Modular Nonmonotonic Logic Programming Revisited*

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{dao,eiter,fink,tkren}@kr.tuwien.ac.at

Abstract. Recently, enabling modularity aspects in Answer Set Programming (ASP) has gained increasing interest to ease the composition of program parts to an overall program. In this paper, we focus on modular nonmonotonic logic programs (MLP) under the answer set semantics, whose modules may have contextually dependent input provided by other modules. Moreover, (mutually) recursive module calls are allowed. We define a model-theoretic semantics for this extended setting, show that many desired properties of ordinary logic programming generalize to our modular ASP, and determine the computational complexity of the new formalism. We investigate the relationship of modular programs to disjunctive logic programs with well-defined input/output interface (DLP-functions) and show that they can be embedded into MLPs.

Key words: Knowledge Representation, Answer Set Programming, Modular Logic Programming

1 Introduction

In the recent years, there has been an increasing interest in studying modularity aspects of Answer Set Programming (ASP), in order to ease the composition of program parts to an overall program. Since the conception of Splitting Sets [1], which generalize stratification and proved to be a useful tool to decompose programs, a number of approaches to enhance ASP and LP in general with modularity have been made [2–8].

However, compared to the area of logic programming (LP) in general (see [4] for a historic account), the work on modular ASP is still less developed. As in general LP, there are two directions, namely Programming-in-the-large and Programming-in-the-small. In the former, compositional operators are provided for combining separate and independent modules based on standard semantics. This direction has been followed, e.g., with answer set programs with Gaifman-Shapiro-style module architecture [2, 3]. Programming-in-the-small aims at enhancing ASP with abstraction and scoping mechanisms similar as in other programming paradigms. This direction has been more widely considered, and modular extensions of ASP based on generalized quantifiers [4], macros [5], and templates [6] have been proposed.

The two directions are quite divergent, as Programming-in-the-large requires to introduce new operators in the language. Modular ASP Programs [4] were an early attempt to narrow the gap between them a bit, using general quantifiers as a device

* This work has been supported by the Austrian Science Fund (FWF) project P20841.

to access from one module P_1 another module P_2 using *module atoms* of the form $P_2[\boldsymbol{p}].q(X)$ (in slightly different syntax), where $\boldsymbol{p}$ is a list of predicates and q is a predicate; intuitively, the module atom evaluates to true for X if, on input of the values of the predicates in $\boldsymbol{p}$ to the module P_2, the atom $q(X)$ will be concluded by P_2 (under skeptical semantics). For a system $P_1[\boldsymbol{q_1}], \ldots, P_n[\boldsymbol{q_n}]$ of such modules, where $\boldsymbol{q_i}$ is a (list of) formal input predicates, answer sets have been defined using a generalization of the Gelfond-Lifschitz reduct. As it has been shown, the resulting framework is quite expressive, as it is EXPSPACE-complete in general.

However, the proposal in [4] has limitations and, due to the use of the Gelfond-Lifschitz reduct, suffers from similar anomalies as answer sets for other extensions of logic programs defined in this way. As for the former, an important restriction that was made in [4] is that calls of modules must be acyclic; that is, following the call chain, one may not return to the same call of a module. In fact, this condition was already imposed at the syntactic level, and does not allow the use of recursion in modules, which is a common and natural technique. Also other approaches exclude (mutually) recursive calls (e.g., disjunctive logic programs with a well-defined input/output interface (DLP-functions) exclude positive such calls [2]; see also Section 6).

Example 1. Consider the following recursive module $P[q/1]$, which determines whether a set has an even number of elements:

$$q'(X) \vee q'(Y) \leftarrow q(X), q(Y), X \neq Y. \qquad skip(X) \leftarrow q(X), \text{not } q'(X).$$
$$odd \leftarrow skip(X), P[q'].even. \qquad even \leftarrow \text{not } odd.$$

Here, $q/1$ is a (formal) unary input predicate that stores the set. The first two rules in the top line effect, by stability of answer sets, that q becomes q' with one element randomly removed (for which $skip$ is true). In the last line, the left rule determines recursively whether q stores an odd number of elements, while the right rule defines $even$ as the complement of odd. Intuitively, if we call P with a predicate p for input, then $even$ is computed true, which is expressed by $P[p].even$, if p stores an even number of elements. Note that P is recursive, and for empty input p it calls itself with the same input (one can easily rewrite this to mutual recursion between two modules for odd and even).

While a main motivation for the proposal in this paper is to allow for recursive calls of program modules with input, another objective is to provide a global semantics for a collection of modules. Comparatively, [4] was more concerned with defining local models of a single module, by importing conclusions of other modules rather than giving a model based semantics to a collection $P_1, \ldots, P_n$ of modules.

Concerning semantics, the use of the Gelfond-Lifschitz reduct effected that local models were in the same vein as Nash equilibria, viz., that a model is (locally) stable if assuming that all modules behave in the same way there is no need for the local program to switch to another model. Specifically, a program P_0 consisting of the clause $q \leftarrow P_1.p[q]$, where $P_1[q_1]$ consists of the single clause $p \leftarrow q_1$, has two answer sets, viz., $\emptyset$ and $\{q\}$. The reason is that q can be concluded in a self-stabilizing way from the call $P_1.p[q]$; however, arguably $\emptyset$ may be considered as the single answer set of P_0.

Such behavior can be excluded using alternative reducts, like the Faber-Leone-Pfeifer (FLP) reduct [9], which has been proposed in the context of ASP with aggregates to ensure that answer sets are minimal models. This reduct formed also the basis for defining the semantics of HEX-programs [10], which generalized the semantics of logic programs with generalized quantifiers to the HiLog setting; however, the setting has been module-centric like [4], and no global semantics for a collection of modules is evident. Motivated by these shortcomings, we reconsider modular ASP and make the following main contributions.

• We define a model theoretic semantics of a system $P_1[\boldsymbol{q_1}], \ldots, P_n[\boldsymbol{q_n}]$ of program modules, which are divided into one or multiple main modules P_i that have no input (i.e., $\boldsymbol{q_i}$ is void), and library modules which may have input (i.e., $\boldsymbol{q_i}$ can be non-void). Informally, the semantics assigns an answer set to each main module and module instance that is called by the program under a call-by-value mechanism as in [4]; the answer set must be reproducible from the rules along its recursive computation.

Example 2 (cont'd). In Example 1 above, an answer set for the module instance of $P[q]$, whose input q stores $S = \{c_1, \ldots, c_n\}$, would have q' storing $S_1 = S \setminus \{c_{\pi(1)}\}$ and call the instance of $P[q]$ with q storing S_1, whose answer set in turn stores $S_2 = S_1 \setminus \{c_{\pi(2)}\} = S \setminus \{c_{\pi(1)}, c_{\pi(2)}\}$ in q', etc., where π is any permutation of $\{1, \ldots, n\}$. The value of even and odd in the answer sets of the instances is determined bottom up from the ground: for the instance of $P[q]$ where $q = \emptyset$, q' and skip are void, and thus odd must be necessarily false; hence, even is true. On the way back, even and odd are complemented with their values at the next recursion level.

While a naive definition of the semantics is straightforward, a more difficult question is to delineate the *relevant* instances of modules for the computation. Intuitively, many (instances of) modules $P_i[\boldsymbol{q_i}]$ in a library might be completely irrelevant for determining the semantics of a particular collection of modules, but prevent the existence of a global semantics if locally, for some input value of $\boldsymbol{q_i}$, the instance has no answer set.

Example 3 (cont'd). Suppose in the module P in Example 1 there would also be a fact $r(a)$ and a rule $\mathit{ok} \leftarrow P'[r].\mathit{nonempty}$ where the module $P'[q/1]$ consists of the rules $\mathit{nonempty} \leftarrow \text{not } \mathit{nonempty}$ and $\mathit{nonempty} \leftarrow q(X)$. Then, an instance P' has an answer set precisely if its input is nonempty. Thus, the call $P'[r].\mathit{nonempty}$ in the rule will always lead to an answer set in which $\mathit{nonempty}$ is true, and hence we expect an answer set for the instance of P with input S. However, as P' has for empty input no answer set, there is no global answer set; intuitively, the instance of P' with empty input is irrelevant, and should not be considered.

To remedy this situation and keep the semantics simple, we use here minimal models as an approximation of answer sets in module instances that are outside of a *context* for which stability of models is strictly required; this context contains always the modules instances along the call graph of the program; the smaller the context, the more permissive is the semantics.

• We analyze semantic properties of the approach, and show that many of the desired properties of ordinary logic programs generalize to our modular ASP. This includes that the answer sets of a positive modular ASP are its minimal models; that Horn programs have a model intersection property, and thus a least model, which can be computed by least fixpoint iteration; that the latter can be extended to stratified programs, which have a canonical model modulo the relevant part.

• We characterize the computational complexity of the new formalism. Our modular ASP programs have the same complexity as ordinary ASP programs if the modules have no input, i.e., deciding answer set existence is Σ_2^p-complete in the propositional case and $\mathrm{NEXP}^{\mathrm{NP}}$-complete in the non-ground (Datalog) case. For programs with arbitrary inputs, the complexity is exponentially higher, viz. $\mathrm{NEXP}^{\mathrm{NP}}$-complete and $2\mathrm{NEXP}^{\mathrm{NP}}$-complete, respectively; note that $\mathrm{EXPSPACE}$ is believed to be strictly contained in $2\mathrm{NEXP}^{\mathrm{NP}}$. The picture is analogous for deciding membership of an atom in the least model of a Horn program, which is P-complete resp. EXP-complete without inputs and EXP-complete resp. $2\mathrm{EXP}$-complete with arbitrary inputs. However, if the inputs are naturally bounded, then the complexity is the same as in the case without inputs, and thus as in ordinary ASP.
• We analyze the relationship between our modular ASP programs and DLP-functions, which are one of the premier formalisms for combining ASP modules. As it turns out, DLP-functions can be very naturally embedded into our formalism, and vice versa a fragment of our modular ASP programs can be embedded into DLP functions. Since our approach admits mutual recursion of calls and also input to modules in terms of call by value, it can be viewed as a generalization of DLP-functions.

We believe that the approach presented in this paper contributes to modular ASP in which modules can be used in an unrestricted and natural way for problem solving, and looping recursion is handled by the very means of logic programming semantics.

2 Modular Nonmonotonic Logic Programs

In this section, we present our framework of modular ASP programs, and define first syntax and then semantics of such programs. We assume that the reader is familiar with basic notions of logic programming and the answer set semantics of nonmonotonic logic programs [11]. The syntax is based on disjunctive logic programs; our modular logic programs (MLPs) consist of modules as a way to structure logic programs. Moreover, such modules allow for input provided by other modules; it is safe to say that one module may call other modules and additionally provide input.

We pose no essential restriction on the rules, and modules may mutually call each other in a recursive way, and, on top of that, provide mutual input. The semantics we provide for MLPs caters for this situation and is thus not straight-forward. By the very notion of module input, it is apparent that modules must be instantiated before they can be "used." To this end, we delineate contexts of models that carry instantiations of modules and serve to define answer sets for modular programs. As noted in [4], answer sets of modular programs based on a Gelfond-Lifschitz-style reduct may be weaker than those of ordinary logic programs, we thus use the FLP-reduct in order to gain the desired property of minimality in answer sets.

Syntax of Modular Nonmonotonic Logic Programs. We consider programs in a function-free first-order (Datalog) setting (this restriction is not essential from a conceptual point of view, but convenient for the purposes of this work).

Let $\mathcal{V}$ be a vocabulary $\mathcal{C}$, $\mathcal{P}$, $\mathcal{X}$, and $\mathcal{M}$ of mutually disjoint sets whose elements are called *constants*, *predicate*, *variable*, and *module names*, respectively, where each $p \in \mathcal{P}$ has a fixed associated arity $n \geq 0$, and each module name in $\mathcal{M}$ has a fixed associated list $\boldsymbol{q} = q_1, \ldots, q_k$ ($k \geq 0$) of predicated names $q_i \in \mathcal{P}$ (the formal input parameters). Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C} \cup \mathcal{P}$) are denoted with first letter in upper case (resp., lower case).

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. Ordinary atoms (simply atoms) are of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms; $n \geq 0$ is the *arity* of the atom. A *module atom* is of the form

$$P[p_1, \ldots, p_k].o(t_1, \ldots, t_l) \ , \tag{1}$$

where $p_1, \ldots, p_k$ is a list of predicate names $p_i \in \mathcal{P}$, called *module input list*, such that p_i has the arity of the formal input parameter q_i, $o \in \mathcal{P}$ is a predicate name with arity l such that for the list of terms $t_1, \ldots, t_l$, $o(t_1, \ldots, t_l)$ is an ordinary atom, and $P \in \mathcal{M}$ is a module name.

Intuitively, a module atom provides a way for deciding the truth value of a ground atom $o(\boldsymbol{c})$ in a program P depending on the extension of a set of input predicates.

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \mathrm{not}\, \beta_{m+1}, \ldots, \mathrm{not}\, \beta_n \ , \tag{2}$$

where $k \geq 1$, $m, n \geq 0$, $\alpha_1, \ldots, \alpha_k$ are atoms, and $\beta_1, \ldots, \beta_n$ are either atoms or module atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\beta_{m+1}, \ldots, \beta_n\}$. If $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a (disjunctive) *fact*; r is *ordinary*, if it contains only ordinary atoms.

We now formally define the syntax of modules.

Definition 1 (module). *A* module *is a pair $m = (P[\boldsymbol{q}], R)$, where $P \in \mathcal{M}$ with associated formal input $\boldsymbol{q}$, and R is a finite set of rules. It is* ordinary*, if all rules in R are ordinary, and* ground*, if all rules in R are ground. A module m is either a* main module *or a* library module*; if it is a main module, then $|\boldsymbol{q}| = 0$.*

Recall that the formal input $\boldsymbol{q}$ is given by a list of predicate names $p_i \in \mathcal{P}$. We refer with $R(m)$ to the rule set of m. When clear from the context, we omit empty [] and () from (main) modules and module atoms. E.g., the module $P[q]$ in Example 1 is a library module; further examples are given below.

Based on modules, we define modular logic programs as follows.

Definition 2 (modular logic program). *A* modular logic program *(MLP) $\mathbf{P}$ is an n-tuple of modules*

$$(m_1, \ldots, m_n) \ , n \geq 1, \tag{3}$$

consisting of at least one main module, where $\mathcal{M} = \{P_1, \ldots, P_n\}$. We say that $\mathbf{P}$ is ground*, if each module is ground.*

Example 4 (cont'd). Suppose that we have besides a module $m_2 = (P[q], R_2)$, where R_2 is taken from the rules in Example 1, a further module $m_1 = (Q[], R_1)$, in which

$$R_1 = \left\{ \begin{array}{ll} s(a).\ \ s(b).\ \ s(c).\ \ s(d). & s_1(X) \vee s_2(X) \leftarrow s(X). \\ ok \leftarrow P[s_1].even, P[s_2].even. & ok \leftarrow \mathrm{not}\ ok. \end{array} \right\} \ .$$

Informally, the disjunctive rule splits the predicate s into two predicates s_1 and s_2; the subsequent rules check that they both store sets of even cardinality. Formally, $\mathbf{P} = (m_1, m_2)$ forms the respective MLP; here, m_1 is the (single) main module.

Example 5. Take an MLP $\mathbf{P} = (m_1, m_2, m_3)$, where both $m_1 = (P_1[], \{a \leftarrow P_2.b.\})$, $m_2 = (P_2[], \{b \leftarrow P_1.a.\})$ are main modules, and $m_3 = (P_3[c], \{c \leftarrow \text{not } c.\})$ is a library module. Intuitively, m_1 and m_2 amount to the logic program $\{a \leftarrow b.\ b \leftarrow a.\}$, while m_3 is a simple constraint with formal input c.

Semantics of Modular Nonmonotonic Logic Programs. We now define the semantics of modular logic programs. It is defined in terms of Herbrand interpretations and grounding as customary in traditional logic programming and ASP.

The *Herbrand base* w.r.t. vocabulary $\mathcal{V}$, $HB_{\mathcal{V}}$, is the set of all possible ground ordinary and module atoms that can be built using $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{M}$; if $\mathcal{V}$ is implicit from an MLP $\mathbf{P}$, it is the *Herbrand base of* $\mathbf{P}$ and denoted by $HB_{\mathbf{P}}$. The grounding of a rule r is the set $gr(r)$ of all ground instances of r w.r.t. $\mathcal{C}$; the grounding of rule set R is $gr(R) = \bigcup_{r \in R} gr(r)$, and the one of a module m, $gr(m)$, is defined by replacing the rules in $R(m)$ by $gr(R(m))$; the grounding of an MLP $\mathbf{P}$ is $gr(\mathbf{P})$, which is formed by grounding each module m_i of $\mathbf{P}$.

The semantics of an arbitrary MLP $\mathbf{P}$ is given in terms of $gr(\mathbf{P})$.

Let $S \subseteq HB_{\mathbf{P}}$ be any set of atoms. For any list of predicate names $\boldsymbol{p} = p_1, \ldots, p_k$ and $\boldsymbol{q} = q_1, \ldots, q_k$, we use the notation $S|_{\boldsymbol{p}} = \{p_i(\boldsymbol{c}) \in S \mid i \in \{1, \ldots, k\}\}$ and $S|_{\boldsymbol{p}}^{\boldsymbol{q}} = \{q_i(\boldsymbol{c}) \mid p_i(\boldsymbol{c}) \in S, i \in \{1, \ldots, k\}\}$.

Next, we define module instantiations. Therefore, we need to index a module with a particular, fixed set of input facts it receives, which is termed a value call.

Definition 3 (value call). *For a* $P \in \mathcal{M}$ *with associated formal input* $\boldsymbol{q}$ *we say that* $P[S]$ *is a* value call with input S*, where* $S \subseteq HB_{\mathbf{P}}|_{\boldsymbol{q}}$*. Let* $VC(\mathbf{P})$ *denote the set of all value calls* $P[S]$ *with input* S *such that* $P \in \mathcal{M}$.[1]

Instantiating an MLP $\mathbf{P}$ is more complex than instantiating $R(m)$ for every module m of $\mathbf{P}$, since all possible inputs for the modules need to be taken into account, yielding different sets of ground rules. Rule bases indexed by value calls account for this.

Definition 4 (rule base). *A* rule base *is an (indexed) tuple* $\mathbf{R} = (R_{P[S]} \mid P[S] \in VC(\mathbf{P}))$ *of sets of ground rules* $R_{P[S]}$.

Definition 5 (instantiation). *For a module* $m_i = (P_i[\boldsymbol{q_i}], R_i)$ *from* $\mathbf{P}$*, its* instantiation with $S \subseteq HB_{\mathbf{P}}|_{\boldsymbol{q_i}}$*, is* $I_{\mathbf{P}}(P_i[S]) = R_i \cup S$*. For an MLP* $\mathbf{P}$*, its* instantiation *is the rule base* $I(\mathbf{P}) = (I_{\mathbf{P}}(P_i[S]) \mid P_i[S] \in VC(\mathbf{P}))$.

Loosely speaking, a module instantiation is given by the rules of the module together with particular, additional input facts. Intuitively, rule bases collect all possible such instantiations with all possible inputs, and can be referenced by $VC(\mathbf{P})$.

We next define (Herbrand) interpretations and models of an MLP.

Definition 6 (interpretation). *An* interpretation $\mathbf{M}$ *of an MLP* $\mathbf{P}$ *is an (indexed) tuple* $(M_i/S \mid P_i[S] \in VC(\mathbf{P}))$*, where all* $M_i/S \subseteq HB_{\mathbf{P}}$ *contain only ordinary atoms.*

An interpretation provides an assignment for every module instance, and thus is likewise indexed, i.e., M_i/S is an interpretation of the module instance referenced by $P_i[S]$.

[1] Note that $VC(\mathbf{P})$ is also used as index set here.

Definition 7 (model). *An interpretation* $\mathbf{M}$ *of an MLP* $\mathbf{P}$ *is a* model *of*

– *a ground atom* $\alpha \in HB_{\mathbf{P}}$ *at* $P_i[S]$, *denoted* $\mathbf{M}, P_i[S] \models \alpha$, *if in case* α *is an ordinary atom,* $\alpha \in M_i/S$, *and if* $\alpha = P_k[\boldsymbol{p}].o(\boldsymbol{c})$ *is a module atom,* $o(\boldsymbol{c}) \in M_k/((M_i/S)|_{\boldsymbol{p}}^{\boldsymbol{q_k}})$;

– *a ground rule* r *at* $P_i[S]$ ($\mathbf{M}, P_i[S] \models r$), *if* $\mathbf{M}, P_i[S] \models H(r)$ *or* $\mathbf{M}, P_i[S] \not\models B(r)$, *where (i)* $\mathbf{M}, P_i[S] \models H(r)$, *if* $\mathbf{M}, P_i[S] \models \alpha$ *for some* $\alpha \in H(r)$, *and (ii)* $\mathbf{M}, P_i[S] \models B(r)$, *if* $\mathbf{M}, P_i[S] \models \alpha$ *for all* $\alpha \in B^+(r)$ *and* $\mathbf{M}, P_i[S] \not\models \alpha$ *for all* $\alpha \in B^-(r)$;

– *a set of ground rules* R *at* $P_i[S]$ ($\mathbf{M}, P_i[S] \models R$) *iff* $\mathbf{M}, P_i[S] \models r$ *for all* $r \in R$;

– *a ground rule base* $\mathbf{R}$ ($\mathbf{M} \models \mathbf{R}$) *iff* $\mathbf{M}, P_i[S] \models R_{P_i[S]}$ *for all* $P_i[S] \in VC(\mathbf{P})$.

Finally, $\mathbf{M}$ *is a* model *of an MLP* $\mathbf{P}$, *denoted* $\mathbf{M} \models \mathbf{P}$, *if* $\mathbf{M} \models I(\mathbf{P})$ *in case* $\mathbf{P}$ *is ground resp.* $\mathbf{M} \models gr(\mathbf{P})$, *if* $\mathbf{P}$ *is nonground. An MLP* $\mathbf{P}$ *is* satisfiable, *if it has a model.*

Intuitively, an interpretation $\mathbf{M}$ satisfies a ground module atom $P_k[\boldsymbol{p}].o(\boldsymbol{c})$ appearing in an instantiation $I_{\mathbf{P}}(P_i[S])$, if the ordinary atom $o(\boldsymbol{c})$ holds for the instantiation of the module m_k with the input which is given by the interpretation of $\boldsymbol{p}$ in M_i/S. On top of this, satisfaction of ordinary atoms, rules, etc., is straightforward.

Example 6. Consider $\mathbf{P}$ from Example 5, then $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset, M_3/\emptyset, M_3/\{c\})$ is a model of $\mathbf{P}$, where $M_1/\emptyset = \{a\}$, $M_2/\emptyset = \{b\}$, and $M_3/\emptyset = M_3/\{c\} = \{c\}$. We have $\mathbf{M}, P_1[\emptyset] \models a$; $\mathbf{M}, P_2[\emptyset] \models b$; $\mathbf{M}, P_1[\emptyset] \models P_2.b$; $\mathbf{M}, P_2[\emptyset] \models P_1.a$; hence $\mathbf{M}, P_1[\emptyset] \models a \leftarrow P_2.b$; $\mathbf{M}, P_2[\emptyset] \models b \leftarrow P_1.a$. Moreover, $\mathbf{M}, P_3[\emptyset] \models c$; $\mathbf{M}, P_3[\emptyset] \models c \leftarrow \text{not } c$ (and similar for $\mathbf{M}$ at $P_3[\{c\}]$); thus $\mathbf{M}, P_1[\emptyset] \models I_{\mathbf{P}}(P_1[\emptyset])$, $\mathbf{M}, P_2[\emptyset] \models I_{\mathbf{P}}(P_2[\emptyset])$, $\mathbf{M}, P_3[\emptyset] \models I_{\mathbf{P}}(P_3[\emptyset])$, and $\mathbf{M}, P_3[\{c\}] \models I_{\mathbf{P}}(P_3[\{c\}])$; therefore $\mathbf{M} \models I(\mathbf{P})$, where $I(\mathbf{P}) = (I_{\mathbf{P}}(P_1[\emptyset]), I_{\mathbf{P}}(P_2[\emptyset]), I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$. Finally, $\mathbf{M} \models \mathbf{P}$.

We next proceed to define answer sets of an MLP $\mathbf{P}$. To this end, we need to compare models and single out minimal models. Furthermore, in order to focus on relevant modules, we introduce the formal notion of a call graph.

Definition 8 (minimal models). *For any interpretations* $\mathbf{M}$ *and* $\mathbf{M}'$ *of* $\mathbf{P}$, *we define that* $\mathbf{M} \leq \mathbf{M}'$, *if for every* $P_i[S] \in VC(\mathbf{P})$ *it holds that* $M_i/S \subseteq M_i'/S$, *and* $\mathbf{M} < \mathbf{M}'$, *if both* $\mathbf{M} \neq \mathbf{M}'$ *and* $\mathbf{M} \leq \mathbf{M}'$. *A model* $\mathbf{M}$ *of* $\mathbf{P}$ *(resp., a rule base* $\mathbf{R}$*) is* minimal, *if* $\mathbf{P}$ *(resp.,* $\mathbf{R}$*) has no model* $\mathbf{M}'$ *such that* $\mathbf{M}' < \mathbf{M}$. *The set of all minimal models of* $\mathbf{P}$ *(resp.,* $\mathbf{R}$*) is denoted by* $MM(\mathbf{P})$ *(resp.,* $MM(\mathbf{R})$*).*

Definition 9 (call graph). *The* call graph of an MLP $\mathbf{P}$ *is a labeled digraph* $CG_{\mathbf{P}} = (V, E, l)$ *with vertex set* $V = VC(\mathbf{P})$ *and an edge* e *from* $P_i[S]$ *to* $P_k[T]$ *in* E *iff* $P_k[\boldsymbol{p}].o(\boldsymbol{t})$ *occurs in* $R(m_i)$; *furthermore,* e *is labeled with an input list* $\boldsymbol{p}$, *denoted* $l(e)$. *Given an interpretation* $\mathbf{M}$, *the* relevant call graph $CG_{\mathbf{P}}(\mathbf{M}) = (V', E')$ of $\mathbf{P}$ w.r.t. $\mathbf{M}$ *is the subgraph of* $CG_{\mathbf{P}}$ *where* E' *contains all edges from* $P_i[S]$ *to* $P_k[T]$ *of* $CG_{\mathbf{P}}$ *such that* $(M_i/S)|_{l(e)}^{\boldsymbol{q_k}} = T$, *and* V' *contains all* $P_i[S]$ *that are main module instantiations or induced by* E'; *any such* $P_i[S]$ *is called* relevant w.r.t. $\mathbf{M}$.

Example 7. Consider $\mathbf{P}$ and $I(\mathbf{P})$ from Example 6. The call graph of $\mathbf{P}$ is $CG_{\mathbf{P}} = (VC(\mathbf{P}), E, l)$, where $E = \{(P_1[\emptyset], P_2[\emptyset]), (P_2[\emptyset], P_1[\emptyset])\}$, and l maps each edge to the void input list. Both $P_1[\emptyset]$ and $P_2[\emptyset]$ are relevant, since they are main modules, while $P_3[\emptyset]$ and $P_3[\{c\}]$ are irrelevant (never called). Thus, we obtain that $CG_{\mathbf{P}}(\mathbf{M}) = (\{P_1[\emptyset], P_2[\emptyset]\}, E, l)$, for any interpretation $\mathbf{M}$ of $\mathbf{P}$.

We refer to the vertex and edge set of a graph G by $V(G)$ and $E(G)$, resp. For defining answer sets, we use a reduct of the instantiated program as customary in ASP. A suggestive way is to apply a traditional reduct to each module instance of $\mathbf{P}$; however, this is not fully satisfactory, as in practice $\mathbf{P}$ might contain module instantiations which have no answer sets for certain inputs, which compromises the existence of an answer set of $\mathbf{P}$. For this reason, we contextualize the notions of reduct and answer sets.

Definition 10 (context). *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$. *A* context for $\mathbf{M}$ *is any set* $C \subseteq VC(\mathbf{P})$ *such that* $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C$.

Then, the reduct of an instantiated program is built w.r.t. a given context.

Definition 11 (context-based reduct). *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$ *and* C *be a context for* $\mathbf{M}$. *The* reduct of $\mathbf{P}$ at $P[S]$ w.r.t. $\mathbf{M}$ and C, *denoted* $f\mathbf{P}(P[S])^{\mathbf{M},C}$, *is the rule set* $I_{gr(\mathbf{P})}(P[S])$ *from which, if* $P[S] \in C$, *all rules* r *such that* $\mathbf{M}, P[S] \not\models B(r)$ *are removed. The* reduct of $\mathbf{P}$ w.r.t. $\mathbf{M}$ and C *is the rule base* $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P[S])^{\mathbf{M},C} \mid P[S] \in VC(\mathbf{P}))$.

That is, outside C the module instantiations of $\mathbf{P}$ resp. $gr(\mathbf{P})$ remain untouched, while inside C the FLP-reduct [9] is applied.

Example 8. Consider $\mathbf{P}$ and $\mathbf{M}$ from Example 6, and a context $C = \{P_1[\emptyset], P_2[\emptyset]\}$. The context-based reduct of $\mathbf{P}$ w.r.t. $\mathbf{M}$ and C is given by the rule base $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P_1[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_2[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_3[\emptyset])^{\mathbf{M},C}, f\mathbf{P}(P_3[\{c\}])^{\mathbf{M},C})$, which is equal to $I(gr(\mathbf{P}))$, i.e., $f\mathbf{P}(P_1[\emptyset])^{\mathbf{M},C} = \{a \leftarrow P_2.b.\}$, $f\mathbf{P}(P_2[\emptyset])^{\mathbf{M},C} = \{b \leftarrow P_1.a.\}$, $f\mathbf{P}(P_3[\emptyset])^{\mathbf{M},C} = \{c \leftarrow \text{not } c.\}$, and $f\mathbf{P}(P_3[\{c\}])^{\mathbf{M},C} = \{c; c \leftarrow \text{not } c.\}$.

Definition 12 (answer set). *Let* $\mathbf{M}$ *be an interpretation of a ground MLP* $\mathbf{P}$. *Then* $\mathbf{M}$ *is an* answer set of $\mathbf{P}$ w.r.t. a context C for $\mathbf{M}$, *if* $\mathbf{M}$ *is a minimal model of* $f\mathbf{P}^{\mathbf{M},C}$.

Note that C is a parameter that allows to select a degree of overall-stability for answer sets of $\mathbf{P}$. The extreme case $C = VC(\mathbf{P})$ requires that all module instances have answer sets. On the other end, the minimal context $C = V(CG_{\mathbf{P}}(\mathbf{M}))$ is the relevant call graph of $\mathbf{P}$; we consider this as the default context and omit C from notation.

Example 9. Consider $\mathbf{P}$ from Example 4. We have that $\mathbf{P}$ has answer sets of four different shapes, each of them having exactly two instances of s_1 and two instances of s_2 for the model $M_Q/\emptyset$ of instantiation $I_{\mathbf{P}}(Q[\emptyset])$. A particular answer set is the indexed tuple with the entries $(M_Q/\emptyset, M_P/\emptyset, M_P/\{q(a)\}, M_P/\{q(b)\}, M_P/\{q(c)\}, M_P/\{q(d)\}, M_P/\{q(a), q(c)\}, M_P/\{q(b), q(d)\}, \dots)$, where $M_Q/\emptyset = \{s_1(a), s_2(b), s_1(c), s_2(d), ok, s(a), s(b), s(c), s(d)\}$, $M_P/\emptyset = \{even\}$, all models for instantiations with singletons $M_P/\{q(a)\}$, $M_P/\{q(b)\}$, $M_P/\{q(c)\}$, $M_P/\{q(d)\}$ contain odd and the resp. *skip*'d element, and both $M_P/\{q(a), q(c)\}$ and $M_P/\{q(b), q(d)\}$ contain $even$.

Example 10. Consider $\mathbf{P}$ and $\mathbf{M}$ from Example 6. Let $\mathbf{M}_0 = (M_1^0/\emptyset, M_2^0/\emptyset, M_3^0/\emptyset, M_3^0/\{c\})$, such that $M_1^0/\emptyset = M_2^0/\emptyset = \emptyset$, $M_3^0/\emptyset = M_3^0/\{c\} = \{c\}$, be another interpretation for $\mathbf{P}$. One can verify that $\mathbf{M}_0$ is also a model of $\mathbf{P}$. Since we fixed the context C to $\{P_1[\emptyset], P_2[\emptyset]\}$, the reduct w.r.t. $\mathbf{M}_0$ is $f\mathbf{P}^{\mathbf{M}_0,C} = (f\mathbf{P}(P_1[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_2[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\{c\}])^{\mathbf{M}_0,C}) = (\emptyset, \emptyset, I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$, and $f\mathbf{P}^{\mathbf{M},C}$ is as in Example 8. The minimal model of $f\mathbf{P}^{\mathbf{M}_0,C}$ is $\mathbf{M}_0$, hence it is an answer set of $\mathbf{P}$ w.r.t. C, whereas the minimal model of $f\mathbf{P}^{\mathbf{M},C}$ is also $\mathbf{M}_0$, i.e., $\mathbf{M}$ is not an answer set of $\mathbf{P}$ w.r.t. C.

3 Semantic Properties

We now consider some properties of modular logic programs. Obviously, they conservatively generalize ordinary logic programs.

Proposition 1. *Let R be an ordinary logic program. Then M is an answer set of R iff $\mathbf{M} = (M_1/\emptyset)$ with $M_1/\emptyset = M$ is an answer set of the MLP (m_1), where $m_1 = (P_1[], R)$ is a main module and P_1 is a module name.*

Some well-known properties from standard answer set programming carry over to the semantics of modular logic programs. This is of avail not only to encompass underlying intuitions, but also for characterizing computational aspects. Two straightforward consequences from the definition of FLP-reduct are the following.

Lemma 1. *If $\mathbf{M} \models f\mathbf{P}^{\mathbf{M},C}$ for some context C for $\mathbf{M}$, then $\mathbf{M} \models \mathbf{P}$.*

Lemma 2. *If $\mathbf{M} \models \mathbf{P}$, then $\mathbf{M} \models f\mathbf{P}^{\mathbf{M}',C}$ for any interpretation $\mathbf{M}'$ and context C.*

Consequently, we obtain that answer sets are minimal models of $\mathbf{P}$.

Proposition 2. *If $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. context C, then $\mathbf{M} \in MM(\mathbf{P})$.*

Furthermore, the semantics is a proper refinement of a naive semantics that would require stability w.r.t. all possible module instantiations disregarding their relevance. This is a simple consequence of the following property.

Proposition 3. *If $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. context $C \subseteq VC(\mathbf{P})$, then $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. every context $C' \subseteq C$ for $\mathbf{M}$, i.e., $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C' \subseteq C$.*

We next consider answer sets that, in a sense, face no inconsistency in the scope of instantiations that are relevant to them. Let $ord(\mathbf{P})$ denote the result of deleting from an MLP $\mathbf{P}$ all rules containing module atoms in $R(m)$ in all modules m of $\mathbf{P}$. We call an answer set $\mathbf{M}$ of $\mathbf{P}$ w.r.t. *C fully stable*, if $V(CG_{\mathbf{P}}(\mathbf{M}')) \subseteq C$ for all $\mathbf{M}' \leq \mathbf{M}$ such that $\mathbf{M}' \models ord(\mathbf{P})^{\mathbf{M},C}$. Then the following holds.

Proposition 4. *Every answer set of $\mathbf{P}$ w.r.t. $C = VC(\mathbf{P})$ is fully stable, and if $\mathbf{M}$ is an answer set of $\mathbf{P}$ w.r.t. C and fully stable w.r.t. $C' \subseteq C$, then $\mathbf{M}$ is fully stable w.r.t. C.*

Obviously, answer sets coincide with the naive semantics if $V(CG_{\mathbf{P}}(\mathbf{M})) = VC(\mathbf{P})$ for all interpretations $\mathbf{M}$ of $\mathbf{P}$, in particular, when all modules are main. Moreover, also for positive MLPs the semantics coincides with the naive semantics. Just like in ordinary logic programs, it behaves like the minimal model semantics in absence of negation.

Proposition 5. *Let $\mathbf{P}$ be positive. Then, the answer sets of $\mathbf{P}$ coincide with $MM(\mathbf{P})$.*

By monotonicity of all module instances, one can easily show that the models of a Horn MLP $\mathbf{P}$ are closed under a suitable notion of intersection. Given interpretations $\mathbf{M}$ and $\mathbf{N}$ of $\mathbf{P} = (m_1, \ldots, m_n)$, let their intersection be the interpretation denoted $\mathbf{M} \cap \mathbf{N}$ such that $(M \cap N)_i/S = \bigcap_{S' \supseteq S}(M_i/S' \cap N_i/S')$, for every $S \subseteq HB_{\mathbf{P}}|_{\boldsymbol{q}_i}$ and $i = 1, \ldots, n$. Then:

Proposition 6. *Suppose $\mathbf{M} \models \mathbf{P}$ and $\mathbf{N} \models \mathbf{P}$, where $\mathbf{P}$ is Horn. Then $\mathbf{M} \cap \mathbf{N} \models \mathbf{P}$.*

As a consequence, a Horn MLP has a canonical answer set.

Corollary 1. *If $\mathbf{P}$ is Horn, then it has a unique answer set, which coincides with its least model.*

Like for ordinary programs, we can compute the answer set of a Horn MLP by means of a bottom up fixed-point computation.

Definition 13 **($T_{\mathbf{P}}$-operator).** *Given a Horn MLP $\mathbf{P}$ and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we define the operator $T_{\mathbf{P}}(\mathbf{M})$ point-wise as follows:*

$$T_{\mathbf{P}}(M_i/S) = \{H(r) \mid r \in I_{\mathbf{P}}(P_i[S]),\ \mathbf{M}, P_i[S] \models B(r)\}.$$

Since the operator is continuous, it has a least fixed-point $\mathit{lfp}(\mathbf{P})$ that results, starting from the empty interpretation $\mathbf{M}_\emptyset$, i.e., where $M_i/S = \emptyset$ for every $P_i[S] \in VC(\mathbf{P})$ in ω steps, i.e., $\mathit{lfp}(\mathbf{P}) = T_{\mathbf{P}}{\uparrow}_\omega(\mathbf{M}_\emptyset)$. We obtain the following result.

Proposition 7. *For a Horn MLP $\mathbf{P}$, $\mathit{lfp}(\mathbf{P})$ is the unique answer set of $\mathbf{P}$.*

For normal MLPs, we generalize the notion of stratification as follows. Intuitively, the usual notion of the dependency graph of a program is extended by nodes $\mathcal{E}$ for the module atoms appearing in $\mathbf{P}$, which serve to take care of the dependencies between input to the module and module output. Furthermore, we assume that each predicate occurs in ordinary atoms of at most one module.

Let $\mathbf{P} = (m_1, \ldots, m_n)$ be an MLP. The *dependency graph of* $\mathbf{P}$ is the following digraph $G_{\mathbf{P}} = (V, E)$. The vertex set V contains all $p \in \mathcal{P} \cup \mathcal{E}$, with p appearing somewhere in $\mathbf{P}$, and $\mathcal{E}$ is the set of module atoms in $\mathbf{P}$. The edge set E is as follows:

Let $r \in R(m_i)$. There is a $\star$-edge $p \rightarrow^\star q$ in $G_{\mathbf{P}}$, $\star \in \{+, -\}$, if either (i) $p(\boldsymbol{t_1}) \in H(r)$ and $q(\boldsymbol{t_2}) \in B^\star(r)$; (ii) $p(\boldsymbol{t_1}), q(\boldsymbol{t_2}) \in H(r)$ and $\star = +$; or (iii) $p(\boldsymbol{t_1}) \in H(r)$ and q is a module atom in $B^\star(r)$. Moreover, for $\alpha = P_j[\boldsymbol{p}].o(\boldsymbol{t}) \in B(r)$, the set E contains all edges $a \rightarrow^+ b$, where (iv) $a = \alpha$ and b appears in $\boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$; (v) $a = \alpha$ and $b = o$; or (vi) $a = q_\ell$ and $b = p_\ell$, where q_ℓ is in $\boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$ and p_ℓ is in $\boldsymbol{p}$.

Definition 14. *We say that an MLP $\mathbf{P}$ is* stratified *if no cycle in $G_{\mathbf{P}}$ has $-$-edges.*

As for ordinary logic programs, given a stratified MLP $\mathbf{P}$, there exists a labelling function l from $HB_{\mathbf{P}}$ to the nonnegative integers, such that $l(\alpha) \geq l(\beta)$ if $a \rightarrow^+ b$ in $G_{\mathbf{P}}$, and $l(\alpha) > l(\beta)$ if $a \rightarrow^- b$ in $G_{\mathbf{P}}$, where $\alpha = a(\boldsymbol{t})$, or $a \in \mathcal{E}$ and a unifies with α, respectively for β and b.

Let k be the maximal value assigned by a particular labelling function, and let $\mathit{Strat}_i = \{a \in HB_{\mathbf{P}} \mid l(a) = i\}$ for $0 \leq i \leq k$, then $\mathit{Strat}_0, \ldots, \mathit{Strat}_k$ is a stratification, i.e., a partitioning of $HB_{\mathbf{P}}$.

Towards an iterated fixed-point computation of answer sets for stratified MLPs, we define the following operator.

Definition 15 **($T^L_{\mathbf{P}}$-operator).** *Given a normal MLP $\mathbf{P}$, a subset L of $HB_{\mathbf{P}}$, and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we define the operator $T^L_{\mathbf{P}}(\mathbf{M})$ point-wise as follows:*

$$T^L_{\mathbf{P}}(M_i/S) = M_i/S \cup \{H(r) \mid r \in I_{\mathbf{P}}(P_i[S]),\ \mathbf{M}, P_i[S] \models B(r),\ B(r) \subseteq L\}.$$

By $T^L_{\mathbf{P}}{\uparrow}_\omega(\mathbf{M})$, we denote the application of $T^L_{\mathbf{P}}$ in ω steps, starting with $\mathbf{M}$. Furthermore, let $\mathbf{M}^0 = \mathbf{M}_\emptyset$ be the empty interpretation, i.e., where $M_i/S = \emptyset$ for every value call $P_i[S] \in VC(\mathbf{P})$, and let $L_i = \bigcup_{0 \leq j \leq i} \mathit{Strat}_j$.We inductively define $\mathbf{M}^{i+1} = T^{L_{i+1}}_{\mathbf{P}}{\uparrow}_\omega(\mathbf{M}^i)$, for $0 \leq i < k$.

Proposition 8. *Let $\mathbf{P}$ be normal and stratified. Then $\mathbf{M}^k$ is an answer set of $\mathbf{P}$, for any stratification $Strat_0, \ldots, Strat_k$ of $HB_{\mathbf{P}}$.*

A further consequence of stratification is that the relevant call graph is unique.

Proposition 9. *Let $\mathbf{P}$ be normal and stratified. Then $V(CG_{\mathbf{P}}(\mathbf{M})) = V(CG_{\mathbf{P}}(\mathbf{M}^k))$, for any answer set $\mathbf{M}$ of $\mathbf{P}$ and any stratification $Strat_0, \ldots, Strat_k$ of $HB_{\mathbf{P}}$.*

Therefore, answer sets of stratified, normal MLPs coincide on relevant instances. The answer set obviously is unique if all value calls of $VC(\mathbf{P})$ are relevant, or if all irrelevant instances have a unique minimal model.

4 Computational Complexity

To begin with, let us restrict our attention to Horn MLPs. Considering the propositional case, if the modules $m_i = (P_i[\boldsymbol{q}_i], R_i)$ in $\mathbf{P}$ have no input (i.e., $\boldsymbol{q}_i$ is void), then $I(\mathbf{P})$ has polynomial size and $lfp(\mathbf{P})$ is computable in polynomial time. For arbitrary propositional $\mathbf{P}$ with no inputs, we can guess and verify an answer set $\mathbf{M}$ of $\mathbf{P}$ in polynomial time with an NP oracle. As MLPs (Proposition 1) subsume ordinary logic programs, we thus obtain by known results (cf. [12]) the same complexity. With slight abuse of notation, for a ground atom α and an interpretation $\mathbf{M}$ of $\mathbf{P}$, we write $\alpha \in \mathbf{M}$ if $\alpha \in M_i/S$ for a given $P_i[S] \in VC(\mathbf{P})$.

Theorem 1. *Given a propositional MLP $\mathbf{P} = ((P_1[], R_1), \ldots, (P_n[], R_n))$, (i) if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = lfp(\mathbf{P})$ of $\mathbf{P}$ is computable in polynomial time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is P-complete; (ii) to decide whether $\mathbf{P}$ has an answer set is Σ_2^p-complete.*

These results generalize to the case where the module inputs in $\mathbf{P}$ have bounded length, i.e., $|\boldsymbol{q}_i| \leq k$ for some constant k, as $I(\mathbf{P})$ and $\mathbf{M}$ have polynomial size. For unrestricted inputs, however, $I(\mathbf{P})$ and $\mathbf{M}$ are exponential and we get a blowup.

Theorem 2. *Given a propositional MLP $\mathbf{P}$ (i) if $\mathbf{P}$ is Horn, the unique answer set $\mathbf{M} = lfp(\mathbf{P})$ of $\mathbf{P}$ is computable in exponential time and to decide whether $\alpha \in \mathbf{M}$ for a ground atom α is EXP-complete; (ii) to decide whether $\mathbf{P}$ has an answer set is $\mathrm{NEXP}^{\mathrm{NP}}$-complete.*

The hardness parts can be shown e.g. by encodings of Turing machines, which adapt constructions in [12]. Superficially, one uses modules $P[\boldsymbol{c}, \boldsymbol{t}]$, where $\boldsymbol{c}$ amounts to a tape cell index and $\boldsymbol{t}$ to a time stamp during a computation; with $|\boldsymbol{c}| = |\boldsymbol{t}| = n$, 2^n cells and 2^n time stamps can be modeled. Further atoms store the cell contents, state of the machine, and the position of the read-write head. The transition function is encoded by rules with access to the contents of neighboring cells, which is realized by respective (recursive) module calls; neighboring cells and time stamps are computed using local rules.

In the Datalog setting, we get for MLPs a similar picture as for ordinary logic programs, where the complexity of Datalog programs is exponentially higher than the one of propositional programs. Intuitively, the process of grounding may introduce exponentially many ground atoms for an atom, which in turn may result in double

Table 1. Complexity of MLPs ($\mathbf{P}$ is Horn in the first two columns, α is a ground atom)

MLP $\mathbf{P}$	Computing $\mathit{lfp}(\mathbf{P})$	Deciding $\alpha \in \mathit{lfp}(\mathbf{P})$	Answer set existence
prop. $\mathbf{P}$, empty inputs	polynomial time	P-complete	Σ_2^p-complete
prop. $\mathbf{P}$	exponential time	EXP-complete	$\mathrm{NEXP}^{\mathrm{NP}}$-complete
non-ground $\mathbf{P}$	double exponential time	2EXP-complete	$2\mathrm{NEXP}^{\mathrm{NP}}$-complete

exponentially many module instances; thus, $I(\mathbf{P})$ and interpretations $\mathbf{M}$ have double exponential size in general. Computing $\mathit{lfp}(\mathbf{P})$ for Horn MLPs $\mathbf{P}$ may thus take double exponential time, and a guess for an answer set has double exponential size. We get the following results.

Theorem 3. *Given a non-ground MLP* $\mathbf{P}$*, (i) if* $\mathbf{P}$ *is Horn, the unique answer set* $\mathbf{M} = \mathit{lfp}(\mathbf{P})$ *of* $\mathbf{P}$ *is computable in double exponential time and to decide whether* $\alpha \in \mathbf{M}$ *for a ground atom* α *is* $2\mathrm{EXP}$*-complete; (ii) to decide whether* $\mathbf{P}$ *has an answer set is* $2\mathrm{NEXP}^{\mathrm{NP}}$*-complete.*

The hardness parts an be shown by lifting the constructions for the propositional case. Here, n-ary predicates $p(X_1, \ldots, X_n)$ are used to store 2^n bits of a number, such that a range of 2^{2^n} tape cells and time stamps can be spanned via module inputs $\boldsymbol{q}$.

Finally, we note that the complexity drops by an exponential to the one of ordinary logic programs, if the arities of input predicates are bounded by a constant (as then $I(\mathbf{P})$ and $\mathbf{M}$ have single exponential size). Our results are compactly summarized in Table 1.

5 Relationship to DLP-Functions

DLP-functions [2] are a proposal for modular logic programs under answer set semantics in conformance with Programming-in-the-large. The approach creates a semantics for a sequence of modules by defining a suitable input-output interface, and allows combining compatible answer sets between joinable modules.

More specifically, a DLP-function has form $\Pi = \langle R, I, O, H\rangle$, where R is a set of propositional disjunctive rules and I, O, H are sets of propositional atoms defining input, output, and hidden atoms, respectively. An operator $\oplus$ forms a new DLP-function from two DLP-functions that respect hidden atoms of each other. In addition, if two such DLP-functions Π_1 and Π_2 are not mutually (positive) dependent, their join $\Pi_1 \sqcup \Pi_2$ is defined. Joinability allows negative loops between DLP-functions but not positive ones; one can use $\oplus$ to generate the join. On top of joinable DLP-functions, the Module Theorem is the basis for computing the answer sets of a sequence of DLP-functions by taking the union of mutually compatible answer sets of each member; hence joinable DLP-functions qualify for having a compositional semantics.

We now show a translation from DLP-functions to MLP modules, and briefly outline a translation from a fragment of MLPs without input to an equivalent sequence of DLP-functions. For space reasons, we must omit recalling the formal machinery of DLP-functions here, but stick to definitions of [2] as much as possible. To be in line with [2], we consider only the propositional case.

Translation from DLP-Functions to MLPs. We now define a translation ∇ which maps sequences of DLP-functions to MLPs. To this end, we map input atoms a appearing in bodies of rules in some DLP-function to module atoms of MLPs, whenever there is an output of another DLP-function which contains a. Other atoms remain unchanged. Then, we add further guessing rules to the modules; intuitively, they guess the truth value for input atoms which have not been fixed by some output.

Let $\Pi = (\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions, where Π_i is a DLP-function $\langle R_i, I_i, O_i, H_i\rangle$, and the join $\bigsqcup_{i=1}^{n} \Pi_i$ is defined. For a propositional atom a in $\mathrm{At}(R_i)$, if $a \in I_i$ and there exists another DLP-function Π_j in Π such that $a \in \mathrm{At}_\mathrm{o}(\Pi_j)$, then $\nabla(a) = P_j.a$ (note that such a Π_j is unique due to the condition $\mathrm{At}_\mathrm{o}(\Pi_k) \cap \mathrm{At}_\mathrm{o}(\Pi_\ell) = \emptyset$ for every $k \neq \ell$); otherwise $\nabla(a) = a$.

Let r be a propositional rule in Π_i of the form (2). We create $\nabla(r)$ by replacing each β_i by $\nabla(\beta_i)$;[2] for Π_i, let $\nabla(\Pi_i) = (P_i, \nabla(R_i))$ where P_i is a module name and $\nabla(R_i) = \{\nabla(r) \mid r \in R_i\} \cup Q_i$, where $Q_i = \{a \vee \bar{a} \mid a \in \mathrm{At}_\mathrm{i}(\Pi_i) \setminus \bigcup_{j \neq i} \mathrm{At}_\mathrm{o}(\Pi_j)\}$ and all $\bar{a}$ are fresh propositional atoms. Finally $\nabla(\Pi) = (\nabla(\Pi_1), \ldots, \nabla(\Pi_n))$, where each $\nabla(\Pi_i)$ is a main module.

Example 11. Let $\Pi = (\Pi_1, \Pi_2)$ be a sequence of DLP-functions consisting of $\Pi_1 = \langle\{a \leftarrow \mathrm{not}\ b\}, \{b\}, \{a\}, \emptyset\rangle$ and $\Pi_2 = \langle\{b \leftarrow \mathrm{not}\ a\}, \{a\}, \{b\}, \emptyset\rangle$. The translation of Π to MLP is $\nabla(\Pi) = (\nabla(\Pi_1), \nabla(\Pi_2))$, where $\nabla(\Pi_1)$ and $\nabla(\Pi_2)$ are the main modules whose associative sets of rules are $\{a \leftarrow \mathrm{not}\ P_2.b\}$ and $\{b \leftarrow \mathrm{not}\ P_1.a\}$, resp. Here, both Π and $\nabla(\Pi)$ possess two answer sets: Π has $\{a\}$ and $\{b\}$, while $\nabla(\Pi)$ has $(\{a\}, \emptyset)$ and $(\emptyset, \{b\})$.

Now, let Π_1 be from above and $\Pi = (\Pi_1)$. In this case, $\nabla(\Pi) = (\nabla(\Pi_1))$, where $\nabla(\Pi_1) = (P_1, \nabla(R_1))$ and $\nabla(R_1) = \{a \leftarrow \mathrm{not}\ b;\ b \vee \bar{b}\}$. Both Π and $\nabla(\Pi)$ have two answer sets; Π has $\{a\}$ and $\{b\}$, while $\nabla(\Pi)$ has $(\{a, \bar{b}\})$ and $(\{b\})$.

The following proposition shows that ∇ is correct.

Proposition 10. *Let $\Pi = (\Pi_1, \ldots, \Pi_n)$ be a sequence of DLP-functions whose join $\bigsqcup_{i=1}^{n} \Pi_i$ is defined. Then, the answer sets of $\nabla(\Pi)$ correspond 1-1 to those of Π.*

Translation from MLPs to DLP-Functions. Compared to DLP-functions, MLPs have a fine-grained input mechanism. DLP-functions import atoms from other DLP-functions by means of an explicit input/output interface; an atom, whose truth value originates from a different DLP-function, can be seen as a call-by-reference. To clarify, take an MLP with library modules $m_k = (P[q], R_k)$ and $m_\ell = (Q[p], R_\ell)$. Consider a module atom $Q[b].a$ appearing in R_k; we are confronted with two different types of input:

(1) m_ℓ retrieves input b from m_k explicitly in form of an additional fact p whenever b holds in some instantiation of $P[q]$, which can be seen as call-by-value, and
(2) m_k retrieves input from m_ℓ implicitly in form of a, which plays a similar role to call-by-reference input in DLP-functions.

Here, we restrict our attention to MLPs with input of type (2). By complexity arguments, translating MLPs with inputs of type (1) into sequences of DLP-functions is likely to cause an exponential blowup in general.

[2] Constraints are allowed in [2]; they can be emulated by adding *fail* (not *fail*) to the head (body) of $\nabla(r)$, where *fail* is a fresh propositional atom.

Given a propositional MLP $\mathbf{P} = (m_1, \dots, m_n)$, where each $m_i = (P_i[], R_i)$ has no formal input parameter, we can do the translation by mapping in R_i each ordinary atom a to $\Delta(a) = a_{P_i}$, each module atom $P_j.b$ to $\Delta(P_j.b) = b_{P_j}$. For each module m_i, the input (output) atoms of the corresponding DLP-function are determined by applying Δ to module atoms occurring in R_i (resp., module atoms $P_i.a$ occurring in $\mathbf{P}$). Based on this idea, $\mathbf{P}$ can be translated into a sequence $\Delta(\mathbf{P})$ of DLP-functions where the composition operator $\oplus$ is defined. However, to have the join operator $\sqcup$ defined and thus answer sets of $\Delta(\mathbf{P})$, the modules in $\mathbf{P}$ must respect a condition akin to "*not mutually dependent*" [2], which is based on the sharing of strongly connected components in the positive dependency graph. On top of this condition, our translation gives a variant of the Module Theorem in [2]. Technical details and proofs are given in an accompanying technical report.

6 Related Work and Conclusion

In the ASP context, several modular logic programming formalisms have been proposed We already discussed the modular logic programs of [4] and DLP-functions [2].

Towards code reusability in ASP, [5] defines modules in terms of macros. On top of this, the authors define ensembles, which group modules comparable to the way classes keep their methods together in object-oriented programming languages, and an inheritance mechanism for ensembles. In a similar way but more focused on aggregates, [6] defines "template" predicates to quickly introduce new predefined constructs and to deal with compound data structures. The DLP^T language based on this notion was implemented on top of DLV. Both [5] and [6] have the restriction that no cycle is allowed between macros/templates.

A different approach is used in [13]. Here, the modules allow to import answer sets from other modules to compute the overall solution. However, this approach considers only modular ASP programs with acyclic dependency graph. Another system called RSig [14] allows to specify modules and provides an information hiding mechanism. Direct communication between modules was not addressed; instead, modules exchange information with a global state via import/export declarations. The semantics of such a system is given by a (polynomial) compilation into an ordinary ASP program.

Another formalism with multiple nonmonotonic logic programs is [15], targeting a Semantic Web environment. It allows to interlink logic programs that may refer to remote knowledge bases distributed on the Web. The authors propose a context-aware form of negation as failure to deal with the inherent incompleteness of data on the Web. The `MWeb` framework [16] is a further attempt to enhance the Semantic Web with scope and context for modular web rule bases. However, it is mainly concerned with support for hidden knowledge and the safe use of strong and weak negation, and modular rule bases are translated into ordinary logic programs, respecting different reasoning modes.

While we have presented the basic approach, several issues remain for further work. An interesting issue is to further analyze contexts and, e.g., to determine conditions for contexts that are fully stable, which desirably should be small. Some (less effective) conditions may be determined by syntactic analysis.

Another issue is extensions of MLP to richer classes of programs, including constructs like strong negation, constraints, external functions, nesting, etc. On the semantical side, we can imagine alternative ways of tolerating violations of stability outside the context. This could be done, e.g., by using partial FLP-reducts (where not all rules with false bodies are dropped, leading to a superset of the answer sets), or by genuine approximations. Variants of stratification and splitting sets would also be interesting.

On the computational side, a detailed complexity study of MLPs that considers various fragments is of interest, where in particular the interplay of major classes of ordinary logic programs with dependency information through module calls deserves attention; various notions similar as in [4] might be considered here. Furthermore, efficient methods and algorithms to compute answer sets of MLPs remain to be developed, as well as implementations. To this end, methods based on reductions to ordinary logic programs and extensions are under investigation.

References

1. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP 1994. MIT Press (1994) 23–37
2. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. In: LPNMR 2007. Springer, Heidelberg (2007) 175–187
3. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for Smodels programs. Theory Pract. Log. Program. **8**(5–6) (2008) 717–761
4. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: LPNMR 1997. Springer, Heidelberg (1997) 290–309
5. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In: ICLP 2006. Springer, Heidelberg (2006) 376–390
6. Calimeri, F., Ianni, G.: Template programs for Disjunctive Logic Programming: An operational semantics. AI Commun. **19**(3) (2006) 193–206
7. Bugliesi, M., Lamma, E., Mello, P.: Modularity in Logic Programming. J. Logic Progr. **19/20** (1994) 443–502
8. Brogi, A., Mancarella, P., Pedreschi, D., Turini, F.: Modular logic programming. ACM Trans. Program. Lang. Syst. **16**(4) (1994) 1361–1398
9. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: JELIA 2004. Springer, Heidelberg (2004) 200–212
10. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with external Evaluations for Semantic Web Reasoning. In: ESWC 2006. Springer, Heidelberg (2006) 273–287
11. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Gener. Comput. **9** (1991) 365–385
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Comput. Surv. **33**(3) (2001) 374–425
13. Tari, L., Baral, C., Anwar, S.: A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling. In: ASP'05. CEUR WS (2005) 277–293
14. Balduccini, M.: Modules and Signature Declarations for A-Prolog: Progress Report. In: Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007)
15. Polleres, A., Feier, C., Harth, A.: Rules with Contextually Scoped Negation. In: ESWC 2006. Springer, Heidelberg (2006) 332–347
16. Analyti, A., Antoniou, G., Damásio, C.V.: A principled framework for modular web rule bases and its semantics. In: KR2008, AAAI Press (2008)