\} \tag{15}$$
$$M_{guess} = \{true(X) \vee false(X) \leftarrow atom(X)\} \tag{16}$$
$$M_{check} = \{inReduct(R) \leftarrow rule(R), (false(X) : bodyN(R,X))\} \tag{17}$$
$$\cup\ \{outReduct(R) \leftarrow rule(R), bodyN(R,X), true(X)\} \tag{18}$$
$$\cup\ \{iter(X,I) \vee niter(X,I) \leftarrow true(X), int(I);\ niter(X,I) \leftarrow false(X), int(I)\} \tag{19}$$
$$\cup\ \{notApp(R) \leftarrow outReduct(R)\} \tag{20}$$
$$\cup\ \{notApp(R) \leftarrow inReduct(R), bodyP(R,X), false(X)\} \tag{21}$$
$$\cup\ \{notApp(R) \leftarrow head(R,X_1), bodyP(R,X_2), iter(X_1,I_1), iter(X_2,I_2), I_2 \geq I_1\} \tag{22}$$
$$\cup\ \{noAS \leftarrow true(X), (notApp(R) : head(R,X))\} \tag{23}$$
$$\cup\ \{noAS \leftarrow inReduct(R), head(R,X), false(X), (true(Y) : bodyP(R,Y))\} \tag{24}$$
$$\cup\ \{noAS \leftarrow true(X), (niter(X,I) : int(I))\} \tag{25}$$
$$\cup\ \{noAS \leftarrow iter(X,I_1), iter(X,I_2), I_1 \neq I_2\} \tag{26}$$
$$\cup\ \{iter_<(X,I) \leftarrow false(X), int(I); iter_<(X,I_2) \leftarrow true(X), iter(X,I_1), int(I_2), I_2 > I_1\} \tag{27}$$
$$\cup\ \{notApp(R) \leftarrow head(R,X_1), iter(X_1,I), I > 0, (iter_<(X_2,I) : bodyP(R,X_2))\} \tag{28}$$
$$M_{sat} = \{true(X) \leftarrow atom(X), noAS;\ false(X) \leftarrow atom(X), noAS\} \tag{29}$$
$$\cup\ \{iter(X,I) \leftarrow atom(X), int(I), noAS;\ niter(X,I) \leftarrow atom(X), int(I), noAS\} \tag{30}$$
$$\cup\ \{inReduct(R) \leftarrow rule(R), noAS;\ outReduct(R) \leftarrow rule(R), noAS\} \tag{31}$$

Before we come to an explanation of M, we discuss the specification of the program-dependent part M^P_{gr}, which is expected to encode the rules of P as facts which are added to M. To this end, we first set the domain of integers to $|A(P)|$ (which is a sufficiently high value as explained below). Then, each rule of P is represented by atoms $head(r,a)$, $bodyP(r,a)$, and $bodyN(r,a)$, where r is a rule from P (used as new atom representing the respective rule). Here, $head(r,a)$, $bodyP(r,a)$ and $bodyN(r,a)$ denote that a is an atom that occurs in the head, positive and negative body of rule r, respectively. Rules 14 and 15 then extract for the sets of all rules and atoms in P. Formally:

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

$$M^P_{gr} = \{int(c) \mid 0 \leq c < |A(P)|\} \cup \{head(r,h) \mid r \in P, h \in H(r)\}$$
$$\cup\ \{bodyP(r,b) \mid r \in P, h \in B^+(r)\} \cup \{bodyN(r,b) \mid r \in P, h \in B^-(r)\}$$

The structure of the static programs M_{guess}, M_{check} and M_{sat} follows then the basic architecture of saturation encodings presented in Section 3. The idea is as follows. Program M_{guess} guesses an answer set candidate I of program P, M_{check} simulates the computation of the reduct P^I and checks if its least model coincides with I, and M_{sat} saturates the model whenever this is *not* the case. 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 (17) and (18) 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, $(\mathit{false}(X) : \mathit{bodyN}(R,X))$ is a *conditional literal* which evaluates to true iff $\mathit{false}(X)$ holds for all X such that $\mathit{bodyN}(R,X)$ is true, i.e., all atoms in the negative body are false. Rules (19) simulate the computation of the least model $\mathit{lfp}(T_{P^I})$ of P^I using fixpoint iteration. To this end, each atom $a \in I$ is assigned a guessed integer to represent an ordering of derivations during fixpoint iteration under T_P. We need at most $|A(P)|$ iterations because the least model of P contains only atoms from $A(P)$ and the fixpoint iteration stops if no new atoms are derivable. However, since not all instances need the maximum of $|A(P)|$ iterations, there can be gaps in this sequence. For instances which need fewer than $|A(P)|$ iterations. Rules (20)-(26) check if the current interpretation is *not* an answer set of P which can be justified by the guessed derivation sequence, and derive *noAS* in this case. Importantly, *noAS* both if (i) I is not an answer set, and if (ii) I *is* an answer set, but one that cannot be reproduced using the guessed derivation sequence. Rules (27) and (28) ensure that true atoms are derived in the earliest possible iteration, which eliminates redundant solutions.

As a preparation for both checks (i) and (ii), rules (20)-(22) 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 in the reduct (rules (20)), if at least one positive body atom is false (rules (21)), or if it has a positive body atom which is derived in the same or a later iteration (rules (22)) because then the rule cannot fire (yet) in the iteration the head atom was guessed to be derived.

We can then perform the actual checks (i) and (ii). (i) For checking if I is an answer set, rules (23) check if all atoms in I are derived by some rule in P^I (i.e., $I \subseteq \mathit{lfp}(T_{P^I})$). Conversely, rules (24) check if all rules derived by some rule in P^I are also in I (i.e., $I \supseteq \mathit{lfp}(T_{P^I})$). Overall, the rules (23)-(24) check if $I = \mathit{lfp}(T_{P^I})$. This check compares I and $\mathit{lfp}(T_{P^I})$ only under the assumption that the guessed derivation sequence is valid.

(ii) This validity remains to be checked. To this end, rules (25) ensure that an iteration number is specified for all atoms which are true in I; in order to avoid default-negation we explicitly check if all atoms $\mathit{niter}(a,i)$ for $0 \leq i \leq |A(P)| - 1$ are true. Rules (26) guarantee that this number is unique for each atom. If one of these conditions does not apply, then the *currently* guessed derivation order does not justify that I is accepted as an answer set, hence it is dismissed by deriving *noAS*, even if the same interpretation might be a valid answer set justified by another (valid) derivation sequence. This is by intend because all real answer sets I are justified by some valid derivation sequence.

One can show that atom *noAS* correctly represents inconsistency of P.

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 the answer sets of M^P contains noAS.

4.2 A Meta-Program for Non-Ground Programs

We extend the encoding of a ground normal logic programs 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. 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 with an arity > 0 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 $\mathscr{P} \cap \mathscr{F} \neq \emptyset$). Then, atoms, interpreted as function terms, can occur as parameters of other atoms.

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

$$\begin{aligned} M^P_{ng} = {} & \{int(c) \mid 0 \leq c < |A(P)|\} \cup \{head(r(\mathbf{V}_r),h) \leftarrow \{head(R,d) \mid d \in B^+(r)\} \mid r \in P, h \in H(r)\}\} \\ & \cup \{bodyP(r(\mathbf{V}_r),b) \leftarrow \{head(R,d) \mid d \in B^+(r)\} \mid r \in P, b \in B^+(r)\} \\ & \cup \{bodyN(r(\mathbf{V}_r),b) \leftarrow \{head(R,d) \mid d \in B^+(r)\} \mid r \in P, b \in B^-(r)\} \end{aligned}$$

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 as unique function symbol and all variables in r as parameters. As for the ground case, the head, the positive and the negative body are extracted from r. However, since variables may occur in any atom of r, we have to add a body to the rules of the representation to ensure safety. To this end, we add a *domain atom* $head(R,d)$ 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. Informally, we create an instance of r for all variable substitutions such that all body atoms of the instance are potentially derivable in the meta-program.

Example 5. Let $P = \{f\colon d(a);\ r_1\colon q(X) \leftarrow d(X), \mathrm{not}\, p(X);\ r_2\colon p(X) \leftarrow d(X), \mathrm{not}\, q(X)\}$. We have:

$$\begin{aligned} M^P_{ng} = {} & \{head(f,d(a)) \leftarrow;\ head(r_1(X),q(X)) \leftarrow head(R,d(X))\} \\ & \cup \{bodyP(r_1(X),d(X)) \leftarrow head(R,d(X));\ bodyN(r_1(X),p(X)) \leftarrow head(R,d(X))\} \\ & \cup \{head(r_2(X),p(X)) \leftarrow head(R,d(X));\ bodyP(r_2(X),d(X)) \leftarrow head(R,d(X))\} \\ & \cup \{bodyN(r_2(X),q(X)) \leftarrow head(R,d(X))\} \end{aligned}$$

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, $d(X)$ in the positive body and $p(X)$ in the negative body. This is denoted by $head(r_1(X),q(X))$, $bodyP(r_1(X),d(X)) \leftarrow head(R,d(X))$ and $bodyN(r_1(X),d(X)) \leftarrow head(R,p(X))$, respectively. The domain of X is defined by all atoms $d(X)$ which are potentially derivable, i.e., by atoms $head(R,d(X))$. □

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 the answer sets of M^P contains noAS.

5 Query Answering over Subprograms

In this section we first introduce a technique for query answering over subprograms based on the inconsistency check from the previous section. We then introduce a language extension with dedicated *query atoms* which allow for easy expression of such queries within a program. Finally we demonstrate this language extension with an example.

5.1 Encoding Query Answering

In the following, a query q is a set of ground literals (atoms or default-negated atoms) interpreted as conjunction; for simplicity we restrict the further discussion to ground queries. For an atom or default-negated atom l, let $\bar{l}$ be its negation, i.e., $\bar{l} = a$ if $l = \text{not}\, a$ and $\bar{l} = \text{not}\, a$ if $l = a$. We say that an interpretation I satisfies a query q, denoted $I \models q$, if $a \in I$ for all atoms $a \in q$ and $\text{not}\, a \notin I$ for all default-negated atoms $\text{not}\, a \in q$. A logic 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.

We can reduce query answering to (in)consistency checking as follows:

Proposition 3. *For a normal logic 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.*

We now can exploit our encoding for (in)consistency checking for query answering.

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

Based on the previous proposition, we introduce a new language feature which allows for expressing queries more conveniently.

Definition 6. *A* query atom *is of form $S \vdash_t q$, where $t \in \{b, c\}$ determines the type of the query, S is a normal logic (sub)program, and q is a query over S.*

We allow query atoms to occur in bodies of ASP programs in place of ordinary atoms (in implementations, S may be specified by its filename). The intuition is that for a program P containing a query atom, we have that $S \vdash_b q$ resp. $S \vdash_c q$ is true (wrt. all interpretations I of P) if $S \models_b q$ resp. $S \models_c q$.

Formally we define the semantics of such a program P using the following translation to an ordinary ASP program, based on Proposition 4. We let the answer sets of a program

P with query atoms be given by the answer sets of the program $[P]$ defined as follows:

$$[P] = P|_{S\vdash_t q \to noAS_{S\vdash_t q}} \cup \bigcup_{S\vdash_b q \text{ in } P} \left(M \cup M_{ng}^{S\cup\{\leftarrow \bar{l} \mid l\in q\}}\right)\Big|_{a\to a_{S\vdash_b q}} \cup \bigcup_{S\vdash_c q \text{ in } P} \left(M \cup M_{ng}^{S\cup\{\leftarrow q\}}\right)\Big|_{a\to a_{S\vdash_c q}}$$

Here, we let $P|_{S\vdash_t q \to noAS_{S\vdash_t q}}$ denote program P after replacing *every* query atom of kind $S \vdash_t q$ (for some t, S and q) by the new ordinary atom $noAS_{S\vdash_t q}$. Moreover, in the unions, expression $|_{a\to a_{S\vdash_b q}}$ denotes that each atom a is replaced by $a_{S\vdash_b q}$ (likewise for $S \vdash_c q$). This ensures that for every query atom $S \vdash_b q$ resp. $S \vdash_c q$ in P, a separate copy of M and $M_{ng}^{S\cup\{\leftarrow \bar{l} \mid l\in q\}}$ resp. $M_{ng}^{S\cup\{\leftarrow q\}}\big)\big|_{a\to a_{S\vdash_c q}}$ is generated whose vocabularies are disjoint. In particular, each such copy uses a separate atom $noAS_{S\vdash_b q}$ resp. $noAS_{S\vdash_c q}$ which represents by Proposition 4 the answer to query q. Thus, after replacing each $S \vdash_t q$ in the original program P by the respective atom $noAS_{S\vdash_c q}$, the program behaves as desired. One can show that $[P]$ resembles the aforementioned intuition:

Proposition 5. *For a logic program P with query atoms we have that the answer sets of P and $[P]$, projected to the atoms in P, coincide.*

Note that, while the definition of the above construction of $[P]$ may not be trivial, this does not harm usability from user's perspective. This is because the above rewriting needs to be implemented only once, while the user can simply use query atoms.

Example 6. Consider the check for Hamiltonian cycles in Example 3. As observed, the presented attempt does not work due to default-negation. For $P = \{noHamiltonian \leftarrow P_{guess} \cup P_{check} \vdash_c sat\}$ (and thus for $[P]$) we have that there is an answer set which contains *noHamiltonian* if and only if the graph at hand does not contain a Hamiltonian cycle. On the other hand, if there are Hamiltonian cycles, then the program has at least one answer set but none of the answer sets contains atom *noHamiltonian*.

Note that the subprogram S, over which query answering is performed, can access atoms from program P. Thus, it is possible to perform computations both before and after query answering. For instance, in the previous example the graph which is checked for Hamiltonian cycles may be the result of preceding computations in P.

5.2 Checking Conditions with Default-Negation

Query answering over subprograms can 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 such cases. 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' [10]). 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 $P_{guess} \cup P_{check}$. To this end, one constructs the program $[\{allOk \leftarrow P_{guess} \cup P_{check} \vdash_c ok\}]$. This program is always consistent, has a unique answer set containing *allOk* whenever the property holds for all guesses in the search space, and has other answer sets none of which contains *allOk* otherwise.

6 Discussion and Related Work

Related to our approach are *nested* HEX*-programs*, which allow for accessing answer sets of subprograms using dedicated *external atoms* [4]. However, HEX is beyond plain ASP and requires a more sophisticated solver. Similar extensions of ordinary ASP exist [15], 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 [8]. 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. The idea of representing a subprogram by atoms in the meta-program is similar to approaches for ASP debugging (cf. [12, 14]). But the actual computation (as realized by program M) 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. Also the *stable-unstable semantics* supports an explicit interface to (possibly even nested) oracles [1]. However, there are no query atoms but the relation between the guessing and checking programs is realized via an extension of the semantics.

Our encoding is related to a technique towards automated integration of guess and check programs [5], but based on a different characterization of answer sets. Also, their approach can only handle ground programs. Moreover and most importantly, they focus on integrating programs, but does 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.

7 Conclusion and Outlook

Saturation is an advanced modeling technique in ASP, 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. In this paper, based on an encoding for consistency checking for normal programs, we realized query answering over subprograms.

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, at least in some cases, to answer multiple queries simultaneously. Another possible starting point for future work is the application of our encoding for more efficient evaluation 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 [11].

References

1. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Stable-unstable semantics: Beyond NP with normal logic programs. TPLP 16(5-6), 570–586 (2016), http://dx.doi.org/10.1017/S1471068416000387
2. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell. 15(3-4), 289–323 (1995)
3. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer. In: 5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009. LNCS, vol. 5689, pp. 40–110. Springer (2009)
4. Eiter, T., Krennwallner, T., Redl, C.: HEX-programs with nested program calls. In: Tompits, H. (ed.) Proceedings of the 19th Intl Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011). LNAI, vol. 7773, pp. 1–10. Springer (2013)
5. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. TPLP 6(1-2), 23–60 (2006)
6. Eiter, T., Redl, C., Schüller, P.: Problem solving using the HEX family. In: Computational Models of Rationality. pp. 150–174. College Publications (2016)
7. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
8. Faber, W., Woltran, S.: Manifold answer-set programs and their applications. In: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 6565, pp. 44–63. Springer (2011)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on AI and Machine Learning, Morgan and Claypool Publishers (2012)
10. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. CoRR abs/1107.5742 (2011)
11. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.J., Hooker, J.N. (eds.) 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.
12. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: AAAI. pp. 448–453. AAAI Press (2008)
13. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9(3–4), 365–386 (1991)
14. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: On debugging non-ground answer-set programs. TPLP 10(4-6), 513–529 (2010)
15. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: Vos, M.D., Provetti, A. (eds.) Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd Intl. ASP'05 Workshop, Bath, UK, September 27-29, 2005. CEUR Workshop Proceedings, vol. 142. (2005)