FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Top-Down Evaluation Techniques for Modular Nonmonotonic Logic Programs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science (Computational Logic) (MSc)

im Rahmen des Studiums

Computational Logic (Erasmus Mundus)

eingereicht von

Tri Kurniawan Wijaya
Matrikelnummer 1028269

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Mitwirkung: Univ. Ass. Dipl.-Ing. Thomas Krennwallner
Minh Dao-Tran, MSc.

Wien, 11.08.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Top-Down Evaluation Techniques for Modular Nonmonotonic Logic Programs

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computational Logic) (M.Sc.)

in

Computatinal Logic (Erasmus Mundus)

by

Tri Kurniawan Wijaya
Registration Number 1028269

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Assistance: Univ. Ass. Dipl.-Ing. Thomas Krennwallner
Minh Dao-Tran, MSc.

Vienna, 11.08.2011

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Tri Kurniawan Wijaya
Toko MELATI, Jl. Muria, Genteng Kulon, Genteng-Banyuwangi 68465, Jawa Timur, Indonesia

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I would to say thanks:

To Prof. Thomas Eiter, who let me work with him and his group on such an interesting topic. Every meeting with him is always exciting and encouraging, and every word from him is always insightful.

To Thomas Krennwallner and Minh Dao-Tran, together with Prof. Eiter they made such a great team to support me from the beginning 'till the end. I cannot imagine finishing all the works without them.

To Peter Schueller, who always there to help me with dlvhex system.

To Eva Nedoma, secretary of our groups; who always eager to help with office and bureaucratic matters. Every morning before anybody comes, she also makes sure that we have enough coffee and milk in our kitchen.

To all of my partners in crime in the third floor: Tran Trung Kien, Carmine Dodaro, Xiao Guohui, Patrik Schneider, and Umut Oztok. I cannot imagine spending my time in the lab without you, guys. All of our jokes, discussions, lunches, and even stupidity, made me 'alive'.

To all of my friends and colleagues for their understanding, support, and encouragement during my 'thesis mode'.

Finally, to my wonderful mother, sister, and families back home who always believe in me.

- Tri -

Abstract

Answer Set Programming (ASP) is a very useful tool for knowledge representation and declarative problem solving. Recently, enabling modularity aspects in ASP has gained increasing interest to help composing (sub-)programs to a combined logic program. Modularity not only allows for problem decomposition, but also facilitates high (code) reusability and provides better support for large-scale projects. Among the contemporary approaches, Modular Nonmonotonic Logic Programs (MLPs) have distinguished strengths, e.g., they allow for mutual recursive calls and utilize predicate symbols as module inputs, resulting in more dynamic problem encodings. MLPs are very expressive and have high computational complexity, thus creating practicable implementations for this formalism is a very challenging task. In this thesis, we develop TD-MLP, a concrete algorithm for computing answer sets for MLPs. TD-MLP is based on a top-down evaluation technique which considers only relevant module calls. In addition, we have devised an optimization technique that splits module instantiations to avoid redundant recomputation. We have incorporated the optimization technique into the original approach and experiments on a benchmark suite show promising results. Furthermore, we also evaluate the performance of different encodings for different problems, involving modular and ordinary encodings. Experiments show in some cases our modular encoding can outperforms the ordinary ones.

Kurzfassung

Answer Set Programming (ASP) ist ein sehr nützliches Werkzeug für die Wissensrepräsentation und zum Lösen von deklarativen Problemstellungen. In letzter Zeit werden Modularitätsaspekte in ASP zunehmend interessant, bei dem es darum geht, (Sub-)Programme zu einem (kombinierten) Logikprogramm zusammenzusetzen. Modularität erlaubt nicht nur das gegebene Problem in seine Teilprobleme zu zerlegen, sondern erleichtert auch die Wiederverwendbarkeit von logischen Programmen und bietet bessere Unterstützung für große Softwareprojekte. Zu den gegenwärtigen Ansätzen in diesem Bereich zählen Modular Nonmonotonic Logic Programs (MLP), welche einige Stärken aufweisen: Sie erlauben wechselseitige rekursive Aufrufe und nutzen Prädikatensymbole als Modul-Input, wodurch dynamischere Kodierungen der Probleme entstehen. MLPs sind sehr ausdrucksstark und haben eine hohe computationale Komplexität, deswegen ist es sehr anspruchsvoll, eine praktikable Implementierung für diesen Formalismus zu erstellen. In dieser Arbeit entwickeln wir TD-MLP, einen konkreten Algorithmus zur Berechnung von Modellen für MLPs. TD-MLP basiert auf Top-down-Auswertungstechniken, die nur relevante Modulaufrufe berücksichtigen. Wir integrieren eine Optimierungstechnik, die Modulinstanzen separiert und damit redundante Berechnungen vermieden werden. Wir haben diese Optimierungstechnik implementiert und Experimente auf Benchmark Instanzen zeigen vielversprechende Resultate. Darüber hinaus evaluieren wir auch unterschiedliche Kodierungen für Probleme, um modularen und mit einfachen logischen Programmen zu vergleichen. Experimente zeigen, dass in einigen Fällen die modulare Kodierung die gewöhnlichen Programme übertreffen können.

Contents

CHAPTER 1

Introduction

The area of Artificial Intelligence (AI) has been growing rapidly since its inception in 1956. Since then, along with molecular biology, AI has regularly been cited as "the field I would most like to be in" by scientists in other disciplines [Russell and Norvig, 2009]. Moreover, John McCarthy called AI as "the science and engineering of making intelligence machines." Currently, AI covers a high variety of subfields, such as problem solving, knowledge representation, automated reasoning, planning, machine learning, natural language processing, computer vision, robotics, and so on. In short, one cannot discuss about an intellectual task done by machine without mentioning AI.

Knowledge representation, problem solving, and automated reasoning are the subfields in AI that became our concern in this thesis. Together, they play an important role in AI. Knowledge representation itself contains a broad discussion on how to store what machine knows, the knowledge of the machines, or description of the world, one might say. Given the stored information, automated reasoning can give an answer to a particular question and even draw a new conclusion. And given a goal description, problem solving find a solution on how to achieve the goal from its current state using any capability it possessed from its reasoning ability.

As one might notice from the title, the topic of this thesis is about modularity in logic programs, in particular, in Answer Set Programs (ASP). Among many approaches, we consider a novel formalism in [Dao-Tran et al., 2009a], modular nonmonotonic logic programs (MLPs). This thesis aims to:

- implements a top-down algorithm to evaluate modules in ASP (in particular, we will use the formalization and approach in [Dao-Tran et al., 2009a,b]),
- devises a technique to optimize and enhance the performance of the algorithm,
- evaluates the implementation through several experiments with benchmark suite.

1.1 Answer Set Programming

Answer set programming (ASP) is one of the AI subfields mentioned previously, namely: knowledge representation, automated reasoning, and problem solving. This due to the fact that ASP:

- has a power to represent knowledge and formalize problems naturally,
- is able to derive new conclusions from facts and rules (description of a world),
- is a declarative programming paradigm that could be used to solve problems in a declarative way [Lifschitz, 2008].

The term ASP itself was first introduced by Vladimir Lifschitz to refer to a new declarative programming paradigm that has its roots in stable model semantics (also known as answer set) of logic programs. One could also say that ASP is a form of declarative programming oriented towards search problems. However, as one could see a little later, ASP is more to the logical representation of a knowledge rather than programming. In order to solve a problem using ASP, a program is devised in such a way that the solutions of the problem can be retrieved from the answer sets of the program. An ASP solver is a system that takes as input a program and computes answer sets for it. *DLV* [Leone et al., 2006], clasp [Gebser et al., 2007], Smodels [Syrjänen and Niemelä, 2001], GnT [Janhunen et al., 2006], Cmodels [Lierler, 2005], ASSAT [Lin and Zhao, 2004] are few examples of ASP solver available nowadays.

As we already know, knowledge representation and problem solving often involves information that is implicitly given. Implicit information, in turn, often requires reasoning to make it explicit. To this end, answer set programming shows its power. To illustrate this, consider the following puzzle in Example 1.1.

Example 1.1 Donald and Daisy Duck took their nephews, age 4, 5, and 6, on an outing. Each boy wore a T-shirt with a different design (camel, giraffe, or panda) on it and of a different color (yellow, white, or green). You are also given the following information:

(a) The 5-year-old wore the T-shirt with the camel design;

(b) Dewey's T-shirt was yellow;

(c) Louie's T-shirt bore the giraffe design;

(d) The panda design was not featured on the white T-shirt.

(e) Huey is younger than the boy in the green T-shirt;

A solution to this puzzle shall consist of a complete description of the T-shirts (color and design) and ages of the three nephews: Dewey, Louie, and Huey.

In order to solve this puzzle, one might encode it into ASP:

Listing 1.1: Disney puzzle

```
% three nephews
nephew(dewey).
nephew(louie).
nephew(huey).
% nephews' age
age(4,X) v age(5,X) v age(6,X) :- nephew(X).
:- age(A,X), age(A,Y), nephew(X), nephew(Y), X!=Y.
% t-shirt design
design(camel,X) v design(giraffe,X) v design(panda,X) :- nephew(X).
:- design(A,X), design(A,Y), nephew(X), nephew(Y), X!=Y.
% t-shirt color
color(yellow,X) v color(white,X) v color(green,X) :- nephew(X).
:- color(A,X), color(A,Y), nephew(X), nephew(Y), X!=Y.
% five further information (a) - (e)
age(5,X) :- design(camel, X), nephew(X).                          % (a)
color(yellow,dewey).                                              % (b)
design(giraffe,louie).                                            % (c)
:- design(panda,X), color(white,X), nephew(X).                    % (d)
:- age(AX,huey), color(green,Y), age(AY,Y), nephew(Y), AX>=AY.    % (e)
```

As one can see the ASP representation of the problem is quite straight-forward. The first 3 lines of the code are to assert the facts that Dewey, Louie, and Huey are the nephews. For the next 6 lines, the code represents the description of the problem that their age should be 4, 5, or 6 (line 4 – 5); the design of their shirt should be camel, giraffe, or panda (line 6 – 7); with yellow, white, or green color (line 8 – 9). Then, the last 5 lines represent the further information given on (a), (b), (c), (d), and (e) respectively.

Please note that the lines that begin with `:-` represent constraints. For example:

```
:- age(A,X), age(A,Y), neph(X), neph(Y), X!=Y.
```

states that if `X` is `A` years old, and `Y` is `A` years old, where `X` and `Y` are the nephews, then both of them cannot be different (`X` and `Y` must be the same person). Running the ASP code in the Listing 1.1 on *DLV*, we will compute exactly one answer set:

```
{
 nephew(dewey), nephew(louie), nephew(huey),
 design(panda,dewey), design(giraffe,louie), design(camel,huey),
 color(yellow,dewey), color(green,louie), color(white,huey),
 age(4,dewey), age(6,louie), age(5,huey)
}
```

If we observe the answer set carefully, then we will realize that it is also the answer of the puzzle! One can interprets the answer set in a very intuitive way, e.g., Dewey wears the panda design, Louie wears the giraffe design, Huey wears the white shirt, Dewey is 4 years old, Huey is 5 years old, and so on.

From Example 1.1 above, one might see that ASP is an elegant tool to represent knowledge, which is very advantageous for the AI community. Not only a very intuitive formalization of a problem (or knowledge), but ASP also fosters an automated reasoning and problem solving at the same time. For details, the formal syntax and semantics of ASP is given in Chapter 2.

1.2 Modularity in Logic Programs

For the past few decades, modularity plays an important role in any programming paradigm. Basically, modularity breaks down a program into modules where each accomplishing one particular task and contains all the source codes and variables needed. By segmenting the program into modules that perform clearly defined functions, we can determine the source of program errors easier and much clearer code can be produced, which in the end enhances the reusability of the codes.

From a problem solving point of view, modular programming also brings a great help to ease problem decomposition in which a big task can be divided into several less complex modules, and similar tasks can be solved together in one module. These properties make modular designed systems more "interesting" than traditional monolithic designs.

Modularity also facilitates the composition of projects into smaller projects. In general, a modularized software project is much easier to handle, since a team member does not need to know about the whole system. This is particularly useful on large-scale projects. She can focus on an assigned smaller task only. Modularity also enables a new member to join the project in the middle of the schedule since she can be assigned into a self-contained module.

The concept of modularity has been growing rapidly in imperative programming and even becomes a must. However, this is not the case with logic programming. For instance, it is a common knowledge that one need to create a monolithic code to solve a problem using logic programming. Despite its advantages as declarative programming paradigm, it will not be very handy when we encounter big and complex problems.

Nevertheless, it does not mean that no researchers are aware of the lack of modularity in logic programming. Researchers in logic and knowledge representation have worked on this topic and gained increasing interest in the last years [Brogi et al., 1994, Bugliesi et al., 1994, Kontchakov et al., 2010]. It has evolved along two different directions:

(i) Various papers have focused primarily on the problems of *programming-in-the-large* [Janhunen et al., 2009, Oikarinen and Janhunen, 2008]. This paradigm introduce compositional operators to combine separate and independent modules.

(ii) Other proposals concentrated on the problem of *programming-in-the-small*, e.g., generalized quantifiers [Eiter et al., 1997], macros [Baral et al., 2006], templates [Calimeri and Ianni, 2006], import rules [Tari et al., 2005] and web rulebases [Analyti et al., 2011]. This paradigm attempts to enrich logic programming with building an abstraction and scoping mechanisms.

This thesis utilizes [Dao-Tran et al., 2009a] as the basis, i.e., modular nonmonotonic logic programs (MLP). This formalism allows providing input and retrieving output from a module

dynamically. The main strength of MLPs compared to other approaches is that MLPs allow recursive calls between modules.

The approach in [Janhunen et al., 2009], DLP-functions, is also one of the premier formalisms to combine ASP modules. It turns out that actually a fragment of MLP can be translated into DLP-functions and vice versa [Dao-Tran et al., 2009a]. However, since MLP admits mutual recursion and module input, MLP can be viewed as a generalization of DLP-functions.

Rooted in theoretical works, some practical implementations of the formalisms have also been made in these directions. However, they do not offer the advantages of modularity to the full extent since some of them are not specifically tailored for modular concepts:

1. *DLT* [Calimeri and Ianni, 2006]: It extends *DLV* with templates predicates, it works as a parser and produces an output that can be computed by *DLV*. The implementation is based on [Calimeri and Ianni, 2006]. However, since the main approach is rewriting the program, it does not support recursive calls between modules.

2. LPEQ [Oikarinen and Janhunen, 2009]: It verifies a variety of equivalences in logic programs in the input language of smodels system [Syrjänen and Niemelä, 2001], including modular equivalence of smodels program modules. It does not allowed positive recursion between modules.

3. Modular AnsProlog [Tari et al., 2005]: It specifically tailored to introduce a modular approach in answer set programming. It uses Smodels [Syrjänen and Niemelä, 2001] as the backbone. The approach had already support input to a module, but only in the form of constants (compared to predicates in MLP).

4. MWeb [Analyti et al., 2011] (short for Modular Web Framework): It views modules as web rulebases. It does not support input to module.

1.3 Thesis Contribution

This thesis has made several contributions, especially to people who are interested in declarative problem solving, logic programming, and knowledge representation:

1. We implement a system that covers the modular approach for logic programs under answer set semantics, MLP [Dao-Tran et al., 2009a]. Since MLP is quite expressive and has a high computational complexity, we evaluate it with the top-down approach described in [Dao-Tran et al., 2009b]. This is the first implementation of such kind of system that allows dynamic input (by the means of predicate symbols) and mutual recursive calls between modules. Since one might have seen the effectiveness of ASP in formally representing knowledge and solving a problem in a declarative way, we are confident that the modular extensions of it can popularize ASP even more.

2. We develop an optimization strategy for the top-down evaluation algorithm in [Dao-Tran et al., 2009b], called *instantiation splitting*. The idea is based on our observation when evaluating a module instantiations. Actually, there is a chance for us to not evaluate the

whole rules. Some part of the rules had already been evaluated when we prepare inputs for module calls. By applying splitting set theorem [Lifschitz and Turner, 1994] on module instantiations, we are able to evaluate only part of the rules that had not been evaluated before. Using this optimization, redundant computation can be avoided.

3. Another concern of such an expressive modular formalism like MLP is the overhead that modules bring in compared to ordinary ASP encodings. For this case, our implementation shows promising results. Re-encoding the ordinary ASP into MLP, experiments suggest that the running time for solving the MLP encoding show a very little time difference only. This has shown us a positive sign for the MLPs development in the future. In addition, there is also a case where MLP confidently outperforms ordinary ASP encodings.

4. Several experiments have been made to investigate the performance of the our implementation. We create a benchmark suite to evaluate the top-down algorithm with and without the optimization. The benchmark suite is also used to measure the performance and scalability of the algorithm for such an unpredictable program. In addition, we also test the system on several well-known problems, such as: hanoi tower, packing problem, and deciding even-odd cardinality of a set. We conduct the experiments using several different ASP solvers, and involving both, MLP and ordinary ASP encodings.

1.4 Organization of the Chapters

This thesis is divided into eight different chapters. From this introduction chapter, we continue with the basic concepts of MLPs, the top-down evaluation approach, the optimization we have made, our system description, and experiment results. We close with conclusions and outlook for further work.

Chapter 2 This chapter contains the basic preliminaries of ASP and MLP. For readers who are not familiar with ASP, this chapter will be a good place to start, while for others who are already familiar with ASP, this chapter provides them with the basic notion that we will use throughout this thesis. The fundamentals of MLP [Dao-Tran et al., 2009a] is then presented in Section 2.2 including many examples.

Chapter 3 As MLP is very expressive, we need nontrivial techniques to evaluate it. This chapter explains the top-down approach that has been proposed in [Dao-Tran et al., 2009b]. In this chapter, we briefly give the basic of splitting set notion that became the underlying idea of the approach, and also some other approaches, e.g., [Oikarinen and Janhunen, 2008]. The top-down evaluation technique in [Dao-Tran et al., 2009b] has became our starting point for implementing the system, and is the first and the only approach that is currently available.

Chapter 4 This chapter discusses our optimization techniques. We give the intuition behind instantiation splitting, formalize it, and give an example on how the evaluation will actually look like at runtime.

Chapter 5 In this chapter we present a new algorithm. This algorithm takes the one in [Dao-Tran et al., 2009b] as the basis and improve it with instantiation splitting techniques. In addition, we also give the proof of soundness and completeness of the algorithm.

Chapter 6 In this chapter, we present the first system, *TD-MLP*, that has been developed to evaluate MLPs. First, we present the architecture of the system. Then, we provide guidance on how to write the input language to the system and how to interpret the results. The format we used is similar to the format of *DLV*. We also explain the parameters of the system which are useful for the users according to their needs.

Chapter 7 The results of experiments with *TD-MLP* are presented in this chapter. It starts from comparing the average of evaluation time using the original approach in [Dao-Tran et al., 2009b] and using the instantiation splitting optimization. Then, we move to see the overhead caused by the modular approach comparing to the original monolithic ASP encoding.

Chapter 8 In the end, we conclude our work and give the interested readers some directions on how and where the work could be advanced in the future.

CHAPTER 2

Preliminaries

This chapter gives basic definitions needed for this thesis. In particular, we present here fundamental concepts of *answer set programs* and *modular nonmonotonic logic programs* [Dao-Tran et al., 2009a]. We give the basic idea of answer set programs first, since modular nonomonotonic logic program (MLP) is defined on top of it.

2.1 Answer Set Programming

Answer Set Programming (ASP) [Brewka et al., 2011] is rooted on the stable model (answer set) semantics for logic programs [Gelfond and Lifschitz, 1988, 1991] and default logic [Reiter, 1980] for the analysis of negation as failure. In ASP, solving search problems are reduced to computing stable models, and answer set solvers (programs for generating stable models) are used to perform search.

In this section we define the syntax of ASP. We took [Eiter et al., 2009] as the main reference for the definitions. Let $\mathcal{V}$ be a vocabulary and $\mathcal{C}, \mathcal{P}, \mathcal{X}$ be mutually disjoint sets whose elements are called *constant*, *predicate*, and *variable* symbols where each predicate symbol has a fixed associated arity $n \geq 0$. We define elements from $\mathcal{C} \cup \mathcal{X}$ as *terms*. An *atom* is of the form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms.

Definition 2.1 (Positive Logic Program) *A* positive (horn) logic program *P is a finite set of clauses (rules) of the form*

$$a \leftarrow b_1, \ldots, b_m \quad (m \geq 0)$$

where $a, b_1, \ldots, b_m$ are atoms. Atom a is called the head *of the rule, while $b_1, \ldots, b_m$ is the* body *of the rule.*

This notion could be interpreted as a *if ... then ...* sentence, i.e., if $b_1, \ldots, b_m$ is true then conclude a. In other words, whenever we have $b_1, \ldots, b_m$, then we also want to have a. A condition whether some atom is true or false is defined over interpretation as follows:

Definition 2.2 (Herbrand Universe, Base, Interpretation) *Given a logic program P, we sat that*

- *the* Herbrand universe *of P, HU_P, is the set of all terms which can be formed from constants and function symbols in P;*
- *the* Herbrand base *of P, HB_P, is the set of all ground atoms which can be formed from predicates and terms $t \in HU_P$;*
- *a (Herbrand)* interpretation *is a first-order interpretation $I = (D, \cdot^I)$ of the vocabulary with domain $D = HU_P$ where each term $t \in HU_P$ is interpreted by itself, i.e., $t^I = t$; and*
- *I is identified with the set $\{p(t_1, \ldots, t_n) \in HB_P \mid \langle t_1^I, \ldots, t_n^I \rangle \in p^I\}$*

Example 2.3 Consider the following program P_{JJ}:

$$\left\{ \begin{array}{rcl} smart(X) & \leftarrow & phd(X) \\ lucky(Y) & \leftarrow & phd(X), couple(X, Y) \\ phd(john) & \leftarrow & \\ couple(john, jane) & \leftarrow & \end{array} \right\}$$

Since the set of constant symbols appearing in P_{JJ} is $\{john, jane\}$, $HU_{P_{JJ}} = \{john, jane\}$. The set of predicate symbols in P_{JJ} is $\{phd, smart, lucky, couple\}$, hence

$$HB_{P_P} = \left\{ \begin{array}{l} phd(john), phd(jane), smart(john), smart(jane), \\ lucky(john), lucky(jane), couple(john), couple(jane), \end{array} \right\},$$

which is basically the set of all possible atoms that can be formed using predicates and terms in P_{JJ}.

This program also has many possible Herbrand interpretations. Some of them are:

- $I_1 = \emptyset$,
- $I_2 = HB_{P_{JJ}}$,
- $I_3 = \{phd(john), lucky(jane), couple(john, jane)\}$,

etc.

The semantics of positive logic programs is defined in terms of grounding. Grounding can also be seen as a way to materialize the universal quantification of variables appearing in a clause.

Definition 2.4 (Ground Instance) *Let $Var(C)$ denotes the set of variables in clause C, then a* ground instance *of a clause C is any clause C' obtained from C by applying a substitution:*

$$\theta : Var(C) \rightarrow HU_P$$

to the variables in C. $gr(C)$ denotes the set of all possible ground instances of C. And for any program P, the grounding *of P is $gr(P) = \bigcup_{C \in P} gr(C)$*

Example 2.5 Let us consider the second rule from program P_{JJ} in the Example 2.3:

$$r = smart(X) \leftarrow phd(X)$$

Its ground instance will be:

$$gr(r) = \left\{ \begin{array}{lcl} smart(john) & \leftarrow & phd(john) \\ smart(jane) & \leftarrow & phd(jane) \end{array} \right\}$$

Next, we define when an interpretation is compatible with a clause, and finally with a program.

Definition 2.6 (Model) *An interpretation I is a (Herbrand)* model *of*

- *a ground (variable-free) clause $C = a \leftarrow b_1, \ldots, b_m$, denoted as $I \models C$, if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $a \in I$;*
- *a clause C, denoted as $I \models C$, if $I \models C'$ for every $C' \in gr(C)$; and*
- *a program P, denoted as $I \models P$, if $I \models C$ for every clause C in P.*

A model I of P is minimal*, if there exists no model J of P such that $J \subset I$. If a program has a single minimal model, such model is also called the* least model *of P.*

Example 2.7 Consider program P_{JJ} and interpretation I_1, I_2, I_3 from Example 2.3.

- I_1 is not a model for P_{JJ} since at least we need to have $phd(john)$ and $couple(john, jane)$ in our model.
- I_2 is a model. In addition, for an arbirary positive logic program P, HB_P is a model of P.
- I_3 is not a model since whenever we have $phd(john)$ then we also need to have $smart(john)$.

In addition, $I_4 = \{phd(john), couple(john, jane), lucky(jane), smart(john)\}$ is a model of P_{JJ}. There are also several facts about the models:

- I_2 is not a minimal model, since $I_4 \subset I_2$ is also a model of P_{JJ}.
- I_4 is a minimal model of P_P, and also the least model of P_{JJ}.

One can easily justify that indeed I_4 is the least model for P_{JJ} (hence also the minimal model of P_{JJ}) by applying the *immediate consequence operator* defined below.

Definition 2.8 (Immediate Consequence Operator) Immediate consequence operator *for a program P, $T_P : 2^{HB_P} \rightarrow 2^{HB_P}$, is defined as:*

$$T_P(I) = \{a \mid \text{there exists some } a \leftarrow b_1, \ldots, b_n \text{ in } gr(P) \text{ such that } \{b_1, \ldots, b_n\} \subseteq I\}.$$

We define $T_P^0 = \emptyset$, and $T_P^{i+1} = T_P(T_P^i)$ for $i \geq 0$. The least model of a program P can be obtained through an iterative process of applying T_P^i, starts from $i = 0$ until it converges (reaches its fixpoint).

Example 2.9 Let us consider program P_{JJ} from Example 2.3. Applying *immediate consequence operator* to P_{JJ}:

- $T_P^0 = \emptyset$
- $T_P^1 = T_P(\emptyset) = \{phd(john), couple(john, jane)\}$
- $T_P^2 = T_P(\{phd(john), couple(john, jane)\}) =$
  $\{phd(john), couple(john, jane), lucky(jane), smart(john)\}$
- $T_P^3 = T_P(\{phd(john), couple(john, jane), lucky(jane), smart(john)\}) =$
  $\{phd(john), couple(john, jane), lucky(jane), smart(john)\} = T_P^2$ (reaches fixpoint)

Sometimes, a counterfactual about a particular condition appears to be more intuitive. In some cases, negation often appear to be more declarative representation, especially when stating all possible negative facts considered to be an expensive task. Consider an example where there are two trains scheduled from Vienna to Salzburg today, at 10.00 and 14.00. Then, we typically assume that there will be no train from Vienna to Salzburg at 9.00. Instead of writing whether there is a train or not for each possible minutes (or even seconds), the schedule is written in such a way that it displays only the time when there is a train. Other than that (if it is not written in the schedule), there will be no train from Vienna to Salzburg. One could intuitively represent a simple representation about this as:

$$noTrainAtTime(X) \leftarrow \text{not } scheduledAtTime(X)$$

Definition 2.10 (Normal Logic Program) *A* normal logic program *is a set of rules of the form*

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \quad (m, n \geq 0)$$

where $a, b_1, \ldots, b_j, c_1, \ldots, c_n$ are atoms. We call not *as "negation as failure", "default negation", or "weak negation".*

Definition 2.11 (Reduct, Stable Model) *The Gelfond-Lifschitz reduct [Gelfond and Lifschitz, 1988], or GL-reduct, or simply* reduct *of a ground program P with respect to an interpretation M, denoted as P^M, is the program obtained from P by*

(i) removing rules with not *p in the body if $p \in M$; and*

(ii) removing literals not *q from all rules if $q \notin M$*

An interpretation M of P is a stable model *of P, if M is a minimal model of P^M.*

Example 2.12 Consider the following program P_N:

$$\left\{ \begin{array}{rcl} child(gill) & \leftarrow & \\ boy(X) & \leftarrow & child(X), \text{not } female(X) \end{array} \right\}$$

Intuitively, the second rule can be read as: If X is a child and we cannot prove that X is a female, then we conclude that X is a boy. One can easily verifies that the ground program of P_N is:

$$\left\{ \begin{array}{lcl} child(gill) & \leftarrow & \\ boy(gill) & \leftarrow & child(gill), \text{not}\, female(gill) \end{array} \right\}$$

Consider interpretations:

- $M_1 = \{child(gill), boy(gill)\}$, then condition *(ii)* from Definition 2.11 is applied on the second rule. The reduct $P_N^{M_1} =$

$$\left\{ \begin{array}{lcl} child(gill) & \leftarrow & \\ boy(gill) & \leftarrow & child(gill) \end{array} \right\}$$

  Since M_1 is the least model of $P_N^{M_1}$, it is a model of P_N.

- $M_2 = \{child(gill), female(gill), boy(gill)\}$, then condition *(i)* from Definition 2.11 is applied on the second rule. The reduct $P_N^{M_2} =$

$$\left\{ child(gill) \quad \leftarrow \quad \right\}$$

  M_2 is a model for $P_N^{M_2}$, but it is not minimal. This because we have $\{child(gill)\} \subset M_2$ is also a model for $P_N^{M_2}$. Hence, M_2 is not a model for P_N

As one can see from the Example 2.12, M_2, which is the HB_{P_N}, is not a model for P_N. In general, for a normal logic program P, HB_P is no longer always a model P (as in positive logic program).

Next, let us take a look at a particular rule that with an empty head.

Definition 2.13 (Constraints) *A* constraint *is a rule of the form*

$$\leftarrow b_1, \ldots, b_m, \text{not}\, c_1, \ldots, \text{not}\, c_n \quad (m, n \geq 0)$$

where $b_1, \ldots, b_j, c_1, \ldots, c_n$ are atoms.

Consider a constraint

$$\leftarrow a, b, c$$

. Intuitively it says that there could not be the case that a, b, and c in the model at the same time.

In other cases, sometimes we also want to state that if a certain condition is satisfied then there could be several consequences applied. This idea is expressed within disjunctive rule that allows disjunction in the rule head, which then becomes an important ingredient of answer set programs.

Definition 2.14 (Disjunctive Rule, Disjunctive Logic Program) *A disjunctive rule is a rule of the form:*

$$a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not}\, c_1, \ldots, \text{not}\, c_n$$

where $a_1, \ldots, a_k, b_1, \ldots, b_m, c_1, \ldots, c_n$ are atoms and $k, m, n \geq 0$. A disjunctive logic program *is a finite set of disjunctive rules. An* answer set program *is of the form of disjunctive logic program.*

Definition 2.15 (Answer Sets) *We define a notion of a model taking multiple atoms in the head into account. An interpretation I is a model of*

- *a ground clause C: $a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not}\, c_1, \ldots, \text{not}\, c_n$, denoted as $I \models C$, if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $\{a_1, \ldots, a_k, c_1, \ldots, c_n\} \cap I \neq \emptyset$*
- *a clause C, denoted as $I \models C$, if $I \models C'$ for every $C' \in gr(C)$; and*
- *a program P, denoted as $I \models P$, if $I \models C$ for every clause C in P.*

An answer set *M of a program P, is a minimal model of P^M.*

For disjunctive rules, definition above states that if the rule body is satisfied, i.e., $\{b_1, \ldots, b_m\} \not\subseteq I$ and $\{c_1, \ldots, c_n\} \cap I \neq \emptyset$, then at least one of the head must be in the model ($\{a_1, \ldots, a_k\} \cap I \neq \emptyset$).

The definition of an answer set above also take *constraint* into account. If the rule we consider is a constraint, then $k = 0$. Then, either

- $\{b_1, \ldots, b_m\} \not\subseteq I$ (at least a positive body atoms not in the model), or
- $\{c_1, \ldots, c_n\} \cap I \neq \emptyset$ (at least a negative body atom in the model).

Example 2.16 and 2.17 give a clear idea about models in disjunctive logic program. For simplicity, these two examples disregard the existence of negation as failure. We give a more complex example involving disjunction in the head and negation as failure in the body in Example 2.18.

Example 2.16 Consider the following programs P_{D_1}:

$$\left\{ \; a \vee b \vee c \quad \leftarrow \quad \right\}$$

Intuitively P_{D_1} can be read as: in all cases, at least one of a, b, or c is true. Hence, interpretation M is a model of P_{D_1}, if it contains at least one element of the set $\{a, b, c\}$. For example:

- $M_1 = \emptyset$ is not a model of P_{D_1}.
- $M_2 = \{a\}$, $M_3 = \{b\}, M_4 = \{c\}$ are models of P_{D_1} and they are minimal (since the only possible proper subset of M_2, M_3, M_4 is $\emptyset$ but $\emptyset$ is not a model of P_{D_1}).
- $M_5 = \{a, c\}$ and $M_6 = \{a, b, c\}$ are also models of P_{D_1}, but they are not minimal models.

Example 2.17 Consider the following programs P_{D_2}:

$$\left\{ \; graduate(tom) \vee company(tom) \leftarrow smart(tom) \; \right\}$$

Program P_{D_2} can be read as: whenever $smart(tom)$ is true, at least one of the $graduate(tom)$ and $company(tom)$ should be true as well. There are several possible interpretations for P_{D_2}:

- $M_1 = \{smart(tom), graduate(tom)\}$, $M_2 = \{smart(tom), graduate(tom), company(tom)\}$ are models of P_{D_2}.

- $M_3 = \{smart(tom)\}$ is not a model of P_{D_2}.
- $M_4 = \emptyset$ is also a model of P_{D_2} and it is minimal.

The following example gives a more comprehensive discussion in determining whether an interpretation is an answer set of a program. It includes disjunction and negation as failure.

Example 2.18 Consider another program P below:

$$\left\{ \begin{array}{rcll} a \vee b & \leftarrow & & (1) \\ p & \leftarrow & \text{not } b & (2) \end{array} \right\}$$

Let us now as examples take 5 possible interpretations of P:

- $M_1 = \{p, a\}$. One can quickly verified that M_1 is an answer set of P. Condition (ii) from Definition 2.11 is applied, and give us the reduct P^{M_1}:

$$\left\{ \begin{array}{rcl} a \vee b & \leftarrow & \\ p & \leftarrow & \end{array} \right\}$$

  M_1 is a minimal model of P^{M_1}. Hence, M_1 is an answer set for P.

- $M_2 = \{p, b\}$. In this case, condition (i) from Definition 2.11 is applied to the rule (2). We have P^{M_2}:

$$\left\{ \begin{array}{rcl} a \vee b & \leftarrow & \end{array} \right\}$$

  M_2 is not an answer set of P because M_2 is not a minimal model of P^{M_2}. This is due to $\{a\}$ and $\{b\}$ are proper subsets of M_2 which are also models of P^{M^2}.

- $M_3 = \{p, a, b\}$. Since $b \in M_3$, condition (i) from Definition 2.11 is applied to rule (2). The reduct $P^{M_3} = P^{M_2}$. Since $\{p, a, b\}$ is also not a minimal model of P^{M_2}, M_3 is not an answer set of P.

- $M_4 = \{a\}$. The reduct $P^{M_4} = P^{M_1}$, and since M_4 is not even a model of P^{M_4}, it is not an answer set of P.

- $M_5 = \{b\}$. The reduct $P^{M_5} = P^{M_3} = P^{M_2}$, and since M_5 is one of the minimal model P^{M_5}, it is an answer set of P.

Example 2.19 give an idea considering a rule of the form of constraint into account.

Example 2.19 Consider a program P_C:

$$\left\{ \begin{array}{rcl} a \vee b & & \\ c \vee d & & \\ & \leftarrow & b, c \end{array} \right\}$$

If we consider only the first and the second rule, then we will have four answer sets:

- $M_1 = \{a, c\}$,
- $M_2 = \{a, d\}$,
- $M_3 = \{b, c\}$, and
- $M_4 = \{b, d\}$

However, we have the third rule that requires to not include b and c at the same time. This removes M_3 from the list of our answer sets, which eventually yields only three answer sets: M_1, M_2, and M_4.

2.2 Modular Nonmonotonic Logic Programs

Now, we present the syntax and semantics of Modular Nonmonotonic Logic Programs (MLP) which was originally introduced in [Dao-Tran et al., 2009a].

2.2.1 Syntax of MLPs

The syntax of MLPs is based on answer set programs. MLPs consist of modules as a way to structure logic programs. In MLPs, each module can receive input provided by other modules. In addition, one module may call other modules and additionally provide input for that call. Modules may also mutually call each other in a recursive way.

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. 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 $\mathbf{q} = q_1, \ldots, q_k$ $(k \geq 0)$ of predicated names $q_i \in \mathcal{P}$ (the formal input parameters). As in logic programs, elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*.

Definition 2.20 (Ordinary and Module Atoms) *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*

$$P[p_1, \ldots, p_k].o(t_1, \ldots, t_n) \ , \tag{2.1}$$

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 n such that for the list of terms $t_1, \ldots, t_n$, $o(t_1, \ldots, t_n)$ 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 $o(\mathbf{t})$ in a module P.

Definition 2.21 (Rule) *A* rule *r is of the form*

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \mathrm{not}\, \beta_{m+1}, \ldots, \mathrm{not}\, \beta_n \ , \tag{2.2}$$

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.

As usual we define the head of the rule, $H(r) = \{\alpha_1, \ldots, \alpha_k\}$, and the body of the rule, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\beta_{m+1}, \ldots, \beta_n\}$. And we define $at(R) = H(r) \cup B^+(r) \cup B^-(r)$ as the set of all atoms occuring in r. A rule r is ordinary *if it contains only ordinary atoms.*

From Definition 2.21, we know that rules in MLPs actually extend disjunctive rules with a slight modification, namely allowing module atoms in the body. We next proceed to define what a module is.

Definition 2.22 (Module) *A* module *is a pair $m = (P[\mathbf{q}], R)$, where $P \in \mathcal{M}$ with associated formal input $\mathbf{q}$ which is a list of predicates, and R is a finite set of rules. A module 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 $|\mathbf{q}| = 0$, then it is a main module. We also define $R(m)$ as the rule set of module m and omit empty* [] *from (main) modules and module atoms if it is unambiguous.*

And finally we define a modular logic program formally.

Definition 2.23 (Modular Logic Program) *A* Modular Logic Program *(MLP)* **P** *is an n-tuple of modules*

$$(m_1, \ldots, m_n) \ , n \geq 1, \tag{2.3}$$

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

Example 2.24 Let us consider a simple example taken from [Dao-Tran et al., 2009a], MLP $\mathbf{P}_A = (m_1, m_2, m_3)$, where:

- $m_1 = (P_1[], \{a \leftarrow P_2.b\})$,
- $m_2 = (P_2[], \{b \leftarrow P_1.a\})$,
- $m_3 = (P_3[c], \{c \leftarrow \text{not } c\})$.

Both m_1 and m_2 are main modules, and m_3 is a library module. While a, b, and c are ordinary atoms, $P_2.b$ and $P_1.a$ are examples of module atoms. Intuitively, m_1 and m_2 resemble the logic program:

$$\left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

while m_3 is a simple constraint with formal input c.

Example 2.25 Let us now consider a more complex MLP. Let $\mathbf{P}_E$ 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:

$$R_1 = \left\{ \begin{array}{lcl} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ even & \leftarrow & P_2[q].even \\ odd & \leftarrow & \text{not } even \end{array} \right\},$$

$$R_2 = \left\{ \begin{array}{rcl} q_2'(X) \vee q_2'(Y) & \leftarrow & q_2(X), q_2(Y), X \neq Y \\ skip_2 & \leftarrow & q_2(X), \text{not } q_2'(X) \\ even & \leftarrow & \text{not } skip_2 \\ even & \leftarrow & skip_2, P_3[q_2'].odd \end{array} \right\},$$

$$R_3 = \left\{ \begin{array}{rcl} q_3'(X) \vee q_3'(Y) & \leftarrow & q_3(X), q_3(Y), X \neq Y \\ skip_3 & \leftarrow & q_3(X), \text{not } q_3'(X) \\ odd & \leftarrow & skip_3, P_2[q_3'].even \end{array} \right\}.$$

In this example, we have one main module only, m_1, while the other two, m_2 and m_3 are library modules with formal input q_2 and q_3 respectively. Intuitively, $\mathbf{P}_E$ determines whether the set q in R_1 has even or odd number of elements.

Let us now take a closer look to Example 2.25 in order to understand the intuition behind $\mathbf{P}_E$ better. In the beginning, m_1 calls m_2 and passes q to check whether the number of facts for predicate q is even. In this case, m_2 and m_3 are the modules that work together in order to decide whether the number of facts for their formal input is even/odd.

The idea behind m_2 is to single out one element from its formal input (done by the first rule), put it into q_2', and then call m_3 to check whether the number of facts in q_2' is odd (which means the number of facts in q_2 is even, see the fourth rule). In addition, if the formal input has no fact, *even* will also be set to true since $skip_2$ is false (see the second and the third rule of R_2). Similar to module m_2, m_3 singles out one fact from its formal input and passes it to m_2 to see whether it has even number of facts (if that is the case, *odd* is set to true).

2.2.2 Semantics of MLPs

The semantics of MLPs is defined in terms of Herbrand interpretations and ground as in ordinary answer set programs.

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 $HB_{\mathbf{P}}$.

Definition 2.26 (Grounding) *The grounding of:*

- *a rule r is the set $gr(r)$ of all ground instances of r w.r.t. $\mathcal{C}$*
- *a rule set R is $gr(R) = \bigcup_{r \in R} gr(r)$*
- *a module m, $gr(m)$, is defined by replacing the rules in $R(m)$ by $gr(R(m))$*
- *an MLP $\mathbf{P}$ is $gr(\mathbf{P})$, which is formed by grounding each module m_i in $\mathbf{P}$*

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

Before going further, we define some notation that will ease our next definitions. Let $S \subseteq HB_{\mathbf{P}}$ be any set of atoms. For any list of predicate names $\mathbf{p} = p_1, \ldots, p_k$ and $\mathbf{q} = q_1, \ldots, q_k$, we define

- $S|_{\mathbf{p}} = \{p_i(\mathbf{c}) \in S \mid i \in \{1, \ldots, k\}\}$

- $S|^{\mathbf{q}}_{\mathbf{p}} = \{q_i(\mathbf{c}) \mid p_i(\mathbf{c}) \in S, i \in \{1, \ldots, k\}\}$

In the sequel, we define the notion of module instantiations. For this purposes, we need to identify each module in MLP $\mathbf{P}$ with input it received. This concept is captured in the definition of *value call* below.

Definition 2.27 (Value Call) *For an MLP $\mathbf{P}$, a $P \in \mathcal{M}$ with associated formal input $\mathbf{q}$ we say that $P[S]$ is a* value call with input S*, where $S \subseteq HB_{\mathbf{P}}|_{\mathbf{q}}$. Let $VC(\mathbf{P})$ denote the set of all value calls $P[S]$ in $\mathbf{P}$ with input S such that $P \in \mathcal{M}$.*

Example 2.28 Take $\mathbf{P}_A$ from Example 2.24. Considering all possible inputs for each module in $\mathbf{P}_A$, we have $VC(\mathbf{P}_A) = \{P_1[\emptyset], P_2[\emptyset], P_3[\emptyset], P_3[\{c\}]\}$.

Definition 2.29 (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 2.30 (Instantiation) *For a module $m_i = (P_i[\mathbf{q_i}], R_i)$ from $\mathbf{P}$, its* instantiation with $S \subseteq HB_{\mathbf{P}}|_{\mathbf{q_i}}$*, is $I_{\mathbf{P}}(P_i[S]) = R_i \cup S$. For an MLP $\mathbf{P}$, its* instantiation *is the rule base $I(\mathbf{P}) = (I_{\mathbf{P}}(P_i[S]) \mid P_i[S] \in VC(\mathbf{P}))$.*

Example 2.31 Let us consider MLP $\mathbf{P}_A$ from Example 2.24. Instantiation of $P_3[\{c\}]$ (P_3 with input $\{c\}$) is:

$$I_{\mathbf{P}_A}(P_3[\{c\}]) = \left\{ \begin{array}{lll} c & \leftarrow & \\ c & \leftarrow & \text{not } c \end{array} \right\}.$$

Please note that in order to instantiate an MLP $\mathbf{P}$, one has to consider all possible inputs for each module in $\mathbf{P}$. We define an *interpretation* and a *model* of $\mathbf{P}$ based on an instantiation of $\mathbf{P}$.

Definition 2.32 (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 $M_i/S \subseteq HB_{\mathbf{P}}$ contains only ordinary atoms.*

Example 2.33 Considering $\mathbf{P}_A$ from Example 2.24, then $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset, M_3/\emptyset, M_3/\{c\})$ where $M_1/\emptyset = \{a\}$, $M_2/\emptyset = \{b\}$, and $M_3/\emptyset = M_3/\{c\} = \{c\}$ is an example of interpretation of $\mathbf{P}_A$.

We can now define models of an MLP $\mathbf{P}$.

Definition 2.34 (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 $\alpha = P_k[\mathbf{p}].o(\mathbf{c})$ is a module atom, $o(\mathbf{c}) \in M_k/((M_i/S)|^{\mathbf{q_k}}_{\mathbf{p}})$;*
- *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.*

Example 2.35 Consider $\mathbf{M}$ from Example 2.33 and $\mathbf{P}_A$ from Example 2.24. $\mathbf{M}$ is a model of $\mathbf{P}_A$.

- First, let us consider m_1 and m_2.

  We have $\mathbf{M}, P_1[\emptyset] \models a$ and $\mathbf{M}, P_2[\emptyset] \models b$. This also means that $\mathbf{M}, P_1[\emptyset] \models P_2.b$ and $\mathbf{M}, P_2[\emptyset] \models P_1.a$. Then $\mathbf{M}, P_1[\emptyset] \models a \leftarrow P_2.b$ and $\mathbf{M}, P_2[\emptyset] \models b \leftarrow P_1.a$.

  Hence $\mathbf{M}, P_1[\emptyset] \models I_{\mathbf{P}}(P_1[\emptyset])$ and $\mathbf{M}, P_2[\emptyset] \models I_{\mathbf{P}}(P_2[\emptyset])$.

- Next, we show that $\mathbf{M}, P_3[\emptyset] \models I_{\mathbf{P}}(P_3[\emptyset])$, and $\mathbf{M}, P_3[\{c\}] \models I_{\mathbf{P}}(P_3[\{c\}])$.

  $\mathbf{M}, P_3[\emptyset] \models c$ and $\mathbf{M}, P_3[\emptyset] \models c \leftarrow \text{not } c$. This is also the case for $P_3[\{c\}]$, i.e., $\mathbf{M}, P_3[\{c\}] \models c$ and $\mathbf{M}, P_3[\{c\}] \models c \leftarrow \text{not } c$.

  Hence, $\mathbf{M}, P_3[\emptyset] \models I_{\mathbf{P}}(P_3[\emptyset])$, and $\mathbf{M}, P_3[\{c\}] \models I_{\mathbf{P}}(P_3[\{c\}])$.

- Since $I(\mathbf{P}) = (I_{\mathbf{P}}(P_1[\emptyset]), I_{\mathbf{P}}(P_2[\emptyset]), I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$, then we have $\mathbf{M} \models \mathbf{P}$.

Finally before we proceed to define an answer set of an MLP $\mathbf{P}$, we define the notion of minimal models. Since an interpretation is now defined as a tuple, we also need to compare interpretations based on the tuples.

Definition 2.36 (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.,* $MM(\mathbf{R})$*).*

When evaluating module calls starting from a particular module, not all module instantiations are needed. We call those which contribute to the evaluation as the *relevant* ones. The following notion of *(relevant) call graph* helps us with respect to this. Then, we proceed to define context and context-based reduct, and finally the notion of answer set for MLPs.

Definition 2.37 (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* $P_k[\mathbf{p}].o(\mathbf{t})$ *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* $(M_i/S)|^{\mathbf{q_k}}_{l(e)} = T$*, 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 2.38 Let us consider P_A from Example 2.24 and $VC(P_A)$ from Example 2.28. The call graph of $\mathbf{P}_A$ is simply $CG_{\mathbf{P}_A} = (VC(\mathbf{P}_A), E, l)$, where $E = \{(P_1[\emptyset], P_2[\emptyset]), (P_2[\emptyset], P_1[\emptyset])\}$, and l maps each edge to the void input list. See Figure 2.1(a) for an illustration.

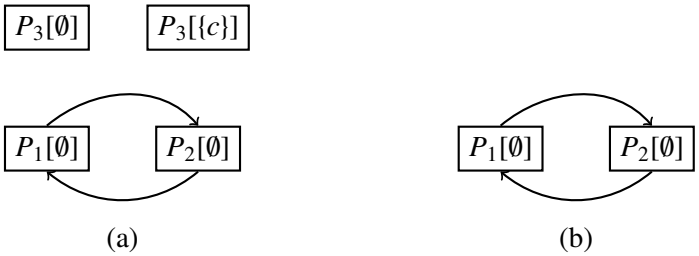


Figure 2.1: (a) $CG_{\mathbf{P}_A}$, (b) $CG_{\mathbf{P}_A}(\mathbf{M})$ as in Example 2.38

Moreover, $CG_{\mathbf{P}_A}(\mathbf{M}) = (\{P_1[\emptyset], P_2[\emptyset]\}, E, l)$ is a relevant call graph of $\mathbf{P}_A$ with respect to $\mathbf{M}$ for any interpreation $\mathbf{M}$ of $\mathbf{P}_A$ since both $P_1[\emptyset]$ and $P_2[\emptyset]$ are main modules (they are always relevant), and $P_3[\emptyset]$ and $P_3[\{c\}]$ are never called (they are always irrelevant). Figure 2.1(b) illustrates this.

Definition 2.39 (Context) *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$*. A* context for $\mathbf{M}$ *is set of value calls* C*, where* $V(CG_{\mathbf{P}}(\mathbf{M})) \subseteq C \subseteq VC(\mathbf{P})$*.*

Below we give a definition of the reduct of a program, which is based on the FLP-reduct [Faber et al., 2004].

Definition 2.40 (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* $f\mathbf{P}^{\mathbf{M},C} = (f\mathbf{P}(P[S])^{\mathbf{M},C} \mid P[S] \in VC(\mathbf{P}))$*.*

Definition 2.41 (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* $f\mathbf{P}^{\mathbf{M},C}$*.*

From Definition 2.40 we know that the reduct is not applied to all instantiations of the program, but only to the instantiations in the context. Since the definition about answer set is defined on top of a context, as a consequences, it could be the case that an answer set of an MLP $\mathbf{P}$ with respect to a particular context C_1 is not an answer set with respect to a different context C_2.

Example 2.42 Consider $\mathbf{P}_A$ from Example 2.24 and two interpretations:

- $\mathbf{M} = (M_1/\emptyset, M_2/\emptyset, M_3/\emptyset, M_3/\{c\})$ from Example 2.33, and
- $\mathbf{M}_0 = (M_1^0/\emptyset, M_2^0/\emptyset, M_3^0/\emptyset, M_3^0/\{c\})$, where $M_1^0/\emptyset = M_2^0/\emptyset = \emptyset$, $M_3^0/\emptyset = M_3^0/\{c\} = \{c\}$.

Let us define a context $C = \{P_1[\emptyset], P_2[\emptyset]\}$.

- One can verify that $\mathbf{M}$ is not an answer set of $\mathbf{P}_A$ w.r.t C. This is due to the fact that:
  (i) $f\mathbf{P}_A^{\mathbf{M},C} = (f\mathbf{P}_A(P_1[\emptyset])^{\mathbf{M},C}, f\mathbf{P}_A(P_2[\emptyset])^{\mathbf{M},C}, f\mathbf{P}_A(P_3[\emptyset])^{\mathbf{M},C}, f\mathbf{P}_A(P_3[\{c\}])^{\mathbf{M},C}) = (I_{\mathbf{P}_A}(P_1[\emptyset]), I_{\mathbf{P}_A}(P_2[\emptyset]), I_{\mathbf{P}_A}(P_3[\emptyset]), I_{\mathbf{P}_A}(P_3[\{c\}]))$, and

(ii) $\mathbf{M}$ is not a minimal model of $f\mathbf{P}_A^{\mathbf{M},C}$ (since $\mathbf{M}_0 < \mathbf{M}$ and $\mathbf{M}_0$ is also a model of $f\mathbf{P}_A^{\mathbf{M},C}$).

- Now, let us consider $\mathbf{M}_0$. The reduct w.r.t. $\mathbf{M}_0$ is $f\mathbf{P}^{\mathbf{M}_0,C} = (f\mathbf{P}(P_1[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_2[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\emptyset])^{\mathbf{M}_0,C}, f\mathbf{P}(P_3[\{c\}])^{\mathbf{M}_0,C}) = (\emptyset, \emptyset, I_{\mathbf{P}}(P_3[\emptyset]), I_{\mathbf{P}}(P_3[\{c\}]))$. Since $\mathbf{M}_0$ is a minimal model of $f\mathbf{P}^{\mathbf{M}_0,C}$ is , it is an answer set of $\mathbf{P}$ w.r.t. C.

Let us now consider a different context, namely $C_2 = \{P_1[\emptyset], P_2[\emptyset], P_3[\emptyset]\}$. In this case $\mathbf{M}_0$ is no longer an answer set of $\mathbf{P}_A$ (with respect to context C_2), since $f\mathbf{P}^{\mathbf{M}_0,C_2} = (\emptyset, \emptyset, \emptyset, I_{\mathbf{P}}(P_3[\{c\}]))$, and $\mathbf{M}_0$ is not its minimal model.

The context C decides the overall-satisfiability of an MLP. From now on, for an MLP $\mathbf{P}$ and its interpretation $\mathbf{M}$, we consider the minimal context $C = V(CG_{\mathbf{P}}(\mathbf{M}))$ as the default context (unless stated otherwise). Note that the maximal reduct $C = VC(P)$ requires all module instances to have answer sets, while the minimal context $C = V(CG_{\mathbf{P}}(\mathbf{M}))$ considers only the relevant call graph of $\mathbf{P}$ with respect to $\mathbf{M}$.

CHAPTER 3

Top-Down Approach for MLPs

This chapter describes a top-down evaluation technique to compute MLP semantics presented in [Dao-Tran et al., 2009b]. First, we recall the concept of splitting sets since they will be used extensively later. Next, we define some notions of splitting a particular class of modular programs called input and call stratified MLPs, which admit a top-down approach to compute its answer sets. Finally, we present the original top-down evaluation algorithm [Dao-Tran et al., 2009b] for this fragment.

3.1 Splitting Sets

Splitting sets were first introduced in [Lifschitz and Turner, 1994] with the purpose of making the evaluation of logic programs faster. The idea is to split a program into two parts and solve each part separately.

Given a rule of the form

$$a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \quad (k, m, n \geq 0)$$

We define the head of the rule, $H(r) = \{a_1, \ldots, a_k\}$, and the body of the rule, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_m\}$ and $B^-(r) = \{c_1, \ldots, c_n\}$. And we define $at(R) = H(r) \cup B^+(r) \cup B^-(r)$ as the set of all atoms occurring in r.

A *splitting set* for a program P is any set U of atoms such that, for every rule $r \in P$, if $H(r) \cap U \neq \emptyset$ then $at(r) \subseteq U$. We call a set of rule $b_U(P) = \{r \in P \mid at(P) \subseteq U\}$ as the *bottom* of P relative to U, and $P \setminus b_U(P)$ is the *top* of P relative to U. Example 3.1 illustrates this concept.

Example 3.1 Consider a program P with the rules

$$\begin{aligned} d &\leftarrow a, c \\ c &\leftarrow \\ a \vee b &\leftarrow \end{aligned}$$

Then $U = \{a, b, c\}$ is a splitting set of this program. The bottom $b_U(P)$ contains the second and the third rule, while the top $P \setminus b_U(P)$ is the first rule.

The idea of splitting set to ease program evaluation is to first evaluate *bottom*, and then based on the answer of the *bottom*, we evaluate *top*. The *bottom* in Example 3.1 has two answer sets, namely $\{a, c\}$ and $\{b, c\}$. The *partial evaluation* of the *top* part of P is defined by function e_U defined next.

Definition 3.2 (Partial Evaluation) *Let U and X be two sets of literal U, X and P be a program. We define a partial evaluation $e_U(P, X)$ as the set of rules $\{r \in P\}$, where*

- *if $B^+(r) \cap U \subseteq X$ and $B^-(r) \cap U \cap X \neq \emptyset$, we replace r with r', where: $H(r') = H(r)$, $B^+(r) = B^+(r) \setminus U$, and $B^-(r') = B^-(r) \setminus U$.*
- *otherwise, remove r*

Example 3.3 Consider program P as in Example 3.1, $U = \{a, b, c\}$ as its splitting set, and $X = \{a, c\}$, then the partial evaluation $e_U(P \setminus b_U(P), \{a, c\})$ is $\{ d \leftarrow \}$.

Next, we recall the splitting set theorem [Lifschitz and Turner, 1994] on top of splitting set, *bottom*, and partial evaluation.

Theorem 3.4 (Splitting Set Theorem [Lifschitz and Turner, 1994]) *Let U be a splitting set for a program P. Then M is an answer set for P iff it can be written as $X \cup Y$ where X is an answer set for $b_U(P)$ and Y is an answer set for $e_U(P \setminus b_U(P), X)$.*

Example 3.5 Consider again program P from Example 3.1, and let $U = \{a, b, c\}$. The *bottom* consists of the second and the third rules and has two answer sets, $\{a, c\}$ and $\{b, c\}$.

- Program $e_U(P \setminus b_U(P), \{a, c\})$ consists of one rule $\{d \leftarrow\}$, and has one answer set, $\{d\}$. Thus, the first answer set of P is $\{a, c, d\}$
- Program $e_U(P \setminus b_U(P), \{b, c\})$ consists of one rule $\{d \leftarrow a, c\}$, and has one answer set, $\emptyset$. Thus the second answer set of P is $\{b, c\}$

3.2 Splitting for input-call-stratified MLPs

Extending the notion of splitting sets to MLP, we observe that it can be done at 2 levels, namely the global and the local splitting. In the following we discuss these two splitting techniques in details.

3.2.1 Global splitting

This section characterizes a class of MLPs named *call-stratified* MLPs. Together with *input-stratified* MLPs that we will introduce in the next section, they admit an efficient top-down evaluation technique. The idea is to restrict cycles on the relevant call graph only to instantiations with empty input. Whenever such cycle occurs, we collect all instantiations forming the cycle and compute the answer sets simultaneously.

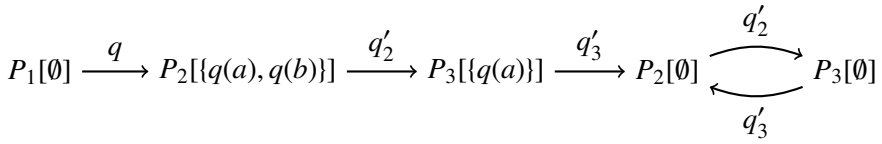


Figure 3.1: $CG_{\mathbf{P}_E}(\mathbf{M})$ from Example 3.7

Definition 3.6 (call stratified MLP [Dao-Tran et al., 2009b]) *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* $P_i[\emptyset]$.

Example 3.7 Consider the MLP $\mathbf{P}_E$ from Example 2.25 and an interpretation $\mathbf{M}$ where

- $M_1/\emptyset = \{q(a), q(b), even\}$,
- $M_2/\{q_2(a), q_2(b)\} = \{q_2(a), q_2(b), q'_2(a), skip_2, even\}$,
- $M_3/\{q_3(a)\} = \{q_3(a), skip_3, odd\}$,
- $M_2/\emptyset = \{even\}$, and
- $M_3/\emptyset = \emptyset$.

We have that the call graph $CG_{\mathbf{P}_E}(\mathbf{M})$ is as depicted in Figure 3.1. Since the loop on $CG_{\mathbf{P}_E}(\mathbf{M})$ contains only $P_2[\emptyset]$ and $P_3[\emptyset]$, $\mathbf{P}_E$ is c-stratified with respect to $\mathbf{M}$.

Before proceeding to an important proposition which gives the idea for performing the top-down evaluation algorithm, we need to define the notion of *answer set of a rule base*.

Definition 3.8 (Answer Set of a Rule Base) *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* $S \neq T$.

Example 3.9 Let us take a look at $\mathbf{P}_E$ and $\mathbf{M}$ from Example 3.7. Based on Definition 3.8, one can see that $M_3/\{q_3(a)\}$ is an answer set of $I_{\mathbf{P}_E}(P_3[\{q_3(a)\}])$ relative to $\mathbf{M}$.

Proposition 3.10 ([Dao-Tran et al., 2009b]) *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* $CG_{\mathbf{P}}(\mathbf{M})$.

On building top down evaluation techniques, cycles in general create problems. However, from Proposition 3.10 one can see that cycles containing instantiation with empty input are safe to be evaluated independently.

3.2.2 Local splitting

Locally inside a module instantiation, the idea of splitting is to prepare input for each module call, i.e., we identify a class of MLPs satisfying that whenever a module call is executed, all of its formal inputs are fully evaluated.

Definition 3.11 (Predicate Definition) *Let* $\mathbf{P}$ *be an MLP, R be a set of ground rules and* α *be a ground module atom of form* $P_k[\mathbf{p}].o(\mathbf{c})$*. For a list of predicate names* $\mathbf{p} = \{p_1, \ldots, p_k\}$*, we define* $def(\mathbf{p}, R) = \{p_\ell(\mathbf{d}) \mid \exists r \in R, p_\ell(\mathbf{d}) \in H(r), p_\ell \in \mathbf{p}\}$.

Definition 3.12 (Splitting Set in MLP) *Let* $\mathbf{P}$ *be an MLP, R be a set of ground rules and* α *be a ground module atom of the form* $P_k[\mathbf{p}].o(\mathbf{c})$*.*

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* $def(\mathbf{p}, R) \subseteq U$*.*

Moreover, for a splitting set U of R, bottom *of R with respect to U is* $b_U(R) = \{r \in R \mid H(r) \cap U \neq \emptyset\}$*, while the* top *of R with respect to U is* $\{R \setminus b_U(R)\}$*.*

Definition 3.13 (Input Splitting Set) *Let* $\mathbf{P}$ *be an MLP, R be a set of ground rules and* α *be a ground module atom of form* $P_k[\mathbf{p}].o(\mathbf{c})$*, and 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* $def(\mathbf{p}, R) \subseteq U$*.*

Example 3.14 Consider $\mathbf{P}_E$ from Example 2.25:

(i) Let $R = I_{\mathbf{P}_E}(P_1[\emptyset])$ and $\alpha = P_2[q].even$ then $U = \{q(a), q(b)\}$ is an input splitting set of R for α. The bottom $b_U(R)$ is

$$\left\{ \begin{array}{lll} q(a) & \leftarrow & \\ q(b) & \leftarrow & \end{array} \right\}.$$

(ii) Let us consider another example where $R = I_{\mathbf{P}_E}(P_2[\{q_2(a), q_2(b)\}])$ and $\alpha = P_3[q_2'].odd$. Then $U = \{q_2(a), q_2(b), q_2'(a), q\}$ is an input splitting set of R for α. The bottom $b_U(R)$ is

$$\left\{ \begin{array}{rcl} q_2(a) & \leftarrow & \\ q_2(b) & \leftarrow & \\ q_2'(a) \vee q_2'(b) & \leftarrow & q_2(a), q_2(b), a \neq b \\ q_2'(b) \vee q_2'(a) & \leftarrow & q_2(b), q_2(a), b \neq a \end{array} \right\}.$$

The theorem below characterizes the splitting set theorem related to an instantiation and its answer set.

Theorem 3.15 ([Dao-Tran et al., 2009b]) *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 3.16 Take R and U from case (ii) in Example 3.14, and $\mathbf{M}$ from Example 3.7.

An answer set of $b_U(R)$ is $N = \{q_2(a), q_2(b), q_2'(a)\}$. On the other hand:

$$R \setminus b_U(R) = \left\{ \begin{array}{lcl} skip_2 & \leftarrow & q_2(a), \text{not } q_2'(a) \\ skip_2 & \leftarrow & q_2(b), \text{not } q_2'(b) \\ even & \leftarrow & \text{not } skip_2 \\ even & \leftarrow & skip_2, P_3[q_2'].odd \end{array} \right\},$$

We have $M_2/\{q_2(a), q_2(b)\}$ is an answer set of $R = I_{\mathbf{P}_E}(P_2[\{q_2(a), q_2(b)\}])$, and also an answer set of $\{R \setminus b_U(R)\} \cup N$.

Next, we define i-stratified MLPs, a class of MLP that could guarantee the existence of input splitting set. We proceed first by introducing the instance dependency graph over an instantiation.

Definition 3.17 (Instance Dependency Graph) *Let* $\mathbf{P}$ *be an MLP and* $\mathbf{M}$ *is an interpretation of* $\mathbf{P}$*. The instance dependency graph of* $\mathbf{P}$ *is the digraph* $G_{\mathbf{P}}^{\mathbf{M}} = (IV, IE)$*, where*

- *The vertex set IV contains of pairs of the form* $(p, P_i[S])$ *or* $(\alpha, P_i[S])$*, where p is a predicate name and* α *is a module atom, appearing in module* m_i*, and S is the input for a value call* $P_i[S\,] \in VC(\mathbf{P})$*.*
- *The edge set IE is define as follows: Let* $r \in R(m_i)$ *and* $\alpha = P_j[\mathbf{p}].o(\mathbf{t}) \in B(R(m_i))$ *, then:*

  (i) $(v, P_i[S]) \rightarrow^{\star} (w, P_i[S])$*, if* $v(\mathbf{t_1}) \in H(r)$ *and* $w(\mathbf{t_2}) \in B^{\star}(r)$

  (ii) $(v, P_i[S]) \rightarrow^{\vee} (w, P_i[S])$*, if* $v(\mathbf{t_1}), w(\mathbf{t_2}) \in H(r)$

  (iii) $(v, P_i[S]) \rightarrow^{\star} (w, P_i[S])$*, if* $v(\mathbf{t_1}) \in H(r)$ *and w is a module atom in* $B^{\star}(r)$*.*

  (iv) $(\alpha, P_i[S]) \rightarrow^{in} (p_\ell, P_i[S])$*, for every* $p_\ell \in \mathbf{p}$ *of* α

  (v) $(\alpha, P_i[S]) \rightarrow^{m} (o, P_j[(M_i/S)|_{\mathbf{p}}^{\mathbf{q_j}}])$

For any module atoms α_1, α_2*, we say that* α_1 *locally depends on* α_2*, if* $\alpha_1 \rightsquigarrow \alpha_2$*, where* $\rightsquigarrow = \rightarrow^{+} \cup \rightarrow^{-} \cup \rightarrow^{\vee} \cup \rightarrow^{in}$*. Then, we define an instance local labelling function* $ill_i : IV \rightarrow \mathbb{N}$ *such that* $ill_i(\alpha_1, P_i[S]) > ill_i(\alpha_2, P_i[S])$ *if* $(\alpha_1, P_i[S]) \rightsquigarrow (\alpha_2, P_i[S])$*.*

Instance dependency graphs capture the dependency relationship between each predicate in an instantiation, a module atom with its input predicates (intuitively, a module atom and its input provider) and a module atom with its output predicates. Finally, we define a notion of input stratified MLPs on top of instance dependency graphs.

Definition 3.18 (Input Stratified MLP [Dao-Tran et al., 2009b]) *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$*. We say that* $\mathbf{P}$ *is* i-stratified (input stratified) w.r.t. $\mathbf{M}$*, iff cycles with in-edges in* $G_{\mathbf{P}}^{\mathbf{M}}$ *contain only nodes of the form* $(X, P_i[\emptyset])$*.*

The definition below characterizes the class of ic-stratified MLPs. The top down evaluation algorithm described in the next section is devised for this class.

Definition 3.19 (Input and Call Stratified MLP [Dao-Tran et al., 2009b]) *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$. *We say that* $\mathbf{P}$ *is* ic-stratified (input and call stratified) w.r.t. $\mathbf{M}$ *iff it is both i-stratified and c-stratified w.r.t.* $\mathbf{M}$.

The existence of an input splitting set for a module atom in an instantiation is important to evaluate the module calls. Within ic-stratified MLP, this condition is guaranteed.

Theorem 3.20 ([Dao-Tran et al., 2009b]) *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.

3.3 Top-Down Evaluation Algorithm

This section briefly discusses a basic top-down evaluation algorithm for ic-stratified MLPs presented in [Dao-Tran et al., 2009b]. Procedure *comp* in Algorithm 3.1 has 6 parameters:

- $\mathbf{P}$: an MLP that is being evaluated.
- C: a set of value calls.
- *path*: a list of sets of value calls. *path* stores the call chain from the main module to the current set value calls evaluated.
- $\mathbf{M}$: an indexed set of interpretations. The interpretation for each instantiation in the call graph is computed gradually by the algorithm, and finally we will have in $\mathbf{M}$ each M_i/S as an answer set of $I_{\mathbf{P}}(P_i[S])$ where $P_i[S] \in CG_{\mathbf{P}}(\mathbf{M})$.
- $\mathbf{A}$: an indexed set of sets of module atom A_i/S which contains all module atoms that have been solved so far in instantiation $P_i[S]$. In the end of the execution of the algorithm, A_i/S shall contain all module atom in $P_i[S]$, which means that all module atoms have been solved.
- $\mathcal{AS}$: set of answer set (if any) of $\mathbf{P}$, as the final result of the algorithm.

In order to solve an MLP $\mathbf{P}$ from a main module $P_1[\emptyset]$, one can call *comp* by specifying $C = \{P_1[\emptyset]\}$, $path = \epsilon$, $\mathbf{M}$ and $\mathbf{A}$ are *nil* at all elements, and $\mathcal{AS} = \emptyset$.

In the process, *comp* makes use of several helper methods:

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.

Algorithm 3.1: *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 $\mathcal{AS}$

1 **if** $\exists P_i[S] \in C$ *s.t.* $P_i[S] \in C_{prev}$ *for some* $C_{prev} \in path$ **then**
2 **if** $S \neq \emptyset$ *for some* $P_i[S\,] \in C$ **then return**
3 **repeat**
4 $C' := tail(path)$ and remove the last element of *path*
5 **if** $\exists P_j[T\,] \in C'$ *s.t.* $T \neq \emptyset$ **then return else** $C := C \cup C'$
6 **until** $C' = C_{prev}$
7 $R := rewrite(C, \mathbf{M}, \mathbf{A})$
8 **if** *R is ordinary* **then**
9 **if** *path is empty* **then**
10 **forall the** $N \in ans(R)$ **do** $\mathcal{AS} := \mathcal{AS} \cup \{\mathbf{M} \uplus mlpize(N, C)\}$
11 **else**
12 $C' := tail(path)$ and remove the last element of *path*
13 **forall the** $P_i[S\,] \in C$ **do** $A_i/S := fin$
14 **forall the** $N \in ans(R)$ **do** $comp(\mathbf{P}, C', path, \mathbf{M} \uplus mlpize(N, C), \mathbf{A}, \mathcal{AS})$
15 **else**
16 pick an $\alpha := P_j[\mathbf{p}].o(\mathbf{c})$ in R with smallest $ill_R(\alpha)$ and find splitting set U of R for α
17 **forall the** $P_i[S\,] \in C$ **do if** $A_i/S = nil$ **then** $A_i/S := \{\alpha\}$ **else** $A_i/S := A_i/S \cup \{\alpha\}$
18 **forall the** $N \in ans(b_U(R))$ **do**
19 $T := N|^{\mathbf{q_j}}_{\mathbf{p}}$
20 **if** $(M_j/T \neq nil) \wedge (A_j/T = fin)$ **then** $C' := C$ and $path' := path$
21 **else** $C' := \{P_j[T]\}$ and $path' := append(path, C)$
22 $comp(\mathbf{P}, C', path', \mathbf{M} \uplus mlpize(N, C), \mathbf{A}, \mathcal{AS})$

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 $\alpha = P_j[\mathbf{p}].o(\mathbf{t})$ in R, such that $\alpha \in A_i/S$, by o prefixed with $P_j[T]$, where $T = (M_i/S)|^{\mathbf{q_j}}_{\mathbf{p_i}}$, and $\mathbf{p_i}$ is $\mathbf{p}$ without prefixes; moreover add any atoms from $(M_j/T)|_o$ prefixed by $P_j[T]$ to R.

To understand the algorithm, we divide the discussion into 3 parts:

1. Preliminary (line 1 – 7). In this part, the algorithm verifies whether the MLP is ic-stratified. There are two possibilities:

   a) If we have no cycle in the value call chain, then the algorithm will run without once executing line 2 – 6.

   b) However, if such a cycle occurs (condition in line 1 is satisfied), then we continue only if all value calls in the cycle has $\emptyset$ as input. Then, those instantiations are collected together in C (line 3 – 6).

Then, in line 7 we rewrite all value calls in the current C. Recall that the instantiations collected in *rewrite* can be from more than one value call only if we found a cycle in the chain of value calls. Otherwise, we have an instantiation from one value call only in C.

2. Ordinary evaluation (line 8 – 14). The result of *rewrite*, R, is ordinary only if all module atoms in the instantiation of all value calls in C are solved. Then:

   - if *path* is empty, then the main module is reached. $\mathbf{M}$ is updated with the answer sets of R (see $\mathbf{M} \uplus mlpize(N, C)$ in line 10, where N is an answer set of R). At this point, $\mathbf{M}$ has become a full answer set of $\mathbf{P}$ and is added to $\mathcal{AS}$.
   - if *path* is not empty, the algorithm continues the computation to the set of value calls from $tail(path)$ which is the parent of the current C (line 12). Since all module atoms have been solved, we mark the current instantiation with *finish* (line 13, represented as a special element *fin*), update our partial model $\mathbf{M}$ with the current result, and call *comp* recursively (line 14).

3. Module atom evaluation (line 15 – 22). The result of *rewrite*, R, is not ordinary. It means that there is still at least a module call that has to be computed. There are 3 main things happen here:

   a) A module atom α which does not depend on other module atom is picked (line 16).

   b) Then α is added to A_i/S where $P_i[S] \in C$ (line 17), which means that α is going to be processed. This information will be used later in further computation, in *rewrite* procedure (where we replace α with its output atom when $\alpha \in A_i/S$ – which means that we have processed α).

   c) From the answer set of the input splitting set, we create a module call of α, $P_j[T]$ (line 18 – 22). There are two possibilities:

      - If the next module call has been solved (denoted by $A_j/T = fin$), then we do not need to proceed further. Instead we stay at the current C and *path*.
      - Otherwise, we proceed further. We add the C to *path*, to keep track the call chain. And we make a call to *comp* to solve the module call P_j/T. $\mathbf{M}$ is updated with the answer set of the bottom.

Example 3.21 We take the main idea from Example 10 in [Dao-Tran et al., 2009b]. Let us consider Algorithm 3.1 on $\mathbf{P}_E$ from Example 2.25 and a current set of value calls $C = \{P_2[\emptyset]\}$, and $path = \{P_1[\emptyset]\}, \{P_2[\{q_2(a), q_2(b)\}]\}, \{P_3[\{q_3(a)\}]\}, \{P_2[\emptyset]\}, \{P_3[\emptyset]\}$. See Figure 3.1 for the illustration of the chain of value calls. Note that since in the current set of value calls we have $P_2[\emptyset]$, and at this point we already have an element in *path* which contains $P_2[\emptyset]$, the condition in line 1 is satisfied. Next, we collect the last two element of *path*, and join them into together in C. Then $C = \{P_2[\emptyset], P_3[\emptyset]\}$. For clarity, we rename predicate q'_2 and q'_3 to r_2 and r_3 respectively and we use superscripts instead of prefixes. The rewriting R w.r.t. C is

$$\left\{\begin{array}{rcl} r_2^{P_2[\emptyset]}(X) \vee r_2^{P_2[\emptyset]}(Y) & \leftarrow & q_2^{P_2[\emptyset]}(X), q_2^{P_2[\emptyset]}(Y), X \neq Y \\ skip_2^{P_2[\emptyset]} & \leftarrow & q_2^{P_2[\emptyset]}(X), \text{not } r_2^{P_2[\emptyset]}(X) \\ even^{P_2[\emptyset]} & \leftarrow & \text{not } skip_2^{P_2[\emptyset]} \\ even^{P_2[\emptyset]} & \leftarrow & skip_2^{P_2[\emptyset]}, odd^{P_3[\emptyset]} \\ r_3(X)^{P_3[\emptyset]} \vee r_3^{P_3[\emptyset]}(Y) & \leftarrow & q_3^{P_3[\emptyset]}(X), q_3^{P_3[\emptyset]}(Y), X \neq Y \\ skip_3^{P_3[\emptyset]} & \leftarrow & q_3^{P_3[\emptyset]}(X), \text{not } r_3^{P_3[\emptyset]}(X) \\ odd^{P_3[\emptyset]} & \leftarrow & skip_3^{P_3[\emptyset]}, even^{P_2[\emptyset]} \end{array}\right\}$$

The only answer set of R is $\{even^{P_2[\emptyset]}\}$, which is in turn placed in $M_2/\emptyset$ as *even*.

On the way back, when evaluating $C = \{P_3[q_3(a)]\}$, after rewrite we will have $odd^{P_3[q_3(a)]}$ *true*, since $skip_3$ is true and $even^{P_2[\emptyset]}$ is true. The result of the rewrite is as the following:

$$\left\{\begin{array}{rcl} q_3^{P_3[q_3(a)]}(a) & \leftarrow & \\ r_3^{P_3[q_3(a)]}(X) \vee r_3^{P_3[q_3(a)]}(Y) & \leftarrow & q_3^{P_3[q_3(a)]}(X), q_3^{P_3[q_3(a)]}(Y), X \neq Y \\ skip_3^{P_3[q_3(a)]} & \leftarrow & q_3^{P_3[q_3(a)]}(X), \text{not } r_3^{P_3[q_3(a)]}(X) \\ odd^{P_3[q_3(a)]} & \leftarrow & skip_3^{P_3[q_3(a)]}, even^{P_2[\emptyset]} \\ even^{P_2[\emptyset]} & \leftarrow & \end{array}\right\}$$

And we have $M_3/\{q_3(a)\} = \{q_3(a), skip_3, odd\}$.

Similarly, when we evaluate $\{P_2[\{q_2(a), q_2(b)\}]\}$ (on the way up after evaluating $P_3[q_3(a)]$), we have $P_2[\{q_2(a), q_2(b)\}]$___*even*, and $M_2/\{q_2(a), q_2(b)\} = \{q_2(a), q_2(b), r_2(a), skip_2, even\}$. In the end, back to $P_1[\emptyset]$, we have $even^{P_1[\emptyset]}$, and $M_1/\emptyset = \{q(a), q(b), even\}$. $\mathbf{M}$ is added to $\mathcal{AS}$.

Continuing with the call chain $P_1[\emptyset] \xrightarrow{q} P_2[\{q(a), q(b)\}] \xrightarrow{r_2} P_3[\{r_2(b)\}] \xrightarrow{r_3} P_2[\emptyset] \rightarrow \cdots$, (as we find the the second answer set from line 18 when evaluating $C = \{P_2[\{q(a), q(b)\}]\}$), *comp* returns another answer set of $\mathbf{P}$ (disregarding irrelevant module instances, i.e., $M_i/S = nil$ iff $P_i[S] \notin V(CG_{\mathbf{P}}(\mathbf{M}))$).

Finally, $\mathcal{AS} =$

$$\left\{\begin{array}{l} \left\{\begin{array}{rcl} M_1/\emptyset & = & \{q(a), q(b), even\}, \\ M_2/\{q_2(a), q_2(b)\} & = & \{q_2(a), q_2(b), r_2(a), skip_2, even\}, \\ M_3/\{q_3(a)\} & = & \{q_3(a), skip_3, odd\}, \\ M_2/\emptyset & = & \{even\}, \\ M_3/\emptyset & = & \emptyset \end{array}\right\}, \\ \left\{\begin{array}{rcl} M_1/\emptyset & = & \{q(a), q(b), even\}, \\ M_2/\{q_2(a), q_2(b)\} & = & \{q_2(a), q_2(b), r_2(b), skip_2, even\}, \\ M_3/\{q_3(b)\} & = & \{q_3(b), skip_3, odd\}, \\ M_2/\emptyset & = & \{even\}, \\ M_3/\emptyset & = & \emptyset \end{array}\right\} \end{array}\right\}.$$

CHAPTER 4

Instantiation Splitting for Input-Call-Stratified MLPs

This chapter explains about instantiation splitting optimization technique. Instantiation splitting is particularly useful to reduce the work delivered to the ASP solver when computing the answer sets of an instantiation, hence helps to improve the overall performance. Section 4.1 gives the intuition behind this technique. We outline the main idea and provide an example on how the instantiation splitting works and give its potential advantages. Section 4.2 presents the formal definition.

4.1 Intuition

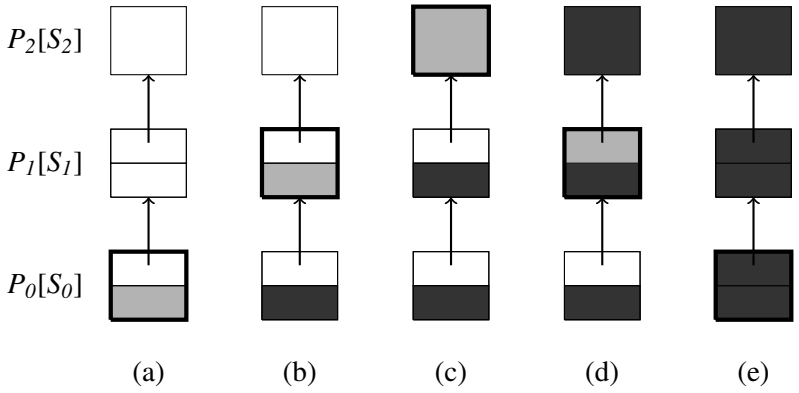
We offer an optimization technique that maximizes the use of our partial model **M** (as in Algorithm 3.1) and splitting set theorem.

Suppose that we have an MLP **P** which consists of three modules: one main module P_0 and two library modules P_1 and P_2. P_0 has a module call to P_1 and P_1 has a module call to P_2. Example 4.1 and Figure 4.1 gives an illustration. This is a simple example whose main purpose is to give an intuition about the idea of instantiation splitting.

Example 4.1 Consider an MLP $\mathbf{P} = (m_0, m_1, m_2)$, where $m_0 = (P_0, R_0)$ is a main modules, while $m_1 = (P_1[q_1], R_1)$, and $m_2 = (P_2[q_2], R_2)$ are library modules, where:

$$R_0 = \left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ q_0(b) & \leftarrow & q_0(a) \\ out_0 & \leftarrow & q_0(b), P_1[q_0].out_1 \end{array} \right\},$$

$$R_1 = \left\{ \begin{array}{rcl} r_1(a) & \leftarrow & \\ r_1(b) & \leftarrow & \\ p_1(X) & \leftarrow & q_1(X), r_1(X) \\ out_1 & \leftarrow & P_2[p_1].out_2 \end{array} \right\}, \text{ and}$$



bold outline: processed instantiation
gray: currently being evaluated
black: deleted

Figure 4.1: Module call instantiation with instantiation splitting

$$R_2 = \left\{ \; out_2 \quad \leftarrow \quad q_2(a) \; \right\}.$$

Intuitively, P_0 passed a set q_0 to P_1. Then, P_1 filters the input, picks only elements that matches with set r_1 and put the result in set p_1. Finally, P_2 concludes out_2 only if the input contain an element a.

Figure 4.1 depicts the evaluation process in 5 steps, from (a) to (e). The input with respect to the displayed instantiations are $S_0 = \emptyset$, $S_1 = \{q_1(a), q_1(b)\}$, and $S_2 = \{q_1(a), q_1(b)\}$. Gray represents a part that is being processed while black represents a part that has been deleted. Let $\mathbf{M}^b$ be an indexed set of ordinary atoms $(M_i^b/S \subseteq HB_\mathbf{P} \mid P_i[S] \in \mathbf{P})$, where each M_i^b/S is $\emptyset$.

First, in step (a), we divide the instantiation $P_0[S_0]$ (represented by the rectangle in the bottom) into two parts: the lower part that represents *bottom* (contain only ordinary rules), and the upper part that represent *top* (that contain a module call to P_1). Take the MLP in Example 4.1, the ordinary part (*bottom*) will be

$$\left\{ \begin{array}{lll} q_0(a) & \leftarrow & \\ q_0(b) & \leftarrow & q_0(a) \end{array} \right\}$$

and the non-ordinary part (*top*) will be $\{out_0 \leftarrow q_0(b), P_1[q_0].out_1\}$. Because the *bottom* contains only ordinary rules, we can solve it right away (colored with gray), add the result to $M_0^b/\emptyset$, and forget it (delete the *bottom* from the instantiation).

In step (b), we can see that the lower part is colored with black (recall that black means it has been deleted. We add $\{q_0(a), q_0(b)\}$ to $M_0^b/\emptyset$ as the answer of $\{q_0(a) \leftarrow, q_0(b) \leftarrow q_0(a)\}$. The idea is to forget $\{q_0(a) \leftarrow, q_0(b) \leftarrow q_0(a)\}$, and only $\{out_0 \leftarrow q_0(b), P_1[q_0].out_1\}$ left to be processed in the instantiation of $P_0[S_0]$. Next, we proceed to call $P_1[S_1]$. Since in $P_1[S_1]$ we have a module call to P_2, we divide it into *bottom* and *top*, solved *bottom*, add the result to M_1^b/S_1, and then delete the *bottom* rules. In this case, the *bottom* is

Table 4.1: Partial interpretation

$M_0^b/\emptyset$	$\{q_0(a), q_0(b)\}$
M_1^b/S_1	$\{q_1(a), q_1(b), r_1(a), r_1(b), p_1(a), p_1(b)\}$
M_2^b/S_2	$\{q_2(a), q_2(b), out_2\}$

Table 4.2: Top after solving $P_2[S_2]$

$P_0[\emptyset]$	$\{out_0 \leftarrow q_0(b), P_1[q_0].out_1\}$
$P_1[S_1]$	$\{out_1 \leftarrow P_2[p_1].out_2\}$
$P_2[S_2]$	$\emptyset$

Table 4.3: Answer set of **P**

$M_0^b/\emptyset$	$\{q_0(a), q_0(b), out_0\}$
M_1^b/S_1	$\{q_1(a), q_1(b), r_1(a), r_1(b), p_1(a), p_1(b), out_1\}$
M_2^b/S_2	$\{q_2(a), q_2(b), out_2\}$

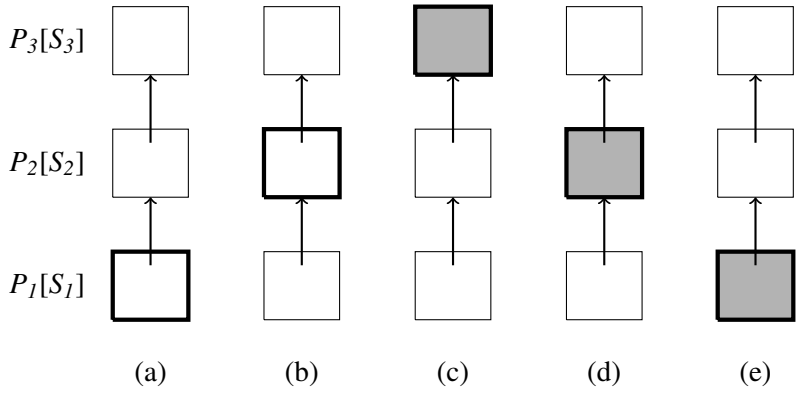
$$\left\{ \begin{array}{lll} r_1(a) & \leftarrow & \\ r_1(b) & \leftarrow & \\ p_1(X) & \leftarrow & q_1(X), r_1(X) \end{array} \right\},$$

and the *top* will be $\{out_1 \leftarrow P_2[p_1].out_2\}$. We have $M_1^b/S_1 = \{q_1(a), q_1(b), r_1(a), r_1(b), p_1(a), p_1(b)\}$ as the answer set of the *bottom*.

In step (c), we have that the instantiation $P_2[S_2]$ contains only ordinary rules, thus we can solve it in one go. We add the result, $\{q_2(a), q_2(b), out_2\}$, to M_2^b/S_2. Table 4.1 summarizes the partial interpretation we have so far, while from Table 4.2 one could already see that a certain number of rules had already been removed compared to the original instantiation. This illustrates the main advantage of instantiation splitting optimization, i.e. to have less and less rules during the computation.

In step (d), after solving $P_2[S_2]$, we go back to $P_1[S_1]$ to solve the *top* part. Our optimization starts to play a role here. Compare to the orginal approach in [Dao-Tran et al., 2009b] that requires to evaluate all rules in $P_1[S_1]$ (see Figure 4.2 (d)), we only need to evaluate the *top* part, that is $\{out_1 \leftarrow P_2[p_1].out_2\}$ (since the *bottom* has already been deleted), only one rule to evaluate compared to the 4 original rules. This optimization technique is particularly useful when we have a big ordinary (*bottom*) part, or when we have several module calls in one instantiation. We add the evaluation result of this *top* part, $\{out_1\}$, into M_1^b/S_1. Combined with the result from the *bottom* part (that has been previously stored), M_1^b/S_1 currently is the full result of $P_1[S_1]$.

In the end, step (e), similarly we form the full result for M_0^b/S_0. We solved the *top* part, add the result $\{out_0\}$ to M_0^b/S_0. This ends our computation and the answer set is shown in Table 4.3.



bold outline: processed instantiation
grey: currently evaluated

Figure 4.2: Module call instantiation without instantiation splitting

4.2 Instantiation Splitting

Next, we formally define splitting sets at the instantiation level and partial interpretations with respect to instantiation splitting sets. Let $mod(r)$ be a set of module atoms in a rule r and $mod(R) = \bigcup_{r \in R} mod(r)$ be a set of all module atoms in a rule set R.

Definition 4.2 (Subordinate Module Atoms Set) *A* subordinate module atoms set *of a ground rule set R is a set $A \subseteq mod(R)$ such that for any $\alpha_1, \alpha_2 \in mod(R)$, if $\alpha_1 \in A$ and $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \in A$ (recall Definition 3.17 for "$\rightsquigarrow$").*

Intuitively, a subordinate module atoms set can be used to capture the chain of solved module atoms. Suppose that A is a subordinate module atoms set for a ground rule set R, and α and β are in R. If α and β do not depend to each other or any other module atoms, then one can freely include them into A. However, if for instance α depends on β ($\alpha \rightsquigarrow \beta$) then whenever we include α into A, we also have to include β. This is similar as in our computation that we have to compute a module call for β before we can compute a module call for α.

In the sequel, we will define an instantiation splitting set with respect to a subordinate module atoms set. This allows us to select the size of a splitting set of an instantiation with respect to a particular subordinate module atoms set.

Definition 4.3 (Instantiation Splitting Sets) *Let* **P** *be an MLP, and A be a subordinate module atoms set for $I_{gr(\mathbf{P})}(P_i[S])$, where $P_i[S] \in VC(\mathbf{P})$. An* instantiation splitting set *of $P_i[S]$ with respect to A is a set $U(A) \subseteq at(I_{gr(\mathbf{P})}(P_i[S]))$, where*

(i) $U(A)$ is a splitting set for $I_{gr(\mathbf{P})}(P_i[S])$, and

(ii) $U(A) \cap mod(I_{gr(\mathbf{P})}(P_i[S])) = A$

Moreover, if $A = mod(I_{gr(\mathbf{P})}(P_i[S]))$*, we define* $U(A) = at(I_{gr(\mathbf{P})}(P_i[S]))$*. As usual, we define the* bottom *of* $I_{gr(\mathbf{P})}(P_i[S])$ *with respect to* $U(A)$ *is* $b_{U(A)}(I_{gr(\mathbf{P})}(P_i[S])) = \{r \in I_{gr(\mathbf{P})}(P_i[S]) \mid H(r) \cap U \neq \emptyset\}$*,*

Next, we lift the notion of a particular module instantiation to MLPs.

Definition 4.4 (Instantiation Splitting Sets for MLPs) *Let* $\mathbf{P}$ *be an MLP. We define a subordinate module atoms set for* $\mathbf{P}$ *as an indexed tuple of subordinate module atoms set* $\mathbf{A} = (A_i/S \mid P_i[S] \in VC(\mathbf{P}))$*. Furthermore, we define an instantiation splitting set of* $\mathbf{P}$ *with respect to* $\mathbf{A}$ *as an indexed tuple of instantiation splitting set* $\mathbf{U_P(A)} = (U_i/S(A_i/S) \mid P_i[S] \in VC(\mathbf{P}))$*, where each* $U_i/S(A_i/S)$ *is an instantiation splitting set of* $P_i[S]$ *with respect to* A_i/S*.*

Example 4.5 give an idea about the notion of *instantiation splitting set*.

Example 4.5 Consider an MLP $\mathbf{P} = (m_1, m_2)$ where $m_1 = (P_1, R_1)$, $m_2 = (P_2[p], R_2)$, and

$$R_1 = \left\{ \begin{array}{lll} a & \leftarrow & \\ b & \leftarrow & a \\ c & \leftarrow & b, P_2[b].c \end{array} \right\}$$

$$R_2 = \left\{ \; c \leftarrow p \; \right\}$$

We have $VC(\mathbf{P}) = \{P_1[\emptyset], P_2[\emptyset], P_2[\{b\}]\}$. Suppose that we have $\mathbf{A} = (A_1/\emptyset, A_2/\emptyset, A_2/\{b\})$, where $A_1/\emptyset = \emptyset$, $A_2/\emptyset = \emptyset$, and $A_2/\{b\} = \emptyset$. Then, $\mathbf{U_P(A)} = (U_1/\emptyset, U_2/\emptyset, U_2/\{b\})$, where $U_1/\emptyset = \{a, b\}$, $U_2/\emptyset = \{c, p\}$, $U_2/\{b\} = \{c, p\}$.

On another case when we have $A_1/\emptyset = \{P_2[b].c\}$ (other elements of $\mathbf{A}$ are unchanged), then we will have $U_1/\emptyset = \{a, b, c, P_2[b].c\}$ (with other elements of $\mathbf{U_P(A)}$ remain the same as before).

Next, we define a partial evaluation for an occurence of a module atom.

Definition 4.6 (Partial Evaluation) *Let* $\mathbf{P}$ *be an MLP,* $\mathbf{M}$ *be an interpretation of* $\mathbf{P}$*, and* R *be a set of ground rules. We define a* partial evaluation *of* R *with respect to* $\mathbf{P}$*,* $\mathbf{M}$*, and* $U_i/S(A_i/S)$ *where* $P_i[S] \in VC(\mathbf{P})$*, denoted* $e^{\mathbf{P},\mathbf{M}}_{U_i/S(A_i/S)}(R, X)$*, as a set of rules* $\{r \in R\}$ *where:*

1. *Let* a *be an ordinary atom in* $B(r)$ *and* $a \in U_i/S(A_i/S)$*, then*
   - $a \in B^+(r)$*; if* $a \in X$*, remove* a *from* $B^+(r)$*, otherwise remove* r *from the resulting set.*
   - $a \in B^-(r)$*; if* $a \notin X$*, remove* a *from* $B^-(r)$*, otherwise remove* r *from the resulting set.*
2. *Let* $\alpha = P_j[\mathbf{p}].o(\mathbf{c})$ *be a module atom in* $B(r)$*,* $T = X|^{\mathbf{q_j}}_{\mathbf{p}}$*, and* $\mathbf{q_j}$ *is the formal input of module* P_j*.*
   - $\alpha \in B^+(r)$*; if* $o(\mathbf{c}) \in M_j/T$*, remove* α *from* $B^+(r)$*, otherwise remove* r *from the resulting set.*
   - $\alpha \in B^-(r)$*; if* $o(\mathbf{c}) \notin M_j/T$*, remove* α *from* $B^-(r)$*, otherwise remove* r *from the resulting set.*

Definition 4.6 extends the partial evaluation from [Lifschitz and Turner, 1994] with the evaluation of module atoms. Intuitively, when the truth value of a module atom α in a rule r is satisfied, we remove α from r. Otherwise, we remove r (because the body is not satisfied).

Let $\mathbf{P}$ be an MLP and $P_i[S] \in VC(\mathbf{P})$. In general, when applying our instantiation splitting optimization to an instantiation $I_{gr(\mathbf{P})}(P_i[S])$ which contains a module call, we apply the following steps:

1. Split it into two parts: *bottom* (ordinary part), and *top* (non-ordinary part).
2. Solve *bottom*, store the result into our incremental interpretation M_i^b/S.
3. Next, in order to evaluate *top*, we recursively solved the remaining module calls one by one and store each module call instantiation that has been solved into A_i/S.
4. Back from evaluating each module call instantiation, instead of evaluating $I_{gr(\mathbf{P})}(P_i[S])$, we evaluate only the rules from $I_{gr(\mathbf{P})}(P_i[S])$ that we have not solved before (the rules from the instantiations without the *bottom* part).

Since we always partition our computation into two parts (*bottom* and *top*), Proposition 4.7 make sure that we always compute towards an answer set, i.e., an answer set of the *bottom* so far combined with the answer set of the *top* part (that is not evaluated yet) will be an answer set of the instantiation.

Proposition 4.7 *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$*, and* $R = I_{gr(\mathbf{P})}(P_i[S])$ *for* $P_i[S] \in VC(\mathbf{P})$*. Then* $X \cup Y$ *is an answer set of* R *relative to* $\mathbf{M}$*, iff* X *is an answer set of* $b_{U_i/S(A_i/S)}(R)$ *and* Y *is an answer set of* $e^{\mathbf{P},\mathbf{M}}_{U_i/S(A_i/S)}(R \setminus b_{U_i/S(A_i/S)}(R), X)$ *relative to* $\mathbf{M}$*.*

Proof. Let R be an instantiation $I_{gr(\mathbf{P})}(P_i[S])$, where $P_i[S] \in VC(\mathbf{P})$. Recall that $U_i/S(A_i/S)$ is an instantiation splitting set for R with respect to A_i/S, and $b_{U_i/S(A_i/S)}(R)$ is the *bottom* of R with respect to $U_i/S(A_i/S)$. The proof is akin to Splitting set theorem in [Lifschitz and Turner, 1994]. In addition, the rules resulting from the partial evaluation are always ordinary. From Definition 4.6 we know that if a rule r contain a module atom α, then either α is removed (if $\alpha \in B^+(r)$ and its output atom is in the model, or $\alpha \in B^-(r)$ and its output atom is not in the model) or the rule is deleted. □

Proposition 4.8 make sure that, in the end, we compute an answer set of an instantiation.

Proposition 4.8 *Let* $\mathbf{M}$ *be an interpretation of an MLP* $\mathbf{P}$*,* $\mathbf{A}$ *be a subordinate module atoms set of* $\mathbf{P}$*, and* $\mathbf{U}_\mathbf{P}(\mathbf{A})$ *be a instantiation splitting set of* $\mathbf{P}$ *with respect to* $\mathbf{A}$*. In addition, let* M_i/S *be an answer set of* $I_{gr(\mathbf{P})}(P_i[S])$ *relative to* $\mathbf{M}$ *for* $P_i[S] \in VC(\mathbf{P})$*. If* $A_i/S = mod(I_{gr(\mathbf{P})}(P_i[S]))$ *and* X *is an answer set of* $b_{U_i/S(A_i/S)}(I_{gr(\mathbf{P})}(P_i[S]))$ *relative to* $\mathbf{M}$ *then* $X = M_i/S$*.*

Proof. We need to show that X is an answer set of $I_{gr(\mathbf{P})}(P_i[S]))$ relative to $\mathbf{M}$. Since $A_i/S = mod(I_{gr(\mathbf{P})}(P_i[S]))$, by definition $U_i/S(A_i/S) = at(I_{gr(\mathbf{P})}(P_i[S]))$.

Since $U_i/S(A_i/S) = at(I_{gr(\mathbf{P})}(P_i[S]))$, for each rule $r \in I_{gr(\mathbf{P})}(P_i[S])$, we have $H(r) \cap U_i/S(A_i/S) \neq \emptyset$. From this fact, we get $b_{U_i/S(A_i/S)}(I_{gr(\mathbf{P})}(P_i[S])) = I_{gr(\mathbf{P})}(P_i[S]))$. Because we have X is an answer set of $b_{U_i/S(A_i/S)}(I_{gr(\mathbf{P})}(P_i[S]))$ relative to $\mathbf{M}$, then X is also an answer set of $I_{gr(\mathbf{P})}(P_i[S]))$ relative to $\mathbf{M}$. □

CHAPTER 5

Evaluating Input-Call-Stratified MLPs with Instantiation Splitting

The authors of [Dao-Tran et al., 2009b], presented a top-down algorithm to evaluate a subclass of MLPs, namely input-call (ic) stratified MLPs. Intuitively, ic-stratified MLPs restrict MLPs in a way that the inputs for a module call are always prepared before the call is made and a cycle between calls can occur only between instantiations with empty input.

The new algorithm *solveMLP* presented in this chapter takes the algorithm in [Dao-Tran et al., 2009b] as the basis and then optimizes it by introducing our instantiation splitting technique.

5.1 Evaluation Algorithm

The main difference of *solveMLP* and the algorithm in [Dao-Tran et al., 2009b] (see Algorithm 3.1 for details) is that we incorporate instantiation splitting optimization into our evaluation algorithm. Parameters used in *solveMLP* are almost the same as paramaters used in Algorithm 3.1, except for **U** (see Algorithm 5.1 for the details of *solveMLP*). In this case, **U** represents (and used as) the instantiation splitting set (recall Definition 4.4).

Algorithm 5.1 proceeds in three parts as in Algorithm 3.1.

1. In the *preliminary* part, we make sure that the input program is c-stratified, and collect the instantiation if a cycle of value call occurs (see Algorithm 5.2). Next, we perform rules rewriting. The *rewrite* procedure in Algorithm 5.3 works on ground rule bases and applying partial evaluation akin to [Lifschitz and Turner, 1994] to the output atom of a module atom that has been solved.

2. In the *ordinary evaluation part*, *solveMLP* proceeds almost the same as Algorithm 3.1, except that in *solveMLP* we also pass the instantiation splitting set **U** in the recursion.

Algorithm 5.1: *solveMLP*(in: $\mathbf{P}, C, path, \mathbf{M}, \mathbf{A}, \mathbf{U}$, 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 $\mathbf{A}$, instantiation splitting set $\mathbf{U}$, set of answer sets $\mathcal{AS}$

Output: set of answer sets $\mathcal{AS}$

```
1  (C, path) := stratificationCheck(C, path)
2  R := rewrite(P, C, M, A, U)
3  if R is ordinary then                                   /*ordinary evaluation*/
4      if path = ϵ then
5          foreach N ∈ ans(R) do
6              AS := AS ∪ {M ⊎ mlpize(N, C)}               /*collect answer sets*/
7      else
8          forall the P_i[S] ∈ C do A_i/S := fin            /*mark as finished*/
9          C' := tail(path) and remove the last element of path
10         foreach N ∈ ans(R) do
11             M' := M ⊎ mlpize(N, C)
12             solveMLP(P, C', path, M', A, U, AS)
13 else                                                    /*module atom evaluation*/
14     (A, U, α, U) := prepareSplittingSet(R, C, A, U)
15     foreach N ∈ ans(b_U(R)) do
16         (C', path') := prepareNextCall(α, C, path, N, A, U)
17         M' := M ⊎ mlpize(N, C)
18         solveMLP(P, C', path', M', A, U, AS)
```

3. In the *module atom evaluation* part, we look for a module atom α that does not depend on another module atom and prepare an input splitting set for α. If such module atom or input splitting set does not exist, the evaluation is stopped (because the input program is not i-stratified). In addition to update $\mathbf{A}$, we also update the instantiation splitting set $\mathbf{U}$ in this part (see Algorithm 5.4). Then, we prepare the next module call (see Algorithm 5.5). Another call to *solveMLP* is made for further evaluation.

Next, we present the sub-algorithms in details.

5.2 Sub-Algorithms

Algorithm 5.1 uses 4 sub-algorithms which are used to checks the stratification of the input program (Algorithm 5.2), rewrites rules (Algorithm 5.3), prepares input splitting set (Algorithm 5.4), and prepares the next value call to be processed (Algorithm 5.5).

Algorithm 5.2: *stratificationCheck*(in/out: $C, path$)

Input: set of value calls C, list of sets of value calls *path*
Output: **return** (set of value calls C, list of sets of value calls *path*)

```
1  if ∃P_i[S] ∈ C s.t. P_i[S] ∈ C_prev for some C_prev ∈ path then      /* detect cycle */
2      if S ≠ ∅ for some P_i[S] ∈ C then
3          exit with failure
4      repeat                                                  /* collect value calls */
5          C' := tail(path) and remove the last element of path
6          if ∃P_j[T] ∈ C' s.t. T ≠ ∅ then
7              exit with failure
8          else
9              C := C ∪ C'
10     until C' = C_prev
11 return (C, path)
```

5.2.1 Stratified Checking

Algorithm 5.2 performs on the fly stratification checking to make sure that the program given by users is ic-stratified. It takes as input a set of value calls C and a list of sets of value calls *path*. It first checks whether a cycle exists (line 1). If so, the algorithm exit with failure when the cycle contains an instantiation with non empty input, as our MLP violates the ic-stratified condition (line 3 and 7); otherwise, all instantiations with empty input that form the cycle are combined/collected to be solved together. The change (if any) with respect to C and *path* are returned (line 11).

In this process, *path* plays an important role since it maintains the call chain. More details on appending a set of value calls into *path* can be traced in Algorithm 5.5, and Algorithm 5.1 (line 9) for deleting an element of path.

Example 5.1 Suppose that we have a call graph

$$P_0[\emptyset] \rightarrow P_1[\{q(a)\}] \rightarrow P_0[\emptyset].$$

Clearly this is not a c-stratified MLP. When evaluating $C = \{P_0[\emptyset]\}$ (the second $P_0[\emptyset]$), and $path = [\{P_0[\emptyset]\}, \{P_1[\{q(a)\}]\}]$, Algorithm 5.2 will detect a cycle in the call chain (because $P_0[\emptyset]$ exists in the first element of *path*. We will pass the strafication checking in line 2 since in C we have only $P_0[\emptyset]$. Then, we enter the **repeat**...**until** block and proceed to line 5 to collect the set of value calls that forms the cycle. When encountering $P_1[\{q(a)\}]$, we exit with failure.

Example 5.2 Consider another case, suppose that the input to Algorithm 5.2 are $C = \{P_2[\emptyset]\}$, and $path = [\{P_0[\emptyset]\}, \{P_1[\{q(a)\}]\}, \{P_2[\emptyset]\}, \{P_1[\emptyset]\}]$.

We detects a cycle (because $P_2[\emptyset]$ exists in the third element of *path*. We collect all value calls starting from the last value call in *path*, until the value call having $P_2[\emptyset]$ in it. In this case we collect in C (and then remove it from *path*) the fourth and the third element of *path*. As the end result, we will get $C = \{P_2[\emptyset], P_1[\emptyset]\}$ and $path = [\{P_0[\emptyset]\}, \{P_1[\{q(a)\}]\}]$.

5.2.2 Rewriting

In our top-down approach, the procedure *rewrite*() presented in Algorithm 5.3 has an important role in evaluating the rules of the current instantiation. In this algorithm we rewrite the instantiation of each value call in C (set of value calls that currently under evaluation).

The main difference from the *rewrite*() in [Dao-Tran et al., 2009b] is that in Algorithm 5.3 we consider only the *top* part of the ground rule base. This makes the algorithm more effective since we do not compute the *bottom* part that has been evaluated before.

Taking Definition 4.3 and 4.4 into account, in this algorithm we write

- $\mathbf{U}$ instead of $\mathbf{U_P(A)}$,
- U_i/S instead of $U_i/S\,(A_i/S\,)$,

since we always refer to a particular MLP $\mathbf{P}$ and subordinate module atoms set $\mathbf{A}$ under evaluation.

1. For each instantiation, we retrieve its ground rule base (line 3)
2. Instead of considering the full instantiation, we consider only the *top* part of the instantiations based on the instantiation splitting set $\mathbf{U}$ (line 4).
3. Then, each rule in the *top* is analyzed (line 5). For each module atom in the rule that has been evaluated, we apply the partial evaluation akin to [Lifschitz and Turner, 1994]. The idea is to remove the positive module atom evaluated to true and negative module atoms evaluated to false (line 12 – 14), or to remove the rule whose body is not satisfied (see line 16 and 18).
4. Finally, we add the partial model that we have found so far. One might notice from Algorithm 5.1 (line 15 – 18) that the partial model that we have is an answer set of the *bottom* part so far.

Overall, *rewrite* plays an important role in the evaluation process in retrieving the result of previously evaluated module calls and combining it with the current set of rules. Example 6.3 and 5.4 provide a concrete example on how the *rewrite* procedure in Algorithm 5.3 works. Note that for simplicity, in this example we do not show the indexed sets with all of its elements. Instead, we present only the elements that relevant to the example.

Example 5.3 Let us consider a simple MLP $\mathbf{P} = (m_0, m_1)$, where $m_0 = (P_0, R_0)$, and $m_1 = (P_1[q_1], R_1)$, where:

$$R_0 = \left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ out_0(X) & \leftarrow & P_1[q_0].out_1(X) \end{array} \right\}$$

$$R_1 = \left\{ \begin{array}{rcl} s_1(a) & \leftarrow & \\ out_1(X) & \leftarrow & s_1(X), q_1(X) \end{array} \right\}$$

Supposed that we have gone through $P_0[\emptyset] \overset{q_0}{\rightarrow} P_1[\{q_1(a)\}]$ and solved the instantiation $P_1[\{q_1(a)\}$. On the way back to $P_0[\emptyset]$, we evaluate *solveMLP* with:

Algorithm 5.3: *rewrite*(in: $\mathbf{P}, C, \mathbf{M}, \mathbf{A}, \mathbf{U}$, out: R)

Input: MLP $\mathbf{P}$, set of value calls C, partial model $\mathbf{M}$, indexed set of sets of module atoms $\mathbf{A}$, instantiation splitting set $\mathbf{U}$,

Output: **return** set of rules R

1 $R := \emptyset$
2 **foreach** $P_i[S] \in C$ **do**
3 let $R_{P_i[S]}$ be the ground rule base of $P_i[S]$
4 $top := R_{P_i[S]} \setminus b_{U_i/S}(R_{P_i[S]})$ `/* collect top */`
5 **foreach** $r \in top$ **do**
6 $del_r := false$
7 **foreach** *occurence of module atom* $\alpha \in B(r)$ *in* r **do**
8 **if** $\alpha \in A_i/S$ **then**
9 let $\alpha = P_j[\mathbf{p}].o(\mathbf{c})$
10 $T := (M_i/S)|^{\mathbf{q_j}}_{\mathbf{p}}$ where $\mathbf{q_j}$ is the formal input of P_j
11 **if** $(o(\mathbf{c}) \in M_j/T) \wedge (\alpha \in B^+(r))$ **then** `/* partial evaluation */`
12 $B^+(r) := B^+(r) \setminus \alpha$
13 **else if** $(o(\mathbf{c}) \notin M_j/T) \wedge (\alpha \in B^-(r))$ **then**
14 $B^-(r) := B^-(r) \setminus \alpha$
15 **else**
16 $del_r := true$ `/* mark r as deleted */`
17 **break**
18 **if** $del_r = false$ **then** $R = R \cup r$
19 Add M_i/S as facts to R `/* add an answer set of the bottom */`
20 **return** R

- $C = \{P_0[\emptyset]\}$
- $path = \epsilon$
- $M_0/\emptyset = \{q_0(a)\}$,
  $M_1/\{q_1(a)\} = \{s_1(a), q_1(a), out_1(a)\}$
- $A_0/\emptyset = \{P_1[q_0].out_1(a)\}$
  $A_1/\{q_1(a)\} = fin$
- $U_0/\emptyset = \{q_0(a)\}$
  $U_1/\{q_1(a)\} = \{s_1(a), q_1(a), out_1(a)\}$

Initially we set $R = \emptyset$, then *rewrite* will proceed as follows:

line 3 We retrieve the ground rule base of $P_0[\emptyset]$,

$$R_{P_0[\emptyset]} = \left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ out_0(a) & \leftarrow & P_1[q_0].out_1(a) \end{array} \right\}$$

line 4 We get $b_{U_0/\emptyset}(P_0[\emptyset]) = \{q_0(a) \leftarrow\}$. Hence, $top = \{out_0(a) \leftarrow P_1[q_0].out_1(a)\}$.

line 5 Consider $r = out_0(a) \leftarrow P_1[q_0].out_1(a)$.

line 8 Since we have $P_1[q_0].out_1(a) \in A_0/\emptyset$,

line 10 $T = \{q_1(a)\}$

line 11 The condition is satisfied, i.e., $out_1(a) \in M_1/\{q_1(a)\}$ and $P_1[q_0].out_1(a) \in B^+(r)$. Hence, we have $r = out_0(a) \leftarrow$.

line 18 We have $R = R \cup \{out_0(a) \leftarrow\}$, hence $R = \{out_0(a) \leftarrow\}$.

line 19 We add $M_0/\emptyset$ to R, which makes $R =$

$$\left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ out_0(a) & \leftarrow & \end{array} \right\}.$$

Since C contain only one element, $P_0[\emptyset]$, *rewrite* stops here and returned R as the result.

Example 5.4 Suppose that we have an MLP $\mathbf{P} = (m_0, m_1, m_2)$, where $m_0 = (P_0, R_0)$, $m_1 = (P_1[q_1], R_1)$, and $m_2 = (P_2[q_2], R_2)$, where:

$$R_0 = \left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ out_0 & \leftarrow & P_1[q_0].out_1 \end{array} \right\}$$

$$R_1 = \left\{ \begin{array}{rcl} s_1(X) \vee s_1(Y) & \leftarrow & q_1(X), q_1(Y), X \neq Y \\ out_1 & \leftarrow & P_2[s_1].out_2 \end{array} \right\}$$

$$R_2 = \left\{ \begin{array}{rcl} s_2(X) & \leftarrow & q_2(X) \\ out_2 & \leftarrow & P_1[s_2].out_1 \end{array} \right\}$$

For example, let us compute $rewrite(C, \mathbf{M}, \mathbf{A}, \mathbf{U})$ when: [1]

- $C = \{P_2[\emptyset], P_1[\emptyset]\}$
- $M_1/\emptyset = \emptyset$
  $M_2/\emptyset = \emptyset$
- $A_1/\emptyset = \{P_2[s_1].out_2\}$,
  $A_2/\emptyset = \{P_1[s_2].out_1\}$
- $U_1/\emptyset = \{s_1(a), q_1(a), a \neq a\}$[2]
  $U_2/\emptyset = \{s_2(a), q_2(a)\}$

The computation started by inspecting each elements in C one by one. Initially we set $R = \emptyset$

[1] The call graph is: $P_0[\emptyset] \xrightarrow{q_0} P_1[\{q_1 a\}] \xrightarrow{s_1} P_2[\emptyset] \xrightarrow{s_2} P_1[\emptyset] \xrightarrow{s_1} P_2[\emptyset]$

[2] Please note that "$\neq$" is also a predicate symbol. It is a predicate symbol with special syntax (infix notation) and semantics (tests inequality).

1. For $P_2[\emptyset]$:

   line 3 We retrieve the ground rule base of $P_2[\emptyset]$,

   $$R_{P_2[\emptyset]} = \left\{ \begin{array}{rcl} s_2(a) & \leftarrow & q_2(a) \\ out_2 & \leftarrow & P_1[s_2].out_1 \end{array} \right\}$$

   line 4 We get $b_{U_2/\emptyset}(P_2[\emptyset]) = \{s_2(a) \leftarrow q_2(a)\}$. Hence, $top = \{out_2 \leftarrow P_1[s_2].out_1\}$.

   line 5 Consider $r = out_2 \leftarrow P_1[s_2].out_1$.

   line 8 Since we have $P_1[s_2].out_1 \in A_2/\emptyset$,

   line 10 $T = \emptyset$

   line 16 The condition in line 11 and 13 are not satisfied. Hence we enter line 16 and set $del_r = true$

   line 18 The condition is not satisfied since we have $del_r = true$, hence $R = \emptyset$.

   line 19 We add $M_2/\emptyset$ to R. Since $M_2/\emptyset = \emptyset$, we have $R = \emptyset$.

2. For $P_1[\emptyset]$, the computation is similar as before. From line 3, we get

   $$R_{P_1[\emptyset]} = \left\{ \begin{array}{rcl} s_1(a) \vee s_1(a) & \leftarrow & q_1(a), q_1(a), a \neq a \\ out_1 & \leftarrow & P_2[s_1].out_2 \end{array} \right\}$$

   From line 4, we get $b_{U_1/\emptyset}(P_1[\emptyset]) = \{s_1(a) \vee s_1(a) \leftarrow q_1(a), q_1(a), a \neq a\}$. Hence, $top = \{out_1 \leftarrow P_2[s_1].out_2\}$. Considering $r = out_1 \leftarrow P_2[s_1].out_2$, condition in line 8 is satisfied since we have $P_2[s_1].out_2 \in A_1/\emptyset$. Then, we have $T = \emptyset$ in line 10. In partial evaluation, the condition in line 11 and 13 are not satisfied. Hence we enter line 16 and set $del_r = true$. Condition in line 18 is not satisfied because we have $del_r = true$. Hence R is still empty. In the end, we add $M_1/\emptyset$ to R (line 19). Since $M_1/\emptyset = \emptyset$, we have $R = \emptyset$

3. As the final result, we return $R = \emptyset$ which is much less than plain collection of module instantiations.

5.2.3 Splitting Set Preparation

Basically, splitting set (and the *bottom*) is prepared in Algorithm 5.4. It works as follows:

- Choose a module atom α which does not depend on the other module atoms (line 1). If such module atom does not exist the evaluation exit immediately since the input program is not i-stratified (line 10).

- If such module atom α exist (all of the input predicates could be prepared without depending on other module atoms), then we collect in U an input splitting set of α (line 2). If an input splitting set for α does not exist, the evaluation exit immediately since it means that the input program is not i-stratified (line 8). Otherwise, we mark the module atom α as "processed" (by adding it to A_i/S for each $P_i[S] \in C$) (line 3 – 4), and update our instantiation splitting set $\mathbf{U}$ (line 6).

Algorithm 5.4: *prepareSplittingSet*(in: $\mathbf{P}, R, C,$ in/out: $\mathbf{A}, \mathbf{U},$ out: α, U)

Input: MLP $\mathbf{P}$, set of rules R, set of value calls C, indexed set of sets of module atoms $\mathbf{A}$, instantiation splitting set $\mathbf{U}$

Output: **return** indexed set of sets of module atoms $\mathbf{A}$, instantiation splitting set $\mathbf{U}$, module atom α, set of atoms U

1 **if** $\exists$ *a module atom* α *in* R *s.t.* $\nexists$ *module atom* β *in* R *where* $\alpha \rightsquigarrow \beta$ **then**
2 **if** $\exists$ *an input splitting set* U *of* R *for* α **then**
3 **foreach** $P_i[S] \in C$ **do**
4 **if** $A_i/S = nil$ **then** $A_i/S := \{\alpha\}$ **else** $A_i/S := A_i/S \cup \{\alpha\}$ /* update **A** */
5
6 $\mathbf{U} := \mathbf{U} \uplus mlpize(U \cup \alpha, C)$ /* update **U** */
7 **else**
8 **exit** *with failure* /* not i-stratified */
9 **else**
10 **exit** *with failure* /* not i-stratified */
11 **return** $(\mathbf{A}, \mathbf{U}, \alpha, U)$

Example 5.5, 5.6, and 5.7 give a clear idea on how Algorithm 5.4 works. Example 5.5 provide a case for i-stratified MLP, while the other two examples provide cases for non i-stratified MLPs.

Example 5.5 Suppose that we have an MLP $\mathbf{P} = (m_0, m_1, m_2)$, where $m_1 = (P_0, R_0)$, $m_1 = (P_1[q_1], R_1)$, $m_2 = (P_2[q_2], R_2)$, and

$$R_0 = \left\{ \begin{array}{lcl} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ r & \leftarrow & P_1[q].out_1 \\ s & \leftarrow & P_2[r].out_2 \end{array} \right\}$$

In addition, let us assume that $C = \{P_0[\emptyset]\}$, $A_0/\emptyset = nil$, $U_0/\emptyset = nil$.

Since we have $P_2[r].out_2 \rightsquigarrow P_1[q].out_1$, and $P_1[q].out_1$ does not depend on any other module atom, we set $\alpha = P_1[q].out_1$ (line 1), and we have splitting set $U = \{q(a), q(b)\}$ (line 2). Then, we set $A_0/\emptyset = \{P_1[q].out_1\}$ (line 4), and $U_0/\emptyset = \{q(a), q(b)\}$ (line 6).

Example 5.6 Let us assume that we have $C = \{P_0[\emptyset]\}$, $A_0/\emptyset = nil$, and a rule set $R =$

$$\left\{ \begin{array}{lcl} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ r(a) & \leftarrow & P_1[q].out_1 \\ q(a) & \leftarrow & P_2[r].out_2 \end{array} \right\}$$

In this case, $P_1[q].out_1$ and $P_2[r].out_2$ depend to each other. Choosing either $P_1[q].out_1(X)$ or $P_2[r].out_2(X)$ will not satisfy the condition in line 1 and causes the algorithm to exit (line 10). Indeed, this is an example of non i-stratified program (caused by the third and the fourth rule in R), and the computation will stop immediately, as expected.

Algorithm 5.5: *prepareNextCall*(in: $\mathbf{P}, \alpha, C, path, N, \mathbf{A}$, out: $C', path'$)

Input: MLP $\mathbf{P}$, module atom α, set of value calls C, list of sets of value calls *path*, set of ordinary atoms as the answer set of *bottom* N, indexed set of sets of module atoms $\mathbf{A}$

Output: **return** set of value calls C', list of sets of value calls $path'$

1 $T := N|_{\mathbf{p}}^{\mathbf{q_j}}$, where $\alpha = P_j[\mathbf{p}].o(\mathbf{c_j})$ and $\mathbf{q_j}$ is the formal input parameter of P_j
2 **if** $A_j/T = fin$ **then**
3 **return** $(C, path)$
4 **else**
5 $C' := \{P_j[T]\}$
6 $path' := append(path, C)$
7 **return** $(C', path')$

Example 5.7 Let us assume that we have $C = \{P_0[\emptyset]\}$, $A_0/\emptyset = nil$, and a rule set $R =$

$$\left\{ \begin{array}{lll} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ q(c) & \leftarrow & P_1[q].out_1 \end{array} \right\}$$

This is another simple example of not i-stratified program (caused by the third rule in R). We have only one module atom, simply choosing $\alpha = P_1[q].out_1$ will pass the condition in line 1. However, then we cannot have an input splitting set for $P_1[q].out_1$. This is because the input predicate for $P_1[q].out_1$ is q, but $q(c)$ depends on $P_1[q].out_1$. Condition in line 2 is not satisfied, the evaluation will exit immediately (line 8).

5.2.4 Value Call Preparation

In Algorithm 5.5, we prepare the next value call needed to evaluate for the module atom α returned by *prepareSplittingSet* procedure in Algorithm 5.4. There are two possibilities:

- We have evaluated this value call before (condition in line 2 is satisfied), hence we can continue with the current C and *path*. Instead, we stick to the current value call and later we let the next recursion step of *solveMLP* (Algorithm 5.1) to solve it.
- Otherwise, as one might expected, we proceed to solve the new value call created, $P_j[T]$, by putting it into C' (line 5) as the next value call to be evaluated and add to *path* the current value call, C.

In this algorithm, *fin* (see line 2) is a special value on A_i/S to mark that all module atoms inside instantiation $P_i[S]$ has been evaluated (see Algorithm 5.1, line 8 to see when we put *fin*).

Using a simple program, Example 5.8 and Example 5.9 gives an illustration on how Algorithm 5.4 runs.

Example 5.8 Suppose that we have MLP $\mathbf{P} = (m_0, m_1, m_2)$, where $m_0 = (P_0, R_0)$, $m_1 = (P_1[q_1], R_1)$, and $m_2 = (P_2[q_2], R_2)$. For simplicity, we do not give the complete rule sets.

However, we provide Figure 5.1 to illustrate how the modules call each other, where $S_0 = \emptyset$, $S_1 = \{q_1(a), q_1(b)\}$, and $S_2 = \{q_2(a), q_2(b)\}$.

Now, let us assume:

- $\alpha = P_1[r_0].r_1(X)$,
- $C = \{P_1[\emptyset]\}$,
- $path = \epsilon$,
- $N = \{r_0(a), r_0(b)\}$
- $A_0/\emptyset = \{P_1[r_0].r_1(X)\}$, while the other elements of $\mathbf{A}$ are still. *nil*

Let us assume that currently we are evaluating the top most instantiation on Figure 5.1 and preparing to make a module call to the left ($P_1[S_1]$). From line 1, we have $T = \{q_1(a), q_1(b)\}$. Since $A_1/\{q_1(a), q_1(b)\}$ is still *nil*, we skip line 2 – 4 and execute line 5 – 7 instead. Then we have $C' = P_1[\{q_1(a), q_1(b)\}]$ and $path' = [\{P_1[\emptyset]\}]$.

Example 5.9 Take a look at Example 5.8 and Figure 5.1 again. To illustrate the other cases in Algorithm 5.5, let us consider the step when we evaluate $P_2[S_2]$. When we run *prepareNextCall* with:

- $\alpha = P_1[r_2].r_1(X)$,
- $C = \{P_2[\{q_2(a), q_2(b)\}]\}$,
- $path = [\{P_0[\emptyset]\}]$,
- $N = \{q_2(a), q_2(b), r_2(a), r_2(b)\}$, and
- $A_1/\{q_1(a), q_1(b)\} = \mathit{fin}$ (the other element of $\mathbf{A}$ are irrelevant for this example).

Then we have $T = \{q_1(a), q_1(b)\}$ from restricting N to r_2 (actual input parameter in α) and then replacing r_2 with q_1 (formal input parameter of P_1). Since we know that $A_1/\{q_1(a), q_1(b)\} = \mathit{fin}$, hence $C' = C$ and $path' = path$. This example illustrates the case when we have computed the answer set of $I_{\mathbf{P}}(P_1[S_1])$ in instantiation $P_0[S_0]$, we do not need to compute it again when we see it again in $P_2[S_2]$.

5.3 Soundness and Completeness of Algorithm *solveMLP*

In this section we prove the soundness and completeness of, Algorithm 5.1. To do this, we need the following notion.

A value call $P_j[T]$ is said to be *induced* by a value call $P_i[S]$ iff $P_j[\mathbf{p}].o(\mathbf{c})$ occurs in $R(m_i)$ and $T = N|^{\mathbf{q_j}}_{\mathbf{p}}$, where N is an answer set of $b_U(I_{\mathbf{P}}(P_i[S]))$ and U is an input splitting set of $I_{\mathbf{P}}(P_i[S])$ for $P_j[\mathbf{p}].o(\mathbf{c})$.

The following lemma highlight the case when an element A_i/S of A is set to *fin*, meaning that all module atoms in instantiation $P_i[S]$ has been evaluated.

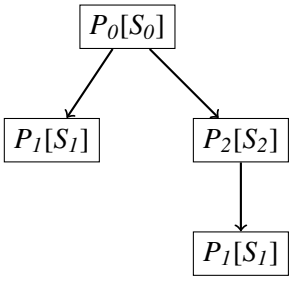


Figure 5.1: Utilized previously computed answer set of a module call

Lemma 5.10 *Let* **P** *be an ic-stratified MLP, and C, path,* **M**, **A**, **U**, $\mathcal{AS}$ *are data structures as defined in Algorithm 5.1. When we call solveMLP*$(\mathbf{P}, C, path, \mathbf{M}, \mathbf{A}, \mathbf{U}, AS)$*, with* $A_i/S = fin$*,* M_i/S *contains an answer set of* $I_{\mathbf{P}}(P_i[S])$ *relative to* **M**.

Proof. We assign *fin* to A_i/S only in line 8 in Algorithm 5.1. When we execute line 8, it means that R as the result from *rewrite* is ordinary (hence, all module atoms has been solved). Then we have two possibilities:

(i) From line 10, $ans(R)$ returns *nil*. This means that R has no answer set. Since we do not proceed with line 12, **A** will not be propagated. The next call to *solveMLP* is not executed, the algorithm will backtrack to a state where $A_i/S \neq fin$.

(ii) $ans(R)$ returns at least one element, which means that R has at least one answer set. Then line 12 will be executed, **A** will be propagated together with $\mathbf{M}'$ that has been updated with N, where N is an answer set of R. Then $\mathbf{M}'$ will be **M** in the next call of *solveMLP*. Recall that $mlpize(N, C)$ convert N to a partial interpretation **N**, projects each atom in N to $P_i[S] \in C$, and put the result into N_i/S. And then $\mathbf{M} \uplus \mathbf{N}$ basically add to M_i/S all atoms in N_i/S.

□

For the soundness and completeness, we adapt Proposition 7 from [Dao-Tran et al., 2009b].

5.3.1 Soundness

Theorem 5.11 *Let* **P** *be an ic-stratified MLP with a main module* $m_{main} = (P_{main}[], R_{main})$*. If* $\mathbf{M} \in \mathcal{AS}$ *then* **M** *is an answer set of* **P***, where* $\mathcal{AS}$ *is the result of solveMLP*$(\mathbf{P}, \{P_{main}[\emptyset]\}, path,$ **M**, **A**, **U**$, \mathcal{AS})$ *with* $\mathcal{AS} = \emptyset$*,* $path = \epsilon$*,* **M**, **A***, and* **U** *to have nil at all components in the beginning (disregarding irrelevant module instances, i.e.,* $M_i/S = nil$ *iff* $P_i[S] \notin V(CG_{\mathbf{P}}(\mathbf{M}))$*).*

Proof. We need to show that if $\mathbf{M} \in \mathcal{AS}$, **M** is an answer set of **P**, that is M_i/S is an answer set of $I_{\mathbf{P}}(P_i[S])$ for $P_i[S] \in V(CG_{\mathbf{P}}(\mathbf{M}))$ relative to **M**.

To make it simple, we show soundness of our algorithm step by step:

1. We assume that the condition in line 1 in Algorithm 5.2 never satisfied, and we disregard line 2 – 5 in Algorithm 5.5.

2. We extend the proof, considering the condition in line 1 in Algorithm 5.2 is satisfied.

3. Finally we show that line 2 – 5 in Algorithm 5.5 is an optimization that enable us to not compute an already computed instantiation twice. We show that with or without this lines, our algorithm is sound.

We now proceed with the proof as outlined above:

1. We proof by structural induction on how each element of $\mathbf{M}$ is formed.

   Induction Base For $C = \{P_i[S]\}$, where instantiation $I_{\mathbf{P}}(P_i[S])$ does not have a module atom in it, *solveMLP* will proceed as follows:

   - *rewrite* will return an ordinary rules R (that is $I_{\mathbf{P}}(P_i[S])$)
   - Regardless *path* is empty or not, from $ans(R)$ we get the answer set(s) of R in N.
   - $mlpize(N, C)$ put N in M_i/S.
   - Hence we get M_i/S as an answer set of $I_{\mathbf{P}}(P_i[S])$ relative to $\mathbf{M}$.

   Induction Hypothesis $M_1/S_1, \ldots, M_k/S_k$ are answer sets for $I_{\mathbf{P}}(P_1[S_1]), \ldots, I_{\mathbf{P}}(P_k[S_k])$ relative to $\mathbf{M}$ respectively.

   Induction Case Let us consider $C = \{P_i[S_i]\}$ where in the instantiation $I_{\mathbf{P}}(P_i[S_i])$ eventually we create module calls to $P_1[S_1], \ldots, P_k[S_k]$, we need to show that M_i/S_i is an answer set of $I_{\mathbf{P}}(P_i[S_i])$ relative to $\mathbf{M}$. *solveMLP* will proceed as follows:

   - First, in line 2, *rewrite* will return non-ordinary rules R (since we had a module calls to $P_1[S_1], \ldots, P_k[S_k]$).
   - Then the algorithm will proceed with solving $P_1[S_1], \ldots, P_k[S_k]$ and got answer sets $M_1/S_1, \ldots, M_k/S_k$. The process itself is started from line 14 in Algorithm 5.1 for choosing the next module atom to evaluate, α, and providing an input splitting set for it. Then for each answer set of the *bottom*, we prepare the next module call for α. More details:
     a) Without loss of generality, let us assume that we $P_k[\mathbf{q_k}].o_1(\mathbf{c_k}) \rightsquigarrow \ldots \rightsquigarrow P_1[\mathbf{q_1}].o_1(\mathbf{c_1})$. Later, proceeding with this ordering will create calls to $P_1[S_1], \ldots, P_k[S_k]$.
     b) First, we will proceed with $\alpha = P_1[\mathbf{q_1}].o_1(\mathbf{c_1})$. We add $P_1[\mathbf{q_1}].o_1(\mathbf{c_1})$ to A_i/S_i (which intuitively means that we proceed to compute $P_1[\mathbf{q_1}].o_1(\mathbf{c_1})$ in $I_{\mathbf{P}}(P_i[S_i])$).
     c) Based on the answer set of the *bottom*, $ans(b_U(R))$, where R is the result of rewriting $I_{\mathbf{P}}(P_i[S_i])$ and U is a result from Algorithm 5.4, we prepare the next call in C' and $path'$ (line 6 and 7 from Algorithm 5.5 and then proceed with line 18 in Algorithm 5.1). In this case, C' will be $P_1[S_1]$ and $path'$ will consist of the current *path* with $P_i[S_i]$ in the tail.

d) We make a call to $P_1[S_1]$ (proceed from line 18 in Algorithm 5.1) and later we get M_1/S_1 as the answer set (from Induction Hypothesis). It is also important to mention here that we always update U_i/S_i in Algorithm 5.4 that is executed in line 14, and add to M_i§$_i$ an answer set of $b_{U_i/S}(R)$. From Proposition 4.7, we can combine the answer set of $b_{U_i/S}(R)$ with the answer set of the *top* part of R to form the complete answer set of R.

e) In the recursive call, please note that when we finish computing $P_1[S_1]$, we will proceed to line 8 – 12 (since the *path* is not empty). Then we enter a recursive call which continues evaluating $P_i[S_i]$.

f) In this recursive call, entering *solveMLP* with $C = \{P_i[S_i]\}$ will produce a different result from *rewrite*. In this phase, we have $U_i/S_i \neq nil$ and $M_i/S_i \neq nil$. This reflects to the splitting set theorem that we want combine the answer set of the *bottom* (currently in M_i/S_i) with the answer set of the *top* (we do not have it yet, but we proceed to that)

g) Next, since we still have $P_k[\mathbf{q_k}].o_1(\mathbf{c_k}) \rightsquigarrow \ldots \rightsquigarrow P_2[\mathbf{q_2}].o_2(\mathbf{c_2})$ in R (if $k > 1$), then again we proceed with the non-ordinary part (line 14 – 18). Please note that we do not have $P_1[\mathbf{q_1}].o_1(\mathbf{c_1})$ in R because of the partial evaluation that we done in Algorithm 5.3. When we have evaluated a module atom, then these two condition always apply:

   (i) the module atom is removed (see line 12 and 14 in Algorithm 5.3), or

   (ii) the rule that contain the module atom is removed (see line 16 and 18 in Algorithm 5.3).

   This is akin to partial evaluation in [Lifschitz and Turner, 1994].

h) and the process continue until we compute $P_k[S_k]$.

– Once we solved module calls $P_1[S_1], \ldots, P_k[S_k]$, *rewrite* will return R in an ordinary form and we have $\{P_1[S_1], \ldots, P_k[S_k]\} \subseteq \mathbf{A}$. Either *path* is empty or not, from line 5 or 10, we will get the answer set of R in N from *ans*. Then, from $\mathbf{M} \uplus mlpize(N, C)$ and justified by Proposition 4.8, we get the complete answer set of $P_i[S_i]$ in M_i/S_i.

2. We extend the proof, considering that $P_i[S]$ is one of instantiations that form a cycle in the value call chain.

   - We extend the *Induction Base*:

     – Consider $P_{l_1}[\emptyset], \ldots, P_{l_k}[\emptyset]$ where $I_{\mathbf{P}}(P_{l_i}[\emptyset])$ has only one module call, that is to $P_{l_{i+1}}[\emptyset]$ for $1 \leq i < k$ and eventually $I_{\mathbf{P}}(P_{l_k}[\emptyset])$ has a module call to $P_{l_1}[\emptyset]$.

     – When we enter *solveMLP* with $C = \{P_{l_1}[\emptyset]\}$, eventually we will make a call to $P_{l_2}[\emptyset], \ldots, P_{l_k}[\emptyset]$ and then to $P_{l_1}[\emptyset]$. For this last call, our *stratificationCheck* algorithm will detect a cycle (see line 1 in Algorithm 5.1, and continue to line 1 in Algorithm 5.2). Checking condition in line 2 in Algorithm 5.2 will succeed since we have a c-stratified program. Next, we collect $P_{l_1}[\emptyset], \ldots, P_{l_k}[\emptyset]$ in C (line 4 – 10, Algorithm 5.2).

- *rewrite* will return an ordinary rules R from instantiations of $I_{\mathbf{P}}(P_{l_l}[\emptyset]), \ldots, I_{\mathbf{P}}(P_{l_k}[\emptyset])$. Ordinary, because partial evaluation has been done to all module atoms $P_{l_i}[\mathbf{q_{l_i}}].o_{l_i}(\mathbf{c_i})$ for $1 \leq i \leq k$. This is due to the case $P_{l_{i+1}}[\mathbf{q_{l_{i+1}}}].o_{l_{i+1}}(\mathbf{c_{i+1}}) \in A_{l_i}/S_{l_i}$ for $1 \leq i < k$, and $P_{l_1}[\mathbf{q_{l_1}}].o_{l_1}(\mathbf{c_1}) \in A_{l_k}/S_{l_k}$. One can see from line 3 – 4 in Algorithm 5.4 that we always put the next module atom that we will call in A_i/S given that $P_i[S]$ is in the current set of value calls being evaluated, C.
- Then either *path* is empty or not, we will have an answet of R in N. And by $\mathbf{M} \uplus mlpize(N, C)$ we will have the answer set of $I_{\mathbf{P}}(P_{l_l}[\emptyset]), \ldots, I_{\mathbf{P}}(P_{l_k}[\emptyset])$ in $M_{l_1}/\emptyset, \ldots, M_{l_k}/\emptyset$ relative to $\mathbf{M}$

- We extend the *Induction Hypothesis*:
  Assume that $M_{g_{i,1}}/S_{g_{i,1}}, \ldots, M_{g_{i,k_i}}/S_{g_{i,k_i}}$ are answer sets for $I_{\mathbf{P}}(P_{g_{i,l}}[S_{g_{i,l}}]), \ldots, I_{\mathbf{P}}(P_{g_{i,k_i}}[S_{g_{i,k_i}}])$ for $1 \leq i \leq n$
- We extend the *Induction Case*:
  Now we consider the case that we have other module calls in a cycle. Suppose that we have a cycle formed by module calls from $P_{m_l}[\emptyset], \ldots, P_{m_n}[\emptyset]$ and from $P_{m_n}[\emptyset]$ back to $P_{m_l}[\emptyset]$. And later, we will consider that each $P_{m_i}[\emptyset]$ also has other module calls to $P_{g_{i,l}}[S_{g_{i,l}}], \ldots, P_{g_{i,k_i}}[S_{g_{i,k_i}}]$ for $1 \leq i \leq n$. We need to show that $M_{m_1}/\emptyset, \ldots, M_{m_n}/\emptyset$ are answer sets for $I_{\mathbf{P}}(P_{m_l}[\emptyset]), \ldots, I_{\mathbf{P}}(P_{m_n}[\emptyset])$.
  - Collection of module instantiation that responsible for the cycle, i.e., $C = \{P_{m_l}[\emptyset], \ldots, P_{m_n}[\emptyset]\}$ will be started similar to *Induction Base*.
  - However, in the end we do not have an ordinary rules R. Instead, we still have $P_{g_{i,1}}[\mathbf{q_{g_{i,1}}}].o_{g_{i,1}}(\mathbf{c_{g_{i,1}}}), \ldots, P_{g_{i,k_i}}[\mathbf{q_{g_{i,k_i}}}].o_{g_{i,k_i}}(\mathbf{c_{g_{i,k_i}}})$, module atoms to solve, for $1 \leq i \leq n$.
  - Similar to the *Induction Case* in the previous case, by maintaining *path* we solve those module atoms one by one. And by *Induction Hypothesis*, we have $M_{g_{i,1}}, \ldots, M_{g_{i,k_i}}$ as answer sets for $I_{\mathbf{P}}(P_{g_{i,l}}[S_{g_{i,l}}]), \ldots, I_{\mathbf{P}}(P_{g_{i,k_i}}[S_{g_{i,k_i}}])$ for $1 \leq i \leq n$.
  - Once we solved $P_{g_{i,1}}[\mathbf{q_{g_{i,1}}}].o_{g_{i,1}}(\mathbf{c_{g_{i,1}}}), \ldots, P_{g_{i,k_i}}[\mathbf{q_{g_{i,k_i}}}].o_{g_{i,k_i}}(\mathbf{c_{g_{i,k_i}}})$, with $1 \leq i \leq n$, on the way back from recursion, when we solved the last module call, i.e. $P_{g_{n,k_n}}[S_{g_{n,k_n}}]$, again we will have $C = \{P_{m_l}[\emptyset], \ldots, P_{m_n}[\emptyset]\}$ and we will have R as the result of *rewrite* is ordinary.
  - Then either *path* is empty or not, we will have an answet of R in N. And by $\mathbf{M} \uplus mlpize(N, C)$ we will have the answer set of $I_{\mathbf{P}}(P_{m_l}[\emptyset]), \ldots, I_{\mathbf{P}}(P_{m_n}[\emptyset])$ in $M_{m_1}/\emptyset, \ldots, M_{m_k}/\emptyset$

3. We prove that adding line 2 – 4 in Algorithm 5.5, *solveMLP* is still sound. Consider an arbitrary value call C_A and path $path_A$ and without loss of generality assume that α is the last module atom that need to be solved from the rule set as the result of rewriting C_A. In Algorithm 5.4, α is added to A_i/S for each $A_i/S \in C_A$.

   Now, let us assume that $P_j[T]$ is a value call for created for α. Without line 2 – 4 in Algorithm 5.5, actually we have to evaluate $P_j[T]$ and get an answer set of $P_j[T]$ in M_j/T before we back to the execution of *solveMLP*with $C = C_A$ and $path = path_A$ to continue

evaluating value calls in C_A. Now, we need to show that with line 2 – 4 in Algorithm 5.5, eventually we will evaluate $C = C_A$ with $path = path_A$ and M_j/T contain an answer set of $P_j[T]$ (*). Consider that we execute line 2 – 4 in Algorithm 5.5 and $A_j/T = fin$.

- Executing Algorithm 5.5 means that currently we are executing *prepareNextCall* procedure in line 16 in *solveMLP*.
- Since $A_j/T = fin$, M_j/T contain an answer set of $P_j[T]$ (Lemma 5.10).
- $C' = C_A$ and $path' = path_A$.
- Return from *prepareNextCall* we execute line 18 in *solveMLP*. Which bring us to the condition that we want (*).

□

5.3.2 Completeness

Theorem 5.12 *Let* **P** *be an ic-stratified MLP with a main module* $m_{main} = (P_{main}[], R_{main})$*. Let* $\mathcal{AS}$ *be the result of solveMLP*$(\mathbf{P}, \{P_0[\emptyset]\}, path, \mathbf{M}, \mathbf{A}, \mathbf{U}, \mathcal{AS})$ *with* $\mathcal{AS} = \emptyset$*,* $path = \epsilon$*,* **M***,* **A***, and* **U** *to have nil at all components in the beginning. If* $\mathcal{AS} = \emptyset$ *then* **P** *has no answer set.*

Proof. First, assume that $\mathcal{AS} = \emptyset$, then we need to show that MLP **P** has no answer set. There are two possibilities:

1. The algorithm reach line 6. Then we have R as the result of *rewrite* in line 2 is ordinary and *path* is empty. Condition *path* is empty could only be achieved when we evaluate C where $P_{main}[\emptyset] \in C$.

   a) $C = P_{main}[\emptyset]$ then basically $R = I_{\mathbf{P}}(P_{main}[\emptyset])$:

      - $P_{main}[\emptyset]$ has no module call. Since $\mathcal{AS} = \emptyset$, then it means that R has no answer set. Since R actually is $I_{\mathbf{P}}(P_{main}[\emptyset])$, it means that $I_{\mathbf{P}}(P_{main}[\emptyset])$ has no answer set, which means MLP **P** also has no answer set.
      - $P_{main}[\emptyset]$ has module calls. All module calls in $P_{main}[\emptyset]$ have been evaluated (hence R is ordinary). But still, R has no answer set. Similar to the previous cases, since $(P_{main}[\emptyset])$ has no answer set, MLP **P** also has no answer set.

   b) C contain several module instantiation, including $P_{main}[\emptyset]$. This is a condition where there is a cycle that contains $P_{main}[\emptyset]$. **P** also has no answer set. This is due to the fact that:

      - C contain $P_{main}[\emptyset]$ and value call(s) induced by $P_{main}[\emptyset]$
      - *rewrite* took all instantiation of all $P_i[S] \in C$, put the result in R.
      - Since R has no answer set, it means that we do not find an answer set for $I_{\mathbf{P}}(P_{main}[\emptyset])$ and instantiations from value call(s) *induced* by it, hence **P** also has no answer set.

2. The execution of Algorithm 5.1 never reach line 6. Never reach line 6 means that the algorithm actually stop, without once executing line 6. Since $path = \epsilon$ (condition in line 4 in order to reach line 6) can only be reached when evaluating the main module, this also means that during the execution of the algorithm, when the algorithm evaluate the main module ($P_{main}[\emptyset] \in C$), R as the result from *rewrite* is not ordinary (condition in line 3).

   We have R from *rewrite* is not ordinary means that there is a module atom that still need to be evaluated. Let us consider the last time $path = \epsilon$ before the algorithm actually stop ($*$). Then:

   - The algorithm proceed to line 14.
   - Let $\alpha = P_j[\mathbf{q_j}].o(\mathbf{c})$. From line 14 we get U as the input splitting set for α. In line 16, we execute *prepareNextCall* from Algorithm 5.5 to prepare the next value call to evaluate. However, we need to make sure that we will eventually evaluate all possible value calls that can be created. This condition is guarantee by line 15 and 16 in Algorithm 5.1. Line 15 iterates over all possible answer sets of the *bottom* of U. Then, in line 16 we prepares the value call which inputs based on those answer set.

     Then we have two possibilities:

     (i) If the value call has been evaluated before, looking at Algorithm 5.5 in more detail, condition in line 2 in Algorithm 5.5 is satisfied. But then this means that $path' = path$, returning from *prepareNextCall* to *solveMLP*, eventually we will execute the recursive case in line 18 which gave us another execution of *solveMLP* with $path = \epsilon$. This contradicts our previous assumption ($*$), i.e., the last time execution of *solveMLP* with $path = \epsilon$.

     (ii) If the value call has never been evaluated before, then we have:
     - $C' = \{P_j[T]\}$ where $T := N|^{\mathbf{q_j}}_{\mathbf{p}}$, and
     - the current C is appended to *path* (currently $path = \epsilon$) and we put the result to $path'$, which makes $path'$ contain one element, C.

     From this step, there are two possibilities:

     (a) P_j has no module atom. Then, the condition in line 3 will be satisfied. Line 8 and 9 will be executed. In line 9 we remove the last element of *path*. Since the current *path* contain only one element, then *path* becomes ϵ. Suppose that we have $\mathbf{M}$ as an answer set of $\mathbf{P}$, then in $CG_{\mathbf{P}}(\mathbf{M})$ we will have $P_{main}[\emptyset] \rightarrow P_j[T]$ and M_k/T as the answer set of $P_j[T]$ relative to $\mathbf{M}$. Since the iteration in line 15 guarantees that we explore all possible answer sets of the *bottom* of U and line 16 prepares the value call that can be created from α, eventually we will visit this particular value call $P_j[T]$. Then:
     - the result of rewriting, R, is ordinary,
     - in line 9 we delete one element of *path* (hence *path* becomes empty),
     - *ans* in line 10 will return at least an element,

– line 12 will be executed, which means another execution of *solveMLP* with $path = \epsilon$

This contradicts our assumption ($*$). Hence, **P** must not has an answer set.

(b) P_j has module atom(s). If we have an answer set **M** of **P**, then by the computation that follows the call chain $P_{main}[\emptyset] \rightarrow P_j[T] \rightarrow \ldots$ in $CG_{\mathbf{P}}(\mathbf{M})$, even though we always add each value call induced by $P_j[T]$ to *path*, eventually we will get an answer set for each instantiation created from the value calls induced by $P_j[T]$. Then:

– the rewriting process of instantiation $P_j[T]$ will result in an ordinary R,
– line 9 will be executed (remove an element of *path*, which caused *path* to be empty),
– *ans* in line 10 will return at least an answer set,
– line 12 will be executed, and
– we will have another execution of *solveMLP* with empty *path*.

This contradicts our assumption ($*$), hence **P** must not has an answer set.

□

CHAPTER 6

Implementation

This chapter discusses the implementation of an MLP solver that has been developed on this thesis. We start the chapter by describing the system architecture, then continue with specifying the allowed input format, and finally end it with explaining how one can run the system.

The system developed is called *TD-MLP*, a short for *top-down evaluation solver for MLPs*. We did not build the system from scratch, but rather based it on *dlvhex*[1]. Our implementation uses C++ as its programming language, and makes use of several libraries, such as Boost[2] and Bitmagic[3] library.

6.1 System Architecture

This section describes the architecture of *TD-MLP* as an extension from *dlvhex* system [Eiter et al., 2006]. *TD-MLP* introduces two new concepts/items, namely module atom and module header. Section 6.1.1 explains the main architecture and Section 6.1.2 and 6.1.3 describe in more detail two important components: *Syntax Checking* and *Evaluator*.

6.1.1 Main Architecture

Figure 6.1 displays the main architecture of *TD-MLP*, which includes the following components:

1. *Syntax Checking* (*SC*): makes sure that users write MLP in a correct syntax (see Section 6.2 for syntax) so that the evaluation algorithm can also run correctly.

2. *Main Modules Collector*: identifies main modules defined by users.

3. *Evaluator*: computes answer sets of an MLP.

[1] *dlvhex* is a prototype implementation of HEX-program [Eiter et al., 2005], which are an extension of ASP towards integration of external computation source.

[2] http://www.boost.org/

[3] http://bmagic.sourceforge.net

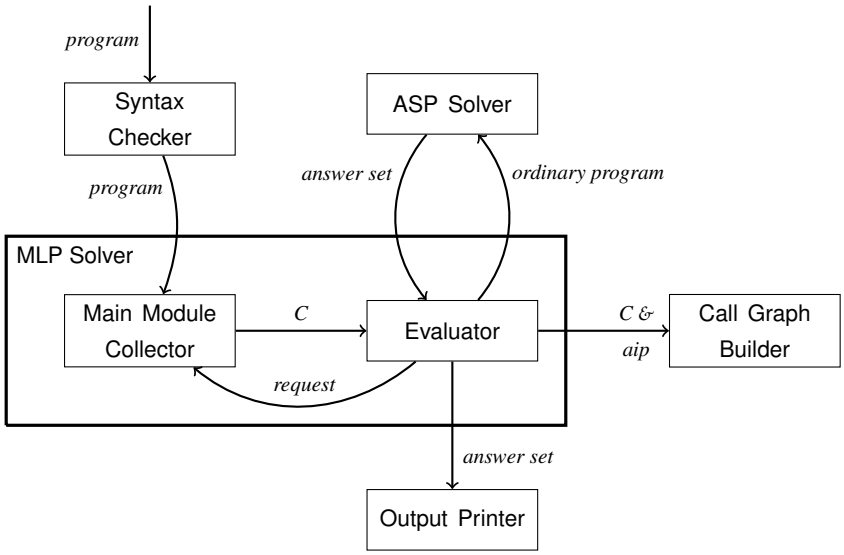


Figure 6.1: System Architecture

4. *ASP Solver*: called by *Evaluator*, computes answer sets of an ordinary program.

5. *Call Graph Builder*: produces a call graph from the evaluation that has been done.

6. *Output Printer*: outputs results in a readable format for the user.

Different types of data used in the systems are:

1. *C*: represents a set of value calls (referring to *C* in Algorithm 5.1).

2. *aip*: actual input parameter, a set of atoms that is used as the input parameters for value calls created.

3. *request*: notification for *Main Modules Collector* from *Evaluator* that the current evaluation is finished and asks for another main module to be solved (if any).

From Figure 6.1 we can see that, first, an input program is passed to *SC*. This component makes sure that the program provided is syntactically correct. A syntactically correct input is important so that the solver interpret the problem (which is reprented in the input) correctly and deliver the intended results.

After the syntax is proven to be correct, *Main Modules Collector* identifies all main modules in the input program one by one and passed it to the *Evaluator*. During evaluation, *Evaluator* calls the *ASP Solver* component to solve an ordinary program (either to solve the bottom or to solve the ordinary program in the last phase of the computation where all module calls have been solved). The current set of value calls, *C*, of the computation and the actual input parameter, *aip*, for each value call are always passed to the *Call Graph Builder* in order to build a call graph of the evaluation.

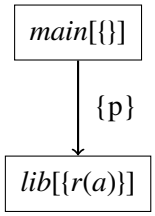


Figure 6.2: An example of *Call Graph*

Whenever the evaluation is done, the answer set is passed to the *Output Printer* to be polished and delivered to users. Next, the evaluator requests the next main module to solve (if any).

Example 6.1 Suppose that we have an MLP $\mathbf{P} = (m_1, m_2)$ where $m_1 = (main, R_1)$, $m_2 = (lib[r], R_2)$, and

$$R_1 = \left\{ \begin{array}{rcl} p(a) \vee p(b) & \leftarrow & \\ getA & \leftarrow & lib[p].q(a) \end{array} \right\}$$

$$R_2 = \left\{ \begin{array}{rcl} q(X) & \leftarrow & r(X) \end{array} \right\}$$

To evaluate **P** using *TD-MLP*, firstly SC checks whether the input program is syntactically correct (see Section 6.2 for more details on syntax). Assume for now that this is the case. Next, *Main Modules Collector* collects all main modules in the program. In this case, we found only $main[\emptyset]$. *Main Modules Collector* passes a value call containing a main module, $main[\emptyset]$ to the *Evaluator*.

If there are several main modules, they are then passed one by one. Starting from the first main module, *Evaluator* solves it, and then sends a request for the next main module to be solved.

On the process, we know that based on the algorithm described in Chapter 5, *Evaluator* has to call an *ASP Solver* to solve an ordinary program. And as explained before, this ordinary program can be a *bottom* part of a set of rules, or even the complete set from *rewrite* (when the module atoms occurred have been solved). As an example, from the R_1 above the *bottom* that needs to be solved by the *ASP Solver* is $\{p(a) \vee p(b)\}$. In this case, $\{p(a)\}$ and $\{p(b)\}$ are the answer sets of the *bottom* and will be returned by *ASP Solver*, one by one.

While evaluating an MLP, *Evaluator* also sends a value call that is currently being processed, together with the actual input to the *Call Graph Builder*. As an example, during its first evaluation, *Evaluator* will send $C_1 = \{main[\emptyset]\}$. Next, suppose that we got $\{p(a)\}$ as the first answer set from the *ASP Solver*, then as the next value call we will have $C_2 = \{lib[\{r(a)\}]\}$ (and $\{p(a)\}$ as the actual input parameter). *Evaluator* will also send this to the *Call Graph Builder*. Then, based on this information, *Call Graph Builder* will build the call graph. A call graph that is obtained from this example can be seen in Figure 6.2.

Once the evaluation is finished, the final answer set is transferred from the *Evaluator* to the *Output Printer* in order to be printed for end users. A real example on how the final answer set looked like can be seen in Section 6.2.

Then, we proceed to the second answer set $\{p(b)\}$. Proceed similarly, we have $C_3 = lib[\{r(b)\}]$ as the next value call.

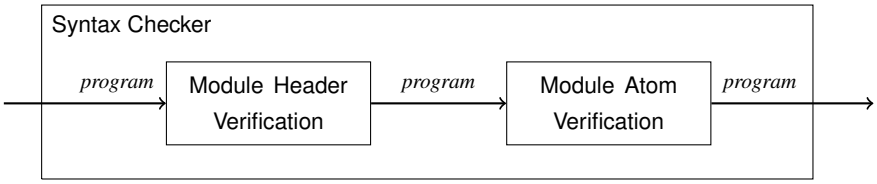


Figure 6.3: Architecture of *Syntax Checking*

When we finish in evaluating all possible answer sets, *Evaluator* sends a *request* to *Main Module Collector*, *solved = true*. If there is another main module that needs to be solved, *Main Modules Collector* will send a set of value calls, *C*, containing only the main module considered to the *Evaluator* and the evaluation process is started again. If there is no main module left, the evaluation process ends.

6.1.2 Syntax Checking

Figure 6.3 depicts the architecture of the component *Syntax Checker*, *SC*, which contains two sub-components, namely:

1. *Module Header Verification*; make sure that:

   (i) the declaration of module headers specified by users are correct, in the sense that there should be no two or more modules having the same name, and

   (ii) the arity of formal input predicates (in the module header) match with the arity of the predicates in the rule sets (of that module).

2. *Module Atom Verification*; for each module atom, make sure that:

   (i) the actual input parameters have the same arity as the corresponding formal input parameters specified in the header of the called module, and

   (ii) the arity of output predicate matches the arity of the corresponding original predicate in the called module.

Checking the syntax is done as follows:

1. The ordinary part of the program is verified by *dlvhex*.

2. The input program is passed through *Module Header Verification* component. Syntax of all module headers are verified.

3. After passed the second verification, the program is passed through *Module Atom Verification*. All module atoms, its input and output predicate, are verified.

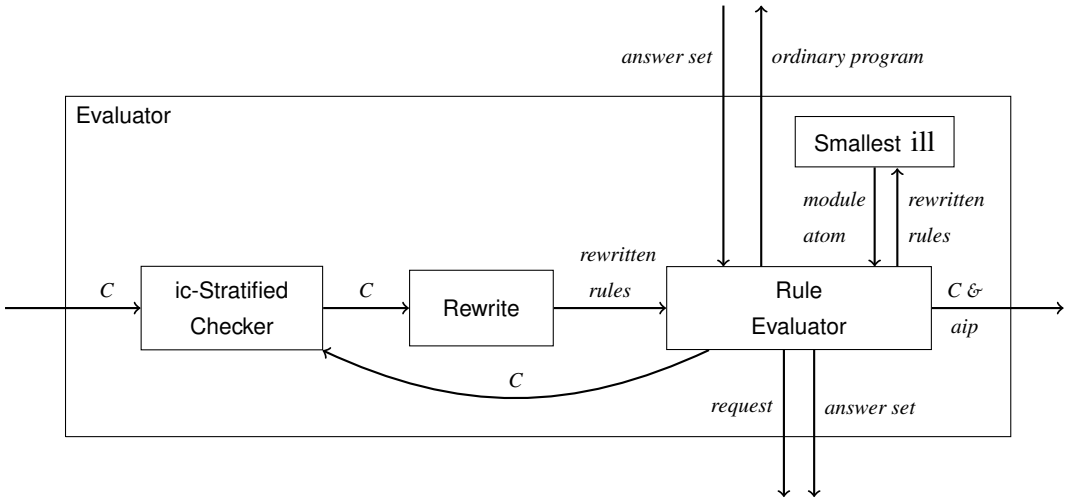


Figure 6.4: Architecture of *Evaluator*

6.1.3 Evaluator

Figure 6.4 describes the architecture of *Evaluator* which represents the core component of our MLP solver. The sub-components inside *Evaluator* are:

1. *ic-Stratified Checker*: performs value call chain analysis to make sure that the MLP provided by users is ic-stratified.

2. *Rewrite*: instead of dealing with a ground program as in Algorithm 5.3, we manage non-ground programs to have a light-weight implementation (ground programs causes an exponential increase on the data). The idea to use prefix as in [Dao-Tran et al., 2009b] to differentiate rules and atoms from different module instantiations. For a computed module atom $\alpha = P_j[\mathbf{p}].o(\mathbf{c})$ in an instantiation $P_i[S]$, instead of directly applying partial evaluation where we remove α or rules, we replace α with its output atom and add $(M_j/T)|o$ as facts (where $T = (M_i/S)|_{\mathbf{p}}^{\mathbf{q_j}}$ and $\mathbf{q_j}$ is the formal input of P_j) to the resulting rule set. In other words:

   (i) for facts and ordinary atoms: prefix the predicates with the instantiation,

   (ii) for module atoms: if this module atom has been solved before,

      i. take the output predicate, prefix it with the called instantiation, and

      ii. add facts from the answer set of the called instantiation that corresponds to the output predicates, prefixed it with the called instantiation.

   See Example 6.2 for illustration.

3. *Rule Evaluator*: analyzes the rewritten rules, decides either to pass them to the *ASP Solver* in case the rules are ordinary, or proceeds for further computation otherwise.

4. *Smallest ill*: given a set of rules R, *Smallest ill* finds a module atom which has the smallest instance local labeling rank, ill_R (see Definition 3.17).

In *Evaluator*, there is a data type that we have not discussed before, namely *rewritten rules*. This is a set of rules as the result from the *Rewrite* component described above. Example 6.2 illustrated this.

Example 6.2 Consider an MLP $\mathbf{P} = (m_0, m_1, m_2)$, where $m_0 = (main, R_0)$, $m_1 = (lib_1[p], R_1)$, $m_2 = (lib_2[p_2], R_2)$, and $R_1 = \{q(X) \leftarrow p(X), lib_2[p].r_2(a)\}$. For simplicity, we do not state R_0 and R_2 here since their contents can be disregarded for this example and instead of prefix, we will use superscript here.

Now, let us consider a set of value calls $C = \{lib_1[\{p(a)\}]\}$. The instantiation of $lib_1[\{p(a)\}]$ is:

$$\left\{ \begin{array}{lll} p(a) & \leftarrow & \\ q(X) & \leftarrow & p(X), lib_2[p].r_2(a) \end{array} \right\}.$$

From this instantiation, after *Rewrite* there are two possibilities:

(i) If module atom $lib_2[p].r_2(a)$ has not been solved yet, and for instance $M_{lib_1}/\{p(a)\} = nil$, and $M_{lib_2}/\{p_2(a)\} = nil$, then the *rewritten rules* are:

$$\left\{ \begin{array}{lll} p^{lib_1[\{p(a)\}]}(a) & \leftarrow & \\ q^{lib_1[\{p(a)\}]}(X) & \leftarrow & p^{lib_1[\{p(a)\}]}(X), lib_2[p].r_2(a) \end{array} \right\}$$

Rewrite takes the instantiation and prefixes the predicates of the fact and ordinary atoms with the instantiation. *Rewrite* does not perform further tasks.

(ii) If the module atom $lib_2[p].r_2(a)$ has been solved before, and for instance $M_{lib_1}/\{p(a)\} = \{p(a)\}$, and $M_{lib_2}/\{p_2(a)\} = \{r_2(a), r_2(b), s_2(a), s_2(b)\}$, then the *rewritten rules* are:

$$\left\{ \begin{array}{lll} p^{lib_1[\{p(a)\}]}(a) & \leftarrow & \\ q^{lib_1[\{p(a)\}]}(X) & \leftarrow & p^{lib_1[\{p(a)\}]}(X), r_2^{lib_2[\{p_2(a)\}]}(a) \\ r_2^{lib_2[\{p_2(a)\}]}(a) & \leftarrow & \\ r_2^{lib_2[\{p_2(a)\}]}(b) & \leftarrow & \end{array} \right\}$$

Since the module atom $lib_2[p].r_2(a)$ has been solved, after prefixing the fact and ordinary atoms, *Rewrite* replaces $lib_2[p].r_2(a)$ with its output predicate, $r_2(a)$. Next, $r_2(a)$ is prefixed with the called instantiation $lib_2[\{p_2(a)\}]$. And from $M_{lib_2}/\{p(a)\}$, *Rewrite* takes atoms which has predicate r_2 and put them as facts, prefixes them with $M_{lib_2}/\{p(a)\}$.

In addition, instead of managing instantiation splitting set, we manage a set of rules that still need to be solved (the *top* part). The reason behind this is because managing ground atoms is very expensive in term of space and time since the number of ground atoms could be exponential. See Algorithm 6.1 for an idea how the new rewriting approach is performed.

In Algorithm 6.1, for each module instantiation considered, one of the two cases apply:

(i) The instantiation has been partially computed before. Instead of considering the full instantiation, we consider only the *top* part of the instantiations (line 3 – 4). Intuitively, Top_i/S contains the unsolved part of instantiation $P_i[S]$

Algorithm 6.1: *rewrite*(in: $C, \mathbf{M}, \mathbf{A}, \mathbf{Top}$) : R

Input: set of value calls C, partial model $\mathbf{M}$, indexed set of sets of module atoms $\mathbf{A}$, indexed set of sets of rules $\mathbf{Top}$

Output: **return** set of rules R

1 $R = \emptyset$
2 **foreach** $P_i[S] \in C$ **do**
3 **if** $Top_i/S \neq nil$ **then** /* collect top */
4 $R = R \cup Top_i/S$
5 **else**
6 prefix $I_{\mathbf{P}}(P_i[S])$ with $P_i[S]$, add the result to R
7 **if** $M_i/S \neq nil$ **then** /* add an answer set of bottom */
8 prefix M_i/S with $P_i[S]$, add the result as facts to R
9 **foreach** *module atom* $P_j[\mathbf{p}].o(\mathbf{t}) \in R$ **do**
10 **if** $P_j[\mathbf{p}].o(\mathbf{t}) \in A_i/S$ **then** /* replace module atoms */
11 $T = (M_i/S)|_{\mathbf{p}}^{\mathbf{q_j}}$ and $\mathbf{q_j}$ is the formal input parameter of P_j
12 replace $P_j[\mathbf{p}].o(\mathbf{t})$ with $prefixed(o(\mathbf{t}), P_j[T])$
13 prefix $(M_j/T)|_o$ with $P_j[T]$, add the result as facts to R
14 **return** R

(ii) Otherwise, we consider the complete (prefixed) instantiation of the calls only when we have not evaluate these instantiation before (line 6). Even though we assume that the set of predicates in each module is disjoint, it could be the case that we consider more than one instantiations from the same module, i.e., $P_i[S]$ and $P_i[T]$ where $S \neq T$. Hence, the idea to prefix predicates is needed to differentiate which rules and atoms come from which instantiation.

Next, in line 7 – 8, we add the partial model (as the answer set of the *bottom* part) we found so far. We also make sure that for each module atom that had been evaluated before, the result is added (line 11 – 13):

- replace the module atoms with the their output atom (line 12), and
- add the interpretation of the call instantiation produced by those module atoms (line 13).

The following example illustrates the rewriting process involving prefixes and storing the *top*.

Example 6.3 Let us consider a simple MLP $\mathbf{P} = (m_0, m_1)$, where $m_0 = (P_0, R_0)$, and $m_1 = (P_1[q_1], R_1)$, where:

$$R_0 = \left\{ \begin{array}{rcl} q_0(a) & \leftarrow & \\ out_0(X) & \leftarrow & P_1[q_0].out_1(X) \end{array} \right\}$$

$$R_1 = \left\{ \begin{array}{lcl} s_1(a) & \leftarrow & \\ out_1(X) & \leftarrow & s_1(X), q_1(X) \end{array} \right\}$$

In the beginning of the computation with:

- $C = \{P_1[\emptyset]\}$
- $path = \epsilon$

In addition we have **Top** as an indexed set of sets of rules which will be used to store *top* rules. Let us start from the beginning of the computation with **Top**, **M** (partial model), and **A** (indexed set of sets module atoms) to have *nil* at all components (recall **M** and **A** as in Algorithm 5.1).

Initially, in line 1 we set $R = \emptyset$, then *rewrite* will proceed as follows:

line 3 – 6 Since $Top_0/\emptyset = nil$ then we add the prefixed instantiation of $P_0[\emptyset]$.

$$R = \left\{ \begin{array}{lcl} q_0^{P_0[\emptyset]}(a) & \leftarrow & \\ out_0^{P_0[\emptyset]}(X) & \leftarrow & P_1[q_0].out_1(X) \end{array} \right\}$$

line 7 We do not add anything to R since $M_0/\emptyset = nil$

line 9 We have one module atom $P_1[q_0].out_1(X)$ in R.

line 10 Since $A_0/\emptyset = nil$, we do not proceed further.

Since C contain only one element, $P_0[\emptyset]$, *rewrite* stops here and returned R as the result.

Next, since R as the result of *rewrite* is not ordinary the evaluation will continue to evaluate the module atom in R, $P_1[q_0].out_1(X)$. The input splitting set for $P_1[q_0].out_1(X)$ is evaluated, $A_0/\emptyset$ is updated to $\{P_1[q_0].out_1(X)\}$, and $Top_0/\emptyset$ is updated to $\{out_0^{P_0[\emptyset]}(X) \leftarrow P_1[q_0].out_1(X)\}$. The evaluation proceed with computing module call $P_1[\{q(a)\}]$.

Now, let us assume that the computation of module call to $P_1[\{q(a)\}]$ is completed and we are back to $P_0[\emptyset]$, with:

- $C = \{P_1[\emptyset]\}$
- $path = \epsilon$
- $M_0/\emptyset = \{q_0(a)\}$,
  $M_1/\emptyset = \{s_1(a), q_1(a), out_1(a)\}$
- $A_0/\emptyset = \{P_1[q_0].out_1(X)\}$
  $A_1/\{q_1 a\} = fin$
- $Top_0/\emptyset = \{out_1^{P_1[\emptyset]}(X) \leftarrow P_1[q_0].out_1(X)\}$
  $Top_1/\{q_1 a\} = \emptyset$

Please note that since in *rewrite* we do not ground the rules, we can have non-ground module atoms in **A**. Initially we set $R = \emptyset$, then *rewrite* will proceed as follows:

line 3 Since $Top_0/\emptyset \neq nil$, then $R = R \cup \{out_1^{P_0[\emptyset]}(X) \leftarrow P_0[q_0].out_1(X)\}$, yields $R = \{out_1^{P_0[\emptyset]}(X) \leftarrow P_0[q_0].out_1(X)\}$

line 7 – 8 We prefix $M_0/\emptyset$ with $P_0[\emptyset]$. Then, we have $R = R \cup \{q_0^{P_0[\emptyset]}(a)\}$, hence:

$$R = \left\{ \begin{array}{rcl} out_1^{P_1[\emptyset]}(X) & \leftarrow & P_1[q_0].out_1(X) \\ q_0^{P_1[\emptyset]}(a) & \leftarrow & \end{array} \right\}.$$

line 9 We have $P_1[q_0].out_1(X)$ in R.

line 10 We have $P_1[q_0].out_1(X) \in A_0/\emptyset$.

line 11 $T = \{q_1(a)\}$.

line 12 We take the output atom from $P_1[q_0].out_1(X)$, and prefix it with $P_1[\{q_1(a)\}]$. This yields $out_1^{P_1[\{q_1(a)\}]}(X)$. Then, we replace $P_1[q_0].out_1(X)$ with $out_1^{P_1[\{q_1(a)\}]}(X)$. Hence:

$$R = \left\{ \begin{array}{rcl} out_0^{P_1[\emptyset]}(X) & \leftarrow & out_1^{P_1[\{q_1(a)\}]}(X) \\ q_0^{P_1[\emptyset]}(a) & \leftarrow & \end{array} \right\}.$$

line 13 We take $(M_1/\{q_1(a)\})|_{out_1}$, prefix it with $P_1[\{q_1(a)\}]$, and add it to R. Then:

$$R = \left\{ \begin{array}{rcl} out_0^{P_1[\emptyset]}(X) & \leftarrow & out_1^{P_1[\{q_1(a)\}]}(X) \\ q_0^{P_1[\emptyset]}(a) & \leftarrow & \\ out_1^{P_1[\{q_1(a)\}]}(a) & \leftarrow & \end{array} \right\}$$

In the end, we return R as the result.

As a summary, the *Evaluator* works as follows:

- In the beginning of the evaluation *ic-Stratified Checking* component always makes sure that we work on ic-stratified. If the program is not ic-stratified, we simply stop the evaluation process.
- Next, we rewrite the rules, add additional facts from previous computation (all is done by the *Rewrite* component).
- Then, *Rule Evaluator* evaluates the *rewritten rules*. If in this phase an answer set is found, the answer set is sent to the *Output Printer*. However, if further computation is needed, we go back to *ic-Stratified Checking* passing the next value call C to be evaluated.

6.2 Input/Output Format

In this section we show how to specify an MLP as an input to *TD-MLP* and how to interpret the returned result.

6.2.1 Input

We base our syntax on *dlvhex* syntax[4] and explain here only the extension with respect to MLP. A module requires a header and the corresponding rule set. More than one module can be specified. In addition, an input file can also contain one or more modules.

- *module header*: consists of module name and formal input parameter. Each module can have zero, one, or more formal input parameter.

  ```
  #module(<MODULENAME>, [<PRED>/<ARITY>, ...]).
  ```

  Example:

  - Module *graphR* with empty input parameter:
    `#module(graphR, []).`
  - Module *modReachable* with two input parameters (predicate p which has arity one and q which has arity two):
    `#module(modReachable, [p/1, q/2]).`

- *module atom*: consist of module name, (possibly zero, one, or more) input predicates, and an output atom.

  ```
  @<MODULENAME>[<INPUTPREDICATE, ...>]::<OUTPUTATOM>
  ```

  Example: Suppose that r is a 1-arity predicate, s is a 2-arity predicate, and *outB* is an atom in the rule set of module *modReachability*, then we can make a call to *modReachability* with r and s as inputs and get the truth value of *outB* by specifying:

  ```
  @modReachable[r,s]::outB
  ```

Example 6.4 and 6.5 give a clear idea of correct inputs for *TD-MLP*.

Example 6.4 Suppose that we have an MLP $\mathbf{P} = (m_1, m_2)$, where: $m_1 = (graphR, R_1)$, $m_2 = (modReachable[first, edge], R_2)$, and

$$R_1 = \left\{ \begin{array}{rcl} firstVertex(a) & \leftarrow & \\ edge(a,b) & \leftarrow & \\ edge(b,c) & \leftarrow & \\ edge(c,d) & \leftarrow & \\ ok & \leftarrow & modReachable[firstVertex, edge].reachable(c) \end{array} \right\},$$

$$R_2 = \left\{ \begin{array}{rcl} reachable(X) & \leftarrow & first(X) \\ reachable(X) & \leftarrow & reachable(X), edge(X,Y) \end{array} \right\}.$$

Actually, R_1 represents a graph showed in Figure 6.5. In the fifth rule of R_1 we perform a test whether vertex c is reachable from a (note that predicate *firstVertex* contain a only). To perform

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

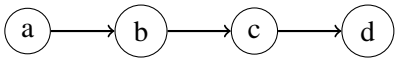


Figure 6.5: Simple graph

this task, this rule calls *modReachable*. We pass *firstVertex* and *edge* as the inputs. Then, module *modReachable* computes the reachability based on these inputs and returns the answer via the output predicate *reachable*(c) We can see from the graph presented in Figure 6.5 that indeed c is reachable from a.

A representation of **P** in a correct *TD-MLP* input format will be:

```
#module(graphR,[]).
      firstVertex(a).
      edge(a,b).
      edge(b,c).
      edge(c,d).
      ok :- @modReachable[firstVertex,edge]::reachable(c).

#module(modReachable, [first/1, edge/2]).
      reachable(X) :- first(X).
      reachable(Y) :- reachable(X), edge(X,Y).
```

Example 6.5 The MLP showed in Example 6.1 can be written (in a correct format) as:

```
#module(main,[]).
      p(a) v p(b).
      getA :– @lib[p]::q(a).
#module(lib,[r/1]).
      q(X) :– r(X).
```

6.2.2 Output

To display an output of *TD-MLP*, we print each answer set in a separate line, and inside parantheses. An example of an output of *TD-MLP* can be seen in Example 6.6.

Example 6.6 The MLP showed in Example 6.1 has two answer sets, namely:

- $\mathbf{M}^1$, where $M^1_{main}/\emptyset = \{p(b)\}$, and $M^1_{lib}/\{r(b)\} = \{r(b), q(b)\}$, and
- $\mathbf{M}^2$, where $M^2_{main}/\emptyset = \{p(a), getA\}$, and $M^2_{lib}/\{r(a)\} = \{r(a), q(a)\}$.

When the input is provided to *TD-MLP* in the correct format as in Example 6.5, we get the result as shown below:

```
(main[{}]={p(b)}, lib[{r(b)}]={r(b),q(b)})

(main[{}]={p(a),getA}, lib[{r(a)}]={r(a),q(a)})
```

Each answer set consists of <INSTANTIATION>={<INTERPRETATION>}. For example, in the first answer set above, we have two instantiations, i.e., main[{}] and lib[{p(b)}]. {p(b)} is the interpretation of main[{}] and {p(b),q(b)} is the interpretation of lib[{p(b)}]. Another example of the result produced by *TD-MLP* can be seen in Example 6.7.

Example 6.7 The MLP about reachability in graphs presented in Example 6.4 has an answer set: **M**, where:

$$M_{graphR}/\emptyset = \left\{ \begin{array}{l} firstVertex(a), \\ edge(a,b), edge(b,c), edge(c,d), \\ ok \end{array} \right\}, \text{and}$$

$$M_{modReachable}/\{first(a), edge(a,b), edge(b,c), edge(c,d)\} = \left\{ \begin{array}{l} first(a), \\ edge(a,b), edge(b,c), edge(c,d), \\ reachable(a), reachable(b), reachable(c), reachable(d) \end{array} \right\}$$

The answer set that we got from *TD-MLP* from solving its representation in Example 6.4 is again intuitively similar:

```
(graphR[{}]={firstVertex(a),edge(a,b),edge(b,c),edge(c,d),ok},
modReachable[{first(a),edge(a,b),edge(b,c),edge(c,d)}]={first(a),edge(a,b),
edge(b,c),edge(c,d),reachable(a),reachable(b),reachable(c),reachable(d)})
```

6.3 Parameters

Dlvhex system is executed using command line:

```
dlvhex [OPTION] FILENAME [FILENAME ...]
```

Argument inside [...] is optional. In case an MLP is written in several files, then all of the files should be listed under FILENAME.

There are several parameters in the system that is relevant to our MLP solver.

- --mlp: activates MLP mode.
- --num=<N>: specifies the maximal number N of answer sets expected. if --num=0 or not specified, then *TD-MLP* returns all answer set.
- --solver=<SOLVER>: specifies the ASP solver to be used (depends on systems availability). For example: --solver=libclingo or --solver=dlv. *DLV* is used by default.
- --split: activate instantiation splitting optimization.

For example, executing:

```
dlvhex --mlp simple.mlp
```

will give us all of answer sets of an MLP in file simple.mlp, while:

```
dlvhex --mlp --split --num=3 one.mlp two.mlp
```

will make *TD-MLP* using instantiation splitting optimization in its computation and return at most 3 answer sets of an MLP formed by modules specified in `one.mlp` and `two.mlp`.

6.4 Usage

In this section we will give another example of solving practical mathematical problem using MLP, namely compares the cardinality between two sets. This problem could be represented easily in MLP by single out an element from each set one by one without introducing any specific ordering mechanisms. Listing 6.1 and 6.2 illustrate this:

- In `main.mlp`, two sets are specified: `q` and `r`. Then, module `cardinality` is called with `q` and `r` as the input parameters.
- In `cardinality.mlp`, `q1` and `q2` are the formal input predicates which receive input from `q` and `r` (from module `main`) respectively. Then we single out one element from `q1` and `q2`, put it into `q1i` and `q2i` respectively. We continue by calling `cardinality` recursively with sets whose cardinality have been already decreased by one (`q1i` and `q2i`). This proceeds until there is no element in either set. If both sets have no element in the same step, we will have `equal` in hand. Otherwise, `equal` will not be derived. If it has been derived, `equal` is transferred backwards along the calling chain to deliver the final result.

Listing 6.1: main.mlp

```
#module(main, []).
q(a). q(b).
r(a). r(b).
equalQR :- @cardinality[q,r]::equal.
notEqualQR :- not equalQR.
```

Listing 6.2: cardinality.mlp

```
#module(cardinality, [q1/1, q2/1]).
q1i(X) v q1i(Y) :- q1(X), q1(Y), X != Y.
q2i(X) v q2i(Y) :- q2(X), q2(Y), X != Y.
skip1 :- q1(X), not q1i(X).
skip2 :- q2(X), not q2i(X).
equal :- skip1 , skip2 , @cardinality[q1i,q2i]::equal.
equal :- not skip1 , not skip2.
```

Using command:

```
dlvhex --mlp main.mlp cardinality.mlp
```

we will have `equalQR` in all answer sets:

- (main[{}]={q(a),q(b),r(a),r(b),equalQR},
  cardinality[{q1(a),q1(b),q2(a),q2(b)}]={q1(a),q1(b),q2(a),q2(b),skip1,skip2,
  equal,q2i(b),q1i(b)},
  cardinality[{q1(b),q2(b)}]={q1(b),q2(b),skip1,skip2,equal},
  cardinality[{}]={equal})
- (main[{}]={q(a),q(b),r(a),r(b),equalQR},
  cardinality[{q1(a),q1(b),q2(a),q2(b)}]={q1(a),q1(b),q2(a),q2(b),skip1,skip2,
  equal,q1i(b),q2i(a)},
  cardinality[{}]={equal},
  cardinality[{q1(b),q2(a)}]={q1(b),q2(a),skip1,skip2,equal})
- (main[{}]={q(a),q(b),r(a),r(b),equalQR},
  cardinality[{q1(a),q1(b),q2(a),q2(b)}]={q1(a),q1(b),q2(a),q2(b),skip1,skip2,
  equal,q2i(b),q1i(a)},
  cardinality[{}]={equal},
  cardinality[{q1(a),q2(b)}]={q1(a),q2(b),skip1,skip2,equal})
- (main[{}]={q(a),q(b),r(a),r(b),equalQR},
  cardinality[{q1(a),q1(b),q2(a),q2(b)}]={q1(a),q1(b),q2(a),q2(b),skip1,skip2,
  equal,q2i(a),q1i(a)},
  cardinality[{}]={equal},
  cardinality[{q1(a),q2(a)}]={q1(a),q2(a),skip1,skip2,equal})

CHAPTER 7

Experiments

In this chapter we show empirical evaluations that have been done on *TD-MLP*. We evaluate the performance of the solver with and without instantiation splitting implementation. The chapter starts by explaining how the system is set up then followed by several evaluation settings. First, we show the evaluation on abstract and random programs. Next, we run experiments on well-known problems with different encodings, i.e. modular and non-modular ones. We run all of our experiment on a machine with Intel Xeon 3.00GHz quad-core processor and 16GB RAM. The system is running on Ubuntu 10.10.

7.1 Random Programs

Experiments on random program are motivated by the curiosity to see how *TD-MLP* behaves on such unexpected program. The experiments in general involves several call pattern settings which consists of several parameter settings, such as the number of modules involved, the number of constants and predicate symbols, negation as failure, and disjunctive or non-disjunctive case.

7.1.1 Experiment Characteristics

On experiments random programs experiments we introduce several different settings on how modules call each other. Given an MLP $\mathbf{P} = (m_0, \ldots, m_n)$:

- *line*: The call is always generated from m_i to m_{i+1} where $0 \leq i < n$, and from m_n to m_n (in order to demonstrate the ability of the algorithm to evaluate an instantiation loop). Figure 7.1(a) gives an example for a *line* pattern for $n = 4$.

- *ring*: This module call pattern is similar to *line* pattern, except that in this case m_n calls m_0. See Figure 7.1(b) for an example of a *ring* call pattern with $n = 4$

- *diamond*: One diamond-shape is actually created by 4 modules. If the number of modules is more than 4, then a chain of diamonds is created. So, given $1 \leq i \leq n$:
  - if i can be divided by 3, then m_i has two module calls, namely to m_{i+1} and m_{i+2} (to create the branching mechanism).
  - if $i - 1$ can be divided by 3, then m_i has one module call to m_{i+2}.
  - for $i \geq 2$, if $i - 2$ can be divided by 3, then m_i has one module call to m_{i+1}.

  Please note that whenever we create a call to m_j where $j > n$, we cancel the call creation, and create a self-recursive call instead. Figure 7.1(c) gives an illustration of a diamond call pattern with $n = 6$.

- *star*: a pattern which from a main module, all library modules are called, and library modules create a recursive call to itself:
  - in m_0 we create module calls to $m_1, \ldots, m_{n-1}$
  - in $m_1, \ldots, m_n$ there is only one module call in each module, to itself.

  An example of star call pattern with $n = 4$ can be seen in Figure 7.2(a).

- *tree*: given a specific parameter *nChild*, we create a tree-shaped call in which each node has at most *nChild* children. In Figure 7.2(b) we can see an example of tree-shaped call pattern with $nChild = 3$ and $n = 6$. Since $m_2, \ldots, m_6$ do not have any child, we create a recursive call for them.

- *random*: the existence of module calls between each module m_i to m_j, where $0 \leq i, j \leq n$ is created based on a random number generator and *density* parameter. The range of *density* parameter is between 0 and 1. The higher the *density* parameter, the more likely an edge from m_i to m_j created. In this pattern we cannot really know how the graph looked like, in terms of which module calls which module. Because, apart from the *density* parameter, it also depends on the random number generator. However, Figure 7.3 provide an illustration of one example of this pattern generated with $n = 4$ and $density = 0.25$.

Apart from the module call pattern, we also do the evaluation with several different parameters:

- c: the number of constant symbols on each modules, where $c \in \{10, 20, 30, 40\}$.
- p: the number of predicate symbols on each modules, where $p \in \{10, 20, 40\}$.
- v: the existence of disjunctive head and negation as failure, where $d \in yes, no$
- m: the number of modules, where $m \in \{5, 10, 15, 20, 25\}$
- other special parameters:

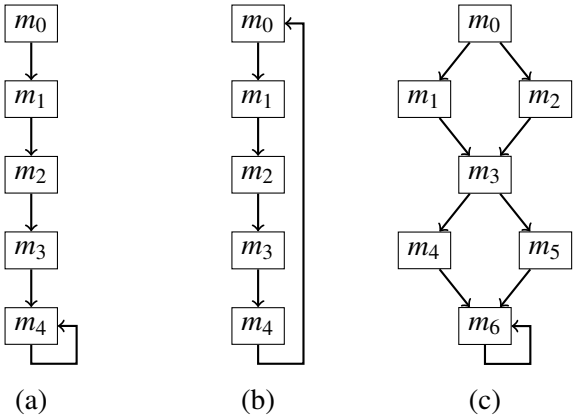


Figure 7.1: Module call pattern: (a) line, (b) ring, (c) diamond

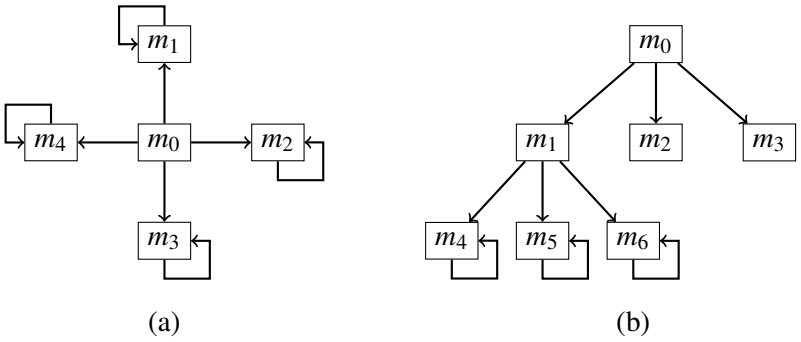


Figure 7.2: Module call pattern: (a) star, (b) tree

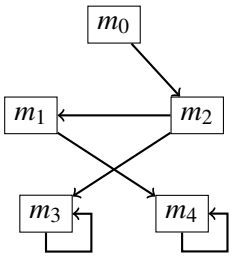


Figure 7.3: Module call pattern: random

– *nChild*: the number of children for each node in the tree pattern, where $nChild \in \{2, 3, 5\}$, and

– *density*: the percentage of whether a call is made from a module to another module, and from a module to itself. $density \in \{0.10, 0.15, 0.20, 0.25\}$. This parameter exists only in *random* pattern).

Table 7.1: Detail information from experiments on finding the first answer set of a program over different call patterns

Call Pattern	$V(CG_{\mathbf{P}}(\mathbf{M}))$	$\lvert\mathbf{M}\rvert$	# Call to ASP solver
line	11.87	808.11	23.74
ring	11.25	771.03	12.25
diamond	13.60	876.27	31.76
star	17.66	976.60	44.57
tree	15.84	951.35	38.28
random	5.50	520.00	11.50

Let $\mathbf{M}$ be an answer set of an MLP $\mathbf{P}$ run on experiments,
column $V(CG_{\mathbf{P}}(\mathbf{M}))$ represents the average number of vertex in $CG_{\mathbf{P}}(\mathbf{M})$,
column $\lvert\mathbf{M}\rvert$ represents the average number of ground atom that is set to true in $\mathbf{M}$.

The value of each parameter is chosen in such a way that we can observe the behavior of *TD-MLP* from small and simple programs to very complicated and large programs.

7.1.2 Experiment Results

During the experiments, we run *TD-MLP* with its default setting (without instantiation splitting optimization), and with instantiation splitting optimization (activates `--split` option). Figure 7.4 shows the time spent by *TD-MLP* to get the first answer set on overage over all parameter setting on each call pattern. It turned out that turning on instantiation splitting optimization gives us better evaluation time in general.

Table 7.1 gives us more detailed description about what is going on in the experiments. The table gives us different information obtained on average over different parameter settings when *TD-MLP* found the first answer set of a program. Intuitively, the first column represents the average number of module instantiations created, the second column represents the average number of ordinary ground atoms set to true in the first answer set, and the third column is the average number of calls made to the ASP solver. One can see these data provide us more fine-grained information on how the evaluation time on such pattern is related to such information. From Table 7.1 we can see that the time spent on a call pattern is relatively higher than the other when the number of module instantiations, ordinary atoms considered, and also calls made to the ASP solver higher.

In addition, we also investigated the relation between the number of modules to be solved and the performance of our solver. We run the experiment over all call patterns. The result is depicted in Figure 7.5. The evaluation time showed is the average of time spent by *TD-MLP* to produce an answer set, over all call patterns. As expected, the evaluation time increases with the increasing number of modules to be solved in general. However, in this setting instantiation splitting optimization also outperforms the default setting. The interesting point here is that the larger number of modules we considered, the more performance we gained from instantiation splitting optimization. This is a good sign since it means that the bigger program we have, the more advantage we obtain from applying the optimization.

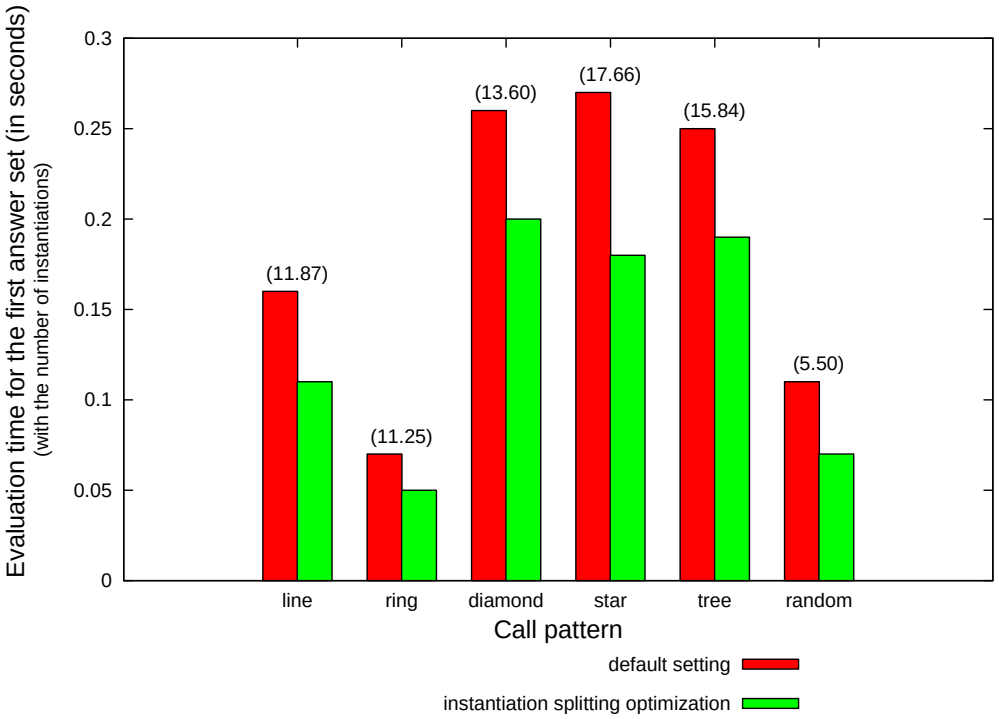


Figure 7.4: Experiment result on different call pattern

One might also curious on how *TD-MLP* scales, in terms on the time spent by *TD-MLP* to find an answer set one after the other. For this purpose, we stored the time spent by *TD-MLP* to output each answer set. As one can see from Figure 7.6 that *TD-MLP* behaves nicely with the increasing number of answer found. The evaluation time increases linearly by the number of answer set found.

7.2 Hanoi Tower

"Hanoi Tower" is a classical problem on search. Given 3 pegs and several disks, the goal is to move all disks from the left most peg to the right most peg with the help of middle peg. Initially all disks are in the left most peg, and placed in an order such that the bigger disk always placed below the smaller disk. The rules are:

1. move one disk at a time.

2. only the top disk on a peg can be moved.

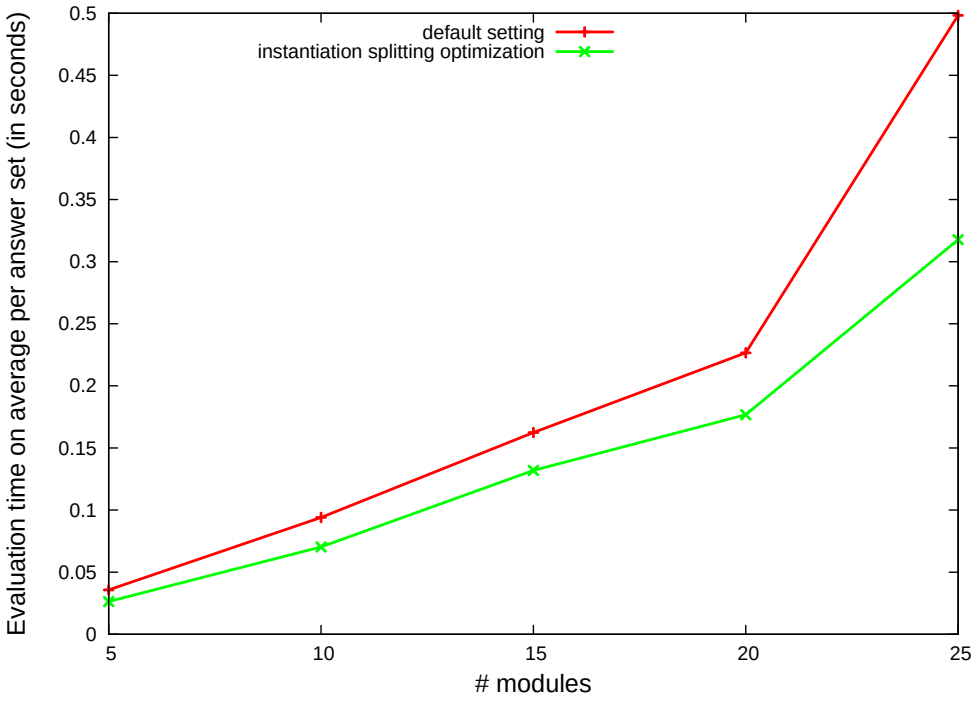


Figure 7.5: Evaluation time with increasing number of modules

3. larger disk cannot be placed on top of a smaller one.

It is known that, for a classical Hanoi tower problem with n disks, the plan of moving all n disks from the left-most peg to the right-most peg consists of $2^n - 1$ moves. In this experiment, we set 3 pegs: a, b, and c. With a as the left most, c as the right most, and b as the middle peg. For disks, we use different setting of disks: 2, 3, 4, and 5.

In this experiment, we run not only *TD-MLP* with several different setting but also *DLV*[1] and *dlvhex*[2] as comparison. The encoding for this problem is taken from [Cabalar, 2011] with minor modification (so that it can be run under *DLV* and *dlvhex*). The reader can take a look at the encoding in Appendix B.1.1 and B.1.2 for ordinary ASP and MLP, respectively.

In addition to *DLV*, in this experiment we also run *TD-MLP* with *clingo*[3] as the ASP solver. Below is the complete list of *TD-MLP* setting run for this experiment:

(a) solver used: *DLV*; instantiation splitting optimization: *no*.

[1] http://www.dbai.tuwien.ac.at/proj/dlv/

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

[3] http://potassco.sourceforge.net/

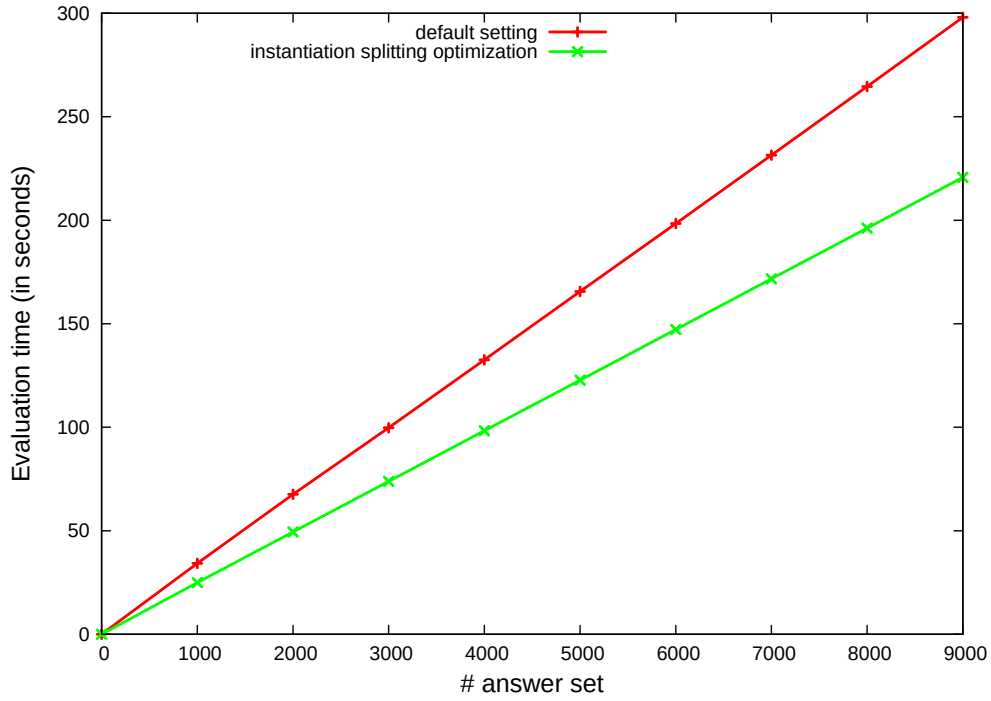


Figure 7.6: Time vs answer set on *tree* call pattern

(b) solver used: *DLV*; instantiation splitting optimization: *yes*.

(c) solver used: *clingo*; instantiation splitting optimization: *no*.

(d) solver used: *clingo*; instantiation splitting optimization: *yes*.

The result of the experiment can be seen in Figure 7.7. As the result and previous experiments suggest, we gain better or at least the same time evaluation using instantiation splitting optimization. The performances of *TD-MLP* compare to *DLV* and *dlvhex* are also not so far behind. This is a good sign although it is not surprising since the encodings of the problem (ordinary and MLP case) are nearly the same. However, one can spot the most interesting cases here is that the performance of *TD-MLP*(c) and *TD-MLP*(d) are far better than *TD-MLP*(a) and *TD-MLP*(b) and even *DLV*, and *dlvhex*. This is due to the different setting of the solver. In this particular case, *clingo* performs better.

Experiment with 4 and 5 disks justify our decision not to fix the choice of our ASP solver into one particular solver (the reason behind providing parameter `--solver` for the system). The experiment suggests that changing the ASP solver used from *DLV* to *clingo* gives us a huge

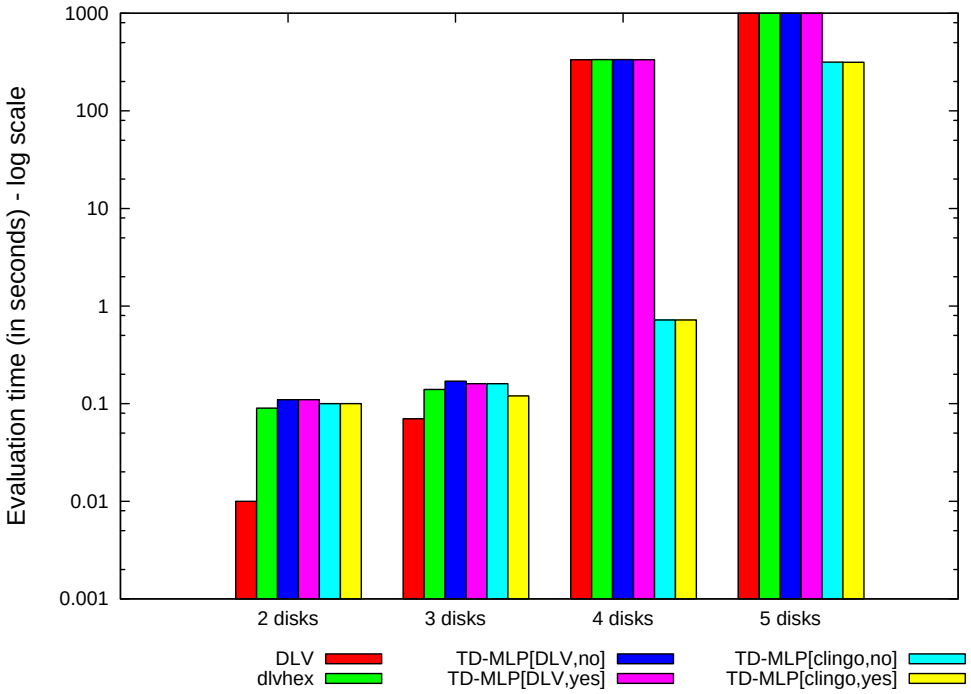


Figure 7.7: Hanoi tower experiments

evaluation time difference for this particular problem. Since there many ASP solvers currently being independently improved by its developer, experimenting with different solvers could be a hint to gain better performance, especially when our favorite ASP solver perform surprisingly bad on a particular case.

7.3 Packing Problem

The problem (and its encoding in ASP syntax) is taken from the ASP Competition 2011 [4]. Packing problem consists of a rectangular area and a set of squares in which require us to pack all squares into the rectangular area without any overlapping squares. The dimension of the rectangular area and each square is given. White space left in the rectangular area is allowed.

The original encoding from the competition has been slightly modified in order to conform with *dlvhex* syntax. An interested reader can take a look the code at Appendix B.2.1. Then we compare the execution time of solving this problem using *DLV*, *dlvhex*, and *TD-MLP*. For MLP

[4] https://www.mat.unical.it/aspcomp2011/

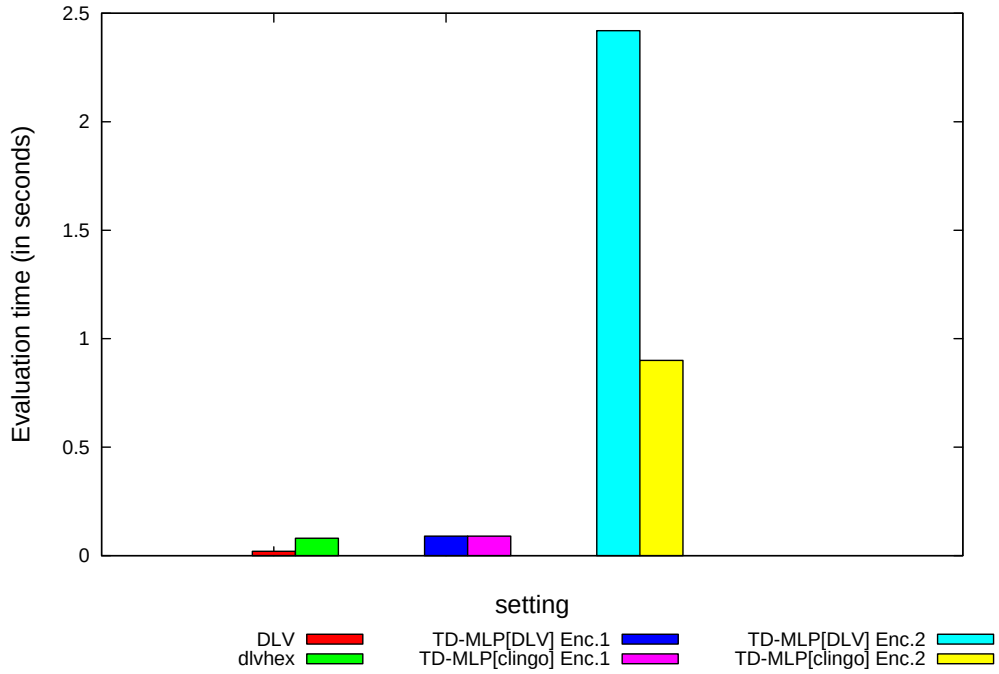


Figure 7.8: Execution time for Packing Problem

setting, we had created two encodings which can be seen in Appendix B.2.2 (*encoding 1*) and B.2.3 (*encoding 2*). The main idea is to differentiate between the input part and the problem solving part.

From the experiment results, *dlvhex* and *TD-MLP* (*encoding 1*) run a little bit slower than *DLV*. This could be understood since both of them actually utilize *DLV* in their computation. The time difference between them and *DLV* could be regarded as the overhead occurring when they call *DLV*.

Since *encoding 1* does not really differ that much from the original ASP encoding, its execution time also does not have a big difference, as expected. But this is not the case with *encoding 2*. We see an obvious different evaluation time between *encoding 1* and *encoding 2*. The big difference basically caused by module *generatePos* in *encoding 2*. This module creates a choice point (disjunction case) that forces our solver to make a call to *DLV* many times (to combine the output from *generatePos* and module *solvePacking*). However, using *clingo* in this setting we could reduce the evaluation time for *encoding 2* for more than 50%.

Table 7.2: Even-Odd experiments using *TD-MLP*

$\|q\|$	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$	$\|\mathbf{M}\|$	# Call to ASP solver	Evaluation time (in seconds)
20	23	553	45	0.16
40	43	1893	85	0.64
60	63	4033	125	1.74
80	83	6973	165	3.99
100	103	10713	205	7.62
120	123	15253	245	13.40
140	143	20593	285	22.38
160	163	26733	325	34.93
180	183	33673	365	52.35
200	203	41413	405	73.63

Let $\mathbf{M}$ be an answer set of an MLP $\mathbf{P}_E$ run on experiments,
column $V(CG_{\mathbf{P}_E}(\mathbf{M}))$ represents the average number of vertex in $CG_{\mathbf{P}_E}(\mathbf{M})$,
column $|\mathbf{M}|$ represents the average number of ground atom that is set to true in $\mathbf{M}$.

7.4 Even-Odd

Finally, we tested the performance of *TD-MLP* on Even-Odd program in Example 2.25. Even-Odd program, as we know, tests whether a set q has an even or odd number of elements. The idea of the experiment is to increase the number of elements (constant symbols) in q, and then observe the evaluation time needed to get the first answer set of the program, how much time does *TD-MLP* need to solve the problem with increasing number of elements in the set.

In this experiment, we fixed the system setting to:

(i) instantiation splitting optimization is activated, and

(ii) *clingo* is used as the backbone solver.

This is due to the fact that the default setting took much more time than the setting described previously. For example, the default setting took around 50 seconds to solve Even-Odd program with 100 elements in q, compare to only less than 10 seconds using the setting with *clingo* and instantiation splitting optimization.

The details of the experiment result is given in Table 7.2. One can see how the number of instantiations, the number of ordinary ground atoms considered, and called to *clingo* grow as the number of constant symbols in q increases.

The number of instantiation grows with only three difference from the number of constants in q. This is due to the nature of Even-Odd program which single-out the elements in q one by one, creating n instantiation, where n is the number of elements considered in q. The other three instantiations: one instantiation is created for the main module in the beginning, and two instantiations for P_2 and P_3, in the form of empty input, are created when all elements has been

singled-out (then they form a cycle of value call). Please recall that P_1 is the main module, P_2 and P_3 alternately test whether the number of elements is even or odd respectively.

The number of call to the ASP solver is almost doubled compared to the instantiation since we need to evaluate each instantiation twice (except the last one). One call to evaluate the *bottom* (to provide an input for the module call) and one call after one module call has been solved. Note that in Even-Odd program, each module has exactly one module atom.

From Table 7.2, we also see that the evaluation time grows rapidly with the increasing number of constants (elements in q) considered. This can be understood since not only the number of call to the answer set solver grew, but also because we always sent a program with bigger Herbrand base (as the number of constants increases) to the answer set solver, which then caused a bigger ground instance, and longer time is needed for evaluations.

Looking at such dramatic increase in the time spent by *TD-MLP*, it triggered our curiosity to do the experiment on an ordinary ASP solver as well. For this purpose, we create an encoding for Even-Odd program in ordinary ASP.

Listing 7.1: Even-Odd program with ordinary-monolithic encoding

```
% put q here: q(a). q(b). ...
% (a) create a pair
p(X,Y) v p2(X,Y) :- q(X), q(Y), X!=Y.
% (b) make sure that the element is unique
:- p(X,Y), p(X,Z), Y!=Z.
:- p(X,Y), p(Z,X), Y!=Z.
:- p(Y,X), p(X,Z), Y!=Z.
:- p(Y,X), p(Z,X), Y!=Z.
% (c) mark which element is in
in(X) :- p(X,Y).
in(Y) :- p(X,Y).
% (d) cannot be two elements out in the same time
:- p2(X,Y), not in(X), not in(Y).
% (e) get the result
odd :- q(X), not in(X).
even :- not odd.
```

Given a predicate q with some elements, then the idea behind the program in Listing 7.1 is to:

(a) Put all elements of q in a set of two element (pair). In the program above p represent the set of pair that we want, and p_2 represent outliers.

(b) Make sure that once an element has been put into a pair, it cannot be appear in another pair.

(c) Mark whether an element from q has been put into p.

(d) Force that at least only one element out of p. Because, if there is two elements out of p, then actually both of them can form a pair. This requirement is required so that p is condense. In other words, if q has even number of elements, all of them will be in the pair.

(e) The result is *odd* if there is one element of q that is not in p. Otherwise, it is *even*.

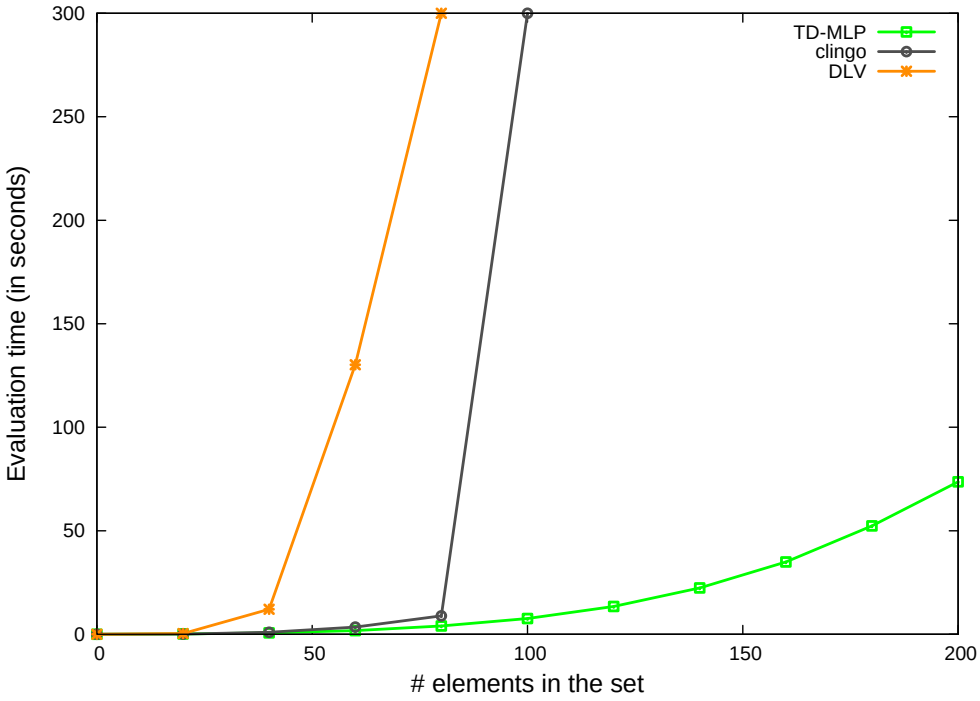


Figure 7.9: Evaluation time comparison on Even-Odd program on *TD-MLP*, *clingo*, and *DLV*

In addition to the *pairing* solution in Listing 7.1, we also create another ordinary encoding for Even-Odd problem (see Appendix B.3.2). Suppose that we want to know whether the set q has an even or odd number of elements, then the idea of this encoding is:

(a) Given a set of label with successor relation, we assign those label to each element of q and make sure that each element of q got exactly one label.

(b) We assign each element as an *odd* or *even* element based on the successor relations of the label. We start by defining the element which has the smallest label as *odd*. Then, we define the element which has the label which as the successor of the smallest label as *even*. We continue this labeling process onto each element alternately between *odd* and *even* until all elements in q has been labeled.

(c) The final result is taken from the element which has the largest label. If this element is assigned to *odd*, then the final result is *odd*. Otherwise, the final result is *even*.

This *labeling* solution, actually is an approach to solve Even-Odd problem that is more similar to our MLP encoding (compare to the encoding in Listing 7.2). It alternately assigns

Table 7.3: The evaluation time for solving Even-Odd problem with ordinary encoding (in seconds)

|q|	Encoding1		Encoding2	
	clingo	DLV	clingo	DLV
5	0.01	0.01	1.18	10.05
10	0.02	0.02	3.94	100.12
15	0.03	0.09	8.52	> 300
20	0.07	0.30	> 300	> 300

Encoding1 = Listing 7.1
Encoding2 = Appendix B.3.2

odd or *even* for each element in the set q. However, we do not compare the performance of this encoding against *TD-MLP* because its performance is not as good as the encoding in Listing 7.1. *clingo* and *DLV*, both of them need much longer time to solve the Even-Odd problem with this solution. See Tabel 7.3 for details.

Next, we use both *clingo* and *DLV* to run on the program in Listing 7.1 with increasing number of elements in q. Then, we compare the evaluation time with the MLP encoding (see Appendix B.3.1) run on *TD-MLP*. The result is depicted in Figure 7.9. In this case, MLP shows its strength. *TD-MLP* outperform *clingo* and *DLV* and has much better evaluation time. One could understand that the main reason why both of our ordinary ASP solver have very low performance in this problem is because of the choice rule created by the first rule. MLP encoding for this problem as in Example 2.25 also has a choice point. However, the choice point that it has does not really problematic as in the first rule in Listing 7.1. Instead of creating a pair (combination of two), MLP cleverly create another set copy, minus one element. Then, call another module to inspect it. This process goes on recursively until all elements has been singled out (see Example 2.25 for detail explanation). This is possible because one of the strengths of MLP that it admits mutual recursive calls between modules. However, this cannot be done in an ordinary-monolithic ASP encoding. Since the problem is to decide whether a set has even or odd number of elements, transferring the elements into pairs is a solution. We know that if all elements can be transferred into pairs, then the number of elements on the set is even. Otherwise, it is odd. Labeling also became another solution. But we know that in this case, the performance of labeling solution is not as good as the pairing solution.

7.5 Summary

From the experiments that have been done, there are several facts that could be pointed out:

- Most of the times, *TD-MLP* with instantiation splitting optimization performs better (at least it has the same performance) compared to *TD-MLP* with default setting. And, the bigger the program is, the more performance we gain from the optimization.

- Time difference on getting the next answer set from the current answer set increases linearly with the number of answer sets.
- It is a good approach not to fix the choice of the ASP solver used in *TD-MLP*. Since currently the characteristics and performance of different ASP solver is quite varied, trying different ASP solvers could yield far better performance (even hundred times better).
- Although it mostly depend on the encoding, the performance of *TD-MLP* and ordinary ASP solvers are also depend on the nature of the problem. With a nearly similar encoding, performance of *TD-MLP* is not far behind the ordinary ASP solver. However, with careful implementation and utilized the full features of MLP, we have shown that *TD-MLP* do provide another option on how to solve a problem, and confidently outperforms the performance of an ordinary ASP solvers.

CHAPTER 8

Conclusion and Further Work

This chapter concludes the thesis and summarizes the main points. In Section 8.1 we conclude our work, state the progress and results we have gained so far. In addition, for such a new and challenging research, it is also important to give an overview on what still needs to be done and what can come next. We give an overview about the further work in Section 8.2.

8.1 Conclusion

In this thesis we consider the concept of modular nonmonotonic logic programs (MLPs). It is a novel formalism defined in [Dao-Tran et al., 2009a] for logic programs to embrace many advantages offered by modularity paradigm. Using MLPs, logic programs can now be defined as modules. Program decomposition becomes concrete and easier than before. MLPs also offer a possibility to handle dynamic input for module calls and allow recursive calls between modules. Seems to be very common in most of imperative programming languages, but such features are quite recent in nonmonotonic logic programming.

In this thesis, we have developed an evaluation system for MLPs called *TD-MLP*. Since MLPs has high computational complexity, a nontrivial way of evaluation is needed. We based our system on the top-down evaluation technique in [Dao-Tran et al., 2009b]. It was the first and the only approach available at the time this thesis is written.

Furthermore, based on deep analysis we noticed that there is a chance for performance improvement by forgetting the rules that had been evaluated before (instead of always considering the complete rules). From this observation, we develop an *instantiation splitting* technique that can be integrated into the algorithm in [Dao-Tran et al., 2009b]. Empirical evaluations show that *instantiation splitting* optimization offers better the performance in general. Experiments also suggest that the bigger the program is, the more performance one can gain from the optimization.

Since evaluation of MLPs is quite expensive compared to the evaluation of ordinary ASP in the worst case scenario, we also carried out experiments involving both encodings. We re-encode the ordinary ASP encodings into MLP and run it on *TD-MLP*. Experiments suggest that

MLP produce only a little time overhead. On another experiment that we set on solving Even-odd (cardinality of a set) problem, MLP outperforms ordinary ASP encodings. We have created two ordinary encodings to solve this problem, i.e., pairing and labeling solutions, and an MLP encoding which single out an element of the set one by one. The experiment shows that the ordinary encodings cannot solve the problem with 100 elements in the set under 300 seconds, while the MLP encodings can solve it on less than 10 seconds only.

8.2 Further Works

There are still many open problems that need to be tackled. Some of them are, but not limited to:

- The evaluation technique introduced in [Dao-Tran et al., 2009b] is quite successful. However, it is restricted to ic-stratified MLPs. To push MLPs even further, one need to find more expressive fragments of MLPs that still allow for efficient evaluation techniques.
- We have stored and reused information about solved module atoms in an indexed set of module atoms (see data structure **A** in Algorithm 5.1). However, one can go further with tabled evaluation techniques introduced in [Chen and Warren, 1996]. The idea is to store the results of previous computations to be used later. However one needs to be careful not to store all information, i.e., answer sets from all instantiations obtained since the number of answer sets caused by disjunctive rules or unstratified negation can be exponential. To limit the size of such table and further investigation for a replacement strategy, i.e., choosing which elements need to be kept or discarded can be another topic of interest.
- When there are two or more module atoms that can solved in simultaneously (they do not depend to each other, and all of their inputs can be completely prepared), several ideas can be applied:
  - Choose a module atom that can narrow the computation tree. This can be achieved by defining a heuristics approach or machine learning algorithms to estimate the number of choice points that will be created (potentially) from a particular instantiation.
  - Investigate the possibility to apply parallel (possibly distributed) algorithms to evaluate the module atoms at the same time. However, a redundant computation can occur when more than one process computes the same instantiation. One has consider this trade off and take it into account when devising such algorithms.
- As we here seen in the experiments, choosing a different ASP solver as the backbone can yield a better evaluation time (even by two orders of magnitude). Since the performance behaviors of ASP solvers are quite diverse, we have to keep in mind that parameterized backbone solver is a always a good idea.

Bibliography

Anastasia Analyti, Grigoris Antoniou, and Carlos Viegas Damasio. Mweb: A principled framework for modular web rule bases and its semantics. *ACM Trans. Comput. Logic*, 12:17:1–17:46, January 2011. ISSN 1529-3785.

Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. H.: Macros, macro calls, and use of ensembles in modular answer set programming. In *International Conference on Logic Programming*. Springer Verlag, 2006.

Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Communication of the ACM*, 2011. (to appear).

Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16:1361–1398, July 1994. ISSN 0164-0925.

Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *J. Log. Program.*, 19/20:443–502, 1994.

Pedro Cabalar. Answer set; programming? In Marcello Balduccini and Tran Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pages 334–343. Springer Verlag, 2011. ISBN 978-3-642-20831-7.

Francesco Calimeri and Giovambattista Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Commun.*, 19:193–206, August 2006. ISSN 0921-7126.

Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43:20–74, January 1996. ISSN 0004-5411.

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 145–159, Berlin, Heidelberg, July 2009a. Springer-Verlag. ISBN 978-3-642-02845-8. doi: http://dx.doi.org/10.1007/978-3-642-02846-5_16. URL `http://dx.doi.org/10.1007/978-3-642-02846-5_16`.

Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Relevance-driven evaluation of modular nonmonotonic logic programs. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages

87–100, Berlin, Heidelberg, September 2009b. Springer-Verlag. ISBN 978-3-642-04237-9. doi: http://dx.doi.org/10.1007/978-3-642-04238-6_10. URL http://dx.doi.org/10.1007/978-3-642-04238-6_10.

Thomas Eiter, Georg Gottlob, and Helmut Veith. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '97, pages 290–309, London, UK, 1997. Springer-Verlag. ISBN 3-540-63255-7.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pages 90–96, 2005.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A system for integrating multiple semantics in an answer-set programming framework. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *WLP*, volume 1843-06-02 of *INFSYS Research Report*, pages 206–210. Technische Universität Wien, Austria, 2006.

Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer Set Programming: A Primer*, pages 40–110. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03753-5. doi: 10.1007/978-3-642-03754-2_2.

Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In José Júlio Alferes and João Alexandre Leite, editors, *Proceedings of the 9th Conference on Logics in Artificial Intelligence (JELIA 2004)*, volume 3229 of *LNCS*, pages 200–212. Springer, September 2004. doi: 10.1007/b100483.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Log.*, 7(1):1–37, 2006.

Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran. Modularity aspects of disjunctive stable models. *J. Artif. Int. Res.*, 35:813–857, August 2009. ISSN 1076-9757.

Roman Kontchakov, Frank Wolter, and Michael Zakharyaschev. Logic-based ontology comparison and module extraction, with an application to dl-lite. *Artif. Intell.*, 174(15):1093–1141, 2010.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7:499–562, July 2006. ISSN 1529-3785.

Yuliya Lierler. cmodels - sat-based disjunctive answer set solver. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2005. ISBN 3-540-28538-5.

Vladimir Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the eleventh international conference on Logic programming*, pages 23–37, Cambridge, MA, USA, 1994. MIT Press. ISBN 0-262-72022-1.

Fangzhen Lin and Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theory Pract. Log. Program.*, 8:717–761, November 2008. ISSN 1471-0684. doi: 10.1017/S147106840800358X.

Emilia Oikarinen and Tomi Janhunen. A translation-based approach to the verification of modular equivalence. *J. Log. Comput.*, 19(4):591–613, 2009.

Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 3 edition, December 2009. ISBN 0136042597.

Tommi Syrjänen and Ilkka Niemelä. The smodels system. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001. ISBN 3-540-42593-4.

Luis Tari, Chitta Baral, and Saadat Anwar. A language for modular answer set programming: Application to acc tournament scheduling. In *In Proc. of the 3rd International Workshop on Answer Set Programming, volume 142 of CEUR Workshop Proceedings*, pages 277–292, 2005.

APPENDIX A

Experiment Results

Table A.1: Line pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-line-10-5-1-10-0-10-10	10.50	0.53	303.00	15.88	21.00	1.07	21.00	1.07	1.00	0.00	0.06	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.07	0.01	-	-
Module-line-30-5-1-10-0-10-10	10.75	0.46	706.38	84.89	21.50	0.93	21.50	0.93	1.00	0.00	0.09	0.01	-	-
Module-line-40-5-1-10-0-10-10	10.50	0.53	992.25	111.40	21.00	1.07	21.00	1.07	1.00	0.00	0.11	0.02	-	-
Module-line-20-10-1-10-0-10-10	10.62	0.52	735.75	25.03	21.25	1.04	21.25	1.04	1.00	0.00	0.08	0.01	-	-
Module-line-20-20-1-10-0-10-10	10.75	0.46	1274.75	76.27	21.50	0.93	21.50	0.93	1.00	0.00	0.12	0.01	-	-
Module-line-20-40-1-10-0-10-10	10.75	0.46	2492.75	220.57	21.50	0.93	21.50	0.93	1.00	0.00	0.17	0.03	-	-
Module-line-20-5-1-10-0-10-5	5.88	0.35	256.00	28.22	11.75	0.71	11.75	0.71	1.00	0.00	0.03	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-10-0-10-15	15.50	0.53	742.38	27.71	31.00	1.07	31.00	1.07	1.00	0.00	0.14	0.02	-	-
Module-line-20-5-1-10-0-10-20	20.62	0.52	1014.50	58.53	41.25	1.04	41.25	1.04	1.00	0.00	0.20	0.02	-	-
Module-line-20-5-1-10-0-10-25	25.88	0.35	1251.75	46.33	51.75	0.71	51.75	0.71	1.00	0.00	0.27	0.03	-	-
Module-line-20-5-1-5-0-10-10	10.62	0.52	534.38	37.89	21.25	1.04	21.25	1.04	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-15-0-10-10	10.50	0.53	481.25	30.21	21.00	1.07	21.00	1.07	1.00	0.00	0.08	0.01	-	-
Module-line-20-5-1-20-0-10-10	10.38	0.52	505.88	15.08	20.75	1.04	20.75	1.04	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-10-0-5-10	10.62	0.52	488.50	10.45	21.25	1.04	21.25	1.04	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-10-0-20-10	10.62	0.52	555.25	43.34	21.25	1.04	21.25	1.04	1.00	0.00	0.10	0.02	-	-
Module-line-20-5-1-10-0-40-10	10.75	0.46	2929.75	5191.54	21.50	0.93	21.50	0.93	1.00	0.00	1.56	2.41	-	-
Module-line-10-5-2-10-10-10-10	10.62	0.52	291.00	12.94	21.25	1.04	21.25	1.04	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.01	-	-
Module-line-30-5-2-10-10-10-10	10.60	0.55	679.00	25.57	21.20	1.10	21.20	1.10	1.00	0.00	0.09	0.01	-	0.2
Module-line-40-5-2-10-10-10-10	10.75	0.46	855.75	58.45	172.38	422.91	180.38	445.14	9.00	22.23	1.14	2.90	0.1	-
Module-line-20-10-2-10-10-10-10	10.62	0.52	717.50	51.79	102.00	105.03	106.62	110.26	5.62	5.37	0.66	0.78	-	-
Module-line-20-20-2-10-10-10-10	11.00	0.00	1283.62	165.14	22.00	0.00	22.00	0.00	1.00	0.00	0.11	0.02	-	-
Module-line-20-40-2-10-10-10-10	11.00	0.00	2617.50	178.14	32.25	17.65	32.88	18.70	1.62	1.06	0.23	0.11	-	-
Module-line-20-5-2-10-10-10-5	6.00	0.00	269.50	35.93	43.00	87.68	46.88	98.64	4.88	10.96	0.17	0.41	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-2-10-10-10-15	15.67	0.52	766.50	34.37	163.50	194.76	168.67	202.38	6.17	7.63	1.00	1.26	-	0.1
Module-line-20-5-2-10-10-10-20	20.83	0.41	994.00	40.01	159.67	103.42	163.00	106.27	4.33	2.88	1.02	0.70	0.1	-
Module-line-20-5-2-10-10-10-25	25.25	0.46	1287.12	99.31	618.25	1046.89	631.50	1068.60	14.25	21.81	3.97	7.03	0.1	-
Module-line-20-5-2-5-10-10-10	10.88	0.35	543.25	45.85	2130.25	2904.67	2277.12	3094.99	147.88	193.48	13.48	18.44	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-2-15-10-10-10	10.62	0.52	482.75	33.53	43.25	62.54	44.62	66.42	2.38	3.89	0.19	0.35	-	-
Module-line-20-5-2-20-10-10-10	10.50	0.53	500.50	39.37	21.00	1.07	21.00	1.07	1.00	0.00	0.10	0.02	-	-
Module-line-20-5-2-10-10-5-10	10.38	0.52	471.25	19.53	20.75	1.04	20.75	1.04	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-2-10-10-20-10	10.50	0.55	593.83	44.47	13412.33	15401.32	14545.17	16846.21	1034.50	1271.38	140.48	148.04	0.4	-
Module-line-20-5-2-10-10-40-10	10.67	0.52	518.00	26.99	1937.17	4564.05	2083.83	4914.99	147.67	350.94	16.38	38.94	0.2	-

*Module-line-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.2: Line pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-line-10-5-1-10-0-10-10	10.50	0.53	303.00	15.88	21.00	1.07	21.00	1.07	1.00	0.00	0.04	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.05	0.01	-	-
Module-line-30-5-1-10-0-10-10	10.75	0.46	706.38	84.89	21.50	0.93	21.50	0.93	1.00	0.00	0.06	0.01	-	-
Module-line-40-5-1-10-0-10-10	10.50	0.53	992.25	111.40	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.01	-	-
Module-line-20-10-1-10-0-10-10	10.62	0.52	735.75	25.03	21.25	1.04	21.25	1.04	1.00	0.00	0.06	0.00	-	-
Module-line-20-20-1-10-0-10-10	10.75	0.46	1274.75	76.27	21.50	0.93	21.50	0.93	1.00	0.00	0.08	0.01	-	-
Module-line-20-40-1-10-0-10-10	10.75	0.46	2492.75	220.57	21.50	0.93	21.50	0.93	1.00	0.00	0.12	0.03	-	-
Module-line-20-5-1-10-0-10-5	5.88	0.35	256.00	28.22	11.75	0.71	11.75	0.71	1.00	0.00	0.02	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-1-10-0-10-15	15.50	0.53	742.38	27.71	31.00	1.07	31.00	1.07	1.00	0.00	0.09	0.01	-	-
Module-line-20-5-1-10-0-10-20	20.62	0.52	1014.50	58.53	41.25	1.04	41.25	1.04	1.00	0.00	0.16	0.01	-	-
Module-line-20-5-1-10-0-10-25	25.88	0.35	1251.75	46.33	51.75	0.71	51.75	0.71	1.00	0.00	0.21	0.03	-	-
Module-line-20-5-1-5-0-10-10	10.62	0.52	534.38	37.89	21.25	1.04	21.25	1.04	1.00	0.00	0.05	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-1-15-0-10-10	10.50	0.53	481.25	30.21	21.00	1.07	21.00	1.07	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-1-20-0-10-10	10.38	0.52	505.88	15.08	20.75	1.04	20.75	1.04	1.00	0.00	0.06	0.01	-	-
Module-line-20-5-1-10-0-5-10	10.62	0.52	488.50	10.45	21.25	1.04	21.25	1.04	1.00	0.00	0.04	0.00	-	-
Module-line-20-5-1-10-0-10-10	10.25	0.46	482.38	20.28	20.50	0.93	20.50	0.93	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-1-10-0-20-10	10.62	0.52	555.25	43.34	21.25	1.04	21.25	1.04	1.00	0.00	0.07	0.01	-	-
Module-line-20-5-1-10-0-40-10	10.75	0.46	2929.75	5191.54	21.50	0.93	21.50	0.93	1.00	0.00	0.95	1.46	-	-
Module-line-10-5-2-10-10-10-10	10.62	0.52	291.00	12.94	21.25	1.04	21.25	1.04	1.00	0.00	0.04	0.01	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.05	0.01	-	-
Module-line-30-5-2-10-10-10-10	10.60	0.55	679.00	25.57	21.20	1.10	21.20	1.10	1.00	0.00	0.07	0.02	-	0.2
Module-line-40-5-2-10-10-10-10	10.75	0.46	855.75	58.45	172.38	422.91	180.38	445.14	9.00	22.23	0.89	2.30	0.1	-
Module-line-20-10-2-10-10-10-10	10.62	0.52	717.50	51.79	102.00	105.03	106.62	110.26	5.62	5.37	0.52	0.66	-	-
Module-line-20-20-2-10-10-10-10	11.00	0.00	1283.62	165.14	22.00	0.00	22.00	0.00	1.00	0.00	0.10	0.02	-	-
Module-line-20-40-2-10-10-10-10	11.00	0.00	2617.50	178.14	32.25	17.65	32.88	18.70	1.62	1.06	0.18	0.12	-	-
Module-line-20-5-2-10-10-10-5	6.00	0.00	269.50	35.93	43.00	87.68	46.88	98.64	4.88	10.96	0.13	0.30	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-2-10-10-10-15	15.67	0.52	766.50	34.37	163.50	194.76	168.67	202.38	6.17	7.63	0.78	0.98	-	0.1
Module-line-20-5-2-10-10-10-20	20.83	0.41	994.00	40.01	159.67	103.42	163.00	106.27	4.33	2.88	0.81	0.55	-	-
Module-line-20-5-2-10-10-10-25	25.25	0.46	1287.12	99.31	618.25	1046.89	631.50	1068.60	14.25	21.81	3.35	5.97	0.1	-
Module-line-20-5-2-5-10-10-10	10.88	0.35	543.25	45.85	2130.25	2904.67	2277.12	3094.99	147.88	193.48	10.11	13.90	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-2-15-10-10-10	10.62	0.52	482.75	33.53	43.25	62.54	44.62	66.42	2.38	3.89	0.15	0.28	-	-
Module-line-20-5-2-20-10-10-10	10.50	0.53	500.50	39.37	21.00	1.07	21.00	1.07	1.00	0.00	0.07	0.02	-	-
Module-line-20-5-2-10-10-5-10	10.38	0.52	471.25	19.53	20.75	1.04	20.75	1.04	1.00	0.00	0.04	0.00	-	-
Module-line-20-5-2-10-10-10-10	10.50	0.53	480.62	30.29	21.00	1.07	21.00	1.07	1.00	0.00	0.05	0.01	-	-
Module-line-20-5-2-10-10-20-10	10.50	0.55	593.83	44.47	12893.17	14311.97	13970.00	15627.35	946.33	1102.21	139.15	149.70	0.4	-
Module-line-20-5-2-10-10-40-10	10.67	0.52	518.00	26.99	1937.17	4564.05	2083.83	4914.99	147.67	350.94	9.45	22.31	0.2	-

*Module-line-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.3: Ring pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		|**M**|		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-ring-10-5-1-10-0-10-10	10.00	0.00	313.50	12.20	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-30-5-1-10-0-10-10	10.00	0.00	656.50	36.57	11.00	0.00	11.00	0.00	1.00	0.00	0.05	0.01	-	-
Module-ring-40-5-1-10-0-10-10	10.00	0.00	864.75	78.07	11.00	0.00	11.00	0.00	1.00	0.00	0.06	0.01	-	-
Module-ring-20-10-1-10-0-10-10	10.00	0.00	719.75	62.88	11.00	0.00	11.00	0.00	1.00	0.00	0.05	0.00	-	-
Module-ring-20-20-1-10-0-10-10	10.00	0.00	1212.25	142.09	11.00	0.00	11.00	0.00	1.00	0.00	0.07	0.01	-	-
Module-ring-20-40-1-10-0-10-10	10.00	0.00	2403.25	196.76	11.00	0.00	11.00	0.00	1.00	0.00	0.09	0.01	-	-
Module-ring-20-5-1-10-0-10-5	5.00	0.00	223.75	17.62	6.00	0.00	6.00	0.00	1.00	0.00	0.02	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-5-1-10-0-10-15	15.00	0.00	728.50	31.04	16.00	0.00	16.00	0.00	1.00	0.00	0.07	0.01	-	-
Module-ring-20-5-1-10-0-10-20	20.00	0.00	1032.00	80.98	21.00	0.00	21.00	0.00	1.00	0.00	0.11	0.01	-	-
Module-ring-20-5-1-10-0-10-25	25.00	0.00	1261.25	97.02	26.00	0.00	26.00	0.00	1.00	0.00	0.15	0.02	-	-
Module-ring-20-5-1-5-0-10-10	10.00	0.00	497.50	32.28	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-5-1-15-0-10-10	10.00	0.00	481.50	23.42	11.00	0.00	11.00	0.00	1.00	0.00	0.05	0.00	-	-
Module-ring-20-5-1-20-0-10-10	10.00	0.00	458.00	23.40	11.00	0.00	11.00	0.00	1.00	0.00	0.05	0.00	-	-
Module-ring-20-5-1-10-0-5-10	10.00	0.00	471.88	24.43	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-5-1-10-0-20-10	10.00	0.00	518.62	28.29	11.00	0.00	11.00	0.00	1.00	0.00	0.06	0.01	-	-
Module-ring-20-5-1-10-0-40-10	10.00	0.00	2291.75	2076.41	11.00	0.00	11.00	0.00	1.00	0.00	0.49	0.43	-	-
Module-ring-10-5-2-10-10-10-10	10.00	0.00	298.50	19.83	11.75	2.12	12.00	2.83	1.12	0.35	0.04	0.01	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.09	-	-
Module-ring-30-5-2-10-10-10-10	10.00	0.00	664.50	21.69	153.00	355.64	170.75	400.09	18.75	44.45	0.82	1.91	0.1	-
Module-ring-40-5-2-10-10-10-10	10.00	0.00	871.71	50.18	12.71	4.54	13.14	5.67	1.43	1.13	0.07	0.04	0.2	-
Module-ring-20-10-2-10-10-10-10	10.00	0.00	746.00	59.24	14.62	6.37	15.25	7.42	1.62	1.06	0.07	0.03	-	-
Module-ring-20-20-2-10-10-10-10	10.00	0.00	1294.75	121.27	11.00	0.00	11.00	0.00	1.00	0.00	0.07	0.00	-	-
Module-ring-20-40-2-10-10-10-10	10.00	0.00	2286.50	190.43	27.88	47.73	29.75	53.03	2.88	5.30	0.24	0.39	-	-
Module-ring-20-5-2-10-10-10-5	5.00	0.00	242.57	27.49	6.71	1.89	6.86	2.27	1.14	0.38	0.02	0.01	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.09	-	-
Module-ring-20-5-2-10-10-10-15	15.00	0.00	722.14	37.74	36.29	37.91	39.29	43.28	4.00	5.39	0.17	0.19	-	-
Module-ring-20-5-2-10-10-10-20	20.00	0.00	954.57	13.79	36.43	14.01	39.71	19.06	3.00	2.38	0.23	0.13	-	-
Module-ring-20-5-2-10-10-10-25	25.17	0.41	1201.67	42.07	3467.50	8168.90	3990.50	9417.40	384.50	906.90	25.79	60.70	0.1	-
Module-ring-20-5-2-5-10-10-10	10.12	0.35	503.25	34.66	103.38	208.92	124.62	258.25	22.25	49.46	0.65	1.35	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.09	-	-
Module-ring-20-5-2-15-10-10-10	10.00	0.00	498.29	36.99	20.71	18.08	23.43	20.80	3.71	3.45	0.13	0.11	-	-
Module-ring-20-5-2-20-10-10-10	10.00	0.00	472.86	21.09	12.57	2.82	12.86	3.29	1.29	0.49	0.06	0.03	-	0.1
Module-ring-20-5-2-10-10-5-10	10.00	0.00	462.88	27.75	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.09	-	-
Module-ring-20-5-2-10-10-20-10	10.00	0.00	498.00	38.42	573.38	853.39	962.38	1488.74	285.50	506.49	10.45	16.92	-	-
Module-ring-20-5-2-10-10-40-10	10.00	0.00	508.00	17.35	26.67	16.01	30.00	19.52	4.33	3.51	0.33	0.29	0.1	0.1

*Module-ring-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.4: Ring pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		|**M**|		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-ring-10-5-1-10-0-10-10	10.00	0.00	313.50	12.20	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-30-5-1-10-0-10-10	10.00	0.00	656.50	36.57	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-40-5-1-10-0-10-10	10.00	0.00	864.75	78.07	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-10-1-10-0-10-10	10.00	0.00	719.75	62.88	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.00	-	-
Module-ring-20-20-1-10-0-10-10	10.00	0.00	1212.25	142.09	11.00	0.00	11.00	0.00	1.00	0.00	0.05	0.00	-	-
Module-ring-20-40-1-10-0-10-10	10.00	0.00	2403.25	196.76	11.00	0.00	11.00	0.00	1.00	0.00	0.08	0.01	-	-
Module-ring-20-5-1-10-0-10-5	5.00	0.00	223.75	17.62	6.00	0.00	6.00	0.00	1.00	0.00	0.01	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-10-15	15.00	0.00	728.50	31.04	16.00	0.00	16.00	0.00	1.00	0.00	0.06	0.00	-	-
Module-ring-20-5-1-10-0-10-20	20.00	0.00	1032.00	80.98	21.00	0.00	21.00	0.00	1.00	0.00	0.10	0.01	-	-
Module-ring-20-5-1-10-0-10-25	25.00	0.00	1261.25	97.02	26.00	0.00	26.00	0.00	1.00	0.00	0.14	0.01	-	-
Module-ring-20-5-1-5-0-10-10	10.00	0.00	497.50	32.28	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-15-0-10-10	10.00	0.00	481.50	23.42	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-20-0-10-10	10.00	0.00	458.00	23.40	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-5-10	10.00	0.00	471.88	24.43	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-10-10	10.00	0.00	482.12	36.07	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-1-10-0-20-10	10.00	0.00	518.62	28.29	11.00	0.00	11.00	0.00	1.00	0.00	0.04	0.01	-	-
Module-ring-20-5-1-10-0-40-10	10.00	0.00	2291.75	2076.41	11.00	0.00	11.00	0.00	1.00	0.00	0.32	0.26	-	-
Module-ring-10-5-2-10-10-10-10	10.00	0.00	298.50	19.83	11.75	2.12	12.00	2.83	1.12	0.35	0.03	0.01	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.12	-	-
Module-ring-30-5-2-10-10-10-10	10.00	0.00	664.50	21.69	153.00	355.64	170.75	400.09	18.75	44.45	1.14	2.80	0.1	-
Module-ring-40-5-2-10-10-10-10	10.00	0.00	871.71	50.18	12.71	4.54	13.14	5.67	1.43	1.13	0.06	0.05	0.2	-
Module-ring-20-10-2-10-10-10-10	10.00	0.00	746.00	59.24	14.62	6.37	15.25	7.42	1.62	1.06	0.06	0.03	-	-
Module-ring-20-20-2-10-10-10-10	10.00	0.00	1294.75	121.27	11.00	0.00	11.00	0.00	1.00	0.00	0.06	0.01	-	-
Module-ring-20-40-2-10-10-10-10	10.00	0.00	2286.50	190.43	27.88	47.73	29.75	53.03	2.88	5.30	0.19	0.31	-	-
Module-ring-20-5-2-10-10-10-5	5.00	0.00	242.57	27.49	6.71	1.89	6.86	2.27	1.14	0.38	0.02	0.01	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.12	-	-
Module-ring-20-5-2-10-10-10-15	15.00	0.00	722.14	37.74	36.29	37.91	39.29	43.28	4.00	5.39	0.22	0.28	-	-
Module-ring-20-5-2-10-10-10-20	20.00	0.00	954.57	13.79	36.43	14.01	39.71	19.06	3.00	2.38	0.26	0.19	-	-
Module-ring-20-5-2-10-10-10-25	25.17	0.41	1201.67	42.07	3467.50	8168.90	3990.50	9417.40	297.67	694.30	29.37	69.29	0.1	-
Module-ring-20-5-2-5-10-10-10	10.12	0.35	503.25	34.66	103.38	208.92	124.62	258.25	22.25	49.46	0.75	1.66	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.12	-	-
Module-ring-20-5-2-15-10-10-10	10.00	0.00	498.29	36.99	20.71	18.08	23.43	20.80	3.71	3.45	0.12	0.15	-	-
Module-ring-20-5-2-20-10-10-10	10.00	0.00	472.86	21.09	12.57	2.82	12.86	3.29	1.29	0.49	0.04	0.02	-	0.1
Module-ring-20-5-2-10-10-5-10	10.00	0.00	462.88	27.75	11.00	0.00	11.00	0.00	1.00	0.00	0.03	0.00	-	-
Module-ring-20-5-2-10-10-10-10	10.00	0.00	485.00	12.02	20.12	18.17	21.38	20.72	2.25	2.55	0.09	0.12	-	-
Module-ring-20-5-2-10-10-20-10	10.00	0.00	498.00	38.42	573.38	853.39	962.38	1488.74	285.50	506.49	6.66	10.60	-	-
Module-ring-20-5-2-10-10-40-10	10.00	0.00	508.00	17.35	26.67	16.01	30.00	19.52	4.33	3.51	0.22	0.17	0.1	0.1

*Module-ring-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.5: Diamond pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		|M|		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-diamond-10-5-1-10-0-10-10	11.38	0.52	331.88	30.72	26.50	1.07	26.50	1.07	1.00	0.00	0.10	0.03	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.12	0.02	-	-
Module-diamond-30-5-1-10-0-10-10	12.12	0.83	868.12	146.67	28.12	2.17	28.12	2.17	1.00	0.00	0.14	0.03	-	-
Module-diamond-40-5-1-10-0-10-10	11.12	0.83	923.50	74.36	25.88	2.17	25.88	2.17	1.00	0.00	0.14	0.02	-	-
Module-diamond-20-10-1-10-0-10-10	12.62	0.74	777.12	49.20	29.38	2.07	29.38	2.07	1.00	0.00	0.12	0.03	-	-
Module-diamond-20-20-1-10-0-10-10	12.88	0.83	1413.62	202.56	30.38	1.85	30.38	1.85	1.00	0.00	0.18	0.03	-	-
Module-diamond-20-40-1-10-0-10-10	13.38	0.74	2830.00	399.58	31.50	1.69	31.50	1.69	1.00	0.00	0.21	0.03	-	-
Module-diamond-20-5-1-10-0-10-5	6.25	0.46	280.38	33.13	13.50	0.93	13.50	0.93	1.00	0.00	0.03	0.00	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.12	0.02	-	-
Module-diamond-20-5-1-10-0-10-15	18.12	1.13	810.88	63.93	42.88	3.00	42.88	3.00	1.00	0.00	0.20	0.02	-	-
Module-diamond-20-5-1-10-0-10-20	23.50	1.60	1112.50	63.33	54.62	3.93	54.62	3.93	1.00	0.00	0.30	0.02	-	-
Module-diamond-20-5-1-10-0-10-25	29.62	1.77	1631.62	375.30	70.38	4.03	70.38	4.03	1.00	0.00	0.46	0.06	-	-
Module-diamond-20-5-1-5-0-10-10	12.12	1.13	556.75	52.82	28.25	2.82	28.25	2.82	1.00	0.00	0.10	0.02	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.12	0.02	-	-
Module-diamond-20-5-1-15-0-10-10	11.25	0.46	531.38	23.74	25.88	1.36	25.88	1.36	1.00	0.00	0.11	0.02	-	-
Module-diamond-20-5-1-20-0-10-10	12.50	0.76	551.12	39.05	29.25	1.83	29.25	1.83	1.00	0.00	0.12	0.02	-	-
Module-diamond-20-5-1-10-0-5-10	11.88	1.25	515.62	41.53	27.25	2.71	27.25	2.71	1.00	0.00	0.09	0.01	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.12	0.02	-	-
Module-diamond-20-5-1-10-0-20-10	12.50	1.07	634.50	113.88	28.75	2.43	28.75	2.43	1.00	0.00	0.15	0.04	-	-
Module-diamond-20-5-1-10-0-40-10	11.75	0.46	2097.38	1862.22	27.38	1.60	27.38	1.60	1.00	0.00	1.08	1.44	-	-
Module-diamond-10-5-2-10-10-10-10	11.86	0.69	325.29	21.07	28.86	2.41	29.00	2.65	1.14	0.38	0.09	0.04	-	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	12.10	33.55	0.1	-
Module-diamond-30-5-2-10-10-10-10	12.00	1.31	755.62	31.72	325.62	828.92	339.12	866.70	14.50	37.78	2.05	5.35	0.1	-
Module-diamond-40-5-2-10-10-10-10	13.00	1.20	1070.62	131.65	5604.25	11171.62	5912.25	11717.48	309.00	565.46	76.10	137.93	0.3	-
Module-diamond-20-10-2-10-10-10-10	12.14	0.90	816.86	68.41	31.00	5.92	31.14	6.28	1.14	0.38	0.15	0.04	-	0.1
Module-diamond-20-20-2-10-10-10-10	13.38	0.52	1537.38	73.58	37.25	9.78	37.50	10.20	1.25	0.46	0.25	0.09	-	-
Module-diamond-20-40-2-10-10-10-10	13.25	0.71	2919.25	349.94	1902.88	3891.12	2272.75	4807.03	370.88	946.93	41.71	104.82	0.1	-
Module-diamond-20-5-2-10-10-10-5	5.88	0.64	271.25	24.13	15.75	4.77	16.00	5.21	1.25	0.46	0.05	0.03	-	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	12.10	33.55	0.1	-
Module-diamond-20-5-2-10-10-10-15	17.62	1.69	816.38	80.97	1618.12	4439.80	1681.50	4617.44	64.38	177.64	10.74	29.67	0.1	-
Module-diamond-20-5-2-10-10-10-20	26.50	7.18	1237.17	334.75	723.83	1076.23	744.17	1101.76	21.33	26.70	4.61	6.78	0.1	-
Module-diamond-20-5-2-10-10-10-25	29.00	1.53	1344.14	58.21	296.29	314.66	301.00	319.51	5.71	4.96	1.88	2.03	0.1	-
Module-diamond-20-5-2-5-10-10-10	12.57	1.13	567.71	60.42	221.43	477.32	240.00	525.11	19.57	47.83	1.27	2.93	-	0.1
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	12.10	33.55	0.1	-
Module-diamond-20-5-2-15-10-10-10	12.25	0.71	558.00	52.83	46.38	45.57	47.25	48.04	1.88	2.47	0.24	0.29	-	-
Module-diamond-20-5-2-20-10-10-10	12.50	0.53	575.50	35.20	34.75	10.57	35.00	11.02	1.25	0.46	0.20	0.07	-	-
Module-diamond-20-5-2-10-10-5-10	11.88	0.99	540.12	48.24	40.25	16.56	41.38	18.72	2.12	2.42	0.15	0.09	0.1	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	12.10	33.55	0.1	-
Module-diamond-20-5-2-10-10-20-10	11.33	1.21	517.83	41.06	946.83	1342.37	997.17	1417.51	51.33	75.48	6.56	9.21	0.2	-
Module-diamond-20-5-2-10-10-40-10	17.67	8.14	970.33	286.33	926.00	987.10	993.00	1041.44	68.00	58.10	23.59	35.09	0.3	0.1

*Module-diamond-[#constant]-[#predicates]-[#head]-[#body]-[%*not*-[#rules]-[#modules]*

Table A.6: Diamond pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-diamond-10-5-1-10-0-10-10	11.38	0.52	331.88	30.72	26.50	1.07	26.50	1.07	1.00	0.00	0.05	0.01	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.08	0.01	-	-
Module-diamond-30-5-1-10-0-10-10	12.12	0.83	868.12	146.67	28.12	2.17	28.12	2.17	1.00	0.00	0.10	0.02	-	-
Module-diamond-40-5-1-10-0-10-10	11.12	0.83	923.50	74.36	25.88	2.17	25.88	2.17	1.00	0.00	0.08	0.02	-	-
Module-diamond-20-10-1-10-0-10-10	12.62	0.74	777.12	49.20	29.38	2.07	29.38	2.07	1.00	0.00	0.06	0.00	-	-
Module-diamond-20-20-1-10-0-10-10	12.88	0.83	1413.62	202.56	30.38	1.85	30.38	1.85	1.00	0.00	0.11	0.02	-	-
Module-diamond-20-40-1-10-0-10-10	13.38	0.74	2830.00	399.58	31.50	1.69	31.50	1.69	1.00	0.00	0.15	0.04	-	-
Module-diamond-20-5-1-10-0-10-5	6.25	0.46	280.38	33.13	13.50	0.93	13.50	0.93	1.00	0.00	0.03	0.00	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.08	0.01	-	-
Module-diamond-20-5-1-10-0-10-15	18.12	1.13	810.88	63.93	42.88	3.00	42.88	3.00	1.00	0.00	0.16	0.02	-	-
Module-diamond-20-5-1-10-0-10-20	23.50	1.60	1112.50	63.33	54.62	3.93	54.62	3.93	1.00	0.00	0.23	0.02	-	-
Module-diamond-20-5-1-10-0-10-25	29.62	1.77	1631.62	375.30	70.38	4.03	70.38	4.03	1.00	0.00	0.34	0.05	-	-
Module-diamond-20-5-1-5-0-10-10	12.12	1.13	556.75	52.82	28.25	2.82	28.25	2.82	1.00	0.00	0.08	0.01	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.08	0.01	-	-
Module-diamond-20-5-1-15-0-10-10	11.25	0.46	531.38	23.74	25.88	1.36	25.88	1.36	1.00	0.00	0.07	0.01	-	-
Module-diamond-20-5-1-20-0-10-10	12.50	0.76	551.12	39.05	29.25	1.83	29.25	1.83	1.00	0.00	0.09	0.02	-	-
Module-diamond-20-5-1-10-0-5-10	11.88	1.25	515.62	41.53	27.25	2.71	27.25	2.71	1.00	0.00	0.06	0.01	-	-
Module-diamond-20-5-1-10-0-10-10	12.00	0.76	542.25	50.82	28.12	1.96	28.12	1.96	1.00	0.00	0.08	0.01	-	-
Module-diamond-20-5-1-10-0-20-10	12.50	1.07	634.50	113.88	28.75	2.43	28.75	2.43	1.00	0.00	0.12	0.03	-	-
Module-diamond-20-5-1-10-0-40-10	11.75	0.46	2097.38	1862.22	27.38	1.60	27.38	1.60	1.00	0.00	0.62	0.77	-	-
Module-diamond-10-5-2-10-10-10-10	11.86	0.69	325.29	21.07	28.86	2.41	29.00	2.65	1.14	0.38	0.05	0.01	-	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	9.34	25.97	0.1	-
Module-diamond-30-5-2-10-10-10-10	12.00	1.31	755.62	31.72	325.62	828.92	339.12	866.70	14.50	37.78	1.67	4.44	0.1	-
Module-diamond-40-5-2-10-10-10-10	13.00	1.20	1070.62	131.65	5917.25	11897.85	6241.25	12476.02	325.00	598.02	75.87	138.09	0.2	-
Module-diamond-20-10-2-10-10-10-10	12.14	0.90	816.86	68.41	31.00	5.92	31.14	6.28	1.14	0.38	0.10	0.04	-	0.1
Module-diamond-20-20-2-10-10-10-10	13.38	0.52	1537.38	73.58	37.25	9.78	37.50	10.20	1.25	0.46	0.19	0.07	-	-
Module-diamond-20-40-2-10-10-10-10	13.25	0.71	2919.25	349.94	2026.88	4218.80	2427.75	5224.40	378.50	968.41	41.48	104.88	0.2	-
Module-diamond-20-5-2-10-10-10-5	5.88	0.64	271.25	24.13	15.75	4.77	16.00	5.21	1.25	0.46	0.03	0.01	-	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	9.34	25.97	0.1	-
Module-diamond-20-5-2-10-10-10-15	17.62	1.69	816.38	80.97	1618.12	4439.80	1681.50	4617.44	64.38	177.64	8.71	24.13	0.1	-
Module-diamond-20-5-2-10-10-10-20	26.50	7.18	1237.17	334.75	723.83	1076.23	744.17	1101.76	21.33	26.70	3.45	5.10	0.1	-
Module-diamond-20-5-2-10-10-10-25	29.00	1.53	1344.14	58.21	296.29	314.66	301.00	319.51	5.71	4.96	1.41	1.56	0.1	-
Module-diamond-20-5-2-5-10-10-10	12.57	1.13	567.71	60.42	221.43	477.32	240.00	525.11	19.57	47.83	0.98	2.22	-	0.1
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	9.34	25.97	0.1	-
Module-diamond-20-5-2-15-10-10-10	12.25	0.71	558.00	52.83	46.38	45.57	47.25	48.04	1.88	2.47	0.17	0.21	-	-
Module-diamond-20-5-2-20-10-10-10	12.50	0.53	575.50	35.20	34.75	10.57	35.00	11.02	1.25	0.46	0.12	0.06	-	-
Module-diamond-20-5-2-10-10-5-10	11.88	0.99	540.12	48.24	40.25	16.56	41.38	18.72	2.12	2.42	0.12	0.07	0.1	-
Module-diamond-20-5-2-10-10-10-10	11.62	0.74	550.12	62.34	1767.38	4873.31	1832.62	5053.44	66.25	180.13	9.34	25.97	0.1	-
Module-diamond-20-5-2-10-10-20-10	11.33	1.21	517.83	41.06	946.83	1342.37	997.17	1417.51	51.33	75.48	4.45	6.39	0.2	-
Module-diamond-20-5-2-10-10-40-10	18.33	9.29	999.67	333.99	926.00	987.10	993.00	1041.44	68.00	58.10	20.52	32.75	0.3	0.1

*Module-diamond-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.7: Star pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-star-10-5-1-10-0-10-10	15.75	1.28	400.88	35.50	39.50	2.56	39.50	2.56	1.00	0.00	0.17	0.02	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.17	0.03	-	-
Module-star-30-5-1-10-0-10-10	16.38	0.92	966.25	86.15	40.75	1.83	40.75	1.83	1.00	0.00	0.24	0.02	-	-
Module-star-40-5-1-10-0-10-10	15.62	2.07	1122.25	114.03	39.12	3.91	39.12	3.91	1.00	0.00	0.23	0.03	-	-
Module-star-20-10-1-10-0-10-10	16.00	0.93	938.00	55.89	40.00	1.85	40.00	1.85	1.00	0.00	0.22	0.01	-	-
Module-star-20-20-1-10-0-10-10	16.50	0.76	1642.25	190.97	41.00	1.51	41.00	1.51	1.00	0.00	0.26	0.03	-	-
Module-star-20-40-1-10-0-10-10	18.00	0.76	3246.38	318.77	44.00	1.51	44.00	1.51	1.00	0.00	0.30	0.04	-	-
Module-star-20-5-1-10-0-10-5	7.38	0.52	313.12	20.95	17.75	1.04	17.75	1.04	1.00	0.00	0.06	0.01	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.17	0.03	-	-
Module-star-20-5-1-10-0-10-15	23.25	1.49	978.50	116.23	59.50	2.98	59.50	2.98	1.00	0.00	0.31	0.05	-	-
Module-star-20-5-1-10-0-10-20	31.75	1.67	1335.25	95.73	81.38	3.25	81.38	3.25	1.00	0.00	0.50	0.07	-	-
Module-star-20-5-1-10-0-10-25	40.25	2.87	1748.50	152.00	103.38	5.55	103.38	5.55	1.00	0.00	0.67	0.06	-	-
Module-star-20-5-1-5-0-10-10	15.00	1.20	642.00	39.63	38.00	2.39	38.00	2.39	1.00	0.00	0.16	0.01	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.17	0.03	-	-
Module-star-20-5-1-15-0-10-10	14.75	0.89	621.75	16.07	37.50	1.77	37.50	1.77	1.00	0.00	0.20	0.03	-	-
Module-star-20-5-1-20-0-10-10	15.25	1.67	639.12	51.42	38.50	3.34	38.50	3.34	1.00	0.00	0.23	0.05	-	-
Module-star-20-5-1-10-0-5-10	15.62	1.19	596.88	53.05	39.25	2.38	39.25	2.38	1.00	0.00	0.15	0.02	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.17	0.03	-	-
Module-star-20-5-1-10-0-20-10	14.50	1.31	687.12	146.33	37.00	2.62	37.00	2.62	1.00	0.00	0.22	0.02	-	-
Module-star-20-5-1-10-0-40-10	15.00	1.41	1342.25	554.53	38.00	2.83	38.00	2.83	1.00	0.00	0.79	0.56	-	-
Module-star-10-5-2-10-10-10-10	14.57	0.98	368.00	25.66	222.57	488.33	229.57	506.41	8.00	18.08	1.22	2.78	-	-
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	4.48	8.90	0.1	-
Module-star-30-5-2-10-10-10-10	16.14	0.90	919.14	71.92	576.86	1130.72	597.71	1177.69	21.86	47.11	4.33	8.86	0.1	-
Module-star-40-5-2-10-10-10-10	16.29	1.38	1162.57	68.73	51.86	32.28	52.86	34.93	2.00	2.65	0.37	0.28	0.1	0.1
Module-star-20-10-2-10-10-10-10	15.38	0.92	947.62	71.61	835.00	2158.79	869.12	2248.15	27.62	68.23	6.55	17.26	-	-
Module-star-20-20-2-10-10-10-10	16.00	0.76	1710.12	144.13	297.75	713.37	305.75	735.60	9.00	22.23	2.40	5.88	-	-
Module-star-20-40-2-10-10-10-10	17.62	1.30	3379.00	441.85	131.00	167.23	134.75	173.98	4.75	6.94	1.06	1.44	-	-
Module-star-20-5-2-10-10-10-5	7.71	0.95	329.29	44.26	63.29	116.30	67.71	128.02	5.43	11.72	0.37	0.82	-	0.1
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	4.48	8.90	0.1	-
Module-star-20-5-2-10-10-10-15	25.14	2.19	1051.57	205.93	4748.14	8071.01	4906.29	8324.20	159.14	296.88	56.26	112.49	0.2	0.1
Module-star-20-5-2-10-10-10-20	32.12	1.13	1306.75	101.49	571.12	906.50	589.88	948.52	19.75	44.02	3.84	5.99	0.1	-
Module-star-20-5-2-10-10-10-25	40.25	3.85	1598.00	168.47	5328.50	11139.97	5616.25	11762.43	288.75	622.69	46.26	104.18	0.2	-
Module-star-20-5-2-5-10-10-10	16.43	1.40	736.57	134.64	983.14	1610.18	1066.14	1729.58	84.00	134.72	6.63	10.80	-	0.1
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	4.48	8.90	0.1	-
Module-star-20-5-2-15-10-10-10	14.62	0.92	572.38	45.05	37.75	2.49	37.75	2.49	1.00	0.00	0.20	0.03	-	-
Module-star-20-5-2-20-10-10-10	15.25	1.28	588.75	54.46	69.12	87.27	71.00	92.57	2.00	2.83	0.41	0.62	-	-
Module-star-20-5-2-10-10-5-10	16.25	0.46	622.88	42.21	94.00	130.40	96.12	135.61	3.12	5.22	0.47	0.74	-	-
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	4.48	8.90	0.1	-
Module-star-20-5-2-10-10-20-10	16.43	1.13	692.43	114.70	16394.29	20686.18	16952.14	21362.97	558.86	698.72	128.59	160.01	0.4	-
Module-star-20-5-2-10-10-40-10	14.80	0.84	657.00	47.46	132.20	90.06	135.20	92.60	4.00	2.55	1.01	0.58	0.3	-

*Module-star-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.8: Star pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-star-10-5-1-10-0-10-10	15.75	1.28	400.88	35.50	39.50	2.56	39.50	2.56	1.00	0.00	0.11	0.02	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.14	0.02	-	-
Module-star-30-5-1-10-0-10-10	16.38	0.92	966.25	86.15	40.75	1.83	40.75	1.83	1.00	0.00	0.16	0.02	-	-
Module-star-40-5-1-10-0-10-10	15.62	2.07	1122.25	114.03	39.12	3.91	39.12	3.91	1.00	0.00	0.16	0.02	-	-
Module-star-20-10-1-10-0-10-10	16.00	0.93	938.00	55.89	40.00	1.85	40.00	1.85	1.00	0.00	0.16	0.01	-	-
Module-star-20-20-1-10-0-10-10	16.50	0.76	1642.25	190.97	41.00	1.51	41.00	1.51	1.00	0.00	0.19	0.03	-	-
Module-star-20-40-1-10-0-10-10	18.00	0.76	3246.38	318.77	44.00	1.51	44.00	1.51	1.00	0.00	0.23	0.04	-	-
Module-star-20-5-1-10-0-10-5	7.38	0.52	313.12	20.95	17.75	1.04	17.75	1.04	1.00	0.00	0.04	0.01	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.14	0.02	-	-
Module-star-20-5-1-10-0-10-15	23.25	1.49	978.50	116.23	59.50	2.98	59.50	2.98	1.00	0.00	0.24	0.04	-	-
Module-star-20-5-1-10-0-10-20	31.75	1.67	1335.25	95.73	81.38	3.25	81.38	3.25	1.00	0.00	0.36	0.02	-	-
Module-star-20-5-1-10-0-10-25	40.25	2.87	1748.50	152.00	103.38	5.55	103.38	5.55	1.00	0.00	0.49	0.05	-	-
Module-star-20-5-1-5-0-10-10	15.00	1.20	642.00	39.63	38.00	2.39	38.00	2.39	1.00	0.00	0.12	0.02	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.14	0.02	-	-
Module-star-20-5-1-15-0-10-10	14.75	0.89	621.75	16.07	37.50	1.77	37.50	1.77	1.00	0.00	0.12	0.01	-	-
Module-star-20-5-1-20-0-10-10	15.25	1.67	639.12	51.42	38.50	3.34	38.50	3.34	1.00	0.00	0.15	0.02	-	-
Module-star-20-5-1-10-0-5-10	15.62	1.19	596.88	53.05	39.25	2.38	39.25	2.38	1.00	0.00	0.12	0.01	-	-
Module-star-20-5-1-10-0-10-10	15.50	1.41	657.38	56.43	39.00	2.83	39.00	2.83	1.00	0.00	0.14	0.02	-	-
Module-star-20-5-1-10-0-20-10	14.50	1.31	687.12	146.33	37.00	2.62	37.00	2.62	1.00	0.00	0.13	0.03	-	-
Module-star-20-5-1-10-0-40-10	15.00	1.41	1342.25	554.53	38.00	2.83	38.00	2.83	1.00	0.00	0.34	0.24	-	-
Module-star-10-5-2-10-10-10-10	14.57	0.98	368.00	25.66	222.57	488.33	229.57	506.41	8.00	18.08	0.89	2.06	-	-
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	3.09	6.10	0.1	-
Module-star-30-5-2-10-10-10-10	16.14	0.90	919.14	71.92	576.86	1130.72	597.71	1177.69	21.86	47.11	3.10	6.23	0.1	-
Module-star-40-5-2-10-10-10-10	16.29	1.38	1162.57	68.73	51.86	32.28	52.86	34.93	2.00	2.65	0.25	0.20	0.1	0.1
Module-star-20-10-2-10-10-10-10	15.38	0.92	947.62	71.61	835.00	2158.79	869.12	2248.15	27.62	68.23	4.68	12.33	-	-
Module-star-20-20-2-10-10-10-10	16.00	0.76	1710.12	144.13	297.75	713.37	305.75	735.60	9.00	22.23	1.85	4.61	-	-
Module-star-20-40-2-10-10-10-10	17.62	1.30	3379.00	441.85	131.00	167.23	134.75	173.98	4.75	6.94	0.85	1.16	-	-
Module-star-20-5-2-10-10-10-5	7.71	0.95	329.29	44.26	63.29	116.30	67.71	128.02	5.43	11.72	0.25	0.55	-	0.1
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	3.09	6.10	0.1	-
Module-star-20-5-2-10-10-10-15	25.14	2.19	1051.57	205.93	4983.14	8541.11	5145.57	8795.04	163.43	300.14	53.24	111.79	0.2	0.1
Module-star-20-5-2-10-10-10-20	32.12	1.13	1306.75	101.49	571.12	906.50	589.88	948.52	19.75	44.02	2.94	4.74	0.1	-
Module-star-20-5-2-10-10-10-25	40.25	3.85	1598.00	168.47	5071.75	10441.89	5344.88	11024.81	274.12	583.16	45.32	104.33	0.2	-
Module-star-20-5-2-5-10-10-10	16.43	1.40	736.57	134.64	983.14	1610.18	1066.14	1729.58	84.00	134.72	5.16	8.68	-	0.1
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	3.09	6.10	0.1	-
Module-star-20-5-2-15-10-10-10	14.62	0.92	572.38	45.05	37.75	2.49	37.75	2.49	1.00	0.00	0.12	0.03	-	-
Module-star-20-5-2-20-10-10-10	15.25	1.28	588.75	54.46	69.12	87.27	71.00	92.57	2.00	2.83	0.29	0.44	-	-
Module-star-20-5-2-10-10-5-10	16.25	0.46	622.88	42.21	94.00	130.40	96.12	135.61	3.12	5.22	0.37	0.61	-	-
Module-star-20-5-2-10-10-10-10	17.25	3.33	699.75	146.76	591.62	1140.24	637.50	1225.77	46.88	87.30	3.09	6.10	0.1	-
Module-star-20-5-2-10-10-20-10	16.43	1.13	692.43	114.70	21681.71	27095.86	22429.29	28010.86	748.57	946.99	128.58	160.14	0.4	-
Module-star-20-5-2-10-10-40-10	14.80	0.84	657.00	47.46	132.20	90.06	135.20	92.60	4.00	2.55	0.53	0.28	0.3	-

*Module-star-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]*

Table A.9: Tree pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\lvert\mathbf{M}\rvert$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-tree-10-5-1-10-0-10-10-b3	14.12	1.55	369.50	28.41	34.25	3.11	34.25	3.11	1.00	0.00	0.13	0.03	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.15	0.03	-	-
Module-tree-30-5-1-10-0-10-10-b3	14.50	0.76	840.62	64.34	35.00	1.51	35.00	1.51	1.00	0.00	0.17	0.02	-	-
Module-tree-40-5-1-10-0-10-10-b3	13.50	0.76	1066.62	78.81	33.00	1.51	33.00	1.51	1.00	0.00	0.19	0.02	-	-
Module-tree-20-10-1-10-0-10-10-b3	14.88	1.46	930.00	61.74	35.75	2.92	35.75	2.92	1.00	0.00	0.20	0.02	-	-
Module-tree-20-20-1-10-0-10-10-b3	15.25	1.28	1625.12	148.55	36.50	2.56	36.50	2.56	1.00	0.00	0.24	0.03	-	-
Module-tree-20-40-1-10-0-10-10-b3	16.38	0.74	3292.25	384.69	38.75	1.49	38.75	1.49	1.00	0.00	0.28	0.04	-	-
Module-tree-20-5-1-10-0-10-5-b3	7.12	0.99	311.50	55.33	16.25	1.98	16.25	1.98	1.00	0.00	0.04	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.15	0.03	-	-
Module-tree-20-5-1-10-0-10-15-b3	20.50	1.41	903.50	143.55	50.00	2.83	50.00	2.83	1.00	0.00	0.26	0.05	-	-
Module-tree-20-5-1-10-0-10-20-b3	28.00	1.60	1192.62	77.72	68.00	3.21	68.00	3.21	1.00	0.00	0.38	0.04	-	-
Module-tree-20-5-1-10-0-10-25-b3	35.75	1.58	1752.50	604.64	87.50	3.16	87.50	3.16	1.00	0.00	0.53	0.04	-	-
Module-tree-20-5-1-5-0-10-10-b3	13.75	1.16	603.50	82.80	33.50	2.33	33.50	2.33	1.00	0.00	0.14	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.15	0.03	-	-
Module-tree-20-5-1-15-0-10-10-b3	13.75	1.39	566.88	58.11	33.50	2.78	33.50	2.78	1.00	0.00	0.14	0.04	-	-
Module-tree-20-5-1-20-0-10-10-b3	14.00	0.93	594.62	58.24	34.00	1.85	34.00	1.85	1.00	0.00	0.18	0.03	-	-
Module-tree-20-5-1-10-0-5-10-b3	14.00	0.76	569.88	40.01	34.00	1.51	34.00	1.51	1.00	0.00	0.11	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.15	0.03	-	-
Module-tree-20-5-1-10-0-20-10-b3	14.62	1.19	793.50	227.32	35.25	2.38	35.25	2.38	1.00	0.00	0.24	0.03	-	-
Module-tree-20-5-1-10-0-40-10-b3	14.12	0.64	3166.25	2334.47	34.25	1.28	34.25	1.28	1.00	0.00	2.95	5.64	-	-
Module-tree-20-5-1-10-0-10-10-b2	12.88	0.83	524.75	30.65	29.75	1.67	29.75	1.67	1.00	0.00	0.13	0.02	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.15	0.03	-	-
Module-tree-20-5-1-10-0-10-10-b5	14.75	1.28	616.12	52.07	36.50	2.56	36.50	2.56	1.00	0.00	0.16	0.03	-	-
Module-tree-10-5-2-10-10-10-10-b3	18.38	7.98	449.88	185.68	478.62	734.31	505.00	778.74	27.38	45.96	2.90	4.61	0.1	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.19	0.04	0.1	-
Module-tree-30-5-2-10-10-10-10-b3	14.14	1.21	887.14	73.45	3295.43	8490.36	3589.71	9263.23	259.00	676.88	28.96	74.11	0.2	-
Module-tree-40-5-2-10-10-10-10-b3	15.12	0.83	1189.50	127.69	2389.62	6572.08	2741.00	7559.45	352.38	987.38	37.83	105.77	0.1	-
Module-tree-20-10-2-10-10-10-10-b3	14.12	0.99	914.75	70.71	8256.75	15196.07	8659.25	15939.37	403.50	745.86	75.15	138.55	0.3	-
Module-tree-20-20-2-10-10-10-10-b3	15.38	0.92	1627.38	155.23	68.88	61.77	69.88	63.70	2.00	1.93	0.50	0.48	-	-
Module-tree-20-40-2-10-10-10-10-b3	16.43	0.53	3293.43	330.48	43.29	10.50	43.43	10.88	1.14	0.38	0.31	0.11	0.1	0.1
Module-tree-20-5-2-10-10-10-5-b3	7.12	1.13	306.12	39.92	45.75	48.62	48.25	52.22	3.50	4.21	0.24	0.32	-	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.19	0.04	0.1	-
Module-tree-20-5-2-10-10-10-15-b3	20.25	1.04	873.62	60.53	9852.50	16347.27	10277.88	17109.86	426.38	771.54	70.11	117.30	0.2	-
Module-tree-20-5-2-10-10-10-20-b3	27.83	1.33	1164.33	34.77	5261.67	11513.32	5484.33	12038.39	223.67	525.43	35.29	77.61	0.2	0.1
Module-tree-20-5-2-10-10-10-25-b3	36.50	2.00	1527.50	92.35	5529.25	11816.77	5825.00	12528.21	296.75	712.93	37.60	81.18	0.1	-
Module-tree-20-5-2-5-10-10-10-b3	16.14	1.77	651.86	98.62	3967.43	5903.56	4127.57	6118.94	161.14	224.39	29.18	47.02	-	0.1
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.19	0.04	0.1	-
Module-tree-20-5-2-15-10-10-10-b3	13.86	0.90	581.29	59.67	36.86	4.74	37.86	7.13	2.00	2.65	0.20	0.07	0.1	-
Module-tree-20-5-2-20-10-10-10-b3	14.62	0.74	615.50	30.97	35.75	1.67	35.75	1.67	1.00	0.00	0.21	0.02	-	-
Module-tree-20-5-2-10-10-5-10-b3	13.25	1.04	544.25	25.04	32.75	2.38	32.75	2.38	1.00	0.00	0.12	0.01	-	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.19	0.04	0.1	-
Module-tree-20-5-2-10-10-20-10-b3	15.20	1.64	653.20	69.20	15314.80	14784.40	16435.00	16179.98	950.60	1208.03	141.22	141.60	0.3	0.1
Module-tree-20-5-2-10-10-40-10-b3	14.75	1.26	667.00	81.45	11284.50	13905.35	12329.00	15341.06	659.25	748.97	110.25	135.37	0.5	-
Module-tree-20-5-2-10-10-10-10-b2	12.71	0.95	559.00	27.10	43.29	21.59	44.14	23.04	1.71	1.25	0.20	0.15	-	0.1
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.19	0.04	0.1	-
Module-tree-20-5-2-10-10-10-10-b5	19.00	11.34	711.75	263.18	1004.12	2101.04	1037.88	2178.61	34.75	77.69	7.26	15.94	0.1	-

*Module-tree-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]-b[#branch]*

Table A.10: Tree pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\|\mathbf{M}\|$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-tree-10-5-1-10-0-10-10-b3	14.12	1.55	369.50	28.41	34.25	3.11	34.25	3.11	1.00	0.00	0.09	0.02	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.12	0.02	-	-
Module-tree-30-5-1-10-0-10-10-b3	14.50	0.76	840.62	64.34	35.00	1.51	35.00	1.51	1.00	0.00	0.14	0.01	-	-
Module-tree-40-5-1-10-0-10-10-b3	13.50	0.76	1066.62	78.81	33.00	1.51	33.00	1.51	1.00	0.00	0.13	0.01	-	-
Module-tree-20-10-1-10-0-10-10-b3	14.88	1.46	930.00	61.74	35.75	2.92	35.75	2.92	1.00	0.00	0.14	0.02	-	-
Module-tree-20-20-1-10-0-10-10-b3	15.25	1.28	1625.12	148.55	36.50	2.56	36.50	2.56	1.00	0.00	0.16	0.03	-	-
Module-tree-20-40-1-10-0-10-10-b3	16.38	0.74	3292.25	384.69	38.75	1.49	38.75	1.49	1.00	0.00	0.23	0.03	-	-
Module-tree-20-5-1-10-0-10-5-b3	7.12	0.99	311.50	55.33	16.25	1.98	16.25	1.98	1.00	0.00	0.03	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.12	0.02	-	-
Module-tree-20-5-1-10-0-10-15-b3	20.50	1.41	903.50	143.55	50.00	2.83	50.00	2.83	1.00	0.00	0.20	0.03	-	-
Module-tree-20-5-1-10-0-10-20-b3	28.00	1.60	1192.62	77.72	68.00	3.21	68.00	3.21	1.00	0.00	0.30	0.02	-	-
Module-tree-20-5-1-10-0-10-25-b3	35.75	1.58	1752.50	604.64	87.50	3.16	87.50	3.16	1.00	0.00	0.41	0.04	-	-
Module-tree-20-5-1-5-0-10-10-b3	13.75	1.16	603.50	82.80	33.50	2.33	33.50	2.33	1.00	0.00	0.10	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.12	0.02	-	-
Module-tree-20-5-1-15-0-10-10-b3	13.75	1.39	566.88	58.11	33.50	2.78	33.50	2.78	1.00	0.00	0.10	0.02	-	-
Module-tree-20-5-1-20-0-10-10-b3	14.00	0.93	594.62	58.24	34.00	1.85	34.00	1.85	1.00	0.00	0.11	0.03	-	-
Module-tree-20-5-1-10-0-5-10-b3	14.00	0.76	569.88	40.01	34.00	1.51	34.00	1.51	1.00	0.00	0.09	0.01	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.12	0.02	-	-
Module-tree-20-5-1-10-0-20-10-b3	14.62	1.19	793.50	227.32	35.25	2.38	35.25	2.38	1.00	0.00	0.14	0.04	-	-
Module-tree-20-5-1-10-0-40-10-b3	14.12	0.64	3166.25	2334.47	34.25	1.28	34.25	1.28	1.00	0.00	1.17	1.78	-	-
Module-tree-20-5-1-10-0-10-10-b2	12.88	0.83	524.75	30.65	29.75	1.67	29.75	1.67	1.00	0.00	0.08	0.02	-	-
Module-tree-20-5-1-10-0-10-10-b3	14.25	1.16	619.38	59.55	34.50	2.33	34.50	2.33	1.00	0.00	0.12	0.02	-	-
Module-tree-20-5-1-10-0-10-10-b5	14.75	1.28	616.12	52.07	36.50	2.56	36.50	2.56	1.00	0.00	0.12	0.01	-	-
Module-tree-10-5-2-10-10-10-10-b3	18.38	7.98	449.88	185.68	478.62	734.31	505.00	778.74	27.38	45.96	2.28	3.64	0.1	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.12	0.03	0.1	-
Module-tree-30-5-2-10-10-10-10-b3	14.14	1.21	887.14	73.45	3295.43	8490.36	3589.71	9263.23	295.29	772.88	23.69	60.83	0.2	-
Module-tree-40-5-2-10-10-10-10-b3	15.12	0.83	1189.50	127.69	2469.62	6798.35	2833.00	7819.66	364.38	1021.32	37.72	105.82	0.2	-
Module-tree-20-10-2-10-10-10-10-b3	14.12	0.99	914.75	70.71	10365.50	19104.63	10871.00	20038.03	506.50	936.59	75.10	138.65	0.3	-
Module-tree-20-20-2-10-10-10-10-b3	15.38	0.92	1627.38	155.23	68.88	61.77	69.88	63.70	2.00	1.93	0.38	0.41	-	-
Module-tree-20-40-2-10-10-10-10-b3	16.43	0.53	3293.43	330.48	43.29	10.50	43.43	10.88	1.14	0.38	0.23	0.08	0.1	0.1
Module-tree-20-5-2-10-10-10-5-b3	7.12	1.13	306.12	39.92	45.75	48.62	48.25	52.22	3.50	4.21	0.18	0.25	-	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.12	0.03	0.1	-
Module-tree-20-5-2-10-10-10-15-b3	20.25	1.04	873.62	60.53	11497.12	20290.11	12072.38	21435.42	522.12	1019.77	62.45	110.37	0.1	-
Module-tree-20-5-2-10-10-10-20-b3	27.83	1.33	1164.33	34.77	5261.67	11513.32	5484.33	12038.39	223.67	525.43	27.02	59.04	0.2	0.1
Module-tree-20-5-2-10-10-10-25-b3	36.50	2.00	1527.50	92.35	5529.25	11816.77	5825.00	12528.21	296.75	712.93	31.35	67.58	0.1	-
Module-tree-20-5-2-5-10-10-10-b3	16.14	1.77	651.86	98.62	3967.43	5903.56	4127.57	6118.94	161.14	224.39	26.86	45.48	-	0.1
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.12	0.03	0.1	-
Module-tree-20-5-2-15-10-10-10-b3	13.86	0.90	581.29	59.67	36.86	4.74	37.86	7.13	2.00	2.65	0.13	0.04	0.1	-
Module-tree-20-5-2-20-10-10-10-b3	14.62	0.74	615.50	30.97	35.75	1.67	35.75	1.67	1.00	0.00	0.13	0.02	-	-
Module-tree-20-5-2-10-10-5-10-b3	13.25	1.04	544.25	25.04	32.75	2.38	32.75	2.38	1.00	0.00	0.09	0.02	-	-
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.12	0.03	0.1	-
Module-tree-20-5-2-10-10-20-10-b3	15.20	1.64	653.20	69.20	18237.00	18108.47	19549.20	19749.58	1008.00	1232.44	127.05	138.59	0.3	0.1
Module-tree-20-5-2-10-10-40-10-b3	14.75	1.26	667.00	81.45	12040.50	15320.64	13161.00	16901.17	866.25	1103.62	95.45	138.98	0.5	-
Module-tree-20-5-2-10-10-10-10-b2	12.71	0.95	559.00	27.10	43.29	21.59	44.14	23.04	1.71	1.25	0.15	0.11	-	0.1
Module-tree-20-5-2-10-10-10-10-b3	13.38	0.92	583.00	24.76	35.50	5.63	35.62	5.95	1.12	0.35	0.12	0.03	0.1	-
Module-tree-20-5-2-10-10-10-10-b5	19.00	11.34	711.75	263.18	1004.12	2101.04	1037.88	2178.61	34.75	77.69	5.46	11.86	0.1	-

*Module-tree-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]-b[#branch]*

Table A.11: Random pattern without instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\|\mathbf{M}\|$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-random-10-5-1-10-0-10-10-d10	2.38	1.77	178.88	27.13	3.75	3.54	3.75	3.54	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-30-5-1-10-0-10-10-d10	4.62	4.07	505.38	151.81	9.25	9.60	9.25	9.60	1.00	0.00	0.04	0.05	-	-
Module-random-40-5-1-10-0-10-10-d10	4.00	2.93	561.38	119.95	7.50	6.41	7.50	6.41	1.00	0.00	0.03	0.03	-	-
Module-random-20-10-1-10-0-10-10-d10	2.50	2.14	432.12	79.95	4.12	4.58	4.12	4.58	1.00	0.00	0.01	0.01	-	-
Module-random-20-20-1-10-0-10-10-d10	3.25	2.87	856.25	202.70	5.62	5.73	5.62	5.73	1.00	0.00	0.03	0.03	-	-
Module-random-20-40-1-10-0-10-10-d10	2.12	1.36	1452.50	296.20	3.50	3.02	3.50	3.02	1.00	0.00	0.02	0.02	-	-
Module-random-20-5-1-10-0-10-5-d10	1.62	0.52	150.00	13.05	2.38	1.19	2.38	1.19	1.00	0.00	0.01	0.00	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-15-d10	3.88	3.48	466.00	88.51	6.88	7.26	6.88	7.26	1.00	0.00	0.02	0.03	-	-
Module-random-20-5-1-10-0-10-20-d10	9.88	9.00	732.38	225.52	18.38	18.16	18.38	18.16	1.00	0.00	0.13	0.15	-	-
Module-random-20-5-1-10-0-10-25-d10	33.62	4.47	1644.38	310.16	84.00	16.96	84.00	16.96	1.00	0.00	1.90	1.08	-	-
Module-random-20-5-1-5-0-10-10-d10	4.25	2.76	333.50	74.31	7.50	6.05	7.50	6.05	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-15-0-10-10-d10	2.88	1.46	304.75	32.88	5.00	3.21	5.00	3.21	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-1-20-0-10-10-d10	1.88	1.13	275.25	30.26	2.88	2.42	2.88	2.42	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-5-10-d10	3.75	1.83	310.50	39.58	6.50	3.55	6.50	3.55	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-20-10-d10	3.00	2.07	294.88	50.10	4.88	3.94	4.88	3.94	1.00	0.00	0.02	0.02	-	-
Module-random-20-5-1-10-0-40-10-d10	2.00	0.76	371.50	143.31	3.12	1.73	3.12	1.73	1.00	0.00	0.05	0.05	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-10-d15	6.38	4.63	403.12	132.12	12.88	10.89	12.88	10.89	1.00	0.00	0.07	0.09	-	-
Module-random-20-5-1-10-0-10-10-d20	9.12	4.52	479.25	114.80	19.88	11.14	19.88	11.14	1.00	0.00	0.14	0.11	-	-
Module-random-20-5-1-10-0-10-10-d25	11.62	6.72	547.50	177.91	32.75	21.12	32.75	21.12	1.00	0.00	0.39	0.30	-	-
Module-random-10-5-2-10-10-10-10-d10	4.62	3.34	219.12	56.63	8.25	6.69	8.38	6.61	1.12	0.35	0.04	0.04	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-30-5-2-10-10-10-10-d10	3.25	2.05	425.25	69.11	14.25	24.56	15.12	27.00	1.88	2.47	0.07	0.15	0.1	-
Module-random-40-5-2-10-10-10-10-d10	1.25	0.46	474.38	29.89	1.50	0.93	1.50	0.93	1.00	0.00	0.01	0.00	-	-
Module-random-20-10-2-10-10-10-10-d10	2.62	2.20	461.50	84.04	5.12	5.57	5.12	5.57	1.00	0.00	0.01	0.02	-	-
Module-random-20-20-2-10-10-10-10-d10	2.75	3.24	822.38	224.45	6.50	11.84	6.62	12.19	1.12	0.35	0.04	0.09	-	-
Module-random-20-40-2-10-10-10-10-d10	3.50	2.93	1578.88	352.96	6.25	6.25	6.38	6.14	1.12	0.35	0.04	0.04	-	-
Module-random-20-5-2-10-10-10-5-d10	1.75	0.89	166.25	34.45	2.62	2.00	2.62	2.00	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-15-d10	6.50	3.66	511.38	113.65	13.12	8.54	13.12	8.54	1.00	0.00	0.06	0.05	-	-
Module-random-20-5-2-10-10-10-20-d10	12.88	8.41	832.50	251.74	1273.38	2357.32	1325.00	2448.52	52.62	93.19	11.48	21.05	-	-
Module-random-20-5-2-10-10-10-25-d10	39.71	7.30	1667.71	242.74	622.86	954.97	636.14	985.45	14.29	30.85	14.83	17.70	0.1	-
Module-random-20-5-2-5-10-10-10-d10	2.75	1.49	282.50	33.53	4.75	3.33	4.75	3.33	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-15-10-10-10-d10	1.25	0.46	257.75	20.83	1.50	0.93	1.50	0.93	1.00	0.00	0.00	0.00	-	-
Module-random-20-5-2-20-10-10-10-d10	4.12	3.04	336.25	78.70	10.00	10.74	10.12	11.04	1.12	0.35	0.04	0.04	-	-
Module-random-20-5-2-10-10-5-10-d10	3.88	2.64	336.12	72.73	6.25	4.46	7.12	3.94	1.00	0.00	0.02	0.02	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-20-10-d10	3.12	2.90	309.00	81.49	4.75	5.28	4.75	5.28	1.00	0.00	0.03	0.05	-	-
Module-random-20-5-2-10-10-40-10-d10	4.43	2.37	372.14	82.46	113.14	152.60	134.86	181.22	22.71	28.82	2.35	3.02	0.1	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-10-d15	7.75	3.77	423.62	99.65	21.38	23.04	21.75	24.06	1.38	1.06	0.13	0.18	-	-
Module-random-20-5-2-10-10-10-10-d20	13.00	4.00	595.00	137.48	2093.75	3893.46	2187.62	4073.31	94.88	180.59	22.81	43.88	0.1	-
Module-random-20-5-2-10-10-10-10-d25	15.00	3.16	673.25	112.76	250.75	460.29	259.62	482.08	9.88	22.00	3.34	5.62	-	-

*Module-random-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]-d[%density]*

Table A.12: Random pattern with instantiation splitting

Example	$V(CG_{\mathbf{P}_E}(\mathbf{M}))$		$\|\mathbf{M}\|$		Call To DLV		AS From DLV		Total AS		Time		Resource	
	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	avg	std.dev	time-out	mem-out
Module-random-10-5-1-10-0-10-10-d10	2.38	1.77	178.88	27.13	3.75	3.54	3.75	3.54	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-30-5-1-10-0-10-10-d10	4.62	4.07	505.38	151.81	9.25	9.60	9.25	9.60	1.00	0.00	0.03	0.04	-	-
Module-random-40-5-1-10-0-10-10-d10	4.00	2.93	561.38	119.95	7.50	6.41	7.50	6.41	1.00	0.00	0.03	0.03	-	-
Module-random-20-10-1-10-0-10-10-d10	2.50	2.14	432.12	79.95	4.12	4.58	4.12	4.58	1.00	0.00	0.01	0.01	-	-
Module-random-20-20-1-10-0-10-10-d10	3.25	2.87	856.25	202.70	5.62	5.73	5.62	5.73	1.00	0.00	0.02	0.02	-	-
Module-random-20-40-1-10-0-10-10-d10	2.12	1.36	1452.50	296.20	3.50	3.02	3.50	3.02	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-1-10-0-10-5-d10	1.62	0.52	150.00	13.05	2.38	1.19	2.38	1.19	1.00	0.00	0.01	0.00	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-15-d10	3.88	3.48	466.00	88.51	6.88	7.26	6.88	7.26	1.00	0.00	0.02	0.02	-	-
Module-random-20-5-1-10-0-10-20-d10	9.88	9.00	732.38	225.52	18.38	18.16	18.38	18.16	1.00	0.00	0.10	0.11	-	-
Module-random-20-5-1-10-0-10-25-d10	33.62	4.47	1644.38	310.16	84.00	16.96	84.00	16.96	1.00	0.00	0.87	0.36	-	-
Module-random-20-5-1-5-0-10-10-d10	4.25	2.76	333.50	74.31	7.50	6.05	7.50	6.05	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-15-0-10-10-d10	2.88	1.46	304.75	32.88	5.00	3.21	5.00	3.21	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-20-0-10-10-d10	1.88	1.13	275.25	30.26	2.88	2.42	2.88	2.42	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-5-10-d10	3.75	1.83	310.50	39.58	6.50	3.55	6.50	3.55	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-20-10-d10	3.00	2.07	294.88	50.10	4.88	3.94	4.88	3.94	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-40-10-d10	2.00	0.76	371.50	143.31	3.12	1.73	3.12	1.73	1.00	0.00	0.04	0.05	-	-
Module-random-20-5-1-10-0-10-10-d10	2.38	1.69	290.25	32.90	3.88	3.60	3.88	3.60	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-1-10-0-10-10-d15	6.38	4.63	403.12	132.12	12.88	10.89	12.88	10.89	1.00	0.00	0.05	0.06	-	-
Module-random-20-5-1-10-0-10-10-d20	9.12	4.52	479.25	114.80	19.88	11.14	19.88	11.14	1.00	0.00	0.09	0.08	-	-
Module-random-20-5-1-10-0-10-10-d25	11.62	6.72	547.50	177.91	32.75	21.12	32.75	21.12	1.00	0.00	0.22	0.18	-	-
Module-random-10-5-2-10-10-10-10-d10	4.62	3.34	219.12	56.63	8.25	6.69	8.38	6.61	1.12	0.35	0.02	0.02	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-30-5-2-10-10-10-10-d10	3.25	2.05	425.25	69.11	14.25	24.56	15.12	27.00	1.88	2.47	0.05	0.11	0.1	-
Module-random-40-5-2-10-10-10-10-d10	1.25	0.46	474.38	29.89	1.50	0.93	1.50	0.93	1.00	0.00	0.01	0.00	-	-
Module-random-20-10-2-10-10-10-10-d10	2.62	2.20	461.50	84.04	5.12	5.57	5.12	5.57	1.00	0.00	0.01	0.01	-	-
Module-random-20-20-2-10-10-10-10-d10	2.75	3.24	822.38	224.45	6.50	11.84	6.62	12.19	1.12	0.35	0.03	0.06	-	-
Module-random-20-40-2-10-10-10-10-d10	3.50	2.93	1578.88	352.96	6.25	6.25	6.38	6.14	1.12	0.35	0.04	0.04	-	-
Module-random-20-5-2-10-10-10-5-d10	1.75	0.89	166.25	34.45	2.62	2.00	2.62	2.00	1.00	0.00	0.01	0.00	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-15-d10	6.50	3.66	511.38	113.65	13.12	8.54	13.12	8.54	1.00	0.00	0.05	0.04	-	-
Module-random-20-5-2-10-10-10-20-d10	12.88	8.41	832.50	251.74	1273.38	2357.32	1325.00	2448.52	52.62	93.19	8.75	15.98	-	-
Module-random-20-5-2-10-10-10-25-d10	39.71	7.30	1667.71	242.74	622.86	954.97	636.14	985.45	14.29	30.85	7.15	11.07	0.1	-
Module-random-20-5-2-5-10-10-10-d10	2.75	1.49	282.50	33.53	4.75	3.33	4.75	3.33	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-15-10-10-10-d10	1.25	0.46	257.75	20.83	1.50	0.93	1.50	0.93	1.00	0.00	0.00	0.00	-	-
Module-random-20-5-2-20-10-10-10-d10	4.12	3.04	336.25	78.70	10.00	10.74	10.12	11.04	1.12	0.35	0.03	0.03	-	-
Module-random-20-5-2-10-10-5-10-d10	3.88	2.64	336.12	72.73	6.25	4.46	7.12	3.94	1.00	0.00	0.02	0.01	-	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-20-10-d10	3.12	2.90	309.00	81.49	4.75	5.28	4.75	5.28	1.00	0.00	0.02	0.02	-	-
Module-random-20-5-2-10-10-40-10-d10	4.43	2.37	372.14	82.46	113.14	152.60	134.86	181.22	22.71	28.82	1.89	2.87	0.1	-
Module-random-20-5-2-10-10-10-10-d10	2.38	1.41	298.38	29.86	4.12	3.72	4.12	3.72	1.00	0.00	0.01	0.01	-	-
Module-random-20-5-2-10-10-10-10-d15	7.75	3.77	423.62	99.65	21.38	23.04	21.75	24.06	1.38	1.06	0.11	0.18	-	-
Module-random-20-5-2-10-10-10-10-d20	13.00	4.00	595.00	137.48	2093.75	3893.46	2187.62	4073.31	94.88	180.59	16.96	31.87	0.1	-
Module-random-20-5-2-10-10-10-10-d25	15.00	3.16	673.25	112.76	250.75	460.29	259.62	482.08	9.88	22.00	2.36	4.26	-	-

*Module-random-[#constant]-[#predicates]-[#head]-[#body]-[%*not*]-[#rules]-[#modules]-d[%density]*

APPENDIX B

Encoding

B.1 Hanoi Tower

B.1.1 Ordinary ASP

```
%---- Initial fact

succ(0,1). succ(1,2). succ(2,3). succ(3,4).
succ(4,5). succ(5,6). succ(6,7). succ(7,8).
succ(8,9). succ(9,10). succ(10,11). succ(11,12).
succ(12,13). succ(13,14). succ(14,15).

ndisk(4).
pathlength(15).

disk(1).
disk(X1) :- disk(X), succ(X,X1), X1<=Y, ndisk(Y).

peg(a).
peg(b).
peg(c).

situation(0).
situation(X1) :- situation(X), succ(X,X1), X1<=Y, pathlength(Y).

transition(0).
transition(X1):- transition(X), succ(X,X1), X1<Y, pathlength(Y).

location(Peg) :- peg(Peg).
location(Disk) :- disk(Disk).

%---- Initial situation
on(X,a,0) :- ndisk(X).
on(X,X1,0) :- X<Y, disk(X), X1=X+1, ndisk(Y).

%---- Inertial fluent: on(X,L,I) = disk X is on location L at time I
% inertia
on(X,L,T1) :- on(X,L,T), not otherloc(X,L,T1), location(L), transition(T), disk(X), T1=T+1.
otherloc(X,L,I) :- on(X,L1,I), L1!=L, situation(I), location(L), location(L1), disk(X).
% on unique location
:- on(X,L,I), on(X,L1,I), L!=L1, situation(I), location(L), location(L1), disk(X).

%---- Defined fluents
% inpeg(L,P,I) = location L is in peg P at time I
% top(P,X,I) = location L is the top of peg P. If empty, the top is P
inpeg(P,P,I) :- situation(I), peg(P).
```

```
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I), situation(I), location(L), disk(X), peg(P).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I), situation(I), location(L), peg(P).
covered(L,I) :- on(X,L,I), situation(I), location(L), disk(X).

%---- Generating actions
% pick one at each transition T
move(a,b,T) v move(a,c,T) v move(b,c,T) v move(b,a,T) v move(c,b,T) v move(c,a,T) :- transition(T).


%---- Effect axiom
on(X,L,T1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T), location(L), transition(T), disk(X), peg(P1), peg(P2), T1=T+1.

%---- State constraint: no disk X on a smaller one
:- on(X,Y,I), X>Y, situation(I), disk(X), disk(Y).

%---- Executability constraint
% the source peg cannot be empty
:- move(P1,P2,T), top(P1,P1,T), transition(T), peg(P1), peg(P2).

%---- Goal: at last situation, all disks in peg c
onewrong :- not inpeg(X,c,Y), disk(X), pathlength(Y).
:- onewrong.
```

B.1.2 MLP

```
#module(mainProgram, []).
succ(0,1). succ(1,2). succ(2,3). succ(3,4).
succ(4,5). succ(5,6). succ(6,7). succ(7,8).
succ(8,9). succ(9,10). succ(10,11). succ(11,12).
succ(12,13). succ(13,14). succ(14,15).

pathlength(15).
ndisk(4).
ok :- @solveHanoi[succ, ndisk, pathlength]::ok.


#module(solveHanoi,[succ/2, ndisk/1, pathlength/1]).
%---- Initial condition
disk(X)       :- @init[succ, ndisk, pathlength]::disk(X).
peg(X)        :- @init[succ, ndisk, pathlength]::peg(X).
situation(X)  :- @init[succ, ndisk, pathlength]::situation(X).
transition(X) :- @init[succ, ndisk, pathlength]::transition(X).
location(X)   :- @init[succ, ndisk, pathlength]::location(X).
on(X,Y,Z)     :- @init[succ, ndisk, pathlength]::on(X,Y,Z).

%---- Inertial fluent: on(X,L,I) = disk X is on location L at time I
% inertia
on(X,L,T1) :- on(X,L,T), not otherloc(X,L,T1), location(L), transition(T), disk(X), T1=T+1.
otherloc(X,L,I) :- on(X,L1,I), L1!=L, situation(I), location(L), location(L1), disk(X).
% on unique location
:- on(X,L,I), on(X,L1,I), L!=L1, situation(I), location(L), location(L1), disk(X).

%---- Defined fluents
% inpeg(L,P,I) = location L is in peg P at time I
% top(P,X,I) = location L is the top of peg P. If empty, the top is P
inpeg(P,P,I) :- situation(I), peg(P).
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I), situation(I), location(L), disk(X), peg(P).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I), situation(I), location(L), peg(P).
covered(L,I) :- on(X,L,I), situation(I), location(L), disk(X).

%---- Generating actions
% pick one at each transition T
move(a,b,T) v move(a,c,T) v move(b,c,T) v move(b,a,T) v move(c,b,T) v move(c,a,T) :- transition(T).

%---- Effect axiom
on(X,L,T1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T), location(L), transition(T), disk(X), peg(P1), peg(P2), T1=T+1.

%---- State constraint: no disk X on a smaller one
:- on(X,Y,I), X>Y, situation(I), disk(X), disk(Y).

%---- Executability constraint
% the source peg cannot be empty
```

```
:- move(P1,P2,T), top(P1,P1,T), transition(T), peg(P1), peg(P2).

%---- Goal: at last situation, all disks in peg c
onewrong :- not inpeg(X,c,Y), disk(X), pathlength(Y).
:- onewrong.
ok :- not onewrong.


#module(init,[succ/2, ndisk/1, pathlength/1]).
%---- Initial fact
disk(1).
disk(X1) :- disk(X), succ(X,X1), X1<=Y, ndisk(Y).

peg(a).
peg(b).
peg(c).

situation(0).
situation(X1) :- situation(X), succ(X,X1), X1<=Y, pathlength(Y).

transition(0).
transition(X1):- transition(X), succ(X,X1), X1<Y, pathlength(Y).

location(Peg) :- peg(Peg).
location(Disk) :- disk(Disk).

%---- Initial situation
on(X,a,0) :- ndisk(X).
on(X,X1,0) :- X<Y, disk(X), X1=X+1, ndisk(Y).
```

B.2 Packing

B.2.1 Ordinary ASP

```
area(6,4).
max_square_num(3).
square(1,4). square(2,2). square(3,2).

int(0). int(1). int(2). int(3).
int(4). int(5). int(6).

pos(I,X,Y) v npos(I,X,Y) :-
square(I,D), area(W,H), int(X), int(Y),
X >= 0, Y >= 0, X1 = X + D, Y1 = Y + D, W >= X1, H >= Y1.

:- pos(I,X,Y), pos(I,X1,Y1), X1 != X.
:- pos(I,X,Y), pos(I,X1,Y1), Y1 != Y.

pos_square(I) :- pos(I,X,Y).
:- square(I,D), not pos_square(I).

%top left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2), I1 != I2,
W1 = X1+D1, H1 = Y1+D1, X2 >= X1, X2 < W1, Y2 >= Y1, Y2 < H1.

%bottom left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2), I1 != I2,
W1 = X1+D1, H1 = Y1+D1, H2 = Y2+D2, X2 >= X1, X2 < W1, H2 > Y1, H2 <= H1.

:- overl(I1,I2).
```

B.2.2 MLP: Encoding 1

```
#module(main, []).
```

```
area(6,4).
max_square_num(3).
square(1,4). square(2,2). square(3,2).

int(0). int(1). int(2). int(3).
int(4). int(5). int(6).

pos(X,Y,Z) :- @solvePacking[int, square, area]::pos(X,Y,Z).

#module(solvePacking, [int/1, square/2, area/2]).
pos(I,X,Y) v npos(I,X,Y) :-
square(I,D), area(W,H), int(X), int(Y),
X >= 0, Y >= 0, X1 = X + D, Y1 = Y + D, W >= X1, H >= Y1.

:- pos(I,X,Y), pos(I,X1,Y1), X1 != X.
:- pos(I,X,Y), pos(I,X1,Y1), Y1 != Y.

pos_square(I) :- pos(I,X,Y).
:- square(I,D), not pos_square(I).

%top left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2),
I1 != I2, W1 = X1+D1, H1 = Y1+D1, X2 >= X1, X2 < W1, Y2 >= Y1, Y2 < H1.

%bottom left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2), I1 != I2,
W1 = X1+D1, H1 = Y1+D1, H2 = Y2+D2, X2 >= X1, X2 < W1, H2 > Y1, H2 <= H1.

:- overl(I1,I2).
```

B.2.3 MLP: Encoding 2

```
#module(main, []).

area(6,4).
max_square_num(3).
square(1,4). square(2,2). square(3,2).

int(0). int(1). int(2). int(3).
int(4). int(5). int(6).

pos(X,Y,Z) :- @solvePacking[int, square, area]::pos(X,Y,Z).

#module(solvePacking, [int/1, square/2, area/2]).
pos(X,Y,Z) :- @generatePos[int, square, area]::pos(X,Y,Z).

%top left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2),
I1 != I2, W1 = X1+D1, H1 = Y1+D1, X2 >= X1, X2 < W1, Y2 >= Y1, Y2 < H1.

%bottom left
overl(I1,I2) :-
pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2), I1 != I2,
W1 = X1+D1, H1 = Y1+D1, H2 = Y2+D2, X2 >= X1, X2 < W1, H2 > Y1, H2 <= H1.

:- overl(I1,I2).

#module(generatePos, [int/1, square/2, area/2]).
pos(I,X,Y) v npos(I,X,Y) :-
square(I,D), area(W,H), int(X), int(Y),
X >= 0, Y >= 0, X1 = X + D, Y1 = Y + D, W >= X1, H >= Y1.

:- pos(I,X,Y), pos(I,X1,Y1), X1 != X.
:- pos(I,X,Y), pos(I,X1,Y1), Y1 != Y.

pos_square(I) :- pos(I,X,Y).
:- square(I,D), not pos_square(I).
```

B.3 Even-Odd

B.3.1 MLP

```
#module(p1,[]).
% put q here: q(a). q(b). q(c). ...
even :- @p2[q]::even.
odd :- not even.

#module(p2,[q2/1]).
q2i(X) v q2i(Y) :- q2(X), q2(Y), X!=Y.
skip2 :- q2(X), not q2i(X).
even :- not skip2.
even :- skip2, @p3[q2i]::odd.

#module(p3,[q3/1]).
q3i(X) v q3i(Y) :- q3(X), q3(Y), X!=Y.
skip3  :- q3(X), not q3i(X).
odd :- skip3, @p2[q3i]::even.
```

B.3.2 Ordinary ASP - Labeling Solution

```
% put q here: q(a). q(b). q(c). ...

% put successor relation here: succ(1,2). succ(2,3). succ(3,4). ...

% get id (elements of succ)
id(X) :- succ(X,Y).
id(Y) :- succ(X,Y).

% less than relation
lessthan(X,Y) :- succ(X,Y).
lessthan(X,Z) :- succ(X,Y), lessthan(Y,Z).

% guess an ordering
pid(X,Y) v opid(X,Y) :- q(X), id(Y).

% cannot be two elements assigned to the same number
:- pid(X,Y), pid(X,Z), Y!=Z.

% cannot be two number assigned to the same elements
:- pid(Y,X), pid(Z,X), Y!=Z.

% get to know who is in
qin(X) :- pid(X,Y).
idin(Y):- pid(X,Y).

% every q must be in
:- q(X), not qin(X).

% the least id should be in
:- id(X), id(Y), lessthan(X,Y), idin(Y), not idin(X).

% get the smallest in the ordering,
% assume the smallest number always in pid
nsmallest(Y) :- succ(X,Y).
smallest(X) :- not nsmallest(X), id(X).

% get the largest in the ordering,
% considering ids that occur in the set only
nlargest(X) :- succ(X,Y), idin(X), idin(Y).
largest(X) :- not nlargest(X), idin(X).

% toggle between even-odd
oddP(X) :- pid(X,Y), smallest(Y), q(X).
evenP(X) :- pid(X,Y), oddP(Z), pid(Z,PrevY), succ(PrevY,Y).
oddP(X) :- pid(X,Y), evenP(Z), pid(Z,PrevY), succ(PrevY,Y).

% get the end result
odd :- oddP(X), pid(X,Y), largest(Y).
even :- not odd.
```