Relevance-driven Evaluation of Modular Nonmonotonic Logic Programs*

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. Modular nonmonotonic logic programs (MLPs) under the answer-set semantics have been recently introduced as an ASP formalism in which modules can receive context-dependent input from other modules, while allowing (mutually) recursive module calls. This can be used for more succinct and natural problem representation at the price of an exponential increase of evaluation time. In this paper, we aim at an efficient top-down evaluation of MLPs, considering only calls to relevant module instances. To this end, we generalize the well-known Splitting Theorem to the MLP setting and present notions of call stratification, for which we determine sufficient conditions. Call-stratified MLPs allow to split module instantiations into two parts, one for computing input of module calls, and one for evaluating the calls themselves with subsequent computations. Based on these results, we develop a top-down evaluation procedure that expands only relevant module instantiations. Finally, we discuss syntactic conditions for its exploitation.

1 Introduction

Modularity is an important element of high-level programming languages that has beneficial effects on problem decomposition, which allows one to structure a program into parts that solve subproblems appropriately. Its importance has also been recognized in the area of logic programming (see [1] for a historic account), and in particular in Answer Set Programming (ASP), as witnessed by the early conception of Splitting Sets [2], a generalization of the notion of stratification for program decomposition.

Since then, modularity aspects have been considered in several works, cf. [1, 3–8], that aim at practicable formalisms for modular logic programming. The approaches are classified into Programming-in-the-small, building on abstraction and scoping mechanisms (e.g., generalized quantifiers [1], macros [5], and templates [6] have been developed in ASP), and Programming-in-the-large, where compositional operators serve the combination of separate and independent modules based on standard semantics. A prominent representative of the latter in ASP is DLP-functions [3, 4].

Recently, modular logic programs (MLPs) have been introduced in [9], which can be viewed as a generalization of DLP-functions. They overcome a restriction of a

* This research has been supported by the Austrian Science Fund (FWF) project P20841, the Vienna Science and Technology Fund (WWTF) project ICT08-020, and the EC ICT Integrated Project Ontorule (FP7 231875).

preliminary approach in [1], in which module calls must be acyclic (which prohibits the use of recursion through modules), as well as anomalies of the semantics due to the Gelfond-Lifschitz reduct, which is replaced by the FLP reduct [10]. The latter was also used for the semantics of HEX-programs [11], a generalization of [1] to the HiLog setting. However, both [1] and [11] defined models of a single module resp. program, and no global semantics for a collection of modules resp. programs is evident.

As the semantics of MLPs is based on module instantiations (which takes possible input values into account), a naive evaluation following the definition is—similar as with grounding of ordinary ASP programs—infeasible in practice; in general, a module may have double exponentially many instances. Towards implementation, efficient evaluation strategies are thus essential, which are sensitive to (sub-)program classes that do not require a simple guess-and-check procedure on the instantiation, but allow for a guided model building process. Starting from the main module, instances of modules may be created on demand as needed by module calls, focusing on relevant module instances.

Restrictions on programs, like stratification of normal MLPs in [9], may be helpful in this regard. However, the notion of stratification is very strict. It requires that *all* module instances are stratified. Moreover, the fix-point semantics for stratified programs given there is inherently bottom-up and only applies to normal programs, excluding a large class of programs which exploit recursion in a common and natural way and are evaluable top-down, even if they are not normal or unstratified in the sense of [9].

For illustration, consider the following example with an MLP consisting of three modules, one is main and the other two are libraries. Each consists of a module name, with (optional) formal input parameters, and a set of rules. One can inquiry a library module for the extensions of its predicates, with input fed into the module via the input parameters. This example exploits the mutual recursive calls between two library modules to determine whether a set has even cardinality.

Example 1. Let $\mathbf{P}$ be an MLP consisting of three modules $m_1 = (P_1, R_1)$, $m_2 = (P_2[q_2], R_2)$, and $m_3 = (P_3[q_3], R_3)$, where $R_1 = \{q(a).\ \ q(b).\ \ ok \leftarrow P_2[q].even.\}$,

$$R_2 = \left\{ \begin{array}{c} q_2'(X) \vee q_2'(Y) \leftarrow q_2(X), q_2(Y), \\ \qquad\qquad X \neq Y. \\ skip_2 \leftarrow q_2(X), \text{not } q_2'(X). \\ even \leftarrow \text{not } skip_2. \\ even \leftarrow skip_2, P_3[q_2'].odd. \end{array} \right\},\ R_3 = \left\{ \begin{array}{c} q_3'(X) \vee q_3'(Y) \leftarrow q_3(X), q_3(Y), \\ \qquad\qquad X \neq Y. \\ skip_3 \leftarrow q_3(X), \text{not } q_3'(X). \\ odd \leftarrow skip_3, P_2[q_3'].even. \\ \\ \end{array} \right\}$$

Intuitively, m_1 calls m_2 to check if the number of facts for predicate q is even. The call to m_2 'returns' $even$, if either the input q_2 to m_2 is empty (as then $skip_2$ is false), or the call of m_3 with q_2' resulting from q_2 by randomly removing one element (then $skip_2$ is true) returns odd. Module m_3 returns odd for input q_3, if a call to R_2 with q_3' analogously constructed from q_3 returns $even$. In any answer set of $\mathbf{P}$, ok is true.

This program is not normal, and shifting head disjunctions yields a program which is not stratified as per [9]. However, along the mutual recursive chain of calls $P_3[q_2'].odd$, $P_2[q_3'].even$ the inputs q_2' and q_3' gradually decrease until the base, i.e., the empty input, is reached. Taking such decreasing inputs of the relevant module calls into account, we can evaluate MLPs efficiently along the relevant call graph using a finer grained notion of stratification, tolerating also disjunctive or unstratified rules in modules.

Capturing this intuition formally and developing a suitable evaluation algorithm for respective MLPs are the main contributions of this work, which are as follows:

- We develop appropriate notions of *call stratification* and *input stratification* for MLPs, and generalize the well-known Splitting Theorem [2] to this setting. Moreover, we establish a sufficient condition to determine call (and input) stratification at the schematic level, i.e., without the requirement to consider all module instantiations (Section 3).
- By the previous results, module instances calling other modules can be locally split into an input preparation part and a calling part. Based on this, a top-down evaluation procedure is developed, expanding only relevant module instances (Section 4).
- Finally, we discuss syntactic conditions for its exploitation, outline a module rewriting technique for self-recursive modules, and consider related work (Section 5).

The envisaged programming style of call-stratified MLPs is to exploit the natural way of specifying recursive problems with decreasing input. Applications emerge, e.g., in temporal reasoning with an ontology of (partially ordered) time points, reasoning about recurrent properties of sets, or games with 'decreasing input'.

Modular ASP in which modules can be used in an unrestricted and natural way for problem solving, including recursion, is an important requirement for the further development of the ASP paradigm. In this paper, we contribute to this goal, providing efficient evaluation techniques, which are essential for its realization.

2 Preliminaries

Modular logic programs (MLPs) [9] consist of modules as a way to structure nonmonotonic logic programs under answer set semantics [12]. Moreover, such modules allow for input provided by other modules, and may call each other in a (mutually) recursive way.

Syntax. We consider programs in a function-free first-order (Datalog) setting. Let $\mathcal{V}$ be a vocabulary $\mathcal{C}$, $\mathcal{P}$, $\mathcal{X}$, and $\mathcal{M}$ of mutually disjoint sets of *constants*, *predicate*, *variable*, and *module names*, respectively, where each $p \in \mathcal{P}$ has a fixed 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 predicate 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).

Each $t \in \mathcal{C} \cup \mathcal{X}$ is a *term*. An ordinary atom (simply atom) has 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 its *arity*. A *module atom* has the form $P[p_1, \ldots, p_k].o(t_1, \ldots, t_l)$, where $P \in \mathcal{M}$ is a module name with associated $\boldsymbol{q}$, $p_1, \ldots, p_k$ is a list of predicate names $p_i \in \mathcal{P}$, called *module input list*, such that p_i has the arity of q_i in $\boldsymbol{q}$, and $o \in \mathcal{P}$ is a predicate name such that $o(t_1, \ldots, t_l)$ is an ordinary atom. 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

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \mathrm{not}\, \beta_{m+1}, \ldots, \mathrm{not}\, \beta_n \quad (k \geq 1, m, n \geq 0), \qquad (1)$$

where all α_i are atoms and each β_j is an ordinary or a module atom. 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 *fact*; r is *ordinary*, if it contains only ordinary atoms. We denote by $at(r)$ the set $H(r) \cup B(r)$.

We now formally define the syntax of modules and MLPs. A *module* is a pair $m = (P, R)$, where $P \in \mathcal{M}$ with associated input $\boldsymbol{q}$, and R is a finite set of rules. It is either a *main module* (then $|\boldsymbol{q}| = 0$) or a *library module*, and is *ordinary* (resp., *ground*), iff all rules in R are ordinary (ground). We refer with $R(m)$ to the rule set of m, and omit empty [] and () from (main) modules and module atoms if unambiguous.

A *modular logic program (MLP)* is a tuple $\mathbf{P} = (m_1, \ldots, m_n)$, $n \geq 1$, where all m_i are modules and at least one is a main module, where $\mathcal{M} = \{P_1, \ldots, P_n\}$. $\mathbf{P}$ is *ground*, iff each module is ground.

Example 2. For instance, the MLP in Example 1 consists of three modules $\mathbf{P} = (m_1, m_2, m_3)$ where m_2 and m_3 are library modules, and m_1 is a (ground) main module.

Semantics. The semantics of MLPs 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 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 $S|^{\boldsymbol{q}}_{\boldsymbol{p}} = \{q_i(\boldsymbol{c}) \mid p_i(\boldsymbol{c}) \in S, i \in \{1, \ldots, k\}\}$.

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 some S (note that $VC(\mathbf{P})$ is also used as index set). A *rule base* is an (indexed) tuple $\mathbf{R} = (R_{P[S]} \mid P[S] \in VC(\mathbf{P}))$ of sets of rules $R_{P[S]}$. 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 $I(\mathbf{P}) = (I_{\mathbf{P}}(P_i[S]) \mid P_i[S] \in VC(\mathbf{P}))$.

We next define (Herbrand) interpretations and models of MLPs.

Definition 1 (model). *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* $\mathbf{M}$ *of an MLP* $\mathbf{P}$ *is a* model *of*

- *a ground atom* $\alpha \in HB_{\mathbf{P}}$ *at* $P_i[S]$*, denoted* $\mathbf{M}, P_i[S] \models \alpha$*, iff (i)* $\alpha \in M_i/S$ *when* α *is ordinary, and (ii)* $o(\boldsymbol{c}) \in M_k/((M_i/S)|^{\boldsymbol{q_k}}_{\boldsymbol{p}})$*, when* $\alpha = P_k[\boldsymbol{p}].o(\boldsymbol{c})$ *is a module atom;*
- *a ground rule* r *at* $P_i[S]$ *(*$\mathbf{M}, P_i[S] \models r$*), iff* $\mathbf{M}, P_i[S] \models H(r)$ *or* $\mathbf{M}, P_i[S] \not\models B(r)$*, where (i)* $\mathbf{M}, P_i[S] \models H(r)$*, iff* $\mathbf{M}, P_i[S] \models \alpha$ *for some* $\alpha \in H(r)$*, and (ii)* $\mathbf{M}, P_i[S] \models B(r)$*, iff* $\mathbf{M}, P_i[S] \models \alpha$ *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* $\mathbf{M} \models \mathbf{P}$*, iff* $\mathbf{M} \models I(\mathbf{P})$ *in case* $\mathbf{P}$ *is ground resp.* $\mathbf{M} \models gr(\mathbf{P})$*, if* $\mathbf{P}$ *is nonground. An MLP* $\mathbf{P}$ *is* satisfiable*, iff it has a model.*

For any interpretations $\mathbf{M}$ and $\mathbf{M}'$ of $\mathbf{P}$, we define $\mathbf{M} \leq \mathbf{M}'$, iff $M_i/S \subseteq M'_i/S$ for every $P_i[S] \in VC(\mathbf{P})$, and $\mathbf{M} < \mathbf{M}'$, iff $\mathbf{M} \neq \mathbf{M}'$ 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 $\mathbf{M}' < \mathbf{M}$.

We next proceed to define answer sets for MLPs. In order to focus on relevant modules, we introduce the formal notion of a call graph. Intuitively, a call graph represents the relationship between module instantiations and potential module calls: nodes correspond to instantiations and an edge indicates that there is a presumptive call from one module instantiation to another. Labels on the edges distinguish different syntactical calls. Given an interpretation, one can determine the actual calls as edges with labels such that the respective predicates match in the interpretation of the corresponding module instantiations. Edges satisfying this condition, their incident nodes, and the nodes representing main modules constitute the relevant call graph.

Definition 2 (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* $\mathbf{p}$*, 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* $(M_i/S)|^{\boldsymbol{q_k}}_{l(e)} = T$*, 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 3. Let in Example 1 $S^i_{\emptyset} = \emptyset$, $S^i_a = \{q_i(a)\}$, $S^i_b = \{q_i(b)\}$, and $S^i_{ab} = \{q_i(a),\, q_i(b)\}$. Then $VC(\mathbf{P}) = \{P_1[\emptyset], P_2[S^2_v], P_3[S^3_w]\}$, where $v, w \in \{\emptyset, a, b, ab\}$, and $CG_{\mathbf{P}}$ has edges $P_1[\emptyset] \xrightarrow{q} P_2[S^2_v]$, $P_2[S^2_v] \xrightarrow{q'_2} P_3[S^3_w]$, and $P_3[S^3_w] \xrightarrow{q'_3} P_2[S^2_v]$. For the interpretation $\mathbf{M}$ such that $M_1/\emptyset = \{q(a), q(b), ok\}$, $M_2/S^2_{ab} = \{q_2(a),\, q_2(b), q'_2(a), skip_2, even\}$, $M_2/\emptyset = \{even\}$, and $M_3/S^3_a = \{q_3(a), skip_3,\, odd\}$, the nodes of $CG_{\mathbf{P}}(\mathbf{M})$ are $P_1[\emptyset]$, $P_2[S^2_{ab}]$, $P_2[\emptyset]$, and $P_3[S^3_a]$.

For answer sets of an MLP $\mathbf{P}$, we use a reduct of the instantiated program as customary in ASP. As $\mathbf{P}$ might have inconsistent module instantiations, compromising the existence of an answer set of $\mathbf{P}$, we contextualize reducts and answer sets. We denote the vertex and edge set of a graph G by $V(G)$ and $E(G)$, respectively.

Definition 3 (context-based reduct). *A* context *for an interpretation* $\mathbf{M}$ *of an MLP* $\mathbf{P}$ *is any set* $C \subseteq VC(\mathbf{P})$ *such that* $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C$*. 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* $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P[S])^{\mathbf{M},C} \mid P[S] \in VC(\mathbf{P}))$.

That is, outside C the module instantiations of $\mathbf{P}$ resp. $gr(\mathbf{P})$ remain untouched, while inside C the FLP-reduct [10] is applied.

Definition 4 (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 $\mathbf{M}$*, iff* $\mathbf{M}$ *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 minimal context $C = V(CG_{\mathbf{P}}(\mathbf{M}))$ is the relevant call graph of $\mathbf{P}$. From now on we consider this as the default context and omit C from notation.

Example 4. Recall $\mathbf{M}$ from Example 3. For every $P_i[S] \in V(CG_{\mathbf{P}}(\mathbf{M}))$, M_i/S is a $\subseteq$-minimal set that satisfies $f_{\mathbf{P}}(P_i[S])^{\mathbf{M}}$. Thus, any such $\mathbf{M}$ is an answer set of $\mathbf{P}$ iff for every $P_k[T] \in VC(\mathbf{P}) \setminus V(CG_{\mathbf{P}}(\mathbf{M}))$, M_k/T is a $\subseteq$-minimal set satisfying $I_{\mathbf{P}}(P_i[S])$.

3 Splitting for Modular Nonmonotonic Logic Programs

We investigate splitting for MLPs at two different levels: the global (module instantiation) level along the relevant call graph, and the local level ('inside' module instantiations) w.r.t. the (instance) dependency graph. These two notions reveal a class of MLPs, for which an efficient top-down algorithm can be developed for answer set computation.

3.1 Global splitting for call-stratified MLPs

We start by introducing *call stratified* MLPs, whose module instantiations can be split into different layers and evaluated in a stratified way.

Definition 5. *Let $\mathbf{M}$ be an interpretation of an MLP $\mathbf{P}$. We say that $\mathbf{P}$ is* c-stratified (call stratified) w.r.t. $\mathbf{M}$ *iff cycles in $CG_{\mathbf{P}}(\mathbf{M})$ contain only nodes of the form $P_i[\emptyset]$.*

The intuition is to evaluate module instantiations of c-stratified MLPs in a particular order along the call chain, such that potential 'self-stabilizing' effects of cycles have to be taken into account only at the base, i.e., for module instantiations with empty input.

Example 5. Consider the MLP $\mathbf{P}$ and the interpretation $\mathbf{M}$ from Example 3. It is easily verified that $\mathbf{P}$ is c-stratified w.r.t. $\mathbf{M}$. One possible call chain for evaluation is

$$P_1[\emptyset] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q_2'} P_3[\{q_2'(a)\}] \xrightarrow{q_3'} P_2[\emptyset] \ .$$

Definition 6. *Let $\mathbf{M}$ be an interpretation of an MLP $\mathbf{P}$ and R be a rule set. We say that M_i/S is an* answer set of R relative to $\mathbf{M}$*, iff $\mathbf{M}$ is an answer set of the rule base $(R_{P[S]} \mid P[S] \in VC(\mathbf{P}))$, where $R_{P_i[S]} = R$ and $R_{P_j[T]} = M_j/T$ for $i \neq j$ or $S \neq T$.*

In particular, M_i/S is an answer set of $R = I_{\mathbf{P}}(P_i[S])$ relative to $\mathbf{M}$, if it is an answer set of R while other instances are fixed by corresponding elements in $\mathbf{M}$, i.e., all module calls in R are fixed.

Example 6. Consider $\mathbf{P}$ from Example 1 and $\mathbf{M}$ from Example 3, then M_2/S^2_{ab} is an answer set of $I_{\mathbf{P}}(P_2[S^2_{ab}])$ relative to $\mathbf{M}$.

Proposition 1. *Let $\mathbf{M}$ be an interpretation of a c-stratified MLP $\mathbf{P}$. Suppose that along $CG_{\mathbf{P}}(\mathbf{M})$, M_i/S is an answer set of $I_{\mathbf{P}}(P_i[S])$ relative to $\mathbf{M}$ for each $P_i[S] \in V(CG_{\mathbf{P}}(\mathbf{M}))$. If there is an answer set of $\mathbf{P}$ that coincides with $\mathbf{M}$ for every $P_i[\emptyset]$ on a cycle in $CG_{\mathbf{P}}(\mathbf{M})$, then $\mathbf{P}$ has an answer set that coincides with $\mathbf{M}$ on $CG_{\mathbf{P}}(\mathbf{M})$.*

Proposition 1 already indicates a top-down way to evaluate c-stratified MLPs. For a concrete procedure, we need a notion of "local" splitting inside module instances, introduced in the next section.

3.2 Local splitting for input and call stratified MLPs

Towards local splitting, we will first extend the notion of Splitting Sets [2] to MLPs. Then, for pratical purposes, we are interested in splitting a module instance w.r.t. module calls. To this end, we introduce a general and another specific notion of *input splitting sets* in Definition 7. Given a set R of ground rules and a list of predicate names $\boldsymbol{p} = \{p_1, \ldots, p_k\}$, let $def(\boldsymbol{p}, R) = \{p_\ell(\boldsymbol{d}) \mid \exists r \in R, p_\ell(\boldsymbol{d}) \in H(r), p_\ell \in \boldsymbol{p}\}$.

Definition 7 (splitting set). *Let $\mathbf{P}$ be an MLP, R be a set of ground rules and α be a ground module atom of form $P_k[\boldsymbol{p}].o(\boldsymbol{c})$.*

(a) *A* splitting set of R *is a set $U \subseteq HB_\mathbf{P}$ s.t. (i) for any rule $r \in R$, if $H(r) \cap U \neq \emptyset$ then $at(r) \subset U$; and (ii) if $\alpha \in U$ then $def(\boldsymbol{p}, R) \subseteq U$.*

(b) *Let U be a splitting set of R. We say that U is an* input splitting set of R for α, *iff $\alpha \notin U$ and $def(\boldsymbol{p}, R) \subseteq U$.*

As usual, the bottom of a set of ground rules R w.r.t. a set of atoms $A \subseteq HB_\mathbf{P}$ is $b_A(R) = \{r \in R \mid H(r) \cap A \neq \emptyset\}$.

Example 7. Consider $\mathbf{P}$ from Example 1 and $P_2[S^2_{ab}]$ from Example 3. Let R be the instantiation $gr(I_\mathbf{P}(P_2[S^2_{ab}]))$. A possible splitting set for R is $U = \{q_2(a), q_2(b), q'_2(a), q'_2(b), skip_2\}$. Then the bottom $b_U(R)$ is

$$\{q_2(\alpha).\ \ skip_2 \leftarrow q_2(\alpha), \text{not } q'_2(\alpha).\ \ q'_2(\alpha) \vee q'_2(\beta) \leftarrow q_2(\alpha), q_2(\beta) \mid \alpha \neq \beta \in \{a, b\}\}.$$

Based on the extended notion of a Splitting Set, the Splitting Theorem [2] straightforwardly applies to c-stratified MLPs.

Theorem 1. *Let $\mathbf{M}$ be an interpretation of a c-stratified MLP $\mathbf{P}$, R be the instantiation $gr(I_\mathbf{P}(P_i[S]))$ for $P_i[S] \in VC(\mathbf{P})$, and let U be a splitting set for R. Then M_i/S is an answer set of R relative to $\mathbf{M}$ iff it is an answer set of $\{R \setminus b_U(R)\} \cup N$, where N is an answer set of $b_U(R)$ relative to $\mathbf{M}$.*

Example 8. Consider $\mathbf{P}$ from Example 1, $\mathbf{M}$ from Example 3, and R from Example 7. An answer set of R is $N = \{q_2(a), q_2(b), q'_2(a), skip_2\}$. By updating R to $\{R \setminus b_U(R)\} \cup N$, we obtain $R' = \{q'_2(a).\ \ q_2(a).\ \ q_2(b).\ \ skip_2.\ \ even \leftarrow skip_2, P_3[q'_2].odd.\ \ even \leftarrow \text{not } skip_2.\}$. Then M_2/S^2_{ab} is an answer set of R' relative to $\mathbf{M}$.

In the sequel, we single out a subclass of c-stratified MLPs, namely *input and call stratified (ic-stratified)* MLPs, which guarantee that input splitting sets exist for their local splitting. We define the property of *input stratification* at two different levels of the dependency graph: the schematic level and the instance level. Comparing these two options, checking the property at the schematic level is easier, but is often too strong and misses input stratification at the instance level.

In the remainder of the paper, we assume without loss of generality that each predicate occurs in ordinary atoms of at most one module.

Let $\mathbf{P}$ be an MLP. The dependency graph of $\mathbf{P}$ is the 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 $G_\mathbf{P}$, $\star \in \{+, -, \vee\}$, 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 $\star = \vee$, 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

(iv) $\alpha \rightarrow^{in} q_\ell$, for every $q_\ell \in \boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$,
(v) $\alpha \rightarrow^{m} o$, and
(vi) $q_\ell \rightarrow^{b} p_\ell$, where $q_\ell \in \boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$ and $p_\ell \in \boldsymbol{p}$ of α.

This notion of dependency graph refines the one in [9] concerning the labels of arcs (types of dependencies) and allows us to capture input stratification as follows:

Definition 8. *An MLP* $\mathbf{P}$ *is* si-stratified *(input stratified at the schematic level), iff no cycle in* $G_{\mathbf{P}}$ *has in-edges.*

For example, one can easily verify that the MLP in Example 1 is si-stratified.

For any module atoms $\alpha_1, \alpha_2 \in \mathcal{E}$, we say that α_1 locally depends on α_2, if $\alpha_1 \rightsquigarrow \alpha_2$, where $\rightsquigarrow = \rightarrow^+ \cup \rightarrow^- \cup \rightarrow^\vee \cup \rightarrow^{in}$. For each module m_i of a si-stratified MLP $\mathbf{P}$, we define a local labeling function $ll_i \colon V \rightarrow \mathbb{N}$ s.t. $ll_i(\alpha_1) > ll_i(\alpha_2)$ if $\alpha_1 \rightsquigarrow \alpha_2$.

Instance stratification. Proceeding to finer grained level of instances, we define the instance dependency graph $G^{\mathbf{M}}_{\mathbf{P}} = (IV, IE)$ of $\mathbf{P}$ w.r.t. an interpretation $\mathbf{M}$. The idea is to distinguish different predicate names and module atoms in different module instances by associating them with the corresponding value call. Hence, a node in IV is a pair $(p, P_i[S])$ or $(\alpha, P_i[S])$, where p (resp., α) is a predicate name (resp., module atom) appearing in module m_i, and S is the input for a value call $P_i[S] \in VC(\mathbf{P})$.

$G^{\mathbf{M}}_{\mathbf{P}}$ has edges (i′)–(iv′) similar to (i)–(iv) in $G_{\mathbf{P}}$, except that appropriate value calls are added to predicate names/module atoms; the real difference is made by edges (v′) and (vi′); for a module atom of the form $\alpha = P_j[\boldsymbol{p}].o(\boldsymbol{t})$ in $R(m_i)$, $G^{\mathbf{M}}_{\mathbf{P}}$ has edges

(v′) $(\alpha, P_i[S]) \rightarrow^{m} (o, P_i[(M_i/S)|^{\boldsymbol{q_j}}_{\boldsymbol{p}}])$; and
(vi′) $(q_\ell, P_j[(M_i/S)|^{\boldsymbol{q_j}}_{\boldsymbol{p}}]) \rightarrow^{b} (p_\ell, P_i[S])$, where $q_\ell \in \boldsymbol{q_j}$ of $P_j[\boldsymbol{q_j}]$ and $p_\ell \in \boldsymbol{p}$ of α.

Intuitively, these edges capture the relationship between a module atom and the corresponding (ordinary) output atom, respectively between formal input parameters and actual input provided. Restricting to concrete applicable instances ($P_j[(M_i/S)|^{\boldsymbol{q_j}}_{\boldsymbol{p}}]$), they do not just schematically extend (v) and (vi).

Definition 9. *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$*. We say that* $\mathbf{P}$ *is* i-stratified w.r.t. $\mathbf{M}$*, iff cycles with in-edges in* $G^{\mathbf{M}}_{\mathbf{P}}$ *contain only nodes of the form* $(X, P_i[\emptyset])$*. Moreover,* $\mathbf{P}$ *is* ic-stratified w.r.t. $\mathbf{M}$ *iff it is both i-stratified and c-stratified w.r.t.* $\mathbf{M}$*.*

The following theorem shows that ic-stratification is sufficient for the existence of input splitting sets for module atoms in relevant instances.

Theorem 2. *Let* $\mathbf{M}$ *be an interpretation of an ic-stratified MLP* $\mathbf{P}$*,* $P_i[S]$ *be a value call in* $V(CG_{\mathbf{P}}(\mathbf{M}))$*, and let* $R = gr(I_{\mathbf{P}}(P_i[S]))$*. Then, for every ground module atom* α *occurring in* R*, there exists an input splitting set* U *of* R *for* α*.*

Example 9. In Example 7, U is an input splitting set for $P_3[S_a^3].odd$. As $\mathbf{P}$ is c-stratified w.r.t. $\mathbf{M}$ (cf. Example 5) and si-stratified, $\mathbf{P}$ is ic-stratified w.r.t. $\mathbf{M}$. Thus, by Theorem 2, all module atoms in grounded instances from $V(CG_{\mathbf{P}}(\mathbf{M}))$ have input splitting sets.

Naturally, si-stratification implies i-stratification, but not vice versa. However, the following condition identifies a case in which i-stratification holds at the instance level while si-unstratification holds at the schematic level.

Definition 10. *Consider a si-unstratified MLP* $\mathbf{P}$. *If all cycles in* $G_{\mathbf{P}}$ *which include an in-edge also contain an m-edge, then we say that* $\mathbf{P}$ *is* psi-unstratified.

Proposition 2. *If an MLP* $\mathbf{P}$ *is c-stratified and psi-unstratified, then* $\mathbf{P}$ *is i-stratified w.r.t. all interpretations* $\mathbf{M}$*, hence ic-stratified.*

Since ic-stratification (of an MLP $\mathbf{P}$ w.r.t. $\mathbf{M}$) ensures that no cycle in $G_{\mathbf{P}}^{\mathbf{M}}$ has in-edges, it yields intended local splits, where the input for any module atom is *fully prepared* before this module atom is called. Extending the notion of local labeling to an instance local labeling function $ill_i \colon IV \to \mathbb{N}$ s.t. $ill_i(\alpha_1, P_i[S]) > ill_i(\alpha_2, P_i[S])$ if $(\alpha_1, P_i[S]) \rightsquigarrow (\alpha_2, P_i[S])$, one can exploit input splitting sets, starting with a module atom α where $ill_i(\alpha, P_i[S])$ is smallest. For ic-stratified MLPs, such input splitting sets consist of ordinary atoms only, hence respective answer sets can be computed in the usual way. By iteration, this inspires an evaluation algorithm presented next.

4 Top-Down Evaluation Algorithm

A top-down evaluation procedure *comp* for building the answer sets of ic-stratified MLPs along the call graph is shown in Algorithm 1. Intuitively, *comp* traverses the relevant call graph from top to the base and back. In forward direction, it gradually prepares input to each module call in a set R of rules, in the order given by the instance local labeling function for R. When all calls are solved, R is rewritable to a set of ordinary rules, and standard methods can be used to find the answer sets, which are fed back to a calling instance, or returned as the result if we are at the top level.

The algorithm has several parameters: a current set of value calls C, a list of sets of value calls *path* storing the recursion chain of value calls up to C, a partial interpretation $\mathbf{M}$ for assembling a (partial stored) answer set, an indexed set $\mathbf{A}$ of split module atoms (initially, all M_i/S and A_i/S are nil), and a set $\mathcal{AS}$ for collecting answer sets. It uses the following subroutines:

mlpize(N, C) : Convert a set of ordinary atoms N to a *partial* interpretation $\mathbf{N}$ (having undefined components nil), by projecting atoms in N to module instances $P_i[S] \in C$, removing module prefixes, and putting the result at position N_i/S in $\mathbf{N}$.

ans(R) : Find the answer sets of a set of ordinary rules R.

rewrite$(C, \mathbf{M}, \mathbf{A})$: For all $P_i[S] \in C$, put into a set R all rules in $I_{\mathbf{P}}(P_i[S])$, and M_i/S as facts if not nil, prefixing every ordinary atom (appearing in a rule or fact) with $P_i[S]$. Futhermore, replace each module atom $\alpha = P_j[\boldsymbol{p}].o(\boldsymbol{t})$ in R, such that $\alpha \in A_i/S$, by o prefixed with $P_j[T]$, where $T = (M_i/S)|_{\boldsymbol{p_i}}^{\boldsymbol{q_j}}$, and $\boldsymbol{p_i}$ is $\boldsymbol{p}$ without prefixes; moreover add any atoms from $(M_j/T)|_o$ prefixed by $P_j[T]$ to R.

Algorithm 1: $comp$(in: $\mathbf{P}, C, path, \mathbf{M}, \mathbf{A}$, in/out: $\mathcal{AS}$)

Input: MLP $\mathbf{P}$, set of value calls C, list of sets of value calls $path$, partial model $\mathbf{M}$, indexed set of sets of module atoms $\mathbf{A}$, set of answer sets $\mathcal{AS}$

(a) **if** $\exists P_i[S] \in C$ *s.t.* $P_i[S] \in C_{prev}$ *for some* $C_{prev} \in path$ **then**
  if $S \neq \emptyset$ *for some* $P_i[S] \in C$ **then return**
  repeat
    $C' := tail(path)$ and remove the last element of $path$
    if $\exists P_j[T] \in C'$ *s.t.* $T \neq \emptyset$ **then return else** $C := C \cup C'$
  until $C' = C_{prev}$
$R := rewrite(C, \mathbf{M}, \mathbf{A})$
if R *is ordinary* **then**
  if $path$ *is empty* **then**
(b) **forall** $N \in ans(R)$ **do** $\mathcal{AS} := \mathcal{AS} \cup \{\mathbf{M} \uplus mlpize(N, C)\}$
  else
    $C' := tail(path)$ and remove the last element of $path$
    forall $P_i[S] \in C$ **do** $A_i/S := fin$
    forall $N \in ans(R)$ **do** $comp(\mathbf{P}, C', path, \mathbf{M} \uplus mlpize(N, C), \mathbf{A}, \mathcal{AS})$
else
(c) pick an $\alpha := P_j[\boldsymbol{p}].o(\boldsymbol{c})$ in R with smallest $ill_R(\alpha)$ and find splitting set U of R for α
  forall $P_i[S] \in C$ **do if** $A_i/S = nil$ **then** $A_i/S := \{\alpha\}$ **else** $A_i/S := A_i/S \cup \{\alpha\}$
  forall $N \in ans(b_U(R))$ **do**
    $T := N|_{\boldsymbol{p}}^{\boldsymbol{q_j}}$
    if $(M_j/T \neq nil) \wedge (A_j/T = fin)$ **then** $C' := C$ and $path' := path$
    else $C' := \{P_j[T]\}$ and $path' := append(path, C)$
(d) $comp(\mathbf{P}, C', path', \mathbf{M} \uplus mlpize(N, C), \mathbf{A}, \mathcal{AS})$

The algorithm first checks if a value call $P_i[S] \in C$ appears somewhere in $path$ (Step (a)). If yes, a cycle is present and all value calls along $path$ until the first appearance of $P_i[S]$ are joined into C. If a value call in this cycle has non-empty input, then $\mathbf{P}$ is not ic-stratified for any completion of $\mathbf{M}$, and $comp$ simply returns. After checking for (and processing) cycles, all instances in C are merged into R by the function $rewrite$.

If R is ordinary, meaning that all module atoms (if any) are solved, ans can be applied to find answer sets of R. Now, if $path$ is empty, then a main module is reached and $\mathbf{M}$ can be completed by the answer sets of R and put into $\mathcal{AS}$ (Step (b)). Otherwise, i.e., $path$ is nonempty, $comp$ marks all instances in C as finished, and goes back to the tail of $path$ where a call to C was issued. In both cases, the algorithm uses an operator $\uplus$ for combining two partial interpretations as follows: $\mathbf{M} \uplus \mathbf{N} = \{M_i/S \uplus N_i/S \mid P_i[S] \in VC(\mathbf{P})\}$, where $x \uplus y = x \cup y$ if $x, y \neq nil$ and $x \uplus nil = x$, $nil \uplus x = x$.

When R is not ordinary, $comp$ splits R according to a module atom α with smallest $ill_R(\alpha)$ in Step (c). If $C = \{P_i[S]\}$, then $ill_R = ill_i$, otherwise it is a function compliant with every ill_i s.t. $P_i[S] \in C$. Then, $comp$ adds α to $\mathbf{A}$ for all value calls in C, and computes all answer sets of the bottom of R, which fully determine the input for α. If the called instance $P_j[T]$ has already been fully evaluated, then a recursive call with the current C and $path$ yields a proper rewriting of α. Otherwise, the next, deeper level of recursion is entered, keeping the chain of calls in $path$ for coming back (Step (d)).

Example 10. Consider Algorithm 1 on $\mathbf{P}$ from Example 1 and 3. The call chain $P_1[\emptyset] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q'_2} P_3[\{q'_2(a)\}] \xrightarrow{q'_3} P_2[\emptyset] \xrightarrow{q'_2} P_3[\emptyset] \xrightarrow{q'_3} P_2[\emptyset]$ will be reflected by the list $\{P_1[S^1_\emptyset]\}, \{P_2[S^2_{a,b}\}, \{P_3[S^3_a]\}, \{P_2[S^2_\emptyset]\}, \{P_3[S^3_\emptyset]\}$ in $path$, and a current set of value calls $C = \{P_2[S^2_\emptyset]\}$. At this point, the last two elements of the path will be removed and joined with C yielding $C = \{P_2[S^2_\emptyset], P_3[S^3_\emptyset]\}$. The rewriting R w.r.t. C is[1]

$$\left\{ \begin{array}{lll} q^{i'}_\emptyset(X) \vee q^{i'}_\emptyset(Y) \leftarrow q^i_\emptyset(X), q^i_\emptyset(Y), X \neq Y. & \mathit{skip}^i_\emptyset \leftarrow q^i_\emptyset(X), \text{not } q^{i'}_\emptyset(X). \mid i = 1, 2 \\ \mathit{even}^2_\emptyset \leftarrow \mathit{skip}^2_\emptyset, \mathit{odd}^3_\emptyset. & \mathit{odd}^3_\emptyset \leftarrow \mathit{skip}^3_\emptyset, \mathit{even}^2_\emptyset. & \mathit{even}^2_\emptyset \leftarrow \text{not } \mathit{skip}^2_\emptyset. \end{array} \right\}$$

The only answer set of R is $\{\mathit{even}^2_\emptyset\}$. On the way back, even^2_v is toggled with odd^3_w, and at P_1 the answer set $\{q^1_\emptyset(a), q^1_\emptyset(b), \mathit{ok}^1_\emptyset\}$ is built; $comp$ adds a respective (partial) interpretation $\mathbf{M}$ to $\mathcal{AS}$, i.e., where $M_2/\emptyset = \{\mathit{even}\}$, $M_3/\emptyset = \emptyset$, etc., and $M_1/\emptyset = \{q(a), q(b), \mathit{ok}\}$. Following the chain $P_1[\emptyset] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{q'_2} P_3[\{q'_2(b)\}] \xrightarrow{q'_3} P_2[\emptyset] \rightarrow \cdots$, $comp$ finds another answer set of $\mathbf{P}$.

The following proposition shows that $comp$ works for ic-stratified answer sets.

Proposition 3. *Suppose $\mathbf{P}$ is an MLP with single main module $m_1 = (P_1[], R_1)$. Set $\mathcal{AS} = \emptyset$, $path = \epsilon$, $\mathbf{M}$ and $\mathbf{A}$ to have nil at all components. Then, $comp(\mathbf{P}, \{P_1[]\}, path, \mathbf{M}, \mathbf{A}, \mathcal{AS})$ computes in $\mathcal{AS}$ all answer sets $\mathbf{N}$ of $\mathbf{P}$ s.t. $\mathbf{P}$ is ic-stratified w.r.t. $\mathbf{N}$ (disregarding irrelevant module instances, i.e., $N_i/S = nil$ iff $P_i[S] \notin V(CG_\mathbf{P}(\mathbf{N}))$).*

This can be extended to $\mathbf{P}$ with multiple main modules. Compared to a simple guess-and-check approach, $comp$ can save a lot of effort as it just looks into the relevant part of the call graph. Allowing non-ic-stratified answer sets, e.g., loops with non-empty S, is a subject for further work.

5 Discussion

Determining c-stratification of an MLP $\mathbf{P}$ requires checking for cycles in the call graph, which is rather expensive. In practice, it seems useful to perform a syntactic analysis of the rules as a sound yet incomplete test that ic-stratification is given, and to exploit information provided by the programmer.

In a simple form, the programmer makes an *assertion* for specific module calls that when processing these calls recursively, inputs to module calls will always be fully prepared and no call with the same input (except at the base level) is issued. Ideally, the assertion is made for all calls, as possible e.g. in the odd/even example or the Cyclic Hanoi Tower example. While this may sound to put a burden on programmer, in fact one tends quite often, especially for recursive applications, to drive the chain of calls to a base case (e.g., instances with empty input). In such cases, the programmer can confidently provide this information, which can tremendously improve performance.

If no assertions are provided by the programmer, a syntactic analysis might be helpful to compare the inputs of a module call and the module specification. We discuss one such case here. Consider a module atom $P_j[\boldsymbol{p}].o(\boldsymbol{t})$ in module $m_i = (P_i[\boldsymbol{q_i}], R_i)$. Let q_ℓ and p_ℓ be corresponding predicate names in $\boldsymbol{q_i}$ and $\boldsymbol{p}$, respectively. Assume p_ℓ is concluded

[1] Rather than prefixes, we use superscripts and subscripts like for instances (cf. Example 3).

from q_ℓ in one step, i.e., by a rule r where $p_\ell(\boldsymbol{X}) \in H(r)$ and $q_\ell(\boldsymbol{X}) \in B(r)$. Suppose that for all such rules, (i) $q_\ell(\boldsymbol{X}) \in B^+(r)$, (ii) $\nexists q_\ell(\boldsymbol{Y}) \in B(r) \mid \boldsymbol{Y} \neq \boldsymbol{X}$, and (iii) all variables not in $\boldsymbol{X}$ in r are safe. Then, all module atoms have the same or smaller input compared to $\boldsymbol{q_i}$. If p_ℓ is concluded from q_ℓ through a chain of rules $r_1, \ldots, r_m$ where $p_\ell(\boldsymbol{X}) \in H(r_1)$, $q_\ell(\boldsymbol{Y}) \in B(r_m)$, then conditions similar to (i)-(iii) must be respected by each r_i, $1 \leq i \leq m$, taking shared atoms between $B^+(r_i)$ and $H(r_{i+1})$ into account.

Now when compared to a module instance, the input to calls in it is either (a) the same or (b) smaller, the evaluation process can branch into handling these cases, and program rewriting is applicable. For simplicity, we discuss this here for self-recursive MLPs, i.e., module calls within module P_i (different from main) are always to P_i.

Rewrite self-recursive MLPs. For a call atom $\alpha = P_i[\boldsymbol{p}].o(\boldsymbol{t})$ in $P_i[\boldsymbol{q_i}]$, we can guess whether case (a) applies; if so, we can replace α by $o(\boldsymbol{t})$. The resulting rules contain fewer module atoms (if none is left, they are ordinary and can be evaluated as usual). In case (b), if $\mathbf{P}$ is psi-unstratified, we can apply Algorithm 1 (with ill_i replaced by ll_i) to a rewritten program in which additional constraints ensure the decrease of the input.

More formally, let $\mathbf{P} = (m_1, \ldots, m_n)$ where $m_i = (P_i[\boldsymbol{q_i}], R_i)$, $i \in \{1, \ldots, n\}$. Let α be as above and let noc and $\overline{noc}$ be two fresh predicate names. We define two types of rewriting functions, ν and μ, as follows:

– Let $\nu_i(p(\boldsymbol{t})) = p(\boldsymbol{t})$, for each predicate $p \in \mathcal{P}$, and $\nu_i(\alpha) = o(\boldsymbol{t})$. For a rule r of form (1), let $\nu_i(r) = \alpha_1 \vee \cdots \vee \alpha_k \leftarrow noc, \nu_i(\beta_1), \ldots, \nu_i(\beta_m), \text{not } \nu_i(\beta_{m+1}), \ldots, \text{not } \nu_i(\beta_n)$. Then, $\nu(R_i) = \{\nu_i(r) \mid r \in R_i\}$.
– For a rule r, $\mu(r)$ adds $\overline{noc}$ to $B(r)$, and $\mu(R_i) = \{\mu(r) \mid r \in R_i\}$.

Let r_g be $noc \vee \overline{noc} \leftarrow$ and let $Eq_i(\alpha)$ and $Con_i(\alpha)$ be the following sets of rules, where q_ℓ and p_ℓ are corresponding predicate names in the formal input list $\boldsymbol{q_i}$ of $P_i[\boldsymbol{q_i}]$ and the actual input list $\boldsymbol{p}$ of α:

$$Eq_i(\alpha) = \left\{ \begin{array}{l} \mathit{fail} \leftarrow p_\ell(\boldsymbol{X}), \text{not } q_\ell(\boldsymbol{X}), noc, \text{not } \mathit{fail}. \\ \mathit{fail} \leftarrow q_\ell(\boldsymbol{X}), \text{not } p_\ell(\boldsymbol{X}), noc, \text{not } \mathit{fail}. \end{array} \quad \middle| \; 1 \leq \ell \leq |\boldsymbol{q_i}| \right\} ,$$

$$Con_i(\alpha) = \left\{ \begin{array}{l} \mathit{fail} \leftarrow p_\ell(\boldsymbol{X}), \text{not } q_\ell(\boldsymbol{X}), \overline{noc}, \text{not } \mathit{fail}. \\ \mathit{ok} \leftarrow q_\ell(\boldsymbol{X}), \text{not } p_\ell(\boldsymbol{X}), \overline{noc}. \\ \mathit{fail} \leftarrow \overline{noc}, \text{not } \mathit{ok}, \text{not } \mathit{fail}. \end{array} \quad \middle| \; 1 \leq \ell \leq |\boldsymbol{q_i}| \right\} .$$

Let Eq_i and Con_i stand for the union of $Eq_i(\alpha)$ and $Con_i(\alpha)$ for all module atoms α appearing in R_i, resp. For each module m_i, let $\tau(m_i) = \nu(R_i) \cup \mu(R_i) \cup \{r_g\} \cup Eq_i \cup Con_i$. Finally, let $\tau(\mathbf{P}) = (\tau(m_1), \ldots, \tau(m_n))$.

Proposition 4. *Let $\mathbf{P}$ be a psi-unstratified MLP $\mathbf{P}$ where the input to any module call is either equal or strictly smaller compared to the input of the module instance issuing the call. Then the answer sets of $\tau(\mathbf{P})$ correspond 1-1 to those of $\mathbf{P}$.*

The method can be extended to non-self recursive MLPs, where calls to different modules are allowed. Here, one needs to keep track of the module call chain, or assume an ordering on the module names to determine input decrease; we omit the details.

Concerning complexity, using our algorithm suitably, answer-set existence of ic-stratified MLPs is decidable in EXPSPACE, i.e., more efficiently than arbitrary MLPs (2NEXP^{NP}-complete [9]). Since already best practical algorithms for ordinary, call-free programs (NEXP^{NP}-complete) require exponential space, the algorithm has reasonable resource bounds. A detailed complexity analysis is planned for the extended paper.

6 Related Work and Conclusion

In the ASP context, several modular logic programming formalisms have been proposed (cf. Introduction). We already mentioned the modular logic programs of [1] and DLP-functions [3]. For the former, a rich taxonomy of notions of stratification was given in [1]; however, they essentially address merely the module schema level, and no specific algorithms were described. For DLP-functions, Janhunen et al. [3, 4] developed a Module Theorem which allows to compose the answer sets of multiple modules; however, no specific account of stratification was given in [3, 4]. As MLPs can be viewed as a generalization of DLP-Functions, our results may be transferred to the DLP context.

In an upcoming paper, Ferraris et al. present Symmetric Splitting [13] as a generalization of the Module Theorem [3, 4] allowing to decompose also nonground programs like MLPs do. Similar to [3, 4], this technique is only applicable to programs with no positive cycles in the dependency graph. Studying the relationship between Symmetric Splitting and our notions of stratification is an interesting subject for future work.

Several other issues remain for further work, including extensions and refinements of the stratification approach. For example, while we have focused here on decreasing inputs in terms of set inclusion, the extension of the method to other partial orderings of inputs that have bounded decreasing chains is suggestive. This and investigating complexity issues as well as implementation are on our agenda.

References

1. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: LPNMR'97. Springer (1997) 290–309
2. Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP'94, MIT Press (1994) 23–37
3. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. In: LPNMR'07. Springer (2007) 175–187
4. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for Smodels programs. Theory Pract. Log. Program. **8**(5–6) (2008) 717–761
5. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In: ICLP'06. Springer (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. Log. Program. **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. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular Nonmonotonic Logic Programming Revisited. In: ICLP'09. Springer (2009) 145–159
10. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: JELIA'04. Springer (2004) 200–212
11. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective integration of declarative rules with external evaluations for semantic web reasoning. In: ESWC'06. Springer (2006) 273–287
12. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Gener. Comput. **9** (1991) 365–385
13. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Symmetric Splitting in the General Theory of Stable Models. In: IJCAI'09. AAAI Press (2009) 797–803